REALISTIC URBAN LAYOUT MODELING FROM REAL DATA

Oriol Pueyo Vallet

Universitat de Girona
Doctoral Thesis

REALISTIC URBAN LAYOUT MODELING FROM REAL DATA

Oriol Pueyo Vallet

2015

Universitat de Girona
Doctoral Thesis

REALISTIC URBAN LAYOUT MODELING FROM REAL DATA

Oriol Pueyo Vallet
2015

Doctoral Programme in Technology

PhD Advisor
Dr. Gustavo Patow

A thesis submitted to get the degree of Doctor at the Universitat de Girona

Universitat de Girona

El Dr. Gustavo Patow, professor agregat del departament d'Informàtica, Matemàtica Aplicada i Estadística (IMAE) de la Universitat de Girona,

CERTIFICO:

Que aquest treball, titulat *Realistic urban layout modeling from real data*, que presenta l'Oriol Pueyo Vallet per a l'obtenció del títol de doctor, ha estat realitzat sota la meva direcció i que compleix els requeriments per poder optar a Menció Europea.

Signatura

Girona, 15 de novembre del 2015

Realistic urban layout modeling from real data, Oriol Pueyo Vallet, PhD in Technology, Universitat de Girona, IMAE - Informàtica, Matemàtica Aplicada i Estadística, 2015

En memòria de qui va ser exemple per tots aquells que el vàrem conèixer.
El teu record et mantindrà sempre entre nosaltres.
Una abraçada Carles!
1977 - 2015

(...)
Hi ha un home que ara estén les ales
Un home que ha tocat el cel
No tornarà mai més a casa
Hi ha algú que ja ha burlat el temps.
(...)
Seguirem lluitant
Ensorrarem els murs
Anirem sempre més lluny.
(...)

Sopa de Cabra

Dedicat a les meves princeses. Laura i Abril, us estimo!

ACKNOWLEDGMENTS

After these years working on this PhD, I certainly must be grateful for so many things and to so many people, for directly or indirectly helping me, that it is difficult not to forget anything or anyone –even if it can sound typical. Any way, I will try:

Obviously, my first acknowledgment is to Gus. Thank you so much for being my thesis advisor but also for teaching me in every meeting, to be my PhD travel colleague and my friend. Thank you so much for being there always and in such an optimistic attitude. I would not get it without your, more than once necessary, motivating words.

I want to thank also other CG community members for helping me with valuable discussions, meetings and advice: Pierre Poulin, Yiorgos Chrysanthou, Marta Fort, Carlos Vanegas, and also all anonymous reviewers. I did not mentioned in this list Michael Wimmer consciously: Michael, I want to specially thank you for hosting me in your Computer Graphics Group at Wien Technical University. It has been a really enriching experience to work in such an important research group. Also in this group of acknowledgments I want to add Xavier. Thank you for your advices on my work and organizing my time, your experience has been very helpful.

Vull agrair també al GGG i en particular la companyia i ajut rebuts de tots i cadascun dels companys que heu passat pel despatx 118 al llarg d'aquests anys: Santi, Mei, Albert, Fran, Raissel, Lien i Isma. Especialment a en Santi i la Mei per moments surrealistes que hem viscut junts entre cafès, papers i codi. Ha estat fantàstic! ☺

Tot i no haver-me ajudat directament en la feina feta durant aquests anys de doctorat, no vull deixar d'agrair a la meva família el sol fet de ser-hi. Gràcies papa, mama, Blanca i Adri perquè tot i haver passat millors i pitjors èpoques, sé que sempre tindré amb qui divertir-me, riure, enfadar-me o plorar. Us estimo molt!

Gràcies també a la meva segona família, els Quintanas: tius, cosins i "associats". I molt especialment a en Pau, en Josep i la Nuri. Sempre m'heu fet sentir com un més i heu mostrat interès per l'evolució de la meva tesi.

Sonarà estrany però vull recordar en aquestes línies un nom comú, el rem. El rem m'ha donat molt, grans alegries i decepcions, però sobretot principis, esperit

de superació, exigència i especialment una colla d'amics immillorables. Gràcies per ser com sou i saber que sempre puc i podré comptar amb vosaltres.

També vull recordar-me de la colla de "amics i punt" i dels companys d'entrenament i amics de triatló. Us he tingut abandonats, massa, però us prometo que... Tornaré!

He separat els meus agraïments en blocs: companys de despatx, amics, companys d'esport, i noi... n'hi ha un que apareix per tot arreu. Sik, no sé com t'ho fots però sempre hi ets pel mig, i me n'alegro! ☺

Finalment vull mostrar el meu profund agraïment a aquelles persones amb qui he decidit compartir la meva vida, la meva dona i la meva filla. Laura, vull que sàpigues que valoro moltíssim el teu suport, comprensió, consells i afecte. Els valoro en els bons i especialment en els mals moments que he passat, quan més difícil ha estat aguantar-me i conviure amb mi. Ja ho saps: "els dies bons... gairebé som invencibles!". Abril, petit tresor de la casa. Admiro la teva capacitat de treure'm de pollaguera i en breus instants robar el millor dels meus somriures. Sàpigues que sempre, sempre, hi seré, pel que em necessitis. Aquesta tesi us ha robat molt del meu temps, ara tocarà recuperar-lo. És un autèntic plaer poder compartir aquest viatge anomenat vida al vostre costat. Us estimo amb bogeria!

I don't want to forget thanking Girona and Barcelona local city councils for providing their cadastral data and also the whole Open Street Maps community for doing such an enormous job.

Thanks also to grants and organizations that helped me:

- BR-UdG PhD scholarship grant.
- Ajut per la mobilitat d'investigadors de la UdG.
- ViRVIG and IMAE for providing me an office and other useful infrastructures and knowledges.

This work has been done under the context of the following projects: TIN2010-20590-C02-02, TIN2011-14860-E, TIN2013-47137-C2-2-P, TIN2014-52211-C2-2-R and 2014 SGR 896.

PUBLICATIONS

As a result of this thesis, several papers have been published or are in process to be submitted to well known Computer Graphics conferences or journals.

- **Structuring urban data.** Oriol Pueyo and Gustavo Patow. *The Visual Computer*, volume: 30, number: 2, pages: 159–172, publisher: Springer. February 2014. doi: 10.1007/s00371-013-0791-7 [57]
- **The CityLib system.** Oriol Pueyo and Gustavo Patow. *To be submitted*. [59]
- **City Shrinking.** Oriol Pueyo, Gustavo Patow and Michael Wimmer. *To be submitted*. [58]
- **An overview of generalisation techniques for street networks.** Oriol Pueyo, Xavier Pueyo and Gustavo Patow. *To be submitted*. [60]

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

CAD Computer-Aided Design

CG Computer Graphics

CGA Computer Graphics Architecture

DDA Digital Differential Analyzer

DXF Drawing Exchange Format

GIS Geographical Information System

LOD level-of-detail

OSM OpenStreetMap

POI Points of Interest

PSLG Planar Straight-Line Graph

STAR state-of-the-art report

SVG Scalable Vector Graphics

XML eXtensible Markup Language

XSLT eXtensible Stylesheet Language Transformation

CONTENTS

Part I

WARM-UP

To know what you know and what you do not know, that is true knowledge.

Confucius

ABSTRACT

Urban modeling is a common topic in Computer Graphics applications. Urban modeling includes several sub-topics depending on the urban elements to work with: street network layouts, buildings, nature elements (like rivers, gardens, street vegetation, ...) or traffic simulation, among others. In this thesis we basically work with urban street layouts. Urban modeling can be used to create new cities, but here we are interested in city models based on real world cities. A realistic city model can be a real full model or not. Real full models are those in which every element (street, intersection, building or tree) in the real world has a match with the same element in the synthetic model. These real full models are usually used in geographical information systems, public administrations, architecture, design or even the cinema industry. However, there are also realistic models without necessarily being an exact replica of the real world city. They are based as well on real world cities but their difference is the fact that, without keeping all the city's elements, they keep the essence of the real city from a perceptual point of view. These realistic city models are basically used in the video game industry, but also in cinema, theater or other entertainment industries. Therefore, to get a good realistic urban model, it is necessary to get a real data source and then, depending on the expected usage, make it smaller without loosing its essential characteristics.

To create a realistic geometric city model, it is always needed to provide the geographical information of the real world city. One important source of 2D geographical information is cadastral urban data. This data is always available at public administrations like city councils and governments. In fact, it is the most accurate data source and the only one with legal value. However, this information is not always well structured because it is often plagued with errors and inconsistencies, usually introduced by council workers who care more about visual rendered appearance than data consistency when modifying it with any GIS application. We present in this thesis a robust and generic solution to generate well structured block and building layouts based on a repairing process applied when these data are not correct. Our semi-automatic 2D structuring algorithm corrects errors and ambiguities. It identifies not just simple elements from the input, but also their connectivity and structure in the corresponding real world elements. Then we generate the urban data hierarchically structured in blocks and buildings. From this 2D well structured urban layout, we get a realistic 3D model by extruding each building with its cadastral defined height.

One of the most important uses of realistic city environments is in the video game industry. When a company works on a game whose action occurs in a real world environment, a team of designers must draw and model a simplified model of the

real city. They simplify the city by reducing its area, removing insignificant streets and bringing important places closer together. Otherwise, the narrative rhythm of the history would be too slow when moving from one place to another. The process of designing a real inspired but simplified city model is a manual task starting from zero, where the first step is to draw the street network of the final designed city. We have proposed a solution to automatically generate a first draft of the simplified city street network. We provide the designer team a shrunk layout from which to continue their design. This shrunk city reduces the real city area as much as desired, but always preserving the landmarks and most important streets, keeping their relative positions. In this way, we get a smaller city but with the essence of the real world one. This makes the video game player feel like being there, in the original real city, although it is just a simplified version.

To work with a city model and enable basic operations over it to run as fast as possible, we have designed and implemented a new data type (*city*) which uses different data structures to store its street network. Depending on the algorithm we want to execute on the street network, we implement it over one structure or another, always looking to get the optimal computational cost for each operation. We have also designed and implemented a system to simulate the evolution of a city, by defining a pipeline of operator modules that are applied sequentially to objects of type *city*. All the operators are independent (like black boxes) and interchangeable to ease any pipeline change.

RESUM

El modelatge urbà és un àmbit de recerca present en la comunitat de les aplicacions d'Informàtica Gràfica. Aquest àmbit inclou diversos camps de treball en funció dels elements urbans d'interès que s'hi tracten: xarxes de carrers, edificis, elements naturals (com ara rius, jardins, vegetació, ...) o simulació del trànsit, entre d'altres. En aquesta tesi bàsicament treballem en el context de les xarxes de carrers. El modelatge urbà pot ser usat per crear ciutats noves; però aquí estem interessats en aquells models basats en ciutats reals. Un model de ciutat realista pot ser un model exacte o no. Els models exactes són aquells en que cada element (carrer, intersecció, edifici o arbre) del món real compta amb el seu corresponent element en el model sintètic. Aquests models s'utilitzen normalment en sistemes d'informació geogràfica, administracions públiques, arquitectura, disseny o fins i tot en l'indústria del cinema. Però també hi ha models realistes que no necessàriament són rèpliques exactes de la ciutat en el món real. Són models igualment basats en ciutats reals però amb la diferència que, sense conservar tots els elements de la ciutat, mantenen la seva essència des del punt de vista perceptiu. Aquests models realistes de ciutats s'usen, bàsicament, en l'indústria dels videojocs així com en cinema, teatre i altres produccions artístiques. Així doncs, per obtenir models urbans realistes cal abans que res disposar d'una font de dades real. Després, i depenent de l'ús que es vulgui fer amb aquest model, cal reduir-ne les dimensions, sense perdre en cap moment les seves principals caracterísiques.

Per crear un model geomètric realista d'una ciutat cal sempre proporcionar la informació geográfica de la ciutat real. Una font important d'informació geogràfica 2D són les dades cadastrals de les ciutats. Aquesta informació està disponible en administracions públiques com ara ajuntaments i altres instàncies de govern. De fet aquestes fonts són les més acurades, donat que són les utilitzades a nivell oficial i legal. Però, malauradament, la informació no sempre està ben estructurada perquè sovint està farcida d'errors i inconsistències habitualment introduïts pels encarregats del manteniment de l'aplicació GIS que es preocupen més de la visualització de les dades que de la seva consistència geomètrica. En aquesta tesi presentem una solució robusta i genèrica per crear models ben estructurats d'illes i edificis basada en un procés de reparació quan les dades no són correctes. El nostre algorisme semiautomàtic d'estructuració 2D corregeix errors i ambigüitats. No només identifica els elements de les dades d'entrada, sinó que també troba la connectivitat i estructura dels elements corresponents en el món real. A continació, es generen les dades urbanístiques estructurades de forma jeràrquica en illes i edificis. A partir d'aquesta disposició ben estructurada s'obté un model 3D realista per extrusió de cada edifici usant la seva altura cadastral.

Un dels usos més importants d'entorns realistes de ciutats es troba en l'indústria dels videojocs. Quan es treballa en un joc l'acció del qual es desenvolupa en un entorn del món real, un equip de dissenyadors s'ocupa de dibuixar i modelar un model simplificat de la ciutat real. En aquesta simplificació redueixen l'àrea de la ciutat, eliminen carrers no rellevants i acosten les posicions relatives dels llocs importants de la ciutat. Altrament, el ritme narratiu de la història seria massa lent l'usuari s'hagués de desplaçar d'un lloc a un altre. El procés de disseny d'un model de ciutat inspirada en una de real però simplificada és manual. La primera etapa consisteix en el dibuix de la xarxa de carrers tal i com es desitgi distribuir la ciutat. Hem proposat una solució per generar automàticament un primer esborrany d'aquesta xarxa de carrers de la ciutat simplificada. Proporcionem una disposició de carrers de la ciutat reduïda a partir de la qual l'equip de disseny podrà continuar la seva tasca. Aquesta ciutat "encongida" redueix l'àrea real de la ciutat en la mesura del desitjat per l'usuari preservant sempre els llocs importants i els carrers principals, i mantenint les seves posicions relatives. D'aquesta manera obtenim una ciutat més petita però amb l'essència de la ciutat real. Això fa que l'usuari del videojoc se senti com si realment estigués en la ciutat de referència tot i tractar-se d'una versió simplificada.

Per treballar amb un model de ciutat que permeti aplicar-hi operacions bàsiques que s'executin el més ràpid possible, hem dissenyat i implementat un nou tipus de dades (*city*) que fa servir diverses estructures de dades per emmagatzemar la seva xarxa de carrers. Segons l'algorisme que vulguem aplicar a aquesta xarxa, s'implementa sobre una estructura de dades o una altra per tal d'obtenir un cost computacional òptim en cada cas. També hem dissenyat i implementat un sistema per simular l'evolució d'una ciutat amb un pipeline d'operadors que s'apliquen seqüencialment a objectes del tipus *city*. Aquests operadors són independents (capses negres) i intercanviables per facilitar qualsevol canvi en el pipeline.

1

INTRODUCTION

This chapter is an introduction to the work done in this thesis and what we are going to talk about in this document. We can find in this chapter an introduction to the topic in which this thesis is developed, the urban modeling and specifically for realistic cities. Also in this chapter, we expose the specific issues that we want to face and an explanation of how we want to contribute to the urban modeling community achieving our goals. Finally, we also present an overview of how this document is structured.

1.1 INTRODUCTION

In Computer Graphics (CG) applications, one of the ultimate goals is to generate a synthetic image from a given scene by means of a computer. A scene file contains 3D models in a strictly defined language or data structure, and it is usually described by geometry, camera, textures, lighting and shading properties. All these elements are necessary to get an accurate realistic synthetic image, if one of theme fails, the final results will not look as real as desired. Even so, we can consider the 3D models as the primary elements to create a computer-generated image. Precisely this point is the one of our main interest, how to generate 3D models of real urban environments. CG is a wide community with many different research lines in it. The CG line of research that studies the generation of synthetic urban environments is called Urban Modeling. Since early 2000, computer graphics has witnessed an explosion of research enthusiasm applied to the design of complex and realistic entire city environments, as well as individual building exteriors. This is certainly not surprising considering the great needs from architecture, cinema, video game and other entertainment industries for such synthetic environments. In Urban Modeling and CG applications, but also industry and research, we can see city models of synthetic environments or realistic environments. We want to focus our research on realistic models, which means that we want to get cities representing real world ones.

The scope of this thesis is the design of realistic city environments with data extracted from official documents. The work has been divided in two main parts, the first one is to create a 3D model from unstructured cadastral data. The second is to reduce a real city model area, while maintaining it essential characteristics. Both parts are expected to have a large impact on the different stakeholders, like local city councils or video game developers.

As mentioned, we want to work on the creation of real city models. To generate this kind of models, we have to start from real data. One of the most accurate sources of 2D geographical urban information is cadastral data. It is the most accurate data source, the only one with legal value and, moreover, it is freely available data on public administrations. However, this information is not structured in a useful way to create a 3D city model. Cadastral data is often plagued with errors and inconsistencies introduced by council workers when modifying it with any Geographical Information System (GIS) application. We have found a surprising lack of algorithms that automatically correct these errors and give a good structure to this data. Hence, our first challenge is to develop a robust and automatic method to detect and correct cadastral urban data, cleaning it from errors, ambiguities and inaccuracies. Then, we want to obtain well structured data by recognizing blocks, buildings or any other urban element from which we could be able to generate a 3D model to be rendered in a synthetic scene. In this way, we can get model with an accurate building footprints of any real city of the world based on it.

Nowadays, one of the most powerful industries of the world is the video game industry. Besides, many video games are set in real cities. To make the player feel like moving through a real city, the model designers should make a realistic model of the city inspired by the real one. This realistic model should keep the essence of the real city but reduce considerably its size. This city reduction should be done to avoid situations in which the player may spend too much time in irrelevant streets when going from one action point to another. To create such city, the first step is to create its street network. The designer thus starts by detecting the most important and characteristic streets and places (landmarks) that should be recognizable by the player, making an association between the synthetic city and the real one. A landmark can also be a key place for the game story, although perhaps without being a city famous place. The relative positions between landmarks should also be respected as much as possible; imagine a New York model in which Liberty Statue is in the North of the Manhattan island; this would not make good sense. Getting these landmarks detected, the designer should put them in closer positions one another. It also should keep the most important streets, avenues and parks. Finally it should fill the areas between landmarks with less important streets to get reasonable block densities. Once the city street network is created, the next step is to design the buildings in each city block, putting special effort in having detailed models of the landmarks. This is a manual process that designers often do. Our goal is to automate the process of getting a reduced street network from a real city. Our objective is to do this in a guided way, by removing the less important areas and preserving landmarks, important streets and their relative positions to get a smaller city while maintaining its original essence. The resulting city layout should be a good draft from where designers can continue the city model design.

When working on the shrinking city project, we realized that we needed a basic platform to store a city in a robust and efficient data structure, together with a set of basic algorithms to operate over it. Hence, we decided to implement a full library to work with city data, a platform from which other researchers and developers can start working without being concerned about how to store their city models or having to implement frequently used operations over street networks. The city data type should be an optimized implementation in which all the basic street network operations should go as fast as possible. Besides, we saw that our shrinking process is an evolution of a city state: from an original state to a smaller one. So, we implemented a generic way of simulating a city evolution by applying a set of operators over the target city, sequentially. To do it, we define an interface that any new operator should accomplish, and then define a pipeline in which the library user can, in a flexible way, define the chain of operators to transform the city. This challenge has resulted in the *CityLib* library, which implements the *city* data type, a set of basic operators and the ^{CT}Ops system to make the operator pipeline.

1.2 THESIS OVERVIEW

We have organized this thesis in the following chapters:

CHAPTER 1: INTRODUCTION:
This first chapter is an introductory chapter to place the reader in urban modeling and explain what this thesis is about.

CHAPTER 2: PREVIOUS WORK:
The chapter establishes the basics to understand the urban modeling community work and interests. We will also present the different publications related to the creation of real city models, and the difficulties to work with real urban data providers. Then, we will analyze different publications doing city street network generalizations and transformations with some similarities to our shrinking process. In this final part we will also introduce different data structures used in other papers to store a street network.

CHAPTER 3: STRUCTURING URBAN DATA:
In this chapter, we will explain our semi-automatic method to detect, process and correct 2D urban data. We introduce the common errors, ambiguities and inaccuracies that appear in an administration's cadastral data and how we find and fix them. Then we will propose a process to structure the different entities used to "draw" the maps into a hierarchical structure of blocks and buildings that can finally be extruded to get 3D models of real cities.

CHAPTER 4: SHRINKING CITIES:
In this chapter, we present a process that automates the first steps that designers should do when they want to create a new real city model. We will see how we transform an original city layout in a simplified one by reducing less important areas and streets. The result is a smaller city with its essence preserved by keeping its most important places and their relative positions. We will explain how this process can help the video game industry when designing real city-inspired models.

CHAPTER 5: SYSTEM: THE *citylib* LIBRARY:
This chapter is more technical and it explains the library we have implemented to support the shrinking process. We will present the *CityLib* library with its three main parts: The *city* data type, a set of basic tools that can be applied over a city street network and the ^{CT}Ops (City Operations) pipeline to simulate any urban evolution on a *city* model.

CHAPTER 6: CONCLUSIONS:
The chapter recapitulates and summarizes the goals achieved in previous chapters. We present our main conclusions about the *Structuring* and *Shrinking* processes and also about the implementation of the *CityLib* library.

CHAPTER 7: FUTURE WORK:

Finally, we explain in this last chapter all our ideas about how to continue our work, about the different future work that can be proposed from the processes presented in this thesis.

2

PREVIOUS WORK

In this chapter we introduce the reader to the Urban Modeling area in Computer Graphics (CG) and contextualize the content of this thesis. Firstly, we give a brief introduction to Urban Modeling in the CG community and delimit our area of interest in it. Then we talk about the state of the art and publications related to our work: modeling real cities. Finally, we discuss the relevant contributions of papers that we had to consider before starting this thesis.

2.1 INTRODUCTION

New solutions have been proposed and many commercialized software products are dedicated to solve different problems related to urban and architectural environment representation, design or planning. In this thesis we are interested mainly on procedural techniques applied to city layout modeling, so our main focus is not on techniques for the design of buildings or their photo-realistic rendering. Said differently, in the context of urban representation we are not interested in the conception of synthetic imaginary cities. Nonetheless, the results of our work may be used in these contexts.

Our first interest is on the representation of real cities from cadastral information provided by a city local council, looking for a robust solution that optimizes the guarantee of using our results for applications that either require coherent real cities or may take advantage of it, including entertainment and marketing applications in need of a realistic city model. Our other interest is on how to reduce a real city area without loosing its essence by keeping its points of interest and maintaining their relative positions; a smaller city thus inspired by a real one. We mean by "real city" an existing one or a project for a future urban environment. See Figure 1.

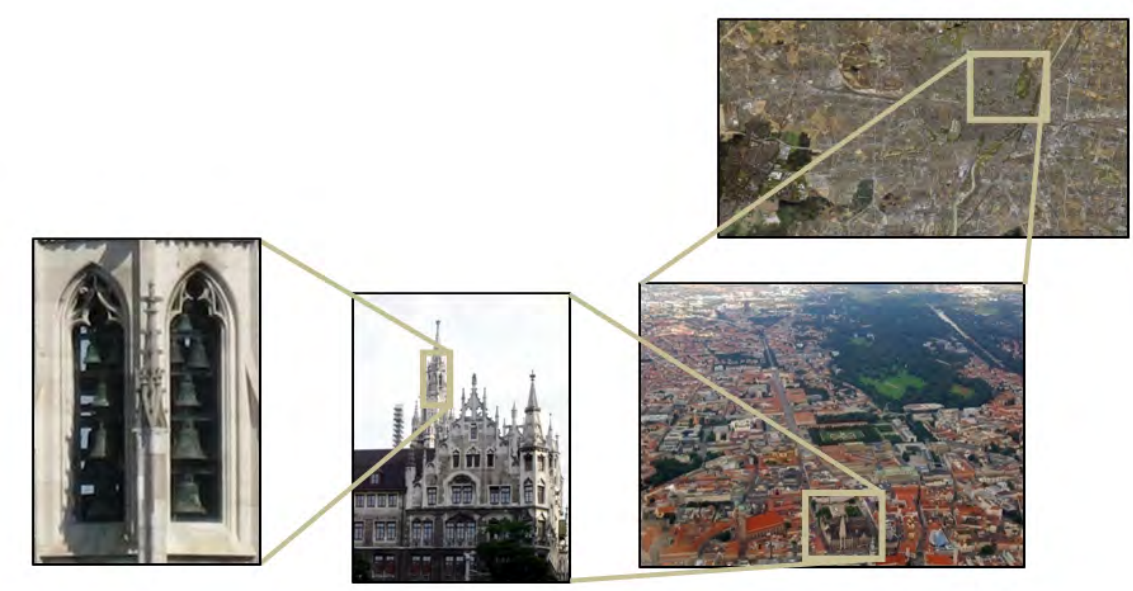

Figure 1: Real city procedural modeling includes considerations for different city levels of detail.
Reprinted from [70].

In this chapter, we focus on the previous work related to our areas of interest and goals. In this context, we thought from the beginning that we had to study some of the work on synthetic city generation and building reconstruction given that some problems could be common with our work.

2.2 URBAN MODELING

Existing methods for procedural urban modeling roughly use the pipeline shown in Figure 2, or at least a part of it. As we can see, the traditional urban modeling pipeline is a 5-step one, where two of its steps (roads and buildings) can be further subdivided.

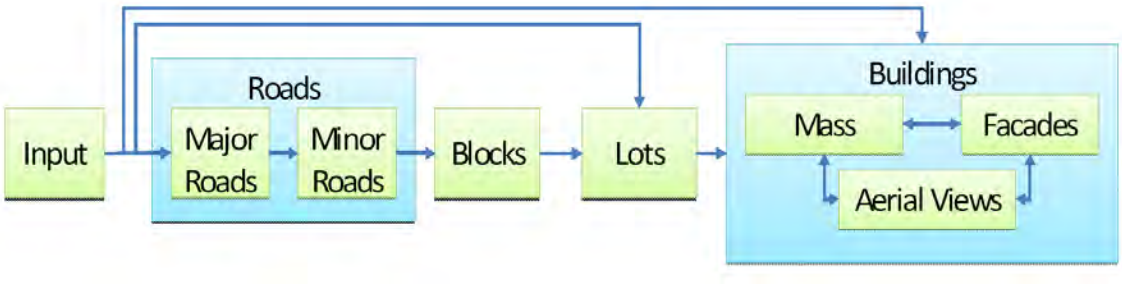

Figure 2: The traditional urban modeling pipeline.
Reprinted from [70].

The first conceptual step, the input, usually works on certain ranges of inputs, like Vanegas et al.'s state-of-the-art report (STAR) [70] explains. Some papers have as input a specific target architectural design, like Dosch et al. [20], Lewis and Séquin [42] and Yin et al. [76]. Aliaga et al. [5] use as input a 3D model, GIS data or imagery. Other papers use socioeconomic data or elevation data, like Vanegas et al. [69], or tensor fields like Chen et al. [17]. Note that none of these approaches aim at using accurate city data, generally available from local city council GIS databases.

Road generation usually is divided in two different stages: Firstly, major roads like avenues are produced, and after, minor roads are built in the areas generated between the major roads. Techniques for generating these roads usually include extended L-systems (Parish and Müller [52]), hyper-streamlines (Chen et al. [17]), directed random walks (Aliaga et al. [5]) and growth from seeds or traffic simulation (Weber et al. [72]). Again, we observe a surprising lack of accurate data processing, but this is understandable, as most of these techniques are aimed at the creation of synthetic cities for entertainment or conceptual design purposes.

Following our pipeline, the next stage, creation of blocks and lots, usually consists either of some sort of recursive block subdivision [52, 72] or a Voronoi-diagram-based subdivision algorithm [5, 69] to fill in the areas between streets. Again, both types of methods are focused on the production of synthetic cities, without any attention to accuracy issues with respect to real cities. Once a layout exists, either generated from scratch or imported, it may be modified using local edition of streets and blocks, or applying more sophisticated techniques. These techniques provide the user with the possibility of adding, removing or changing the locations of elements with a larger (global) effect on the new layout. An example of

this could be to change the location of an important street (let us say a street traversing the whole city for example) like Lipp et al. [43] do. Other proposed powerful tools insert or merge layouts into (with) another layout designed independently, like Aliaga et al. [5, 4] did, and again Lipp et al. [43] later on.

The final stage, building modeling, usually uses shape/split grammars (Müller et al. [51] and Parish and Müller [52]), mass/façade modeling (Whiting et al. [73]), build by numbers (Aliaga et al. [3] and Bekins et al. [8]) or image-based synthesis techniques.

Most of these papers deal with new approaches or improving techniques for the use of parametric tools for the generation of synthetic cities [5, 38, 51, 52, 71, 72, 74, 76], but no one of them directly face the cleaning process of real urban data. Let us first introduce the paper main features and afterwards discuss the elements of these contributions more strictly related to our work.

2.3 STRUCTURING

The first part of the thesis deals with the robustness of city cadastral data in order to be able to derive reliable 3D models of real cities. What we actually realized is that there is an important void in the field of CG concerning the study of restoring cadastral data in order to provide reliable city structures. In this section we review the geometric models of city layouts used in previous work. We analyze the format of the 2D information in relevant urban modeling papers as well as in different papers surveying urban modeling and building generation. We also review papers that explicitly refer to real cadastral data.

Parish and Müller [52] proposed the first procedural approach to city modeling, at least to our knowledge, based on L-systems (the system is named City Engine), previously used in other applications like tree generation and blood vessel networks. L-systems are used for both street generation and building modeling. The system starts generating a network of streets, roads, and highways, based on socio-statistical information. Afterwards, it divides populated areas into smaller areas, which each represents city blocks, taking into account given city rules like, for instance, minimum block area. Finally, buildings are modeled in each block, also using city rules like, for example, the relation between block area and building height. The input information does not include cadastral data representing real blocks of a given city. Blocks are generated using a recursive algorithm that stops at a given area threshold. Therefore, the system creates cities that do not repre-

sent real cities, even though, by introducing some specific feature rules, generated cities could resemble existing ones. See Figure 3. Actually, an important constraint is introduced in the block generation, as they are assumed to be "... convex and rectangularly shaped. The system therefore forbids the creation of concave allotments."

Figure 3: A procedural network model of Manhattan using an L-system for streets creation. *Top*: The L-system-based growth of streets. *Middle*: The resulting roadmap. *Bottom*: The real Manhattan's street network.
Reprinted from [52].

Later, Müller et al. [51] presented a novel grammar called Computer Graphics Architecture (CGA) *shape* for the procedural modeling of high quality buildings with geometric details. It is designed to produce complex architectural environments for computer games and movies at low cost.

Weber et al. [72] introduced interactivity that allows edition during the simulation of cities over time (Aliaga et al. [5] also allows it in a different way). The authors claim also to be the first to propose an urban simulation system that includes realistic geometric configurations that are not forced on a regular grid. Nonetheless, we have understood that Parish and Müller [52] blocks are neither forced to be on a regular grid. As input, the pipeline of the proposed methods receives, in addition to environment information (height map, water map, forest map and land use type), an initial layout configuration ranging from a single street to a larger

network of streets, potentially from real existing streets. Information on blocks is not in the input data. Aliaga et al. [5] presented a novel technique to produce new layouts, combining street and block structure with aerial image of synthetic cities; either to create new urban layouts or to simulate the extension of new ones based on examples (of real or synthetic cities). The system permits several editing operations as joining, expanding and blending layout fragments. Lipp et al. [43] presented a method specifically dedicated to edit with user criteria existing city layouts.

Some of already mentioned references (Kelly and Mc Cabe [38], Watson et al. [71], Yin et al. [76]) along with Vanegas et al. [70], present different overviews of the field as surveys, tutorials or STARs. The most recent ones, so the most relevant from an up-to-date point of view, are proposed by Yin et al. [76] and Vanegas et al. [70]. The former is a survey with an overview and a comparison of techniques and systems designed to deal with the automatic generation of 3D building models from 2D architectural CAD-based drawings, as well as from architectural drawings, from floor-plan images inform of pictures or hand-made (like Dosch et al. [20]). Therefore, the part of this work of particular interest to us is about accurate 2D files of the city layouts. We are specially interested on Yin et al. [76] techniques to deal with the Computer-Aided Design (CAD) files. In Vanegas et al. [70], there is a first step where the paper drawings are scanned and processed in order to produce a file with segments or polylines (here human assistance may be even more unavoidable than in the other systems we will review in the following paragraph). After this phase, the problem is very similar to the one presented, not clean data to process. This is very usual in architectural CAD files. If the CAD files have been produced, modified, merged, etc. accurately the procedure is simpler.

One of the main contributions mentioned in the survey of Yin et al. [76] is the paper by Lewis and Séquin [42], (a similar method was developed at Massachusetts Institute of Technology) where they use as input CAD-based floor plans. The system applies a pre-process to organize the input in dedicated layers, as also done in other work like Mas and Besuievsky [49]. Referring to Lewis and Séquin [42], Yin et al. [76] say that: "Although some pre-process simplifies the recognition algorithm task, the geometry typically suffers from errors and ambiguities". This assertion, in our point of view, is a very important issue to take into account. Another interesting work, also referred in this survey, is the one by Lu et al. [45], based on a heuristic that recognizes certain patterns. The system is highly automated with adequate inputs, but it is fragile when the input quality is insufficient. Other techniques discussed in the survey require additional semantic information associated to the geometry input or deal with simplified input geometries. Yin et al. [76] insist on the need for a cleanup stage as both, vectorized hand-drawn images and computer-sketched drawings, suffer from disjointed segments, overlapping vertices and false intersections. We may add from our experience that

other problems arise like duplicated edges, adding further problems to the reconstruction process. Hence, we can conclude that it is a common understanding that, before working with polygons, designers must launch certain operations to clean up geometrical errors. They can do this cleanup manually or by using algorithms such as snapping to a grid. Although this solution may be suitable for architectural reconstructions, it proves inadequate for dealing with a whole city cadastral map. Also, in the conclusions of their survey, Yin et al. [76] state that few systems fully address the problem of generating 3D building models from 2D architectural drawings, and that they are not completely automated the computation of consistent layouts, being an open issue that still requires significant manual intervention and will continue to do so as long as architectural representations contain ambiguities or inconsistencies. The same can be said about cadastral city data.

Vanegas et al. [70] presented a STAR about very interesting topics and open problems in the field of city simulation. It covers the modeling and simulation of new urban environments, proposes solutions for restoration/conversion of restricted areas of a city, and simulates how a city will grow given an evolution model taking into account not only geometric constraints. Nothing is said concerning eventual errors and ambiguities of 2D layouts. The hypothesis of perfect 2D layouts is assumed.

Previous to these two overviews, other papers look at the problems from different points of view [38, 71, 74]. Kelly and McCabe [38] offer a systematic description of several procedural techniques for city generation. For each technique, they describe road network and building generations. After the description of each system, they provide a synthetic list of conclusions according to a number of criteria: realism, scale, variation, input, efficiency, control and real time. Concerning the input, of major interest for our work, the only significant information reports that it is important to avoid using real-world data as input. We believe that they said so perhaps because of their inherent problems: errors and ambiguities are commonplace in real-world data.

Wonka et al. [74] give, in their tutorial notes, a light introduction to Stiny's shape grammars and the algebra they are based on [65]. The tutorial deals with all the steps, including rendering and texturing. It is aimed at the entertainment (video game) applications. Concerning our topic of interest, the tutorial refers to the use of techniques where the user interacts with the model; thus, no contributions on automating the generation of the floor-plan model are presented. Also, Watson et al. [71] present a short survey of techniques for the generation of urban environments (streets and buildings) for film and game applications. It is devoted more to the generation of new cities than to the reconstruction of existing cities. Nevertheless, there is a paragraph reviewing the previous contributions on the use of 2D layouts as the basis for the modeling of the buildings.

After reviewing the survey references on 2D inputs and how they manage this information, we can mention a few other papers. Fabritius et al. [27] provide a new approach, that has as goal to make easy and real-time the creation of high-quality complex 3D city models from 2D map data and terrain informations. The proposed technique for model generation is based on 2D street and building maps of existing cities. Unfortunately, the authors state that the 2D data from real-world sources contained a considerable amount of degeneracies such as isolated segments, holes, redundant vertices, that would make their algorithm fail without a careful cleaning pre-processing step. Therefore, they use graph-based algorithms for cycle detection to identify loops. Unfortunately the authors do not say which algorithms they use neither how robust these algorithms are. They however say that, for a small number of remaining ambiguous or inconsistent configurations, it turned out that simple manual intervention was needed. Our objective is to avoid such manual cleaning step by providing a completely automatic and robust solution.

We finish the presentation of this previous work with two mutually strongly related papers by Flamanc et al. [29] and Taillandier [66]. These two papers present work done in the French Institut Géographique National (Laboratoire MATIS). Both papers deal with the modeling of 3D buildings/cities using aerial images in order to derive their roofs. When the roofs have been detected and categorized, the authors use cadastral maps to determine the walls of the buildings. In the two contributions we find the same kind of problems explicitly described by the authors. Flamanc et al. [29], after assuming correct 2D cadastral polygons, write that 2D ground map polygon edges are not always the real building facades. Then, they add that some user-assisted tools are absolutely necessary to correct them. In his discussion, Taillandier [66] underlines a limitation of the proposed technique, explicitly stating its inability to account for cadastral map errors.

2.3.1 *Summary*

So far we have tried to group the papers on the basis of their main objectives, always taking into account the proximity of these goals to our subject of interest. Below we summarize the related previous work along the lines of our structuring project, putting a special emphasis on aspects of our specific interest that we have found in the studied papers.

We often find that the systems start by, or have as a target, creating a network of streets based on different criteria. For example, Parish and Müller [52] start with an L-system generating a network of streets, roads and highways based on socio-statistical information. Then, they divide the populated areas into smaller areas called blocks, taking into account given city rules (for example, minimum block area). For Aliaga et al. [5], the objective is to produce new layouts for both, streets and blocks, creating structure of synthetic cities based on examples (from

real or synthetic cities). The system uses editing operations as joining, expanding and blending pieces from other layouts.

Concerning the generation of blocks, we found several approaches (the most advanced using social and administrative constraints). For Parish and Müller [52], the input does not include cadastral data defining real blocks. Blocks are generated using a recursive algorithm that stops at a given area threshold. Actually, an important constraint is introduced in the block generation assuming they are convex and rectangularly shaped. They also force the blocks to fit on a regular grid.

It is obviously of the highest interest for us, in the context of this work, to analyze how previous work deal, if it is the case, with cadastral geometric information. This data is imperative for systems working with real cities, while it is not with synthetic cities. Therefore, given that many papers are applied for entertainment purposes, they may obviate cadastral information. Although, sometimes they use such a type of information in order to generate city layouts similar to existing urban environments. In these cases, approximated cadastral information may be acceptable. As we have seen above, Flamanc et al. [29], when the roofs have been detected, use cadastral maps to determine the walls of the buildings assuming correct 2D cadastral polygons. Unfortunately, that assumption is not usually correct unless a pre-process has been applied. In other contributions [76] a cleanup process is applied to the input data in order to make the geometric cadastral CAD data usable. If the input is a scanned map, then another previous pre-process is needed to generate CAD-like files. As we will later see, even when the cleanup pre-process is applied, the data still remains prone to several errors and inconsistencies, given the numerous difficulties and ambiguities in the original data. Moreover, these papers do not give significant details on the cleanup pre-process itself.

To end this section we list a number of comments to summarize it. They are either compiled from the studied papers or derived from them (according to our best understanding). These comments from our point of view confirm the interest of the problem we have dealt with in our project, and so the interest of our results.

ASSUMPTIONS & LIMITATIONS

- Parish and Müller [52]. An important constraint is introduced in the block generation: they are assumed to be "[...] convex and rectangularly shaped. The system therefore forbids the creation of concave allotments." This can not obviously be applied for many real cities.
- Vanegas et al. [70]. Nothing is stated concerning eventual errors and ambiguities of the 2D layout. The hypothesis of coherent and correct 2D layouts is assumed.

MANUAL INTERVENTION

- Flamanc et al. [29]. They come to realize that cadastral data is not consistent: "Moreover, 2D ground map polygon edges are not always the real building facades" and their solution is to allow manual intervention: "it is absolutely necessary to add some user-assisted tools".
- Wonka et al. [74]. It refers to the use of techniques where the user interacts with the model. No contribution on automating the generation of the floor plan model is suggested.
- Fabritius et al. [27]. They faced the difficulties related to cadastral data: "Unfortunately the provided 2D data contained [...] a considerable amount of degeneracies such as isolated segments, holes, redundant vertices". They do not say which algorithms they used neither how robust these algorithms were to solve these problems. They actually point out: "However, for a small number of remaining and ambiguous or inconsistent configurations it turned out that simple manual intervention".

ROBUSTNESS

- Taillandier [66]. A limitation of their proposed technique is underlined: "its inability to account for cadastral [...] errors".
- Kelly and McCabe [38]. They consider that it is important to avoid using real world data as input. Perhaps because of the problems inherent to them, probably in the form of errors and ambiguities.
- Referring to the paper of Lewis and Séquin [42], Yin et al. [76] say: "Although this simplifies the recognition algorithm task, the geometry typically suffers from errors and ambiguities".
- Referring to the work of Lu et al. [45], Yin et al. [76] state that the system is highly automated with appropriate inputs, but its robustness is fragile when the input quality fails.

Let us conclude this section with Yin et al. [76], who insist in their survey on the need of an "error cleanup" stage as "Both vectorized, hand-drawn images and computer-sketched drawings suffer from disjointed lines, overlapping vertices and false intersections." The cleanup may be done "[...] manually or by using algorithms such as coerce-to-grid". In the conclusions of their survey we find that the "few systems that address the problem of generating 3D building models from 2D architectural drawings aren't completely automated" the computation of consistent layouts being an open issue given that it still requires "significant manual intervention and will continue to do so as long as architectural representations contain ambiguities or inconsistencies."

2.4 SHRINKING

The second part of our work deals with *simplification*, in a broad sense, of city layouts. We call this project *shrinking cities*. In the context of urban modeling we find two correlated concepts that embed simplification: *legibility* and *generalization*. *Urban legibility* was introduced by Lynch [46], stating that a legible city would be one whose districts, landmarks or pathways are easily identifiable and are easily grouped into an overall pattern. Lynch considers that people should intuitively understand a city including not only geometry but also semantic information. We did not find a formal, widely accepted definition for *generalization*. According to Edwardes and Mackaness [23], *generalization* is a complex process that attempts to mediate between detailed spatial data and cartographically represented geographic information. We also find definitions correlating both concepts, like by Raheja and Kumar [62], where they assert that the purpose of cartographic *generalization* is to represent a particular situation adapted to the needs of its users, with adequate *legibility* of the real situation and its perceptional congruity with the representation. We actually have concluded that both words are often used as synonyms even though the use of *generalization* prevails when referring to the process and *legibility* is used as the quality of the result of the process.

Müller et al. [50] differentiate two types of generalization in GIS: cartographical generalization and model-based generalization. We are interested on model-based generalization that can be defined as mainly oriented to a structural-based filtering, while cartographical generalization deals with geometrical simplification in a scale reduction process. Therefore, we will use from now on the words *simplification* and *generalization* indifferently as synonyms.

The contributions we have studied strongly rely on the use of graph theory tools even though other fundamental tools are used, such as information theory principles (Bjørke et al. [11, 12]) or tree data structures (Chang et al. [15, 16]). A reference work for the use of graphs in this field was provided by Mackaness and Beard [47]. In a *primal graph*, nodes represent street intersections and edges stand for streets or street segments while in a *dual graph*, nodes represent streets and edges stand for street intersections. For detailed discussion on the pros and cons of primal and dual graphs we may refer to the work by Porta et al. [54, 55].

Let us clarify here the *dual graph* concept. In mathematical graph theory, the *dual graph* has a node corresponding to each face of its *primal graph* and an edge joining two neighboring faces through each *primal graph* edge. A different graph is the *line graph* (also known as *edge-to-vertex dual graph*), where each vertex represents an edge of its *primal graph* and each edge represents two *primal graph* edges with a common endpoint. See Figure 4 for an example of a *primal graph* and its derived *dual graph* and *line graph*. Hence, what Porta et al. –but also other authors in the urban modeling community– call *dual graph*, is actually a *line graph* or, if we prefer,

an *edge-to-vertex dual graph*. Aliaga et al. [5, 4] and Edwardes et al. [22, 24, 23] use a *dual graph* but they named their graphs differently: *tiles graph* and *adjacency graph* respectively. Jiang and Claramunt [36] use what they call *connectivity graph*, which is in fact a *line graph*.

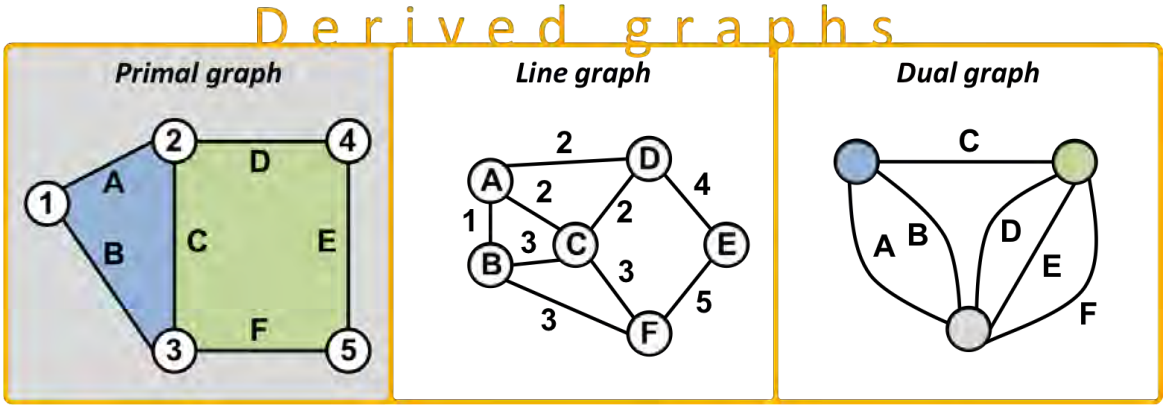

Figure 4: A primal graph (*Left*) and its derived line (*Middle*) and dual graphs (*Right*). Each node in the dual graph represents a colored area in the primal graph.

In a first approach we may classify the papers related with our focus of interest into two groups:

- Street network generalization.
- Application oriented generalization.

2.4.1 *Street network generalization*

In this section we review the background on generalization techniques for street networks, regardless of the application which the simplification is used for. As we will see, a number of techniques have been proposed in order to reduce the complexity of street networks, producing new models with less information while keeping the essential information. This is often associated to the change of scale of the model, reducing the details to be visualized, but it is also employed to obtain schematic representations (and, in general, better visualizations) of city street networks.

Edwardes et al. [22, 23, 24] presented one of the first contributions in this domain. Their simplification criterion is to reduce the number of city partitions by merging small areas into larger ones. In a first step they define strokes by means of concatenating streets with "good continuation", without branches. From these, cycles (polygons) representing partitions of the city are defined. Finally a dual graph is built where nodes represent partitions and links represent adjacency between partitions. Weights derived from the strokes separating adjacent partitions

are assigned to the links. This adjacency graph is the basis for the generalization of the network, which is carried out by removing the less important streets. This is achieved by eliminating partitions with an area below a given threshold while keeping the characteristic structure of the city's network and preserving its density distribution.

The principle of grouping partitions to simplify the model of a city is also applied by Chang et al. [15, 16]. The authors substantiate their technique on the concept of legibility from Lynch's book [46]. They aggregate elements in such a way that provides different levels of abstraction using a level-of-detail (LOD) hierarchy. We consider only the concepts dealing with layout simplification even though they also propose 3D visualization. In their first work [15], street networks and building footprints are considered. Therefore, their input is a collection of paths (the above strokes) representing streets, rivers, etc. and a collection of groups of edges, each one representing the footprint of a building. In their later work [16], the input information is just the set of building footprints of the city. In both cases they generate a hierarchical tree structure where leaves are building footprints and all other nodes represent clusters containing pairs of nearby buildings or clusters. Proximity is bounded by the width of the paths in order to group partitions on the same side of a street. A metric is proposed in both papers in order to obtain an acceptable balanced tree. In the last step, for each non-terminal node, the pair of elements (buildings or clusters) is merged, generating a single polygon (called *merged hull*) that represents the corresponding level of abstraction. Pruning the resulting tree at a given level provides the correlated simplified model of the city layout.

Other criteria for the simplification of street networks are the use of centrality (importance) measures assigned to elements of the network. Jiang and Claramunt [36] use a line graph representation (called *connectivity graph*) where nodes represent named streets (entire streets, not segments) and edges represent the intersection of the nodes (streets) it connects. Here streets play a role similar to the strokes of Edwardes et al. [23] and the paths of Chang et al. [15]. The following centrality measures may be assigned to each node:

- Degree: The number of edges incident upon a node.
- Closeness: The reciprocal of farness. Farness of a node is the sum of its distances from all other nodes, the distance between two nodes being the length of their shortest path.
- Betweenness: The number of times a node is traversed by the shortest path between two other nodes.

For each centrality measure, a hierarchical structure may be derived. The resulting tree is used to simplify the network representation by means of pruning it based on a given threshold of the centrality measure.

An interesting approach that is a bit far away from our current interest is the use of information theory principles to simplify road networks with the concept of perceptual similarity of two elements. Bjørke and Isaksen [12] apply this idea to delete roads that obtain close values when applying the similarity function. Of course, the candidate to be removed could have a semantic value that demands its presence anyway after the simplification process. In order to avoid removing undesired roads, the proposed technique uses additional information related to hierarchy and connectivity assigned to each road.

2.4.2 *Application-oriented generalization*

In different fields, network simplification techniques have been proposed for specific applications. We here review generalization methods designed to improve the usability of route maps, i.e., to enhance their legibility (visualization). To reach this goal there are two phases: determine the important elements (streets and Points of Interest (POI)) of the map and generate clear and uncluttered visualizations of them.

Some interesting work, not strictly related with our goal, have been proposed to enhance the legibility of maps, offering clear visualizations of 3D landmarks like by Grabler et al. [34] and Qu et al. [61]. These papers propose simplification techniques to highlight important streets and landmarks.

The map simplification approaches related to our research focus on the generation of the following routes:

- Route from a starting point to a destination point [2].
- Routes from anywhere in a given region to a destination point [40].
- Routes for multiple destination points [9, 40].

Agrawala et al. [2] proposed route (from a starting point to a destination point) generalization techniques based on cognitive psychology research, showing that an effective route map must clearly communicate all turning points on the route, and that precisely depicting the exact length, angle and shape of each road is much less important. Therefore, they keep turning points and propose three generalization operations: *length generalization*, to shorten or extend roads; *angle generalization*, to open tight angles; and *shape generalization*, to remove useless changes of road topology.

Kopf et al. [40] propose an automatic generation of destination maps. They consider that destination maps are navigational aids designed to show anyone within a region how to reach a location (the destination). This can be seen as a generalization of the problem solved by Agrawala et al. [2], but it is addressed in a different

way. A hierarchical navigation approach is used, selecting first globally important roads (e.g., highways) and taking less-important ways (arterial streets, residential streets) as they approach the destination point. In order to provide a good visualization, they apply well-known scale and orientation techniques to simplify selected streets.

We include in the topic of multiple destination points two similar but different problems. One of them is to automatically provide a legible and efficient set of routes to reach *n* POIs from anywhere in a given area embracing those POIs. The feasibility of solving this was experimentally verified by Kopt et al. using a hierarchical approach [40]. The other problem to solve is to automatically generate a legible and efficient set of routes to connect each of *n* POIs to the other *n-1* POIs. This is dealt with by Birsak et al. [9] to produce tourist brochures. The process starts computing all the shortest paths between the *n* POIs. The result is a cluttered and dense graph that requires a cleaning post-process to eliminate less significant and, eventually, redundant paths. This post-process is applied iteratively until the graph is sparse enough.

2.4.3 *Summary*

In our bibliography research we did not find papers addressing the problem we want to face, i.e., to reduce a city area, getting a new street network layout with closer POIs, while keeping the most important streets and the relative positions of these landmarks; in other words, to automatically get smaller cities, retaining their essence. In this section we have reviewed papers that focus on related problems. These are solved based on principles and/or tools that could be of our interest. We note that these contributions strongly rely on the use of graph theory tools [47], even though other fundamental tools are used, such as information theory principles [12] or tree data structures [15, 16]. Two groups of papers have been studied, one of them dealing with generic simplification operations and the other with a specific application field for the generation of ad hoc route maps. From the point of view of our work, the last group of contributions are mainly built on top of techniques introduced by the former group.

From this previous work study, we conclude that these papers contribute to techniques eventually useful for our work, even though none deals with problem we are interest on [15, 16, 22, 23, 24, 36]. We assume their input information is either a GIS database or a primal graph. All but Chang et al. [16] use, as basic modeling unit, a group of connected street segments called *strokes* [22, 24], *paths* [15] and *streets* (they are actual streets) [36]. Let us use *stroke* here to refer to these three elements. Edwards et al. [22, 24] use strokes to define cycles that bound areas (not necessarily lots, blocks or buildings), while Chang et al. [15] use building footprints in addition to strokes. On the other hand, Chang et al. [16] use building footprints as basic modeling units.

From a primal graph input, all papers build other structures to support their simplification techniques. As central structure of the proposed simplification techniques, Edwardes et al. [22, 24] build a dual graph called *adjacency graph* where nodes represent partitions (areas) and edges represent adjacency between partitions. A line graph called *connectivity graph* (nodes: streets, edges: street intersections) is used by Jiang and Claramunt [36]. Finally, Chang et al. [15, 16] build a *binary tree* where leaves are building footprints and all other nodes represent clusters containing pairs of nearby buildings or clusters. Edwardes et al. simplify the networks by means of aggregating small area nodes to one of its neighbors. Jiang and Claramunt's simplification criteria are to remove nodes based on their centrality values. Finally, Chang et al. base their simplification technique on merging the pairs of buildings or clusters of each node, generating the boundaries of a new cluster representing both.

Part II

CONTRIBUTION

It's a dangerous business, Frodo, going out your door. You step onto the road, and if you don't keep your feet, there's no knowing where you might be swept off to.

J.R.R. Tolkien, *The Lord of the Rings*

3

STRUCTURING URBAN DATA

Geometric city modeling is an open problem without standard solutions. Within this problem, several sub-problems must be faced, like the accurate modeling of streets, buildings and other architectural structures. One important source of geographical information is (measured) cadastral urban data. However, this information is not always well structured, and sometimes it is even simply some form of corrupted GIS data. In this chapter we present a robust and generic solution for the generation of block and building layouts based on a repairing process applied when this data is not correct. Our input data is a top projection map of a city which usually has been created by a mixture of photogrametric restitution and, in a second stage, of hand-drawn polylines, polygons or curves using any GIS application. Moreover, these maps are under continuous modifications, like in the case of public administrations. This process sometimes results in the introduction of mistakes and anomalies, which are hard to correct without the appropriate tools. Our solution is based on a novel semi-automatic 2D restructuring algorithm, which uniformly corrects errors and ambiguities that are commonly present in corrupted cadastral data. This problem is complex because it is necessary to identify not just simple elements from the input file, but also their connectivity and structure in the real world. The output of our algorithm is the urban data restructured into a hierarchy of blocks and buildings, from which we can get a realistic 3D model by extruding each building using the number of floors within the cadastral data.

3.1 INTRODUCTION

As said in this thesis introduction (Chapter 1), in recent years urban authorities have demonstrated an apparent urge to have high quality 3D models of their cities. These models are important for stakeholders such as architects and urban designers, cinema and theater, virtual reality and even videogames. Companies like Google® and Microsoft® use high-end digital cameras to acquire aerial or street-side images for large areas in a cost-efficient way. Once the images are captured, photogrametric restitution (e.g., [35]) plus manual touches complete the model. The Infrastructure for Spatial Information in the European Community (INSPIRE) [18], in 2007, defined rules for addressing the interoperability of spatial data, including cadastral data. When these guidelines are followed, cadastral information is good, reliable and well formatted. However, many local city councils still operate with legacy data that is plagued with errors and inconsistencies (usually introduced by council workers who care more about visual appearance than data consistency). This renders the data unreliable for 3D reconstruction. In these cases, local authorities face a difficult choice of hiring external 3D scanning services, assigning personnel to repair the information, or searching for a cheaper automated solution. As we concluded in previous work (Chapter 2), from a literature search on GIS and city modeling [28, 41, 53, 68, 44], we observe a surprising lack of algorithms that automatically (or semi-automatically) solve the problem: the most promising approaches up to now only provide partial solutions, and commercial packages fall short of correcting anything beyond the simplest situations (e.g., duplicated vertices, missing street intersections or open polygons) [63, 33, 48].

Our objectives are: (1) to develop a robust and semi-automatic method to detect, process and correct 2D urban data, cleaning up errors, ambiguities and inaccuracies that appear in legacy cadastral maps; and (2) to obtain well structured cadastral data, recognizing blocks, buildings and any interesting urban structure accurately while providing a reliable GIS structure. Figure 5 shows a mess of elements used to create a city blocks map. Once we get well structured data, we can take advantage of it. As practical example, we use other CAD layers like the number of floors of each building element to convert the 2D urban data into a 3D urban model.

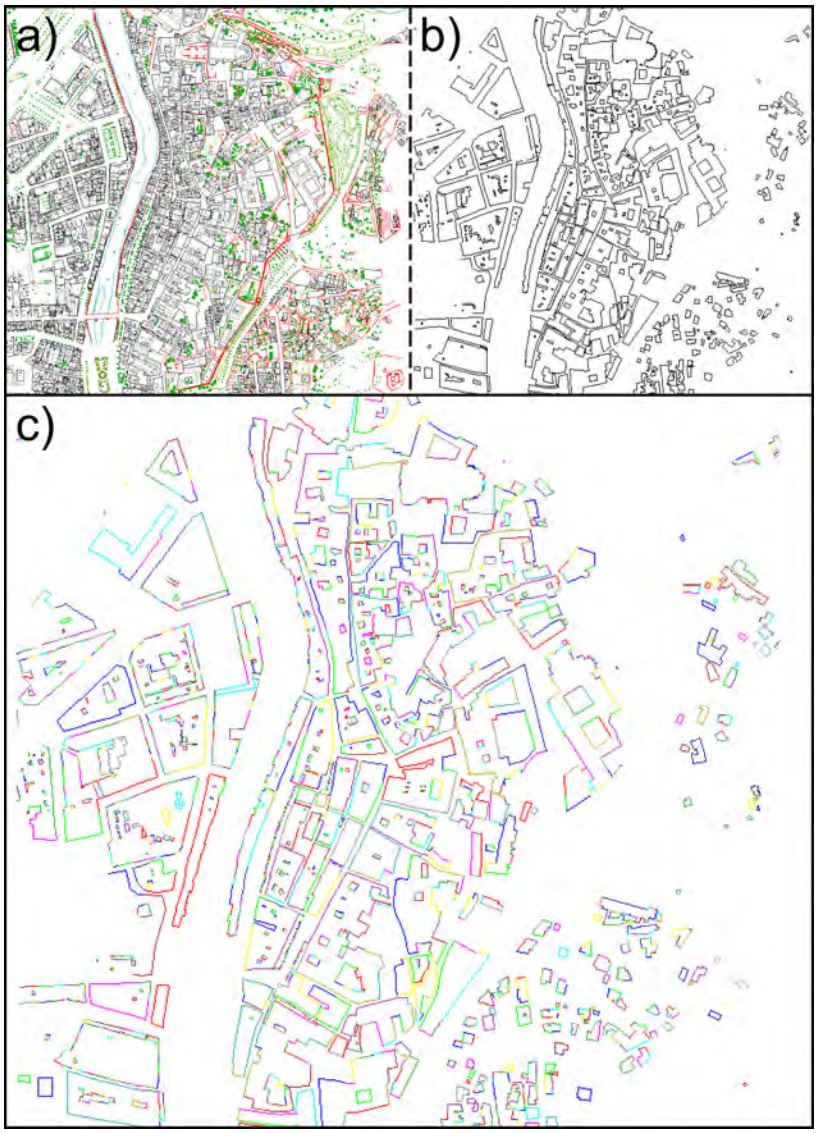

Figure 5: Input data from a CAD file: *(a)* complete CAD map, *(b)* only the block layer is activated in the CAD tool, and *(c)* block layer with each CAD entity using different colors.

3.2 DEFINITIONS

In this section we will introduce a few definitions that will be used throughout the rest of this manuscript. Most of these concepts are closely related to the input cadastral data layers and how we use them.

BLOCK: A block is a physical city block, which can be defined as a set of structures (e.g., buildings) surrounded by streets, or possibly gardens or squares.

SUB-BLOCK: These are the sub-areas into which a block can be subdivided. They are defined in the same block layer data, and they represent a partition of the block area. In Figure 13 (c.i) and (c.ii) we can see two schematic sub-block situations. In the case the block layer contains only the block outlines, there is only one sub-block for each block: the block itself. In Figure 8 we can observe that a sub-block only represents a sub-part of a block. It should not be confused with a lot, which has property implications.

PRE-BLOCK: It is a data structure that we use during computation, to represent all the polyline entities that define a block; and it containes buildings.

LOT: It is each of the minor closed areas into which a block (or sub-block) is subdivided. In general, in cadastral data, lot information is incomplete, and requires the building layer to, together, provide consistent information. Because of this, we decided to ignore this layer and directly work with the building layer, enriched with the polylines from the lot layer.

BUILDING: A building, in our case, is each of the minor closed areas into which a lot is subdivided. Buildings are formed by building elements. Observe that detecting which building elements are part of a given building requires explicit ownership relations to be set in the data, which might not be available.

BUILDING ELEMENT: Each of the closed polygons with different heights that define the parts of a building. If a building element has an associated zero height, then it is a *courtyard*. As a result of the fusion of the lot and building layers, the system will not be able to distinguish the building elements that belong to one building or another.

From now on, we will use the terms building and building element interchangeably, although we will always be referring to the latter.

3.3 OVERVIEW

As we said, our input is a two-dimensional GIS map. In general, local city councils use CAD tools to create and manage files with many layers of information storing not only buildings but also information about streets, public environment, hydrology, green spaces, sewers, electrical installations, etc. For our application, the interesting layers are the ones associated to blocks, lots, buildings and heights (number of floors data).

Manual editing of CAD files often introduces errors in the data. In Figure 6 we can see a simple and schematic example of a badly structured polygon. If we zoom out we easily appreciate that this polygon is a square. This visual information is usually enough for the vast majority of tasks that a council needs to do. But it is not enough for many others, like associating a height to each building to generate a 3D city model. For this purpose, we need closed and well defined polygons. If we zoom in near the square vertices and edges, we notice a set of inaccuracies. Here we can classify the data errors in the six following categories:

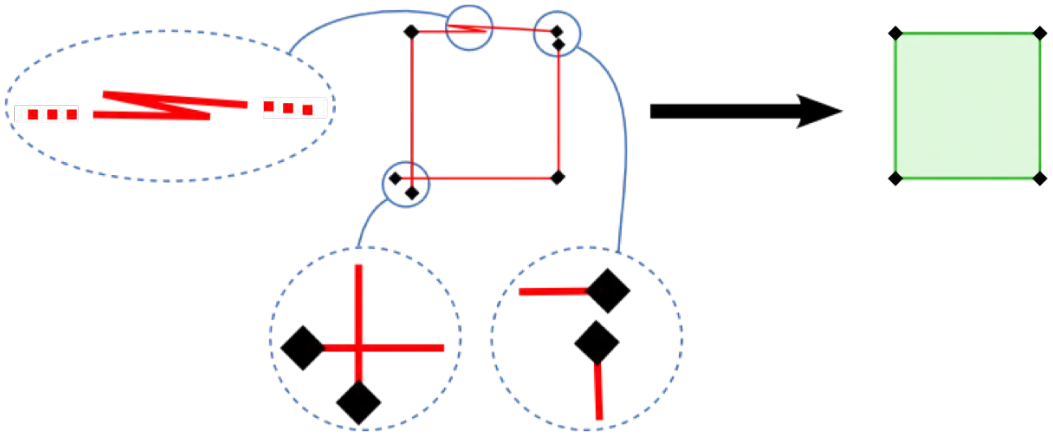

Figure 6: Schematic unstructured polygon example. Non structured connectivity, vertex mismatches and redundant segments (in red) should be corrected to a well structured closed polygon (in green).

- **Redundant vertices**, because of duplicated vertices or many vertices that should be replaced for a single one. This happens, for instance, when a straight segment is generated as multiple smaller segments.
- **Redundant segments**, where there are some sort of overlaps between the segments of a polyline or between different polylines.
- **Joint errors**, where the ends of two polylines do not met at a common vertex.
- **Mixed layers**, where polylines that should be on a layer (e.g., block) are in another one (e.g., building).
- **Isolated polylines**, where we find a set of disconnected polylines instead of a coherent and structured set of closed polygons.
- **Lack of structure**, which is the most usual situation, as the provided polylines are not structured in any way. Although polylines can be connected, they may have no coherent structure.

Our goal is to go from an unstructured CAD file input to the desired output: a set of ground polygons representing hierarchical, coherent and well structured urban

data. In this work we will follow a process with the following stages, with its pipeline shown in Figure 7.

1. **Input parser:**
   This stage parses the input CAD file and returns a structure that we call *CityRawData*, which is a set of CAD polylines representing city building data. The user is expected to provide the names of the interesting layers. As a result, the algorithm gets as input a large data structure that acts as a container of the CAD entities used. (Section 3.4.1)

2. **Clean data:**
   This stage takes the *CityRawData* as input and returns the same set of polylines as output, but without erroneous or redundant information. In order to correct the source data, the algorithm looks over all vertices and polyline segments from the raw data. The vertex cleaning process merges vertices that are too similar to each other according to a user selected *Collapsing Threshold*. The polyline cleaning process removes any duplicated segments. (Section 3.4.2 and Section 3.4.3)

3. **Generate blocks:**
   Because our final goal is to get a consistent simple closed polygon for each building element and each block, this stage takes the cleaned CAD entities from the block layer and returns a set of closed polygons representing blocks. In this stage appears the structuring of the city information. It is important to observe that the set of adjacent building element polygons surrounded by an outline block polygon can be seen as a *Planar Straight-Line Graph* (PSLG). We can divide the process in two sub-tasks:

   a) **Create the *Pre-block* graph:**
      The algorithm groups nearby polylines from the previous stage. First, it processes the polylines with a *plane sweep* algorithm where it considers not only the segment intersections, but also the point-to-point and point-to-segment coincidences, using a user-defined *Block Threshold*. Then, it stores the grouped polylines in a graph structure (*pre-block*), where each node is an intersection or a polyline endpoint. Note that this stage is actually performing a *clustering* over the input data into the final blocks. (Section 3.5.1.1)

   b) **Identify block internal structures:**
      This task finds the graph minimal circuits to get all the subdivisions the current *pre-block* has, and the final block outline that his *pre-block* represents. (Section 3.5.1.2)

4. **Generate buildings:**
   This stage processes the lot and building layers together. The output of this stage is a set of *pre-block* graphs, one for each city block, where each minimal

circuit of the graph –except the block outline– represents a building element footprint. (Section 3.5.2)

5. **Assign heights:**
   This stage takes as input a set of *pre-blocks* and returns, for each *pre-block* building element, its height, which could be retrieved from the number-of-floors text data layer. With these heights, each building element can be extruded, resulting in a 3D model of the city. (Section 3.5.3)

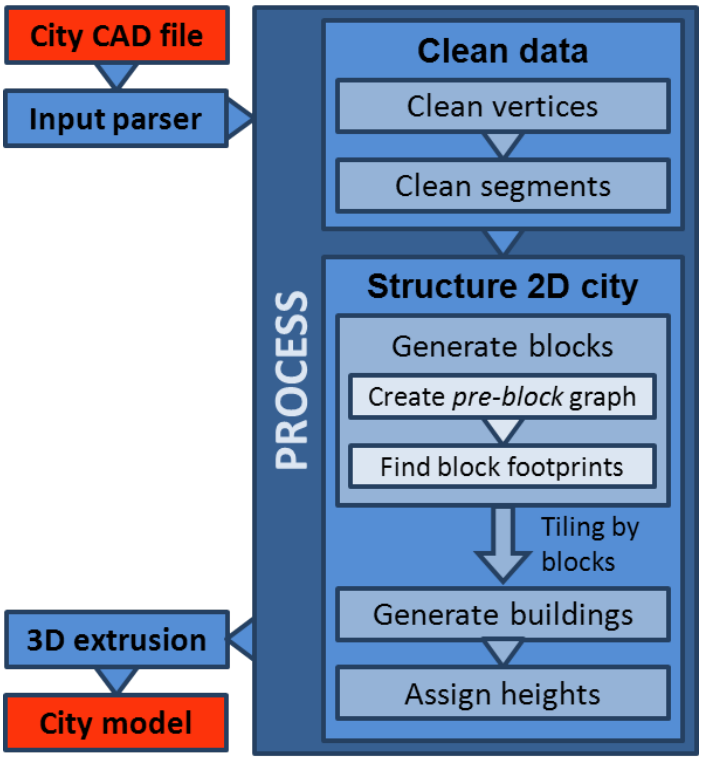

Figure 7: Structuring application pipeline.

3.4 DATA CLEANING

In previous introductory sections we have put special emphasis on the problems we can find in the input data because it is plagued with error. Those errors arise, for instance, when council workers introduce information –any city element– using CAD tools as drawing tools and not as technical tools. In these occasions, they usually only take care of the visual result and do not pay much attention to the structure of each element (block, building, etc.). Therefore, they do not always

choose the most pertinent or correct way of drawing each element. This makes this data very difficult to use directly as input to reconstruct urban models. Figure 5 clearly shows this situation with a section of a Girona city map with all information layers activated (top left) and with just the *Block* layer activated (top right). The urban elements can be easily identified by the human eye. But if we color each entity in the *Block* layer with a different color, we observe that there is not any structural coherence, see Figure 5 *(c)*. To see more examples where this is happening, see Figures 19 *(b)* and 20 *(a)*. So, before structuring the urban data, it must be cleaned to eliminate redundancies, errors, ambiguities and inaccuracies from our input.

3.4.1 *Input data*

Our input source is a CAD file, specifically in an open format Drawing Exchange Format (DXF). These files usually have lot of layers –often some tens– each one for a specific type of information. Details about this format can be found in Autodesk® *DXF Reference* manual for AutoCAD® 2012 [6]. To run our structuring process, the user is required to choose the layers to get block, building and height information from. The algorithm records each entity of the selected layer as a polyline in a rough draft data structure (*CityRawData*) with all its attributes. In Figure 8 we have an example of a block in the Girona city with layers colored differently.

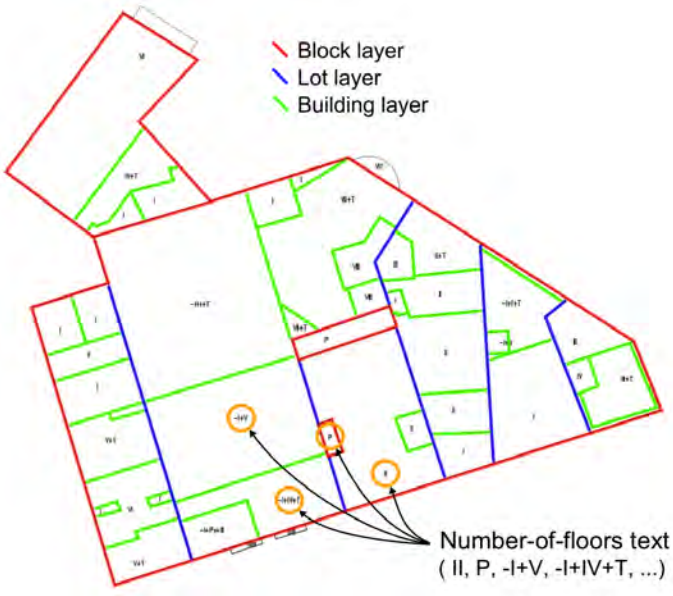

Figure 8: CAD block cadastral data (our input source). Different layers are highlighted with different colors.

3.4.2 *Vertex cleaning process*

The first thing we have to do is to remove the redundant vertices while preserving the shape. In this cleaning process, the algorithm goes layer by layer and polyline by polyline, simplifying them. It can be considered as relevant the work done by Bischoff et al. [10]. The interested reader may refer to the excellent book by Bostch et al. [14] too. In our case, we have implemented our own method that goes through each polyline vertex, considering one of three possible situations:

1. **The analyzed vertex already exists in the stored city structure vertex list**. In this case it is discarded.
2. **The analyzed vertex is very similar to a previously stored one** (i.e., closer than the user's defined *Collapsing Threshold*). In this case the algorithm discards the analyzed vertex in favor of the stored one, but updates its position by averaging both coordinate values.
3. **The analyzed vertex is different from any other city vertex already stored**. In this case the algorithm adds it to the city structure.

As already mentioned, if the second situation occurs, the algorithm tries to collapse the two vertices. But not all vertex collapsing algorithms are appropriate.

A first approach that can come to mind is the a snapping grid method. We can see how it works in Figure 9 *(a)*. The rationale behind the vertex cleaning process is to try to connect vertices that would probably be the same vertex but are not because of human or numerical errors. For this purpose, the snapping method is not feasible, as this can introduce some new errors that will not help reach our goal. Figure 9 *(b)* shows how two different polyline ending vertices candidates to be merged become into different and more separated ones vertices. Another option would be a naive vertex snapping. This decision would also bring new problems as important information could be lost, as Figure 10 (left column) demonstrates.

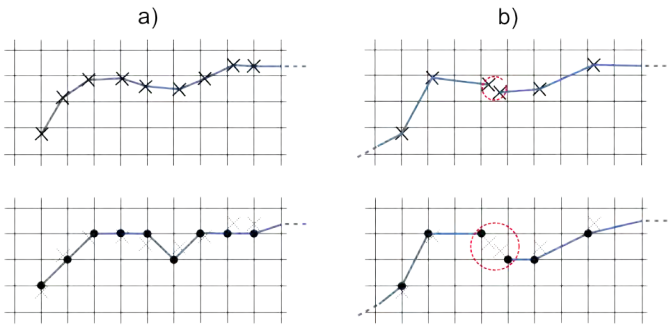

Figure 9: Naive snapping vertices to a grid. *(a)* Simple polyline snapping. *(b)* Snapping two polylines could result in disconnected polylines even though what we want is to join them.

We have implemented our own collapsing method, illustrated in Figure 10 (right column) and explain it in the following lines. This algorithm tries to solve problems of the naive collapsing algorithm. When two vertices are collapsed, the new vertex also stores coordinate values of the original vertices in a list. This way, when a new vertex tries to collapse with a vertex representing previously collapsed ones, the algorithm calculates for all of them their distance to the new vertex. If all collapsed vertices are closer than the user's defined *Collapsing Threshold*, the new vertex is collapsed, updating the representing vertex coordinates and adding the new vertex coordinates to the list. Figure 10 (ii.d) illustrates this situation. Otherwise, the collapsing operation of the representing vertex is reverted and the new vertex and the farther previously collapsed vertex are added to the vertex list, as shown in Figure 10 (ii.b). Applying this method, the algorithm eliminates similar vertices and guarantees that vertices are separated by a minimum distance in between without affecting polylines continuity.

Discussion:
Any simplification method that preserves the curve shape would work correctly in our context, like the polyline simplification method of Douglas and Peucker [21]. However, no method can guarantee no modification of the geometry: by definition, any simplification method will introduce some distortion. Moreover, the larger the user defines the *Collapsing Threshold*, the larger the distortion will be. On the other hand, if the user defines a very small threshold, it will better preserve the shape, but it will store more vertices and more disconnection situations will occur. Obviously, depending on the *Collapsing Threshold* value, results will change. Thus, this method cannot be completely automatic, the user must define the proper *Collapsing Threshold* value and even more, take a look to possible remaining errors after the process. See Figure 21. Note that we are working with a 2D algorithm, mainly because we are looking for polygons defining the projection of blocks and buildings. However, as each vertex has three coordinates, if we do not change its height, the ground may not necessarily be flat. When collapsing different vertices, we calculate the main value for each of the three coordinates.

3.4.3 *Segment cleaning process*

In the same way as with vertices, we want to avoid repeated segments. We designed two methods to remove this redundant information. The first one removes all redundant information a polyline itself may have, while the other removes repeated information that two given polylines may share.

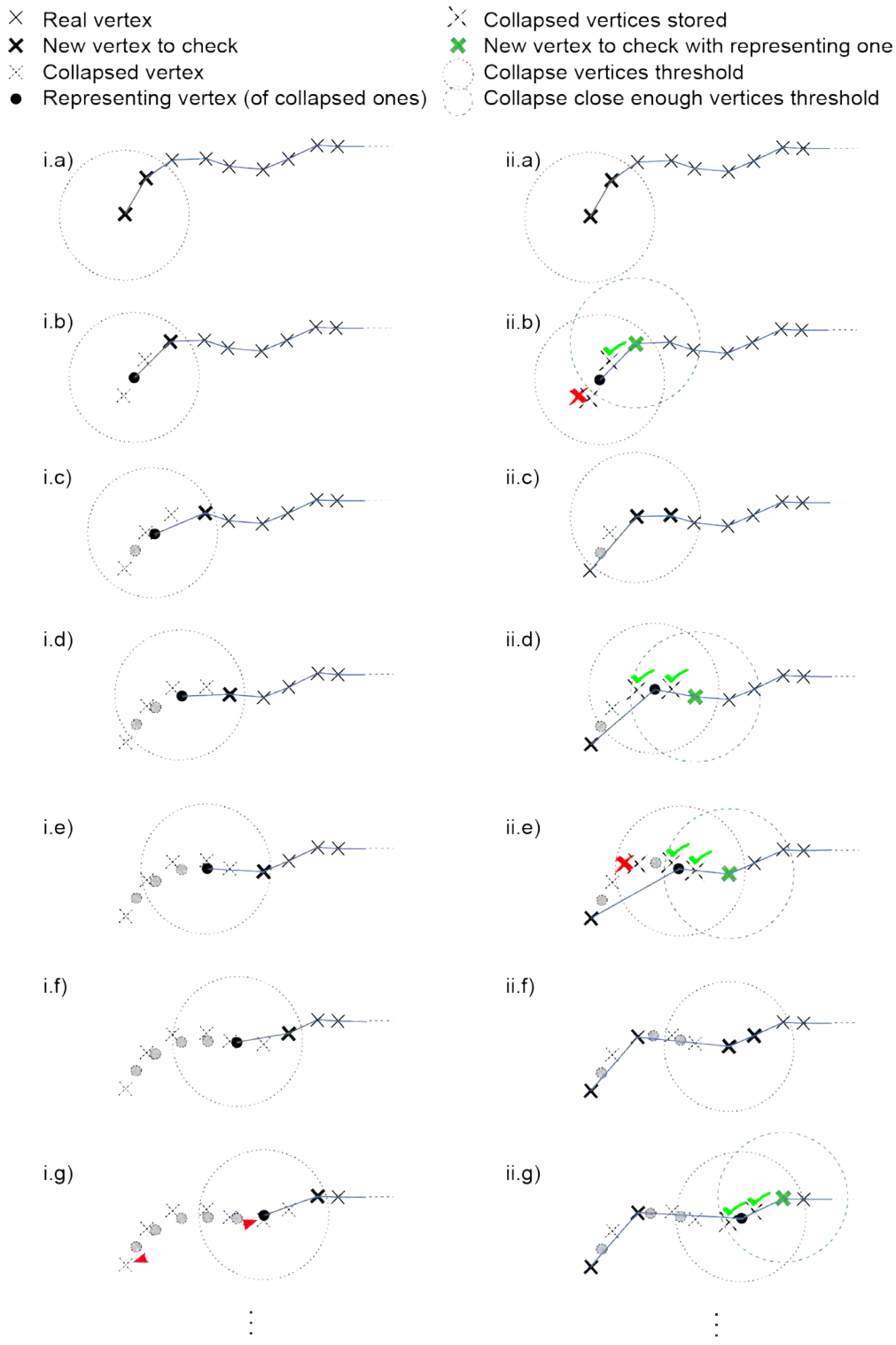

Figure 10: Collapsing method. The left column shows a naive sequential collapsing process, where the new processed vertex (x) is always collapsed with last representing vertex if they are close enough (disk) along the polyline. The right column shows our method. Green ticks and red crosses show if the collapsed vertices are close enough to the new vertex.

3.4.3.1 *Cleaning self-redundant information*

The algorithm of this stage should detect when one polyline segment (or a sequential set of such segments) is represented more than once, eliminating all repetitions. For each pair of polyline segments, it checks whether they are redundant or not, either as a part of the segment or as the entire segment. Two segments are considered as redundant when they have the same slope (or a very similar one) and have a distance between them not larger than the *Collapsing Threshold* defined by the user. When eliminating self-redundant segments, the incoming polyline can result in a set of outgoing polylines, all them pieces of the original one. Figure 11 shows an example of this situation.

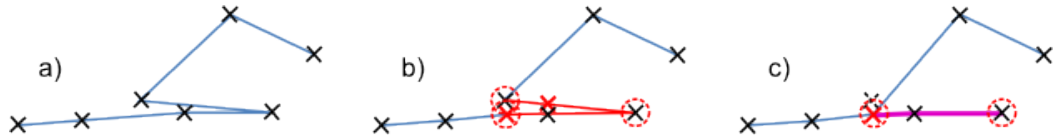

Figure 11: *(a)* Polyline redundant segment example. *(b)* A situation of redundant segment is indicated in red. *(c)* The new generated polyline, indicated in violet, while the original polyline edited is still in blue.

3.4.3.2 *Cleaning redundant information between different polylines*

In this stage, the algorithm moves all polylines from *CityRawData* to the definitive city structure while removing repeated segments overlapping between different polylines. After this process, a set of cleaned polylines without redundancies are stored in the city structure. For each polyline, the algorithm looks for redundant segments between the polyline and the ones already stored in the city structure. Again, like in the previous stage, for each input polyline we may have a list of polylines as output. When a candidate *CityRawData* segment is checked against an already stored segment, there are three possible situations that can be found. Figure 12 illustrates them and here we enumerate them:

1. **The candidate segment is completely redundant:**
   When there is a complete superposition of the analyzed segment of the candidate polyline with respect to already stored polyline segments, no information needs to be added.

2. **The complete stored segment is redundant:**
   When there is a complete superposition the other way around, the non-coincident parts of the candidate segment should be added to the city structure.

3. **The candidate and the stored segments have a coincident fragment:**
   When candidate and already stored segments are partially redundant, the candidate polyline segment should be divided in redundant and non-redundant

fragments. Then the algorithm discards the redundant fragments and adds the other parts to the city structure.

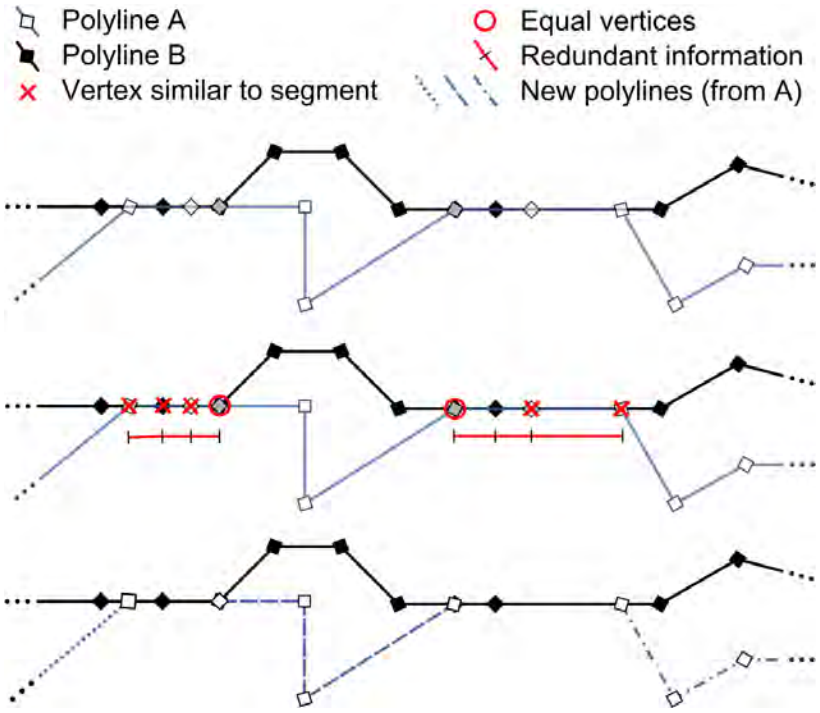

Figure 12: Polylines superposition: Eliminating redundant information (superposed segments).

Once the algorithm has cleaned the vertices and polylines, we have all the needed information to start the city structuring stage.

Discussion:
Again, like in the vertex-cleaning process, the user's defined *Collapsing Threshold* value could affect the results. Also note that, at this point, no polygons are found yet: we just have a set of polylines. Talking about computational cost, despite the fact that the order of this stage is $O(n^2)$, where n is the number of polyline segments, it could be easily improved to $O(\log n)$ using a similarity search solution (e.g., with a space partitioning tree like a quad-tree), once the tree is already created. Finally, note that different polylines could share a T-vertex (three incident segments), or there even can be vertices with an n-degree. But this will not represent a problem to detect block and building footprints, as we will see in Section 3.5.1.2.

3.5 STRUCTURING

The segment cleaning process results in a very fragmented polyline set, but without repeated vertices or segments. Now, we want to give urban structure to that data. The structuring algorithm generates blocks as closed polygons and then buildings, also as polygons. All the buildings are associated to it respective blocks, so each block has it own set of buildings.

3.5.1 *Block generation*

In this first stage the block-layer polylines are processed. After, all blocks are composed of closed polygons representing footprint silhouettes, which each defines the area of the city block projection (see Figure 13). It is important to mention that, in order to process the block layer, the algorithm has to take care of the *Sub-blocks* (see Section 3.2 and Figure 13 (c.i) and (c.ii)). The algorithm performs the steps we describe in the following subsections to get the city blocks.

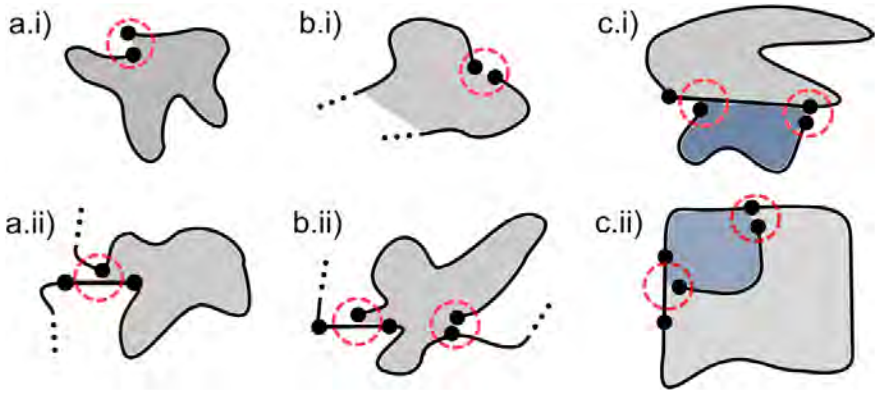

Figure 13: Schematic representation of possible block and sub-block situations. *(a)* Cases with just one polyline. *(b)* Cases with more than one polyline. *(c)* Cases generating *sub-blocks*.

3.5.1.1 Pre-block *graph creation*

In this first stage, we want to group polylines into smaller sets, considering if they are close enough to each other to belong to the same block. This way, we are creating a clustering of all the input data into the candidate future blocks. Each set of polyline candidates to become a block, is grouped in a graph structure called a *pre-block*. Each *pre-block* is a graph $G =< N, P >$ with a list of nodes N and a list of paths P, as can be seen in the right hand of Figure 14.

Each node represents either an intersection between polyline segments, or the endpoint of a polyline. A node stores its respective vertex coordinates, and a list of all paths reaching to the respective "neighbor" nodes. Each path $P = (Pl, < N_i, N_j >)$

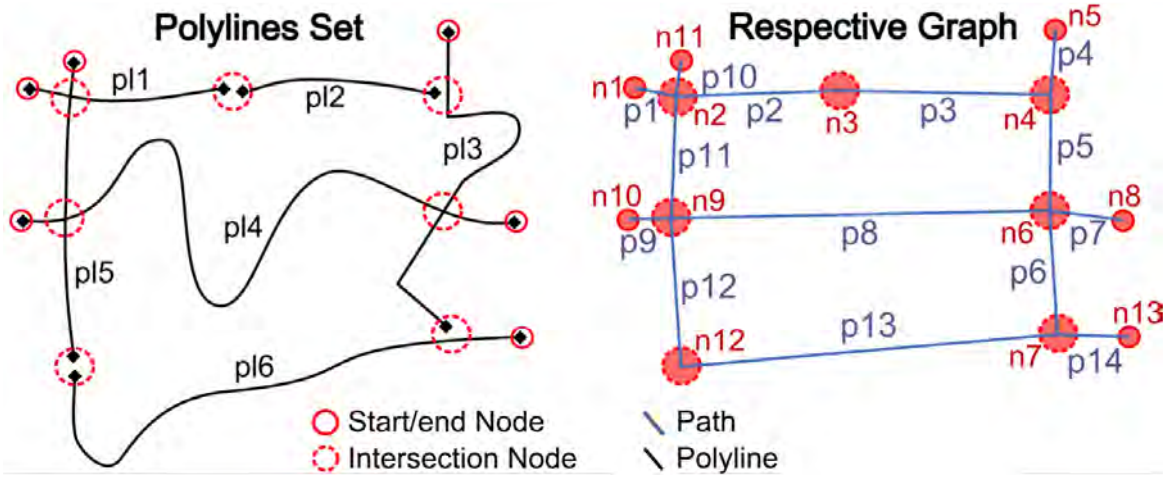

Figure 14: A diagram of a *pre-block*'s graph from a set of polylines. *Left:* The set of polylines with its ending and intersection nodes marked in red. *Right:* The corresponding graph with polylines transformed in graph edges (paths).

represents the polyline fragment *Pl* that links node N_i to node N_j as a list of the original polyline vertices linking N_i and N_j. Note that this graph is a Planar Straight-Line Graph (PSLG) as we avoid intersections between edges.

Before starting with the *pre-block* generation, we link polylines to form continuous and longer polylines. We concatenate any pair of polylines with coincident starting or ending vertices. By doing this, we considerably reduce the number of input polylines before starting this *pre-block* graph creation step, which is the most computationally expensive one of our process pipeline.

Pre-block generation starts by going through the input polyline list, trying to add each polyline to the currently processing *pre-block*. When the polyline has any self-intersection, it is broken down into its respective non-intersecting component polylines, and processed as a set of independent entities. A polyline is added if it is close enough to any path of the current *pre-block* structure. It is possible to know this by searching for intersecting segments or vertices closer to a path segment than the user-defined *Block Threshold*. There may be (one or more of) the following four conditions, where "similar" means closer than the *Block Threshold* (see Figure 15):

1. A vertex is similar to a *pre-block* path vertex.
2. A vertex is similar to its projection on a *pre-block* path segment.
3. A *pre-block* path vertex is similar to its projection on a polyline segment.
4. A segment and a *pre-block* path segment intersect.

Each similar vertex and polyline endpoints are stored as new *pre-block* nodes. Finally the algorithm eliminates the polyline from this stage input polyline list. Note

that there could appear nodes with a degree larger than four but, as we will see in next section, this will not cause any problem to our method.

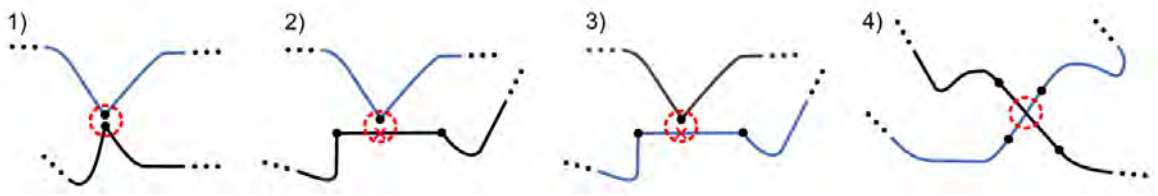

Figure 15: Nearby vertex conditions: adding a candidate polyline to a *pre-block* graph. Candidate polylines are drawn in blue and *pre-block* paths in black.

After adding a new polyline to the *pre-block*, the algorithm continues by checking all remaining polylines. A *pre-block* generation is finished when no more polylines can be added. In this case, the current *pre-block* is considered to be finished and a new *pre-block* is created with the first not-assigned polyline of the input set. The process is repeated until there are no polylines left.

Finally, we must do a "cleaning" process over the graph paths, cutting out all *fringe* paths. A *fringe* path is a path with only one ending node connected to the rest of the graph. In Figure 14, paths p1, p4, p7, p9, p10 and p14 would be eliminated.

Implementation details:
The pre-block graph creation is similar to the known problem of segment intersection in computational geometry, with the only difference being the notion of proximity. The algorithm also needs to detect situations of vertices and segments closer than the given *Block Threshold*. It is convenient to define an "inflated point" as a point plus a user-defined radius, which is our already mentioned threshold. Hence, we could consider this stage as one that computes intersection detections, where an intersection could be generated by an overlap between two inflated points or an inflated point and a segment. The best implementation of this algorithm is a *plane sweep algorithm* using a *sweep line*, with order $O(n \log n)$, where n is the number of points. This algorithm is implemented in the CGAL library [13].

Discussion:
In this stage, results will depend on the user-defined *Block Threshold* value. It is important to observe that, in certain situations, it can be difficult to find a good value for this threshold, and a wrong choice can result in situations where no point similarities are found or the other way around. Situations of false similarity detection can happen when setting a value too large for this threshold. For this block generation stage, we set the *Block Threshold* to be slightly smaller than the smallest street width. We have found that this value makes the algorithm perform remarkably well.

3.5.1.2 *Block footprint and internal structure*

A block is a closed area surrounded by streets. At a high level, we can say that streets define the block outline, but in practice the actual outline comes from the architectural elements that the block contains. Block generation requires that we define the footprint delimiting its area (i.e., its outer boundary), together with a (possibly empty) list of courtyards, and then identify the building elements with their respective heights. The courtyard and building element areas should be a set of sub-areas that partition the overall block area. The algorithm of this step uses the *pre-block* graph previously created to define the polygons representing the block and the interior *sub-block* footprints. It is in this process that the *pre-block* graph is transformed into a block structure with a footprint outline and interior elements. The process computes the graph's closed minimal circuits and stores them in a list. Therefore, a *pre-block* stores a list of circuits, where each circuit is a list of paths defining a closed polygon.

A closed circuit is detected by starting from any node in the graph. For that node, the algorithm chooses the outgoing path segment which has the lowest positive angle with respect to the positive x-axis. Then it continues through the chosen path to the next node. When arrived to the next node, it chooses the outgoing path segment with the minimal angle (counterclockwise) with respect to the incoming one. When a neighbor node has already been visited, it is marked on the node's neighbor list, also marking the path that connects both nodes from its path list. This step is repeated, moving from one node to the next one, until we readh the initial node. At that moment, the circuit is closed and stored in a list of circuits for the current block. After finding a circuit, a new search is started from any node with a remaining unmarked neighbor. Note that all graph edges will be used in both directions for it two adjacent circuits. When all nodes have no more unmarked neighbor nodes, all current *pre-block* circuits have been found. Figure 16 *left* illustrates this process. Note that the degrees of the nodes are irrelevant in this algorithm, as all circuits will be found. Also note that each *pre-block* circuit is a closed polygon with no self-intersections. In the case of a simple block, our method stores the block outline twice, both clockwise and counterclockwise. But when a block has *sub-blocks*, the method gets a closed polygon whose layout is the block outline, plus a set of interior polygons (*sub-blocks*) representing a partition of the block.

Our method lists its outline as the first entry in the *pre-block* circuits. We enforce this situation by starting the first circuit search always from the path having the vertex with the lowest y-coordinate value. Figure 16 *right* shows how it finds the exterior red circuit. In this first circuit search, the ending test to know that the circuit is closed, is not only to arrive to the initial node. We have to also check that the best possible continuing path is the first starting path of the circuit. If not done this way, the full outline may not be well identified in cases of "eight shaped"

circuits. See in Figure 16 *right*: We start from node six and going through path 0, when reaching to node six again, we have to continue with path II and not stop there. The full outline silhouette is found when reaching node six from path VI and the next path to continue with would be again the starting path 0. Then, circuit detection continues as described finding the rest of interior *sub-block* circuits.

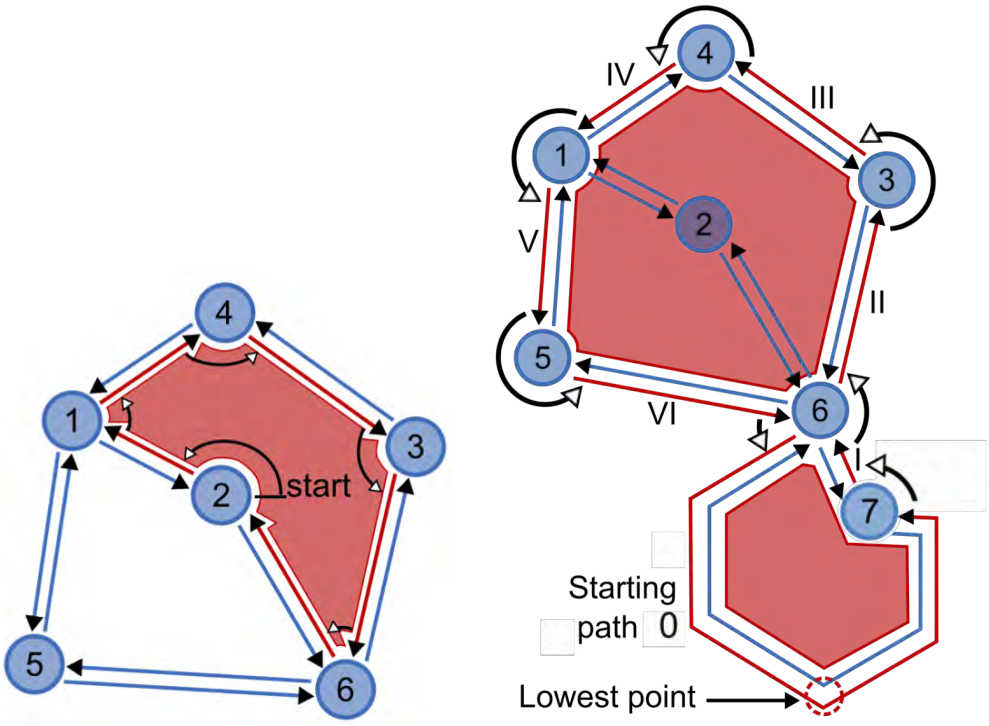

Figure 16: Examples of *pre-block* graph circuit search. *(Left)* An interior closed polygon starting form node two. *(Right)* An outline circuit starting from the lowest point (path 0) and using the node six more than once.

Once all the *pre-block* circuits are found and stored internally, the method searches for possible block courtyards. As courtyards are independent closed polygons, the previous steps stored them as city *pre-blocks* themselves, as Figure 17 *left* shows. The main difference between blocks and courtyards is that a courtyard is a *pre-block* completely enclosed by another *sub-block* (or *pre-block* circuit). Hence, it becomes a *sub-block* of the enclosing outline without connection with other *sub-block* paths. To find all the courtyards in the city, all block structures are analyzed, checking if they are inside another larger block circuit by executing a simple *point-in-polygon* test. As blocks do not have intersections between them, if any block point is contained inside any other one, then we can be sure that it is a courtyard, being completely enclosed in the second one. In that case, we eliminate the courtyard *pre-block* from the city block list and add it to the enclosing block courtyard list. In Figure 17 we can see how green and blue polygons were independent *pre-blocks* before this step and how they are colored in orange like the rest of *sub-blocks* when they have been added to the final block where they belong.

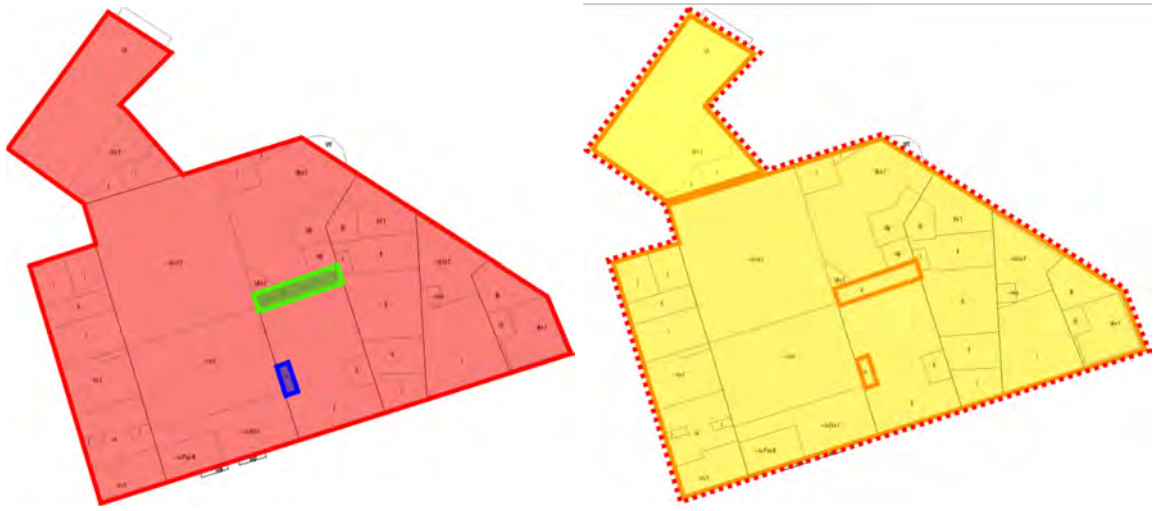

Figure 17: A city block after processing the block layer: *pre-block* outline, *sub-blocks* and courtyards. *Left*: Three *pre-block* outline have been detected, in different colored outlines. *Right*: After courtyard detection, just one *pre-block* (dotted red line) remains with two *sub-blocks* and courtyards inside one of them drawn in different orange each *sub-block*.

With all courtyards associated to their respective blocks, we consider that we have well structured city blocks. As a graphical example, compare Figure 19 *(b)* and *(c)*. Note all CAD entity polylines used to draw a block in Figure 19 *(b)*. After the structuring process of the blocks layer, Figure 19 *(c)* shows in a unique color the block and it *sub-blocks* and courtyards.

Summarizing, each block is stored as a *pre-block* graph and has an outline footprint, a (possibly empty) set of *sub-blocks* and a (again possibly empty) set of courtyards. This way, the simplest block possible only has its outline and one *sub-block*, which is the same as the outline.

We point out that blocks in a city can be seen as a kind of spatial tiling, i.e., a partition of the city data. Thus, after generating the block footprints, any polyline belonging to any layer can be classified and processed after locating it in a block, i.e., the block where it belongs.

Discussion:
Information lost and erroneous recognition of blocks may affect the *pre-block* graph creation. Once a *pre-block* graph is created, if it does not contain any circuit (e.g., it is not well connected), the algorithm will consider it as a large *fringe* path and eliminate it. Any *pre-block* considered in this stage will have at least its own block outline, and all the circuits found will be closed non-intersecting polygons.

3.5.2 *Building generation*

When the processing of the block layer is finished, the algorithm can start working on the other layers. These next layers are called *building information* layers. As we have mentioned, it is not convenient to define lots at this stage because very often this layer and the building layer are incomplete and need each other to properly define the building boundaries as closed polygons. If they are processed separately, it can wrongly remove fringe paths needed in the next layers to define buildings. Actually, what is defined in these layers is a set of closed polygons with associated heights. Each of these polygons is called a building element.

The entities in these layers will always be processed after localization calculation is done, where each entity is associated to an existing *pre-block*, the block it belongs to. For the *building information* layers, the algorithm processes a set of polylines in the same way as described in Section 3.5.1. Therefore, knowing to which block the building information is associated, the building element polylines are processed as follows:

1. Add building element polylines to the *pre-block*.
   This step increases significantly the graph complexity.

2. Find the *pre-block* minimal closed circuits.

As with the block's layer process, the result is a set of graph circuits, where each one is a closed polygon. In this case, a circuit defines a specific building element, instead of a *sub-block*. For this updated graph, the first circuit is the block outline footprint, while the rest of the circuits are building footprints. See Figure 18. Consequently, the result is a coherent structured city data consisting of a set of blocks and a set of building elements for each block. In the results section, Figure 19 *(c)* shows a block of the Girona city structured after processing only the blocks layer. On the other hand, Figure 19 *(d)* shows the same block after all building layers have been processed.

Discussion:
Note that what we called a building is just a portion of the complete block footprint area with a polygonal boundary. The way input CAD maps are created, each of these polygonal boundaries are areas with a specific height. Thus, we call building elements all these height polygons. A real city building could be represented with different *pre-block* circuits if a single building contains different heights. Moreover, the *pre-block* circuits are building projections, so there could be overhangs with no walls on ground floors, that will be represented in the same way as a building element with a normal ground floor.

Figure 18: A city block after processing lot and building layers: a *pre-block* outline, and circuits for *sub-block*, courtyard and building circuits. *Left*: After the blocks layer is processed. Outline in red dotted line. *Sub-blocks* and courtyards are outlined in orange. *Right*: After processing lot and building layers. Each building element detected is in a different color. Note that courtyards (with zero height) are in black.

3.5.3 *Height assignment*

From the data output from this last stage, we get a city structure with blocks as closed polygons internally well structured with a set of building elements and courtyards. building elements are 2D geometric entities. By extruding each building element footprint polygon by a given height, we get prisms. That way, we get a 3D structure like the LoD1 of the CityGML [39] standard.

Processing an extra layer, called *number-of-floors text* (see Figure 8), a height can be associated to each building element. This layer has positioned labels with information about the number of floors in each building element. From a simple *point-in-polygon* test, we know to which building element any label belongs to, and use this information as an educated guess of the actual building element height. Thus, all 2D building elements become 2.5D elements considering that each footprint polygon has an associated height information.

3.5.3.1 *3D model extrusion*

At this point, our 2D structured data can generate a 3D model of the city. Let us assume that, like in most traditional buildings, a building 3D model is a prism, based on its footprint stored as an arbitrary closed polygon without self-intersections. For each city building element, we want to extrude its footprint polygon vertices to this height value. Note that before having well-defined closed polygons this was impossible to be realized. To get the height value we multiply its associated

number of floors by a standard floor height (three meters for the ground floor, 2.5 meters for superior floors), and add a small random offset. This small offset is added because in the real world, there are no fixed heights for all buildings, so we add a certain degree of randomness. The triangulation of the model –ground polygon, roof and facade– is always possible because each building facade is a rectangle and the building roof and footprint are closed polygons. We store this 3D geometric data in an open format like *Wavefront*'s *OBJ*. Figure 19 *(e)* shows a Girona city 3D block model example. The *number-of-floors text* layer labels indicates the number of floors for each building element, but also give information about underground floors, overhangs, yards, roofs, etc. So, the previously explained overhang situation (see the discussion in Section 3.5.2) could be resolved when a 3D model is reconstructed, considering all the information this layer provides.

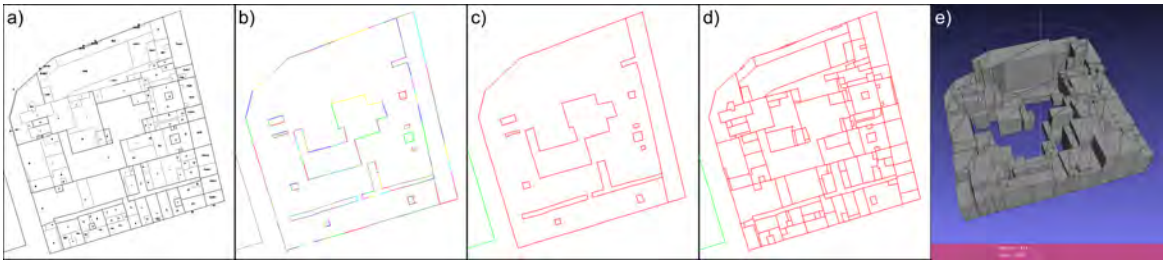

Figure 19: A Girona block (the "*1-block*" case) process visualization. *(a)* Block and building layers with a CAD tool. *(b)* Block layer cleaned but unstructured entity polylines. *(c)* Block layer structured, with courtyards and *sub-blocks*. *(d)* Block and building layers structured. *(e)* The respective 3D model triangulation shown with an *OBJ* viewer.

Discussion:
The final result of our work is equivalent to a LoD 1 of cityGML [39], consisting of block models comprising prismatic buildings with flat roofs. More detailed representations are not possible to be built from our source of data (e.g., roofs [29]). Of course, we can always use procedural techniques (e.g., *Esri*'s *City Engine* [26]) to add details to each building. Also, building heights are not exactly real ones. Using only the height information we have (number of floors), we have only done an approximation. As an obvious example, a cathedral's floor height is much higher than a residential building floor. Thus, in this stage we are not reflecting exact real values. Also, in the case of overhangs, our reconstruction is not accurate: we are creating walls from the ground to the roof for all buildings, while we would need to create walls from the first floor to the roof in these cases, or in case of roof-style terraces, with open walls, but a roof on top of it only for that floor. Although we consider this is beyond the scope of our work, it could be solved by a more accurate process of 3D model reconstruction considering all the information that

the *number-of-floors text* layer provides, unlike non flat roofs that cannot be solved without more information that the one provided from our process input data.

3.6 RESULTS

The results given below are from DXF CAD files provided by Girona and Barcelona local city councils. Girona's file has more than 50 layers, each one having different features. As we explained in Section 3.4.1, we worked with four layers. Three for block and building footprint geometries and the other for the text of the number of floors. In Table 1, some execution statistics are shown. The *1-block* column, refers to the block shown in Figure 19. The *Some blocks* column is a region of 11 blocks of the available data (Figure 20). The *Girona Jewish neighborhood* is the whole Girona old neighborhood city map (Figure 22) and *Girona full city* is the complete city model. About Barcelona's file, we presented the *Barcelona Eixample* column results. This is a section of the famous Eixample neighborhood of Barcelona, design by Ildefons Cerdà. The Girona Jewish and Barcelona Eixample neighborhoods are good examples of very different kinds of cities. The first one has very irregular blocks and buildings, while Barcelona's Eixample is specially famous for its regular blocks delimited by the grid of parallel and perpendicular streets. Figures 24 and 25 show renderings of the resulting models.

As a reference, in Table 1, we have added the current number of buildings in the corresponding part of the real map. This number has been obtained by a manual process, precisely because of the unstructured nature of the input data. We counted what the human eye can appreciate when looking at the CAD map.

All these results were obtained using the following thresholds, which proved to be quite general for the streets and structures of the whole model, even for very narrow ones:

COLLAPSING THRESHOLD: It decides if two vertices should be collapsed in the data cleaning process (Section 3.4). A good value for this user-provided parameter will depend on the precision used in the map generation, so the exact value is not significant, but we used 1 meter as threshold value.

BLOCK THRESHOLD: It decides whether a polyline should belong to a *pre-block* or not in the *pre-block* graph creation process (Section 3.5.1.1). We suggest to use a value slightly smaller than the smallest street width. In our case, we used a threshold value of about 3 meters.

CASES	Girona 1-block	Girona some blocks	Girona Jewish neig.	BCN Eixample	Girona full city
	Number of input file CAD entities				
Block layer	158	580	4235	4557	36988
Building layer	211	522	6228	60135	54284
Total	392	1116	11507	65295	100346
	Real number of elements				
Block layer	1	7	287	178	≃2500
Building layer	141	468	—	—	—
	Number of processed elements				
Block layer	1	7	287	178	2497
Building layer	131	441	4622	24268	38834
	Statistics				
Collapsed vertices	38%	16%	26%	27%	28%
Entities/Building	5.98	5.06	4.98	5.38	5.17
Buildings/Block	131	63	16.1	136.3	15.6
Block Errors	0%	0%	0%	0%	—
Building Errors	7%	6%	—	—	—

Table 1: Structuring process results.

As we focus on the reconstruction of buildings, the layers we are interested in are: blocks, buildings and textual height information. For the Girona's model, the input consisted of more than 100.000 entities. All of them were transformed into polylines. When all polylines had been processed through the *vertex cleaning* step, about 25% of the resulting vertices were collapsed (see Table 1 *Collapsed vert.* row values for all the cases). In our *polyline cleaning* step, the process removes also hundreds of redundant segments.

After then *structuring* process (Section 3.5), we got 2497 blocks with their respective 38834 courtyards and building elements, with their associated heights. Considering that the 100.000 input entities were distributed, 16988 in block layers and 54284 in building layers, we can conclude that:

- **Each block is defined with an average of 15 input entities.** We get this value by dividing the number of block layer input entities by the number of blocks processed (for Girona full city case: 36988/2497).

- **Each set of block building elements is defined with an average of 40 input entities.** We get this value by dividing the total of input entities by the number of processed blocks (for Girona full city case: 100346/2497).

- **Each build is defined with an average of 5 input entities.** We get this value with the following formula. Considering Be as the number of input entities in block layers, be the number of entities in building layers and bi the number of output buildings:

$$\frac{Be + 2be}{bi}$$

  The building layer entities are considered as doubled, due to the fact that each entity segment is always delimiting two buildings, the ones of each segment side. For Girona full city case: $(36988 + 2 * 54284)/38834$. We consider this last value the most important statistic. All cases results are in the *Entit./Build.* row of Table 1.

The two last rows in Table 1 show an estimation of the number of errors done while searching for blocks and buildings in the city. This error is calculated by comparing the obtained buildings with the real ones: we estimated this value to be less than 6% of the total number of buildings in the city. The same calculation is done for the blocks.

In the cases of the *Girona Jewish neighborhood* and the *Girona full city*, the density of buildings per block is lower than the other cases (see Table 1 *Build./Block* row). This is because there are many detached houses considered as blocks. So this is skewed data. Nonetheless, this is data that depends on the urban density and architectural characteristics, it does not have crucial importance for our work. Comparing the *Barcelona Eixample* neighborhood (Figure 25), it has larger blocks (100x100 meters square) than the *Girona Jewish neighborhood* (Figure 24), which has smaller and more elongated blocks. This could result in the false impression that the Eixample model has a higher complexity, specially with respect to its roofs, but a detailed inspection reveals they have roughly the same complexity.

In Figure 19 we can see how our application structures the urban data for a single block and also the resulting 3D model. Figure 19 *(a)* shows how the useful layers look like using a traditional CAD tool. Our way to view different unconnected objects (either CAD entities or blocks) is to use different colors. Thus, in Figure 19 *(b)* the block appears with only the input block layer entities drawn, each one in a specific color. Instead, within our final 2D structured city (Figures 19 *(c)* and 19 *(d)*), each block is drawn with a unique color, in this case red. In Figure 19 *(c)* we can see that, after the process, we have just one unique block with its *sub-blocks* and

courtyards. In Figure 19 *(d)* we can appreciate all the block buildings processed and structured. Finally, in Figure 19 *(e)* we have the *OBJ* model, rendered using *MeshLab* [19].

In Figure 20 *(a)* we can see each input entity in the block layer in a different color, which we labeled as the *Some blocks* case. In Figure 20 *(b)* we can see each structured block colored differently, with its *sub-blocks*, courtyards and buildings. Finally, in Figure 20 *(c)* we have an image of the generated 3D model.

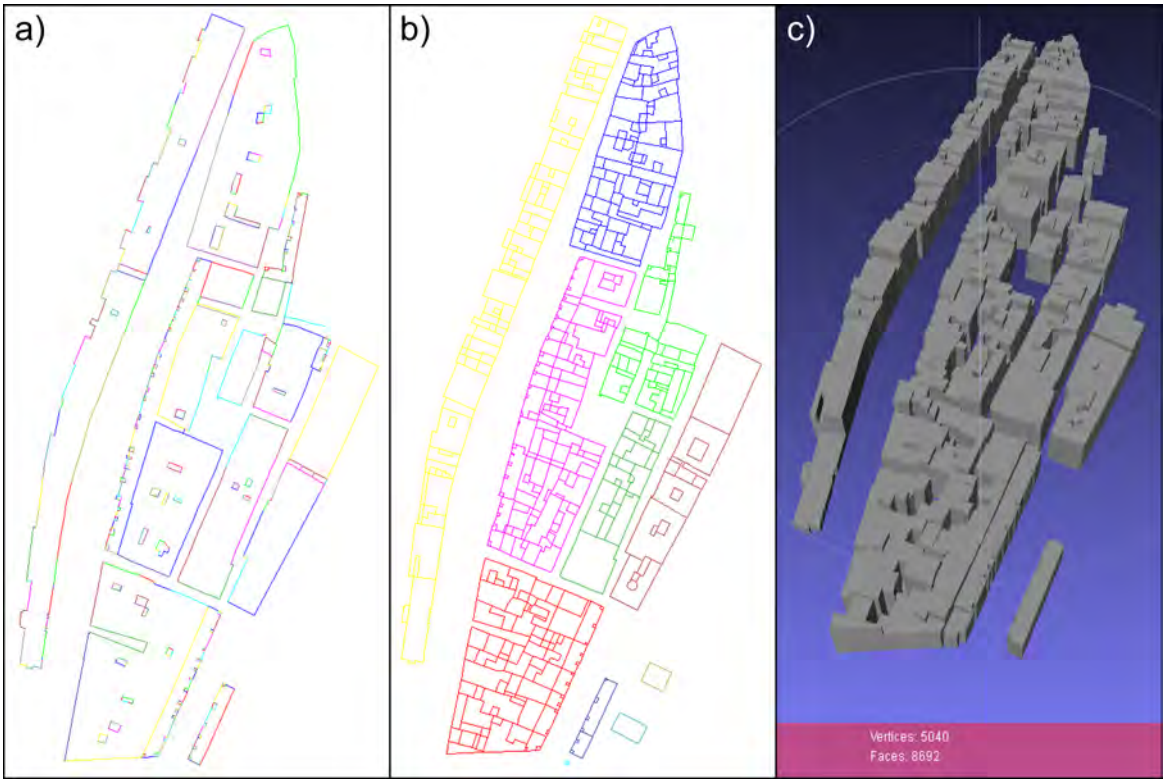

Figure 20: The" *Some blocks*" case: a result of urban data structuring. *(a)* Block layer cleaned but unstructured entity polylines. *(b)* Block and building layers structured. *(c)* The respective 3D model shown in an *OBJ* viewer.

Figures 22 and 24 show that for all the *Girona Jewish neighborhood* case, the same process as the one explained above, with the difference that in Figure 22 *(b)* we show only the block layers because it is easier to compare then with the respective input entities in Figure 22 *(a)*.

To close the comments on the resulting images, Figures 23 and 25 show results of processing the *Barcelona Eixample* case. In Figure 23, we can see how our input –the whole unstructured CAD entities, block and building layer entities– occur in our output –a set of blocks with its respective buildings. The resulting *OBJ* model of this set of the Barcelona Eixample blocks is rendered using *City Engine* [26] in Figure 25. We can observe all the buildings processed there. It is interesting to observe that some blocks seem to be incomplete, but they are not: areas like gar-

dens, squares, and parks are closed polygons without height, like courtyards. If the height is zero, our implementation does not generate any polygon. So, in cases that a garden is located on the limits of the block, which is typical in the octagonal blocks of the Eixample neighborhood, the block may loose its characteristic squared silhouette.

Let us remind in this final section, that a process with improper threshold values would give undesirable outcome. Figure 21 shows the *1-block* case when using too large and too small *Collapsing Threshold* values. In Figure 21 *(a)*, a large threshold value causes too many vertex collapses, and this involves too much loss of geometry. In Figure 21 *(b)*, the small threshold value causes missing geometry. Some polylines are not attached to others, so they are eliminated and some circuits are not found, loosing the corresponding building footprints.

Figure 21: The *1 block* case with undesired results. *(a)* Outcome using a too large threshold. *(b)* Outcome using a too small threshold, there are missing buildings in the blue circle when comparing with *Figure 19* (d).

Talking about timing, it takes about one and a half minutes to process the block's layer of *Girona Jewish neighborhood* case and get the completely restructured city. This time is got by using an Intel(R) Core(TM)2 Quad CPU at 2.33GHz and 3.25GB of RAM (GPU features not used). From this one and a half minutes, twenty-two seconds are for cleaning vertices and polylines and the remaining for the structuring process, one minute and eight seconds. We can appreciate from this that the most expensive process is to group polylines in nearby sets to generate blocks and their posterior connecting process.

The increment of the number of CAD entities on block's layer E_B, means a quadratic increment of the process time, to get all the blocks B structured. This is because we have to check all entities against the others to know which are considered to belong to the same block. On the other hand, the increment of number of building entities E_b affects linearly to the timings. This is because of the previous localiza-

tion process that associate the building entity to an specific block. After getting the number of entities belonging to the same block, then the cost is a constant K_B or K_b considering cities of the same CAD entities per block factor. So, time to get block layer T_B and buildings layer T_b processed is:

$$T_B = {E_B}^2 \cdot K_B$$

$$T_b = B \cdot K_b$$

where,

$$K_B = \left(\frac{E_B}{B}\right)^2 \text{ and } K_b = \left(\frac{E_b}{B}\right)^2$$

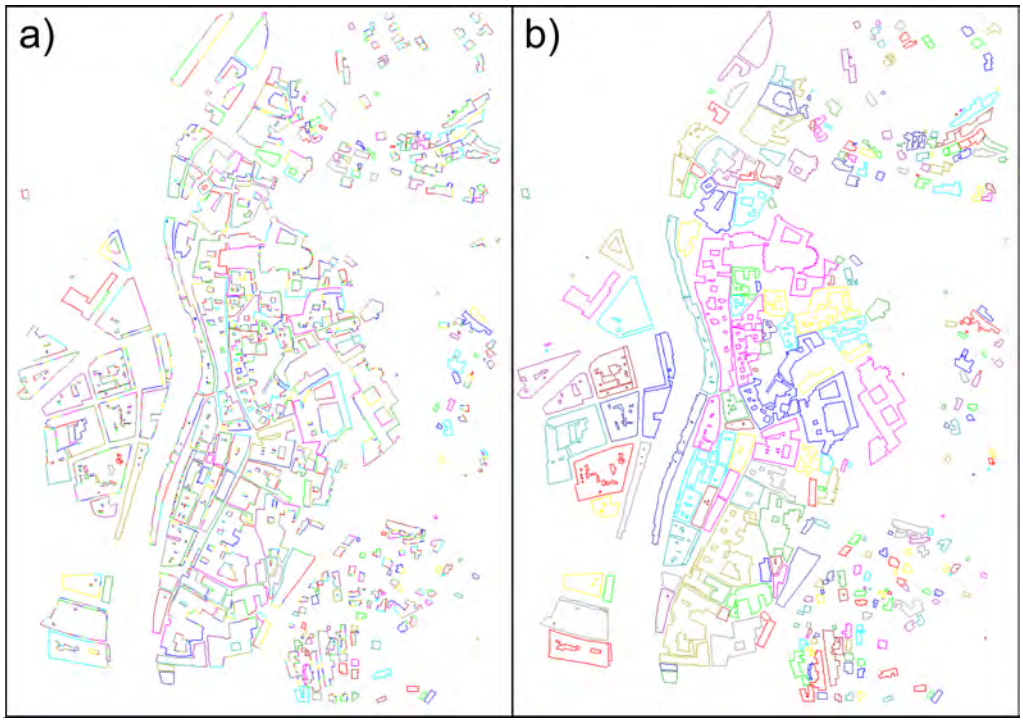

Figure 22: The *Girona Jewish neighborhood* case. *(a)* Block layer cleaned but with unstructured entity polylines. *(b)* Block layer structured, with courtyards and *sub-blocks*. Its 3D model is shown in Figure 24.

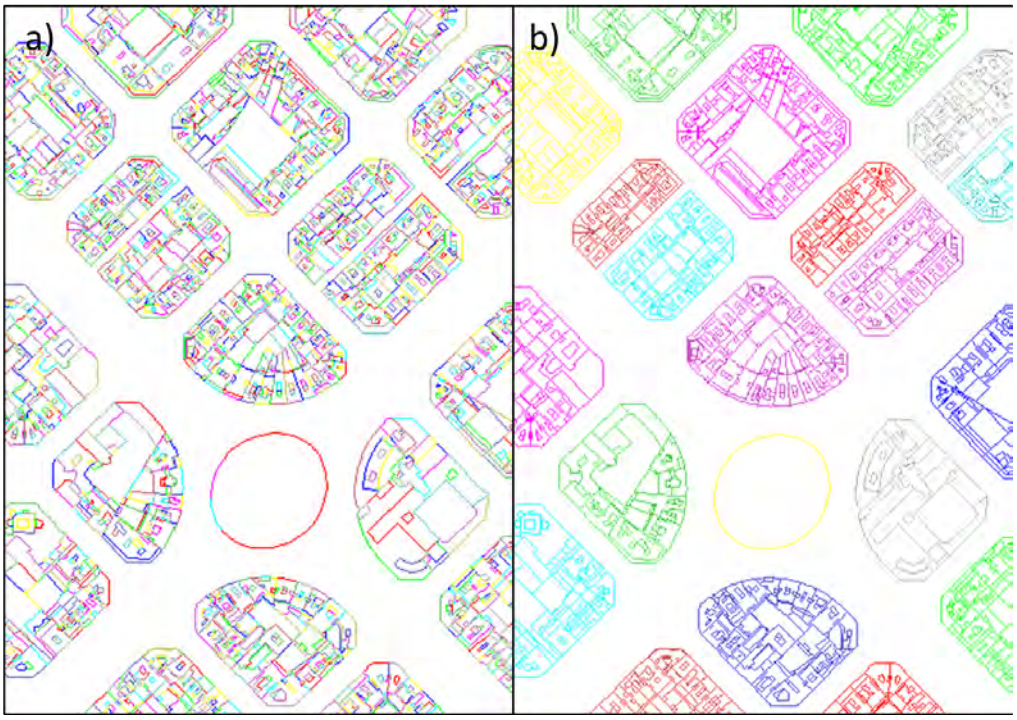

Figure 23: The *Barcelona Eixample* case (one region of it) before and after our process. *(a)* Before the process, unstructured CAD entities in different colors. *(b)* Block and building layers structured after running our process.

Figure 24: *Girona Jewish neighborhood* case in 3D. Observe the reconstructed tallest tower in the center of the image, which represents the Gothic Cathedral.

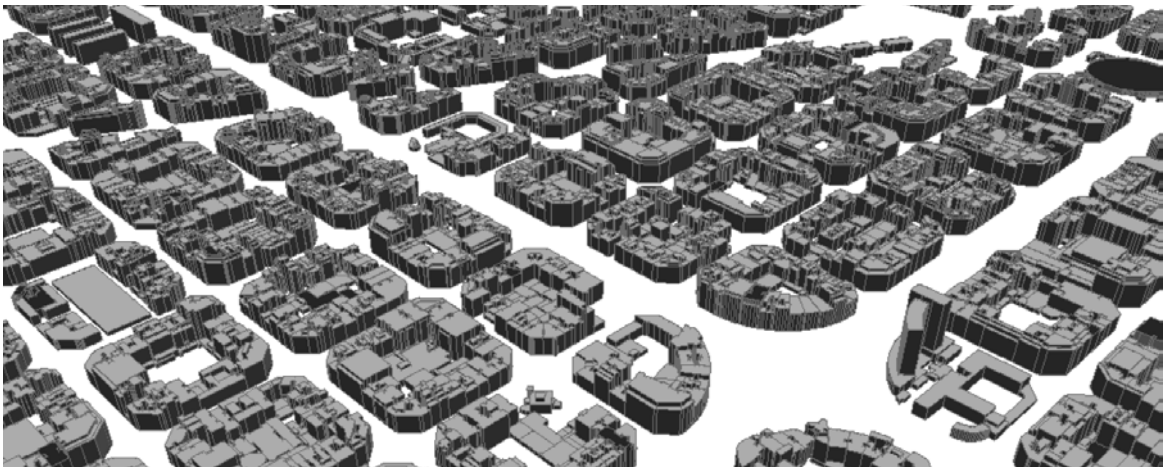

Figure 25: *Barcelona Eixample* neighborhood in 3D. We display only the building models. Note that empty spaces are streets and courtyards, but also squares, parks and any other element processed without heights.

4

SHRINKING CITIES

The video game industry is one of the most powerful industries in the world nowadays. There are many kinds of video games, some set in fantastic environments and others set in real world environments. In particular the second kind needs the work of a designer, who maps out a simplified model of the real city (or any other kind of environment). This is a manual work starting from zero, where the first step is to outline the street network of the final city. In this chapter we present a solution to generate automatically a first approach for the simplified design of a city model. This first draft of the city street network is like a kind of skeleton or template, from which the artist can (and should) start working until getting the desired result. The output that our process provides to designers is a street network that reduces the city area as much as the designer wants, including landmarks and most important streets, but keeping the relative position between them. To proceed it, we select the most important places of the city and connect them with the most important streets. Then, we run the shrinking process which reduces the area in an irregular way, prioritizing the removal of the areas of less important places.

4.1 INTRODUCTION

Nowadays, the video game industry is one of most powerful and wealthy in the world, even more than cinema and music put together. There are many kinds of games. Games for kids, youths, gils, adults and even for babies. There are games for one player and for multiple players. Video games to play off line and on line, collaborating or competing against other players from around the world. We can also find games with different environments and atmospheres. This last characteristic is the one that interests us the most for this chapter. Architecture –meaning both landscapes and structures– is what turns the monotonous grid into the living world of the computer game. Its importance is on a par with character design in creating the player's visual experience. Character design tells you who you are and who you interact with; architecture tells you where you are. But more than that, it also tells you what might happen there, and even sometimes what you ought to be doing [1]. There are video games elapsing in imaginary worlds. Those new worlds have new architecture, sometimes are abstract or surrealistic, and some other times realistic. Likewise, they can transmit wonderful and even funny, or apocalyptic and dangerous worlds. In any case, any new synthetic world is not inspired by a real environment. On the other hand, there are video games inspired by real environments and also historic moments. In this second case, video game companies put efforts on giving to the player the feeling of that real world. To fulfill this purpose, it is important to make the architecture design look familiar to those that know the real place. This can be achieved with allusions to real architectural elements, architecture styles and clichés. Some famous examples are *Grand Theft Auto IV* (RockStar Games®) in Manhattan (New York), and *Assassin's Creed II* (Ubisoft®) in Venice and Firenze. See Figures 26, 27 and 28. These are only a few examples in a large list of realistic experience video games.

To create a top realistic video game a team of hundreds of people is needed. An important subset of this team are designers. Designers are responsible to create the visuals of the environment and characters, but also the history narrative, different game levels, game dynamics, etc. Here we are interested in their creative work to design city models. To create a synthetic model of a realistic city, designers start by drafting the street network. This outline captures the landmarks (essential important places of interest). Once this draft is done, they continue on by designing buildings. In both steps, they may start with manual designs and then move them to 3D modeling, texturing, illumination, etc. The majority of these city designs generate cities considerably smaller than the original cities serving as inspiration source. The reason is that essential places where the actions take place, may be too far apart in the real world, forcing the player to travel long, boring landscapes between key game events. A good example to illustrate this situation is Manhattan. In the real world, if you want to go from Central Park to the southern limit of the island to see the Statue of Liberty, you have to walk through more than fifty blocks. Now, let us imagine that we recreate a real scale city for a video game.

If the video game script requires the character to walk from Central Park to the Statue of Liberty it would imply a several-hour walk. This probably would be one of the most boring games ever created. One reason is that most of the blocks are similar and this walk quickly becomes repetitive. Therefore, in a real game, a designed city must reduce the number of blocks between interesting places to keep the action more dynamic while retaining the essence of the city. If you look at Figure 26, the number of blocks from Central Park to the southern limit of the island has been reduced to about twelve blocks instead of the more than fifty in the real Manhattan.

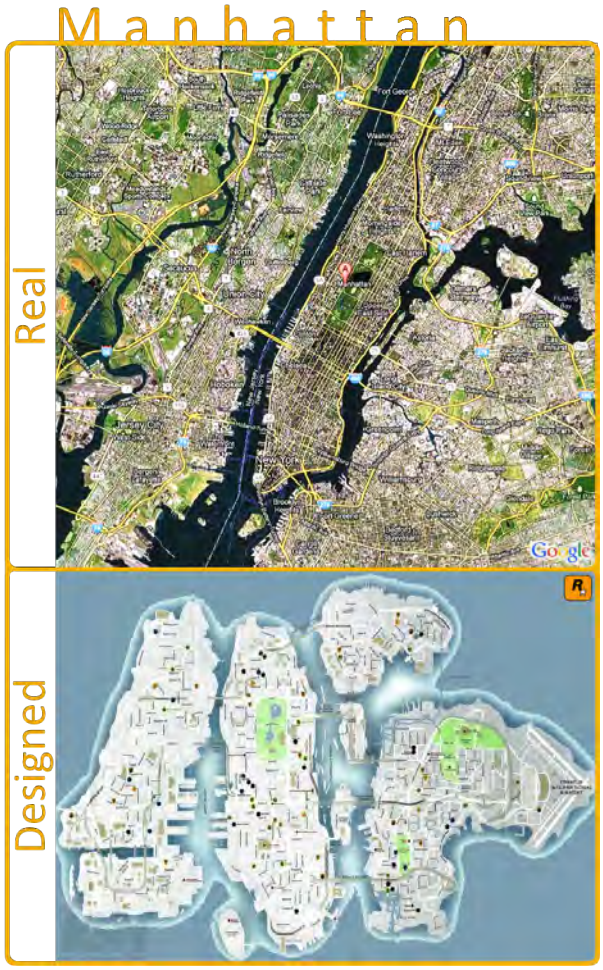

Figure 26: Manhatan (New York). *Top*: Real map from Google Maps®. *Bottom*: *Grand Theft Auto 4* map designed by RockStar Games®.

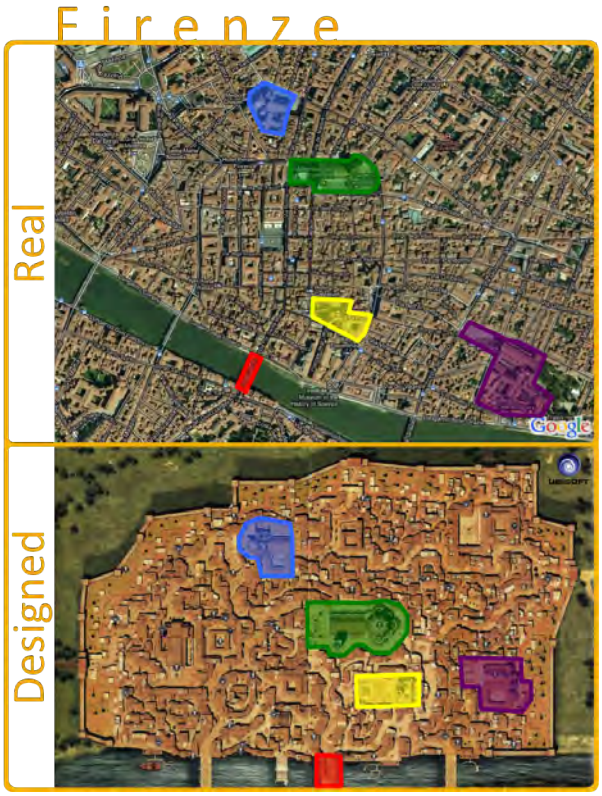

Figure 27: Firenze. *Top*: Real map from Google Maps®. *Bottom*: *Assassin's Creed II* map designed by Ubisoft®. Landmarks highlighted: *Basilica di San Lorenzo* (blue), *Piazza del Duomo* and *Piazza San Giovanni* (green), *Palazzo Vecchio* and *Piazza della Signoria* (yellow), *Ponte Vecchio* (red) and *Basilica di Santa Croce* (purple).

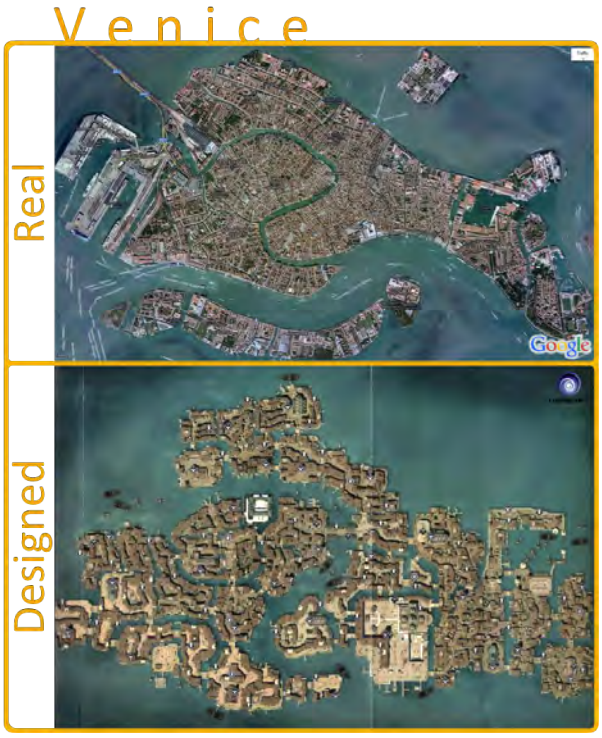

Figure 28: Venice. *Top*: Real map from Google Maps®. *Bottom*: *Assassin's Creed II* map designed by Ubisoft®.

We want to help automatizing this process to let designers start from an initial street network with landmarks. From free GIS data as input, we want to process it and get a *shrunk* city street network. The resulting city draft should have a surface area smaller than its real world counterpart. Also, landmarks should keep their geometries and relative positions within the city distribution. Our process will remove less-important streets and preserve important places. The city area is reduced in an irregular way; the surface removed is from unimportant, more distant elements from the key places. Hence, we do an irregular reduction guided by the area important place density. The output of our process is a city street network with a set of streets and other important city elements: gardens, buildings, bridges, etc. This first approach should be considered only a kind of template or skeleton from which designers should start their work, not the final city model. Designers can –in fact, should– add more streets if needed and edit some street geometries if needed to achieve the desired city model.

In this chapter, we are going to explain the steps to get the *shrunk* city. Implementation details about the city data structure; core street operations and how city operators work in a pipeline are presented in the next chapter, Chapter 5.

4.2 INPUT

To start the shrinking process, the algorithm needs an input consisting of two elements: the city map to shrink and the list of important places to keep in the output resulting city. We explain in the next subsection what each one of these input elements is and how we get them.

4.2.1 *City map*

The first input element that the shrinking process needs is an original real city map. We use open GIS data to get the city map and therefore the city street network. This provider is the OpenStreetMap [31] community. OpenStreetMap is a free, editable map of the whole world that is being built by volunteers largely from scratch and released with an open-content license. This project creates and distributes free geographic data for the world. They started it because most available maps actually have legal or technical restrictions on their use, holding back people from using them in creative, productive, or unexpected ways. This last motivation of OpenStreetMap is what convinced us to use this data source as our input.

When users download data form OpenStreetMap (OSM), they do it in an specific format with the extension .osm. The OSM file is basically an eXtensible Markup Language (XML) file with a header, a list of nodes, a list of ways and a list of relations, each one with a unique identifier.

NODE: defines a specific point in the earth's surface defined by its latitude and longitude. A node has a set of attributes. Within them, the only mandatory ones are the identifier and the latitude and longitude coordinate values. It can also have a list of tags adding information to the node. Generally speaking, if the node is a standalone point, it has its own tags, but if it is used as a point along a way it usually has not.

- *Attributes:* id, lat, lon, ...
- *Tags:* name, traffic_sign, ...

WAY: is an ordered list of nodes (between 2 and 2000) that defines a polyline. Ways can also represent the boundary of an area. In this case, the first and last nodes are the same. It has a set of attributes, including the mandatory identifier value. It has also the ordered list of references to the nodes that defines it shape. Finally it can also have a list of tags enriching the way with extra information.

- *Attributes:* id, ...
- *Node references:* Each one with the node reference identifier.
- *Tags:* highway, name, ...

RELATION: is a multi-purpose data structure that basically documents a relationship between two or more elements (nodes, ways and/or other relations). These elements are grouped between them for any conceptual relationship, so the meaning of the relation should be defined by its tags. Again it has a set of attributes within the identifier. Then a list of relation members, specifying the kind of member of each one, and a list of tags to give extra information about the relation.

- *Attributes:* id, ...
- *Members:* Each one with type (node | way), element reference identifier and role.
- *Tags:* name, route, type, ...

TAG: All types of data elements can have tags. Tags enrich element information giving them their particular meaning. A tag consists of two free format text fields: key and value. There is no fixed dictionary of tags, but there are many conventions documented on the OSM wiki *Map Features* page [31]. For example, $highway = residential$ defines the way as a road whose main function is to give access to people's homes.

You can see a simple incomplete example in the following code:

```
<! - - ########## OSM FILE EXAMPLE ########## - - >

<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6" generator="CGImap 0.0.2">
<bounds minlat="54.0889580" minlon="12.2487570" maxlat="54.0913900" maxlon="12.2524800"/>
    <node id="298884269" lat="54.0901746" lon="12.2482632" user="SvenHRO" uid="46882" visible="true" version="1" changeset="676636" timestamp="2008-09-21T21:37:45Z"/>
    <node id="261728686" lat="54.0906309" lon="12.2441924" user="PikoWinter" uid="36744" visible="true" version="1" changeset="323878" timestamp="2008-05-03T13:39:23Z"/>
    <node id="1831881213" version="1" changeset="12370172" lat="54.0900666" lon="12.2539381" user="lafkor" uid="75625" visible="true" timestamp="2012-07-20T09:43:19Z">
        <tag k="name" v="Neu Broderstorf"/>
        <tag k="traffic_sign" v="city_limit"/>
    </node>
    ...
    <node id="298884272" lat="54.0901447" lon="12.2516513" user="SvenHRO" uid="46882" visible="true" version="1" changeset="676636" timestamp="2008-09-21T21:37:45Z"/>
    <way id="26659127" user="Masch" uid="55988" visible="true" version="5" changeset="4142606" timestamp="2010-03-16T11:47:08Z">
        <nd ref="292403538"/>
        <nd ref="298884289"/>
        ...
        <nd ref="261728686"/>
        <tag k="highway" v="unclassified"/>
        <tag k="name" v="Pastower Strasse"/>
    </way>
    <relation id="56688" user="kmvar" uid="56190" visible="true" version="28" changeset="6947637" timestamp="2011-01-12T14:23:49Z">
        <member type="node" ref="294942404" role=""/>
        ...
        <member type="node" ref="364933006" role=""/>
        <member type="way" ref="4579143" role=""/>
        ...
        <member type="node" ref="249673494" role=""/>
        <tag k="name" v="Kustenbus Linie 123"/>
        <tag k="network" v="VVW"/>
        <tag k="operator" v="Regionalverkehr Kuste"/>
        <tag k="ref" v="123"/>
        <tag k="route" v="bus"/>
        <tag k="type" v="route"/>
    </relation>
    ...
</osm>
```

This is a file structure that makes an expensive computational cost for any basic operation over nodes and ways. To start working on the shrinking process, we obviously decided to implement an efficient data structure to store the city and provides us optimized algorithms for common nodes and ways operations, the *CityLib*. This library is explained in the next chapter, Section 5.1.

One last thing to take in consideration is that an OSM file can store many kinds of geographical information. Highways, railways, aerial ways, buildings, historic elements, power lines, nature, waterways, leisure, shopping and a large etcetera. To know how we differentiate the street information and the rest of information, see Section 5.1.

4.2.2 *Important places*

With a city street network, it is not enough to decide which are the important places to keep when shrinking the city. Besides from the .osm file, the algorithm needs also information about the city important places. For this purpose, we created the .osmi (open street map importance) file that encodes important places information. The .osmi file is also an XML structured file storing a list of important places, inspired by the same organization of .osm files. An important place in an .osmi file is similar to a relation in a .osm file, both files group a set of nodes and ways. Basically, an .osmi file place element stores a list of one or more nodes and way elements which together make an important place, street or region.

PLACE: A place defines an element or a set of elements of the associated .osm file with a high importance or interest weight within the city. Each place has a unique identifier, a list of members configuring the important place and a list of tags giving more information to the place element. We can add as many tags as information we need.

- *Attributes:* id, role (important | interesting)
- *Members:* Each member has an .osm file node or a way reference identifier and it type (node | way)
- *Tags:* appearW, modifyW, name, function, ...

The mandatory and most important tag is *appearW*. This tag value defines how important this place is. This value is between 1 and 0; 1 being the most important. Another relevant tag is *function*. In case that we want to keep a neighborhood or area exactly like it is, we don not need to add all its inner elements. We can define a place, tagging it with key *function* and value *boundary* and defining the boundary polygon of the area. To define the polygon we should list the place members alternating nodes and ways starting and ending with the same node, like with the following example: $n_1, w_1, n_2, w_2, n_3, ..., n_n, w_n, n_1$. The nodes before and after a way delimit the section of the way used to get the polygon of delimiting the important area. Hence, the resultant polygon will be the one defined when concatenating for all the w_i ways, their segments between nodes n_i and n_{i+1}. For this kind of boundary important area elements, the algorithm will consider all elements inside the polygon with the same weight that the boundary has.

You can see a simple example in the following code:

```
<! - - ########## OSMI FILE EXAMPLE ########## - - >

<?xml version="1.0" encoding="UTF-8"?>
<osmi version="0.1" osm="firenze.osm">
    <place id="112" role="interesting">
        <member ref="23396102" type="way"/>
        <tag k="appearW" v="1."/>
        <tag k="modifyW" v="0."/>
        <tag k="name" v="Battistero di San Giovanni"/>
    </place>
    <place id="113" role="interesting">
        <member ref="24758173" type="way"/>
        <tag k="appearW" v="1."/>
        <tag k="modifyW" v="0."/>
        <tag k="name" v="Campanile di Giotto"/>
    </place>
    <place id="100" role="interesting">
        <member ref="270928894" type="node"/>
        <member ref="27648129" type="way"/>
        <member ref="250253340" type="node"/>
        <member ref="23158535" type="way"/>
        <member ref="270929658" type="node"/>
        <member ref="113775998" type="way"/>
        <member ref="270928894" type="node"/>
        <tag k="appearW" v="1."/>
        <tag k="modifyW" v="0."/>
        <tag k="name" v="Piazza del Duomo AREA"/>
        <tag k="function" v="bounding"/>
    </place>
    ...
</osmi>
```

The list of important or interesting places, or, in other words, the content of the .osmi file, can be created manually or automatically by running a process able to find the most important places in the city and finding their .osm element references, like Grabler et al. [34] do.

4.3 CLEAN INPUT CITY

The process must start with a preliminary cleaning step. As we have explained in the previous chapter, any introduced data is prone to inaccuracies. OSM is not an exception. Besides, when we download the data from OSM we select a rectangular area and all the ways inside that rectangle are downloaded. Some of them can be

very long and extend far outside the selected area. To solve these problems, we run a cleaning process to the street tagged ways. This process can be parametrized to do different kinds of cleaning operations:

- **Add intersection:**
  Finds intersecting ways without a node at that position. If it finds it, adds a new intersection node to both way node lists.
- **Remove empty ways:**
  Finds all ways with a void list of nodes or with only one node. It removes all the ways fulfilling this condition.
- **Remove segments outside bounding box:**
  Removes all nodes falling outside the bounding box defined when downloading the map. In consequence, all way segments defined outside the box are removed.

We run this cleaning process by activating the removal of void and single-node ways, also removing the way segments falling outside the area of interest. But we do not activate the addition of way intersections. The reason is to not connect ways that may not be sharing an intersection node. For example, two roads crossing, one being a bridge and the other going down the first one. They are at different heights so are not connected. See results of this step in Figure 29.

Figure 29: Firenze center: The clean-up process removes streets outside the selected bounding box. *Left*: Some ways extending outside the selected bounding box. *Right*: Way segments outside the bounding box are cropped.

4.4 STREETS SELECTION, CREATING THE *skeleton* CITY

We can clearly divide all the process steps in two main parts: selecting the important streets when doing the city shrinking process, and the shrinking itself. This section explains the first step and the sequence of operations that the original city must go through before starting with the shrinking algorithm. The goal of this part is to go from the original full city to an *importance skeleton* city, with only the most important places and streets, and the best streets to connect them. From now on, we are going to talk about the *skeleton* city as the city which stores the most important places and streets, plus the streets to connect the important places.

4.4.1 *Add indirect important streets*

In this first step, expand the list of important places that the .osmi provides us with. This expansion adds some streets that should be considered as important as a consequence of other .osmi elements. Basically, in the .osmi file we have the important places. An important commercial street can be in the file, but also buildings of historical or architectural value, monuments, parks... On the other hand, an .osmi file can define an area of interest without selecting all the interior ways of that area. So, we have to do two tasks here: add streets surrounding a (non-street) landmark and add all streets inside an important delimited area. For example, see Figure 30 for streets close to the *Piazza del Duomo* in Firenze.

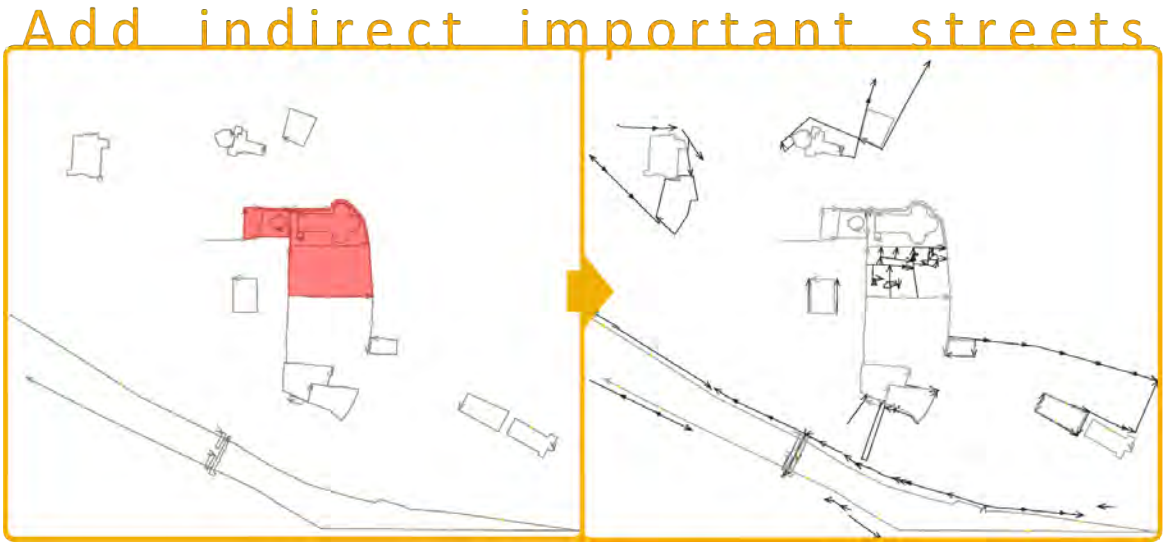

Figure 30: Firenze center: Adding indirect important streets. *Left*: The .osmi important places in gray, with a boundary area highlighted in red. *Right*: The added indirect streets in black. Note all the indirect streets inside the area highlighted on the left image.

To get the streets surrounding a (non-street) landmark, we take all the nodes of the important place and find their nearest streets. Then we add all the resulting

streets as important streets with the same weight as the place analyzed. If the place is a complex place element with more than one member, we do exactly the same for all the place members. The method to get a node's nearest street takes advantage of the *City* proximity acceleration structure of Section 5.1.6. Note that the nearest streets found can be parallel or perpendicular to the building facade. Also, this process may not find all the streets surrounding an important place. Imagine an important building where two consecutive nodes of a large segment find their nearest street perpendicular to the building facade. Then a short street parallel to that segment might not be found. These situations are not a problem for our process, because we are not aiming at surrounding the important places, but we need to have at least one way to arrive to them.

To get all the streets inside an important region defined in the .osmi as a *bounding* place, we run a *point-in-polygon* test to find street nodes falling inside the region. To do it, the first thing needed is to define the polygon delimiting the important surface. This polygon is defined by listing, for each way w_i of the place, the nodes between the previous and next nodes n_i and n_{i+1}. Once we get the polygon, we find the rectangular space that covers the *City* proximity grid (see Section 5.1.6). With this polygon positioned, we run the *point-in-polygon* algorithm only for the nodes in the grid cells covered by the polygon. This avoids many expensive computations.

4.4.2 *Select streets*

After the previous step, we get a list with the important places and also their nearest streets, each one with an importance weight value, encoded in the tag *appearW*. The next step is to select those streets that together will form the original city skeleton to be shrunk. To get them, we have to sort all city streets by their importance index. Then we select a percentage of the streets of the city with the highest importance values. This threshold is the *shrinking factor*, specified by the user. From this selection, we get a list with one element for each city street selected. The list stores four values for each element: importance weight ($\mathtt{impW}$), street length ($\mathtt{len}$), importance density ($\mathtt{impD}$, which is computed as $\mathtt{impW}/\mathtt{len}$) and the important place identifier it belongs to if it is an indirect added important street.

The street importance weight is a value between 0 and 1. All the streets in the important place list will have a value in the range $\mathtt{impW} \in [0.5, 1]$, while unimportant places will get a value in the range $\mathtt{impW} \in [0, 0.5)$. Figure 31 shows the schema of how the importance weight is calculated. In the next paragraphs we extend the explanation about how we built it.

We process the importance value of the street member of an important place by using the following formula:

$$impW = 0.5 + 0.5 \cdot appearW$$

where $appearW$ is the place importance weight and $impW \in [0, 1]$. In case that the important place is tagged with the *indirect* key, we add the important place from which it is related to. If not, we leave this value empty. For streets not member of any important place, we should calculate their weight, too. To do it, we firstly process all city streets to get the following data:

- Number of intersection nodes for each street ($nInt$).
- Length of each street (len).
- Shortest city street length (len_m).
- Largest city street length (len_M).
- Maximum number of intersection nodes in a street ($nInt_M$).

With this data, we can calculate the importance weight of the the unimportant streets, by summing two factors. The first one is an associated weight value depending on the type of street, and thus in the .osm *highway* tag value. The importance associated to each street type is defined by the user. We configure it from more to less important depending on the *highway* tag value: *motorway*, *primary*, *secondary*, *tertiary*, *residential*, with any other value grouped together. The second factor considers the length and number of intersections that the street has. The algorithm normalizes each street length and its number of intersection nodes, considering the maximum and minimum values found in the whole city. We have distributed the influence of each of these factors in the following way:

$$impW = strT + lenF + intF$$

- Street type value $strT$:
  - $strT = 0.05$ for unspecified street type.
  - $strT = 0.10$ for *residential* street.
  - $strT = 0.15$ for *tertiary* street.
  - $strT = 0.20$ for *secondary* street.
  - $strT = 0.25$ for *primary* street.
  - $strT = 0.30$ for *motorway*.
- Length factor $lenF \in [0, 0.1]$ adds a normalized value:

$$lenF = \frac{nInt}{nInt_M} \cdot 0.1$$

- Intersection factor $\mathrm{intF} \in [0, 0.1]$ adds a normalized value:

$$\mathrm{intF} = \frac{\mathrm{len} - \mathrm{len}_m}{\mathrm{len}_M - \mathrm{len}_m} \cdot 0.1$$

The sum of the street type value and the length and the intersection factor will always result in a value between 0.05 and 0.5, that will guarantee that all those unimportant streets have a lower weight than all the important ones.

To summarize, we get each street importance weight impW and importance density impD by using the following formulas:

- **Important place streets weight:**

$$\mathrm{impW} = 0.5 + 0.5 \cdot \mathrm{appW}$$

- **unimportant streets weight:**

$$\mathrm{impW} = \mathrm{strT} + \frac{\mathrm{nInt}}{\mathrm{nInt}_M} \cdot 0.1 + \frac{\mathrm{len} - \mathrm{len}_m}{\mathrm{len}_M - \mathrm{len}_m} \cdot 0.1$$

- **For all streets importance density:**

$$\mathrm{impD} = \frac{\mathrm{impW}}{\mathrm{len}}$$

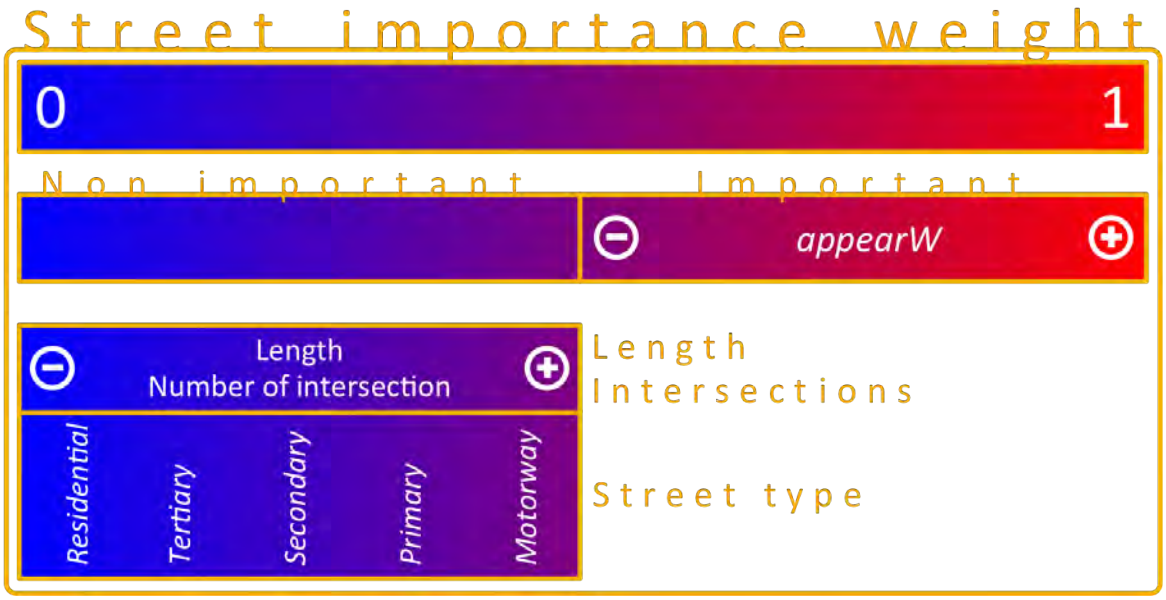

Figure 31: Color gradient chart for street importance weight. From less important in blue to most important in red. Streets between 0.5 and 1.0 are the marked as important in an .osmi file. Unimportant ones get their importance weight between 0 and 0.5 depending on the street type, length and number of intersections.

After getting the list of streets sorted by importance weight, we select a given percentage of streets on the top of the list. This percentage is the user-defined *shrinking factor* (shF). Depending on this value, the algorithm will take more or fewer streets to create the new *importance skeleton* city. In Figure 32 we can see the difference depending on the shF value. With a value of 0.1 it takes only the 10% top weighted streets, while with 0.2 it takes the 20%. These two examples are very useful to be compared with Figure 30 *Right*, which shows the important places and their indirect associated important streets. If the user selects a *shrinking factor* of 0.1, it will not get all the important places and associated streets, while increasing it to 0.2 it will get all of them plus some other unimportant streets. Probably, in this case, the best decision is to define a shF value of 0.15 or higher.

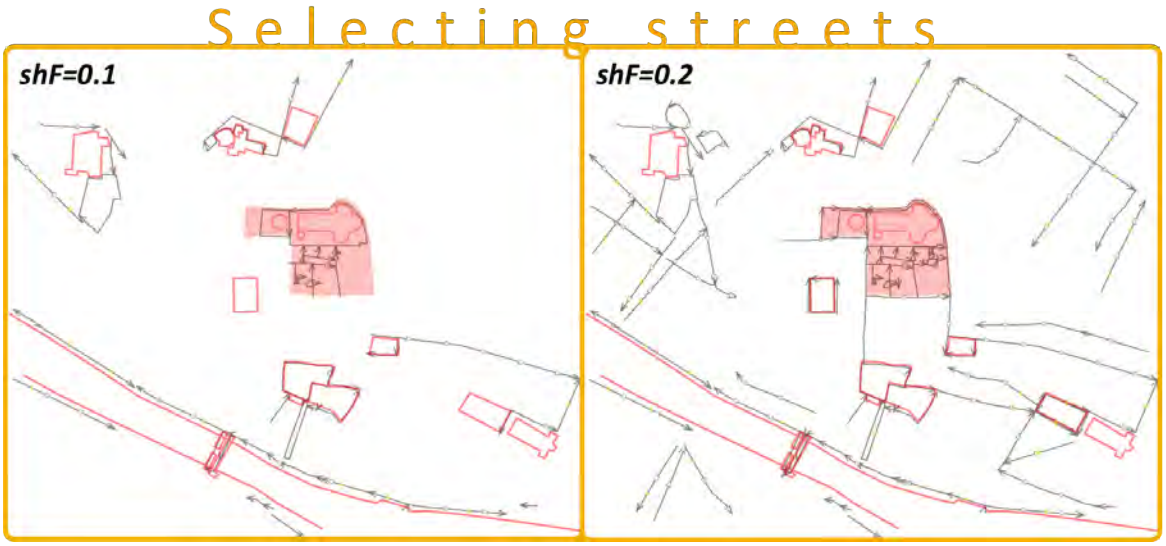

Figure 32: Firenze center: Streets selected to create the *importance skeleton* city of Firenze, using *shrinking factors* of 0.1 and 0.2. The red ways are the important places that are not streets defined in the .osmi file.

4.4.3 *Connect important streets*

At this point in the process, we have a set of ways, with large importances. But it is highly probable that they form (the ways) a disconnected city model. Therefore, in this step we are going to interconnect them. For this, we find all the disconnected clusters and connect them by using a few city streets discarded from the previous step. Each connecting path is added to the resulting *importance skeleton* city. The path to connect two clusters will be the one with the highest importance value.

To get all the disconnected clusters, we run a tool which implements a vector-based *flood fill* algorithm (see Section 5.2.2). You can see in Figure 33 the set of clusters found in the Firenze center example using a *shrinking factor* value of 0.15.

The work of connecting the different clusters found is again a *City* library tool which implements an A* (*A-star*) algorithm. This implementation is explained in more depth in Section 5.2.1. However, let us introduce here that our A* implementation allows to run the algorithm considering the city network either as a directed or indirected graph. On the other hand, we can specify the desired distance function to consider the cost of moving between two consecutive nodes.

To find the shortest path between two street clusters, we first get the centroid of each cluster. Then we sort the clusters from closer to farther from the city bounding box center. In this way we avoid the possibility of getting bad situations like starting the cluster's connection process with the two most distanced clusters – hence, with longest path to connect them. Finally we run the A* algorithm over the original city trying to connect two nodes of two different clusters with the shortest path in the sorted clusters list order. The A* algorithm finds the minimal-cost path from an initial node to a goal node. The cost of the path depends on the distance function we define to calculate the cost from one node to another one. The initial and goal nodes in our case are one node from each one of the pair of clusters to be connected. Note that the selection of the *initial* and *goal* cluster nodes to be connected is not important, because in the defined distance function we give a distance of 0 to all transitions between two nodes inside the same cluster, and to segments already on the *importance skeleton* city. Once we get two clusters connected, we try to connect the resulting new cluster to the next cluster in the sorted list. The process finishes when we can not connect more clusters. Hopefully the resulting *importance skeleton* city will have only one cluster because all the original clusters have been connected, but this is not always the case. If the resulting city still has a disconnected street cluster, and this cluster has no important place in it, so we remove it. This implies to remove all cluster ways and nodes. See the resulting connected city in Figure 33.

Clearly, in this step, the most important thing is to define a distance function according to the characteristics of our desired connecting path between two clusters. The distance function is provided to our A* method. This function defines the cost dist_{AB} of going from node A to node B. For this, we combine two values: the inverse importance of the street segment imp_{AB} (using the importance density impD calculated in the previous step, see Section 4.4.2) and the Euclidean distance d_{AB}. The imp_{AB} gives preference to paths through important streets, while d_{AB} is used to guide the path to the destiny node B. The respective weight of each factor can be decided by the user using parameter $\alpha \in [0, 1]$. The more balanced toward imp_{AB} (higher α), the more computational cost we have. Therefore, we define the distance function like it follows:

$$\mathrm{dist}_{AB} = \alpha \cdot \frac{1}{\mathrm{imp}_{AB}} + (1 - \alpha) \cdot d_{AB}$$

To get the importance value imp, we have to apply the following function:

$$\text{imp}_{AB} = \begin{cases} 0 & \text{if A and B are not neighbor nodes.} \\ 0 & \text{if a street from A to B is already in the } \textit{skeleton} \text{ city.} \\ \text{impD}_{AB} \cdot \text{dist}_{AB} & \text{otherwise.} \end{cases}$$

The A* algorithm finds the shortest path to go from *initial* node I to *goal* node G. The path distance is the overall sum of all segments AB_i of the resulting connecting path. A* gives the shortest path, which is the one with lowest path distance dist_{IG}:

$$\text{dist}_{IG} = \sum_{i=1}^{n} \text{dist}_{AB}^{i}$$

In Figure 33 we can see the result of connecting the clusters after running our algorithm on the Firenze city center with a shF of 0.15. We can see different unconnected sets of streets (the clusters) in Figure 33 *Left*, and in *Right*, the resulting paths in red added to connect the city. The *importance skeleton* city is now a connected city (its network is a connected graph).

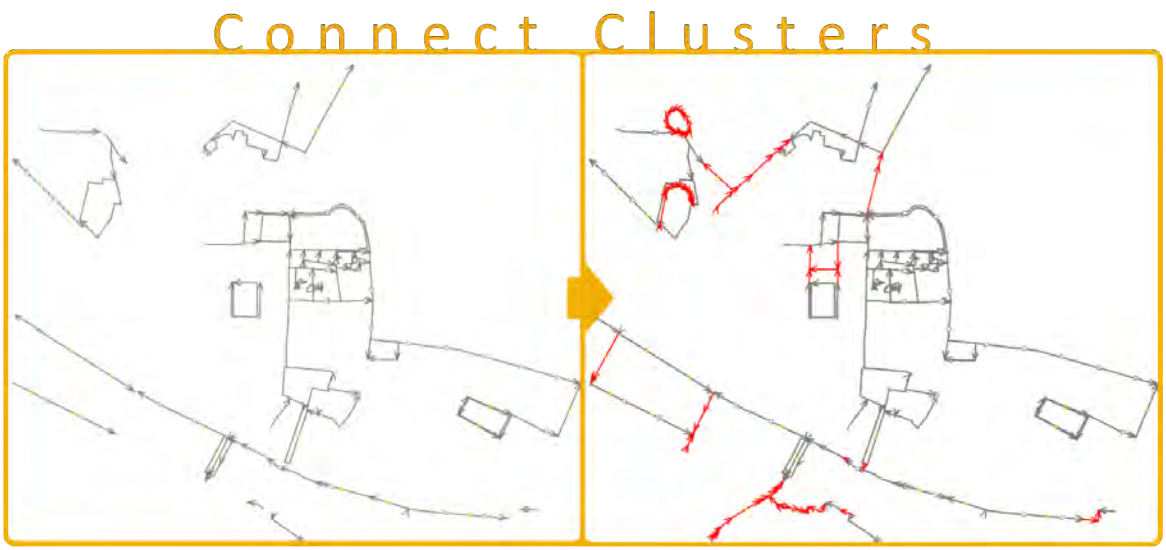

Figure 33: Firenze center: Connecting important street clusters. *Left*: The original *skeleton* city after selecting the most important streets. There are unconnected sets of streets. *Right*: The added streets highlighted in red to connect the city.

4.4.4 *Connecting fringe paths*

At this point we have a new *importance skeleton* city with a connected street network. This new city is a subset of all original city streets, i.e., the most important ones. These streets are the ones to keep in the resulting *shrunk* city. Among them there are some dead-end streets. In the real world, the proportion of dead-end streets in a city is really small. However, because of our previous step, the new *skeleton* city has too many of them. This gives an undesirable look for the city street map.

To resolve this issue, we connect those dead-end streets in the *skeleton* city that were not in the original city. We first detect all these streets by finding the ending nodes with only one connected neighbor. Then the algorithm tries to find a way through the original city streets, from the ending nodes to other nodes existing in the *skeleton* city, but without using segments already in the *skeleton*. Obviously, we can always find a path for this situation, but we restrict the connecting paths to have a maximum of m steps (street segments) between the ending node and the connecting one. This m value defines the "beautification" degree and can be defined by the user by deciding how restrictive to be when removing dead-end streets in the new city. For each ending node n_i, we visit its m-degree neighbors in the full city and store all the nodes that are candidate to connect paths. A candidate path is a path with two ending nodes in the *skeleton* city, one of them being the ending node n_i itself. Hence, for each ending node we have a set (maybe empty) of candidate paths. We want to get the best, shortest connecting path for each dead-end street. To do it, we run an $A^\star$ algorithm for each n_i node pair over the original city. This $A^\star$ uses the Euclidean distance between two node coordinates as distance function, differentiating with the more complex distance function used in the previous step (Section 4.4.3). When the best m-degree connecting path is found, we add the path street segments to the *importance skeleton* city. We want to emphasize that we are adding street segments, not full streets from the full city to the *skeleton* city, giving to each path a brand new way identifier. This is done to avoid generating new dead-end streets. It is mandatory to add connecting streets with no new dead-end nodes. See the result of this step in Figure 34 right. The connecting paths are highlighted in red. In these images we can appreciate how the connected dead-end streets depend on the degree value m. With degree $m = 3$, there are three dead-end streets that cannot be connected. This is because their possible connecting path has a larger number of nodes to connect. If we increase this value to $m = 10$, the algorithm solves for all dead-end streets.

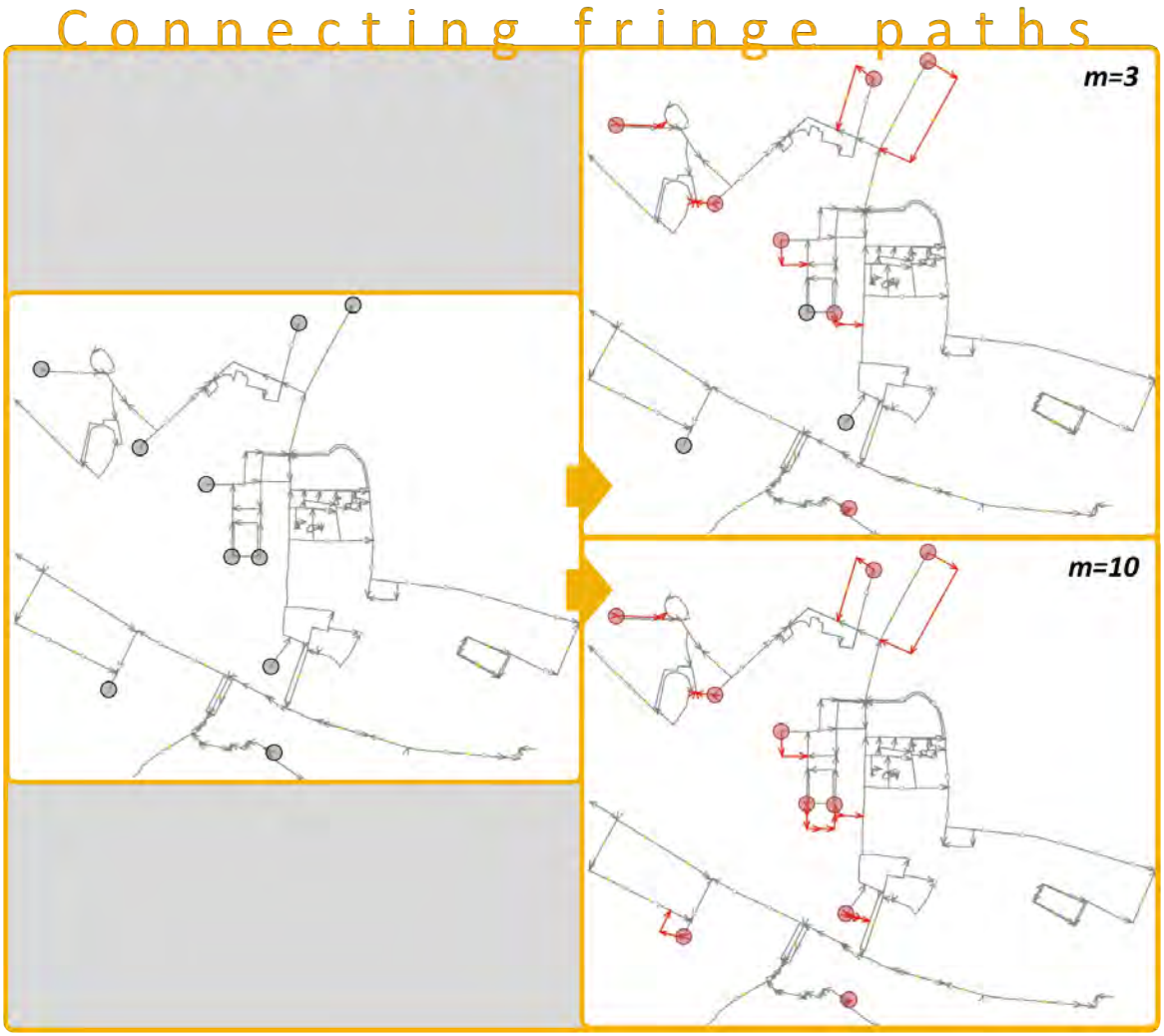

Figure 34: Firenze center: Connecting fringe paths with degrees 3 and 10. *Left*: Black circles mark dead-end streets absent in the original city. *Right*: The dead-end street extension paths added due to the connecting step are highlighted in red. The ones that could not be connected are still black circles. Note that the number of still unconnected dead-end streets is larger with $m = 3$ than with $m = 10$.

4.4.5 *Add important places*

This is the last step before the actual shrinking process of the *skeleton* city. It does not affect the street map. The previous steps created the important streets for the directed shrinking process, so we have a street network without any other important element. Before starting the shrinking process, the algorithm also adds all the non-street important elements, specified in the .osmi file (buildings, parks, squares, rivers, etc.). For the example of the Firenze center, this step will add landmarks like *Basilica di San Lorenzo*, *Piazza del Duomo*, *Arno* river and *Ponte Vecchio*, among others, as can be seen in red in Figure 35 *Right*.

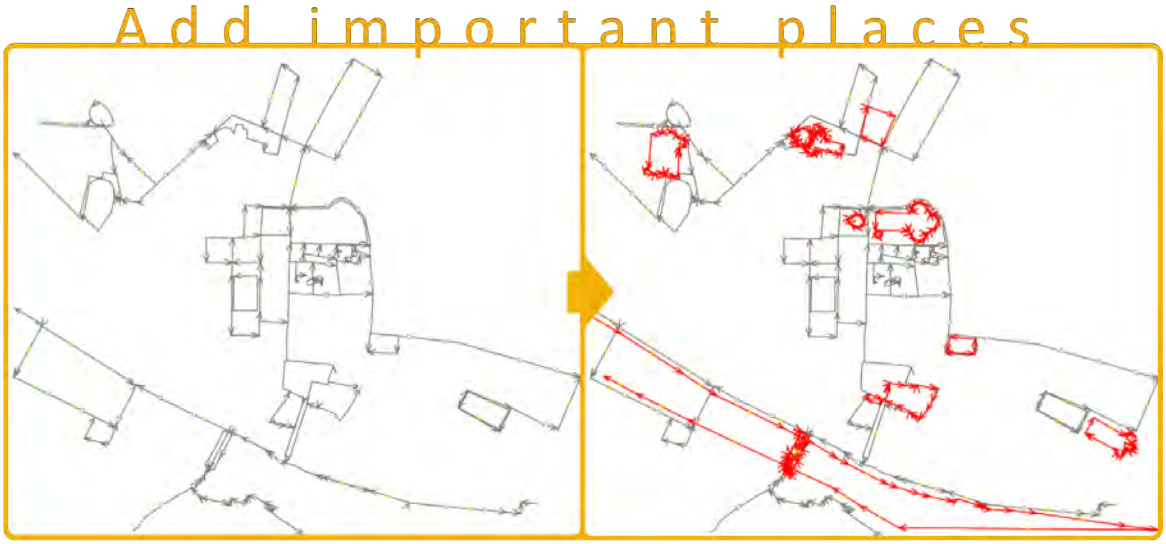

Figure 35: Firenze center: Adding the .osmi non-street important places to the *skeleton* city. *Right*: In red, the important places added. There is no change in the street map.

Summarize *skeleton* city:
Before leaving this section, we want to illustrate the partial result at this point of the shrinking pipeline. In Figure 36 we show, in *Right*, the resulting *skeleton* city that is going to be shrunk with the subset of ways selected from the complete original city (In Figure 36 *Left*). Note that the city area has not suffered any reduction yet. This figure shows the result by using a shrinking factor of $\mathtt{shF} = 0.15$, a balance of importance and distance equivalent $\mathtt{impDistBalance} = 0.5$, and a connecting fringe paths degree of $m = 10$.

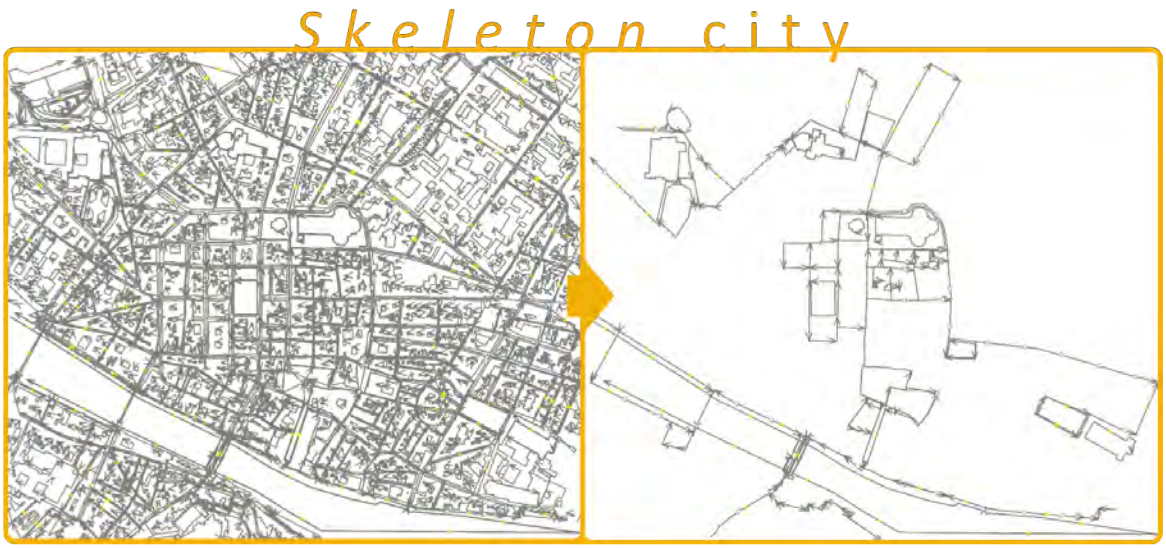

Figure 36: Firenze center: The resulting *skeleton* city. *Left*: The original Firenze center with all its streets, buildings and its river. *Right*: The *skeleton* city with the 15% most important streets and places.

4.5 CITY SHRINKING

From the previous section, we have an *importance skeleton* city from the original city. This *skeleton* city has the most important places and streets of the city with the same sizes and positions that in the original city. Now, we reduce de city area by running the shrinking process. This process is applied over the *skeleton* city instead of the full city because we do not want to consider all the city elements in the same way. The shrinking process is a directed area reduction with the goal of reducing those areas with smaller element densities. To avoid processing those less important areas, we removed all streets and elements without importance.

4.5.1 *Seam Carving algorithm*

Before explaining the city shrinking process, it is essential to understand how the Seam Carving algorithm works [7]. Seam Carving is basically an image retargeting operator. This operator is a non-homogeneous resizing system with content-awareness. Avidan and Shamir [7], argue that an image resize should not only use a geometric constraint, but consider the image content as well. See in Figure 37, a summarizing figure of Seam Carving that explains the process and shows the results of a resized image with a regular scale process or with it algorithm.

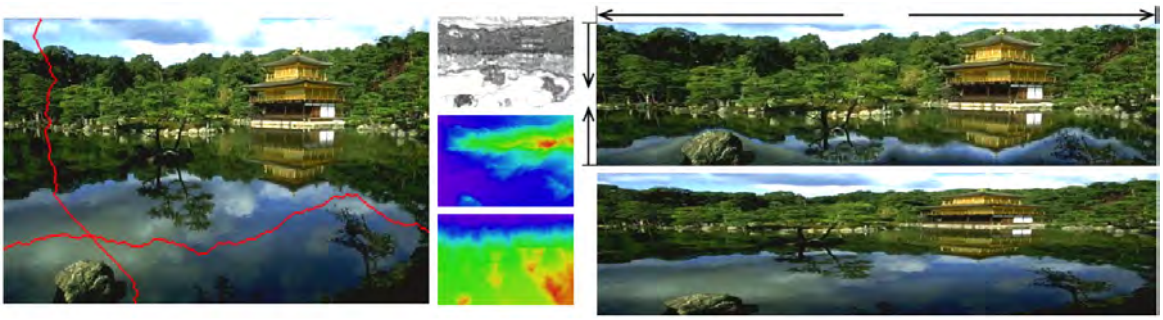

Figure 37: Seam Carving.A seam is a connected path of low-energy pixels in an image. On the left, the original image with one horizontal and one vertical seam. In the middle, the energy function used in this example (the magnitude of the gradient), along with the vertical and horizontal path maps used to calculate the seams. By automatically carving out seams to reduce image size, or expanding on both sides of seams, content-aware resizing is achieved. The example on the top right shows a result of extending in one dimension and reducing in the other, compared to standard scaling on the bottom right.
Reprinted from [7].

A seam is an 8-connected path of pixels from top to bottom or from left to right that optimizes an image energy function. By carving out repeatedly this optimal path pixels, the image can be reduced. And the other way around, by inserting pixels along this optimal path repeatedly, the image can be elongated. The resulting

image depends on the selection and order of the seams, and this selection depends basically on the energy function. In our code, we are interested in reducing a city map. Thus, we should define an appropriate energy function.

Even though Seam Carving is an image operator initially designed to retargeting, Avidan and Shamir also explore more utilities of the algorithm [7], for example removing image elements. Furthermore, the Seam Carving operator has been used for other purposes beyond image retargeting in other papers. As examples, Qu et al. [61] use it to redimension urban landmarks and streets to visit them and Emilien et al. [25] to straighten or distribute natural elements throughout a terrain. We want to shrink the city surface considering the content of the city, by applying non-homogeneous resizing. Therefore, we should find a proper energy function to find seams and remove them. The problem here is that we do not have an image (a set of pixels with their respective color), we have a *City* structure, with a graph representing the city street map. To directly apply this technique to our city, we map the city area to an image. If we divide the city in a grid, then we can consider each cell to be an image pixel. In this way, when removing a seam from the image, we will be removing a proportional city surface area.

4.5.2 *Create city image*

In this subsection we explain how to convert the vector-based city representation into a raster image without losing crucial connectivity information.

To create the image corresponding to the *skeleton* city we take a two-step process using two different tools. The first one converts the city from an OSM file (.osm XML format) to a vectorial image file (.svg), and the second turns the vectorial image (.svg) to a raster image (.png). To transform the .osm file to the .svg file, we use Osmarender tool [32] which is a rule-based rendering tool. It takes as input an OSM dataset (.osm file) and a rule file (.xslt file). It outputs a Scalable Vector Graphics (SVG) image (.svg) that is marked up in accordance with the styles defined in the eXtensible Stylesheet Language Transformation (XSLT) file. XSLT is a language for transforming XML documents, that defines, for each kind of XML element, how it should transform in the .svg file. We have basically defined rules to make as dark as possible the important places and lighter the unimportant elements. After getting the vectorial format .svg image, we transform it from vectorial to raster format, getting a .png file. We use a SVG rasterizer. The resulting image for the Firenze center city (with an *shrinking factor* of 0.15) is shown in Figure 38.

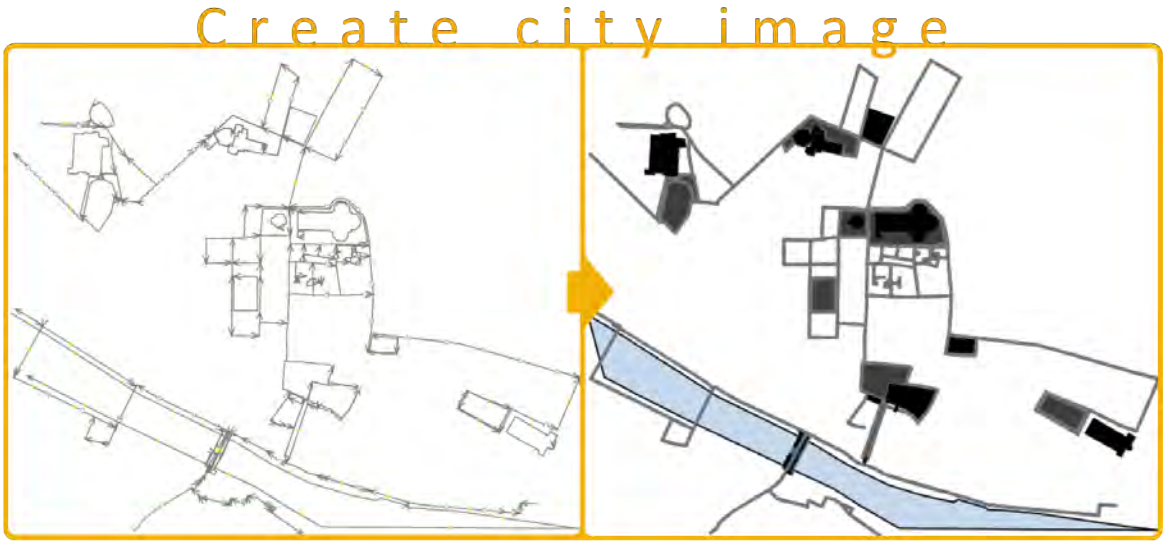

Figure 38: Firenze center: *Skeleton* city transformation from .osm file to a rasterized image in .png file. *Right*: See how the important places are colored in black and empty areas in white. Nearby roads are in darker gray and streets in lighter gray.

4.5.3 *Shrinking pre-process*

We add extra information for each image pixel. This extra data will be used in the Seam Carving process to calculate the energy function and to determine seams to remove. The extra data structure is a matrix of the same size as the image. For each matrix cell (representing a pixel) we store the following six information:

GRAY SCALE VALUE: We translate the city image to a gray scale image and then we invert it. Each pixel stores its gray scale value in the range $[0, 255]$. Because of the inverse operation, black pixels in the original image become white and white ones become black.

DISTANCE FIELD: The pixel's shortest distance to a *skeleton* city element. The algorithm implemented is a standard diffusion algorithm. We normalize distances also in the range $[0, 255]$. As a result, we get a new gray scale image.

PIXEL SKELETON NODES: We store a list of all *skeleton* city node identifiers whose coordinates fall inside the pixel "area".

PIXEL FULL NODES: We store a list of all original city node identifiers whose coordinates fall inside that pixel.

ORIGINAL IMAGE COORDINATES: We also store the coordinates (longitude, latitude) in the original *skeleton* city of the center position of the pixel. This value is important to calculate, after the process, the new position of the pixel, which can be at a smaller latitude and/or longitude if any seam is removed with pixels of lower coordinate values.

FROZEN PIXEL: This is a boolean value that declares that a pixel cannot be removed (it is frozen). Therefore, it cannot be added to a seam path to be removed. The criteria to consider a pixel as frozen is:

- The pixel has a *skeleton* city node of an important non-street element.
- The pixel has a *skeleton* city street intersection node.

4.5.4 *Shrinking Seam Carving process*

We have implemented the Seam Carving algorithm, but with some peculiarities to adapt the energy function to our needs and process the changes of the image in the extra data matrix, to keep coherences between them.

4.5.4.1 *Seam Carving energy function*

As already explained in the beginning of this section, Seam Carving goes repeating the same process of finding a vertical or horizontal seam to remove. This seam is the pixel path to go from top to bottom or from left to right of the image, optimizing an energy function. The original Seam Carving algorithm uses a *Sobel* filter over the image. The *Sobel* filter is an image processing operator that emphasizes the edges and transitions. In our case, we are going to calculate a different energy function because we do not want to consider the pixel gradient; we use a blend between two images, the gray scale and the distance field, and we balance the blending factor to promote one or the other. In this way, we are giving more weight to important places and streets while giving less energy to empty areas (which are the ones without important elements). This is certainly a very specific energy function for our purpose but in practice it turns out to cover our needs. In Figure 39 we can see an example of this process and the resulting energy function image by blending gray scale and distance field images, with a 50/50 balance.

4.5.4.2 *Seam Carving finding seams*

As we can see, we can tune the energy function to make the seam carving algorithm select seams with specific characteristics. By overloading image pixels with the information we want, we can use different data and mix them to get the desired energy function image.

With our energy function, we can find the best seam to remove at each iteration. This path is the one with minimum energy, that is, the minimum sum of pixel values in the energy function image. This process can be done via Dijkstra's algorithm, graph cuts or dynamic programming. Our implementation uses dynamic programming with one special modification: When computing the energy value of a pixel with the frozen value, it sums an infinite value. That way, the algorithm avoids to get any seam passing through these pixels. Dynamic programming

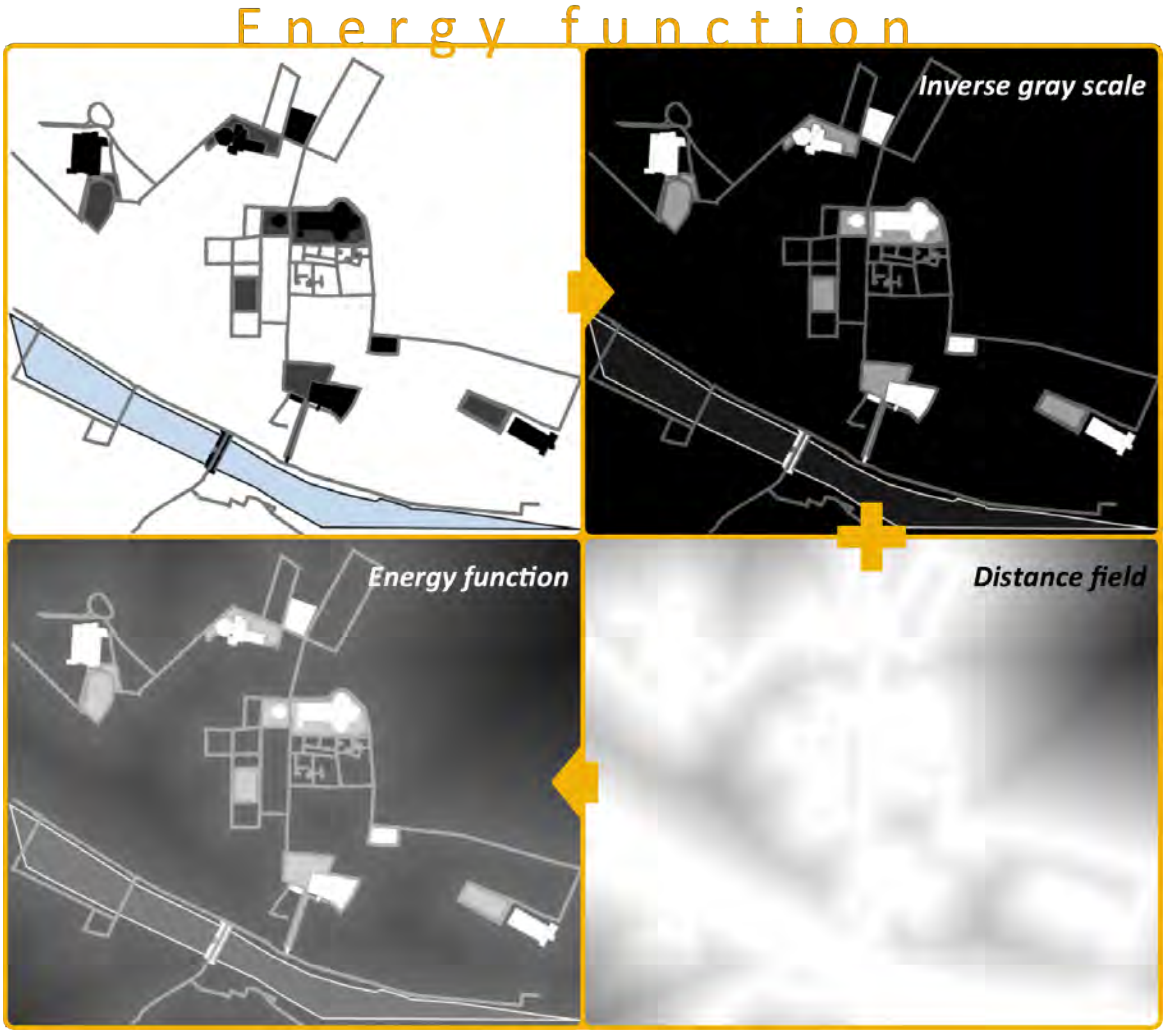

Figure 39: Firenze center: The *skeleton* city Seam Carving energy function. *Top-Left*: Raster image of the original *importance skeleton* city. *Top-Right*: The inverse gray scale image. *Bottom-Right*: The distance field got from the gray scale image. *Bottom-Left*: By blending gray scale and distance field images, we get the final energy function used to find seams.

stores the result of the sub-calculations in order to simplify calculating more complex results. For example, if attempting to compute a vertical seam path, for each pixel in a row we compute the energy of the current pixel plus the energy of one of the three possible pixels above it to obtain the one with minimum accumulated energy. See Figure 40, with in black number the pixel's energy value, and in red the accumulated energy. When arriving to the bottom of the image, the optimal seam is the path ending in the last row pixel with the lowest accumulated energy. We have used a naive implementation with a computational cost of $O(m \times n)$, m and n being the image size. To get the final result we sum this cost for each of the seams S to be found, s_m and s_n being the number of vertical and horizontal:

$$O(m \times n) + O(m-1 \times n) + ... + O(m-i \times n-j) + ... + O(m-s_m \times n-s_n)$$

Hence, the order of this step is: $O(m \times n \times S)$

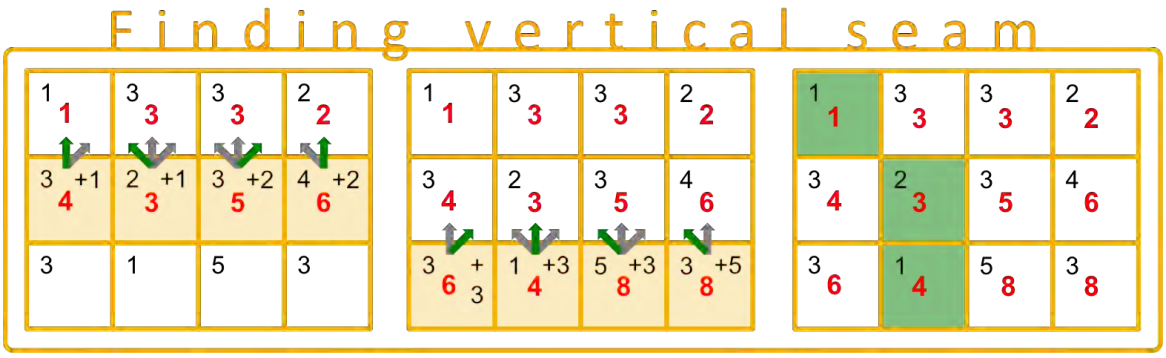

Figure 40: Seam Carving finding vertical seam using dynamic programming. For each pixel, the number in black is its energy value and in red the lowest accumulated energy to arrive to it. *Left*: First row, the pixel accumulated value is the same as the pixel value. The second row gets the lowest accumulated energy, considering their three (or two, along the image borders) previous neighbors. *Middle*: Processing the last row. *Right*: In green the lowest energy seam path.

In Figure 41, we see in red the set of seams to be removed over the *skeleton* city image. Note that seams run over darker pixels in the left image, which is the energy function image.

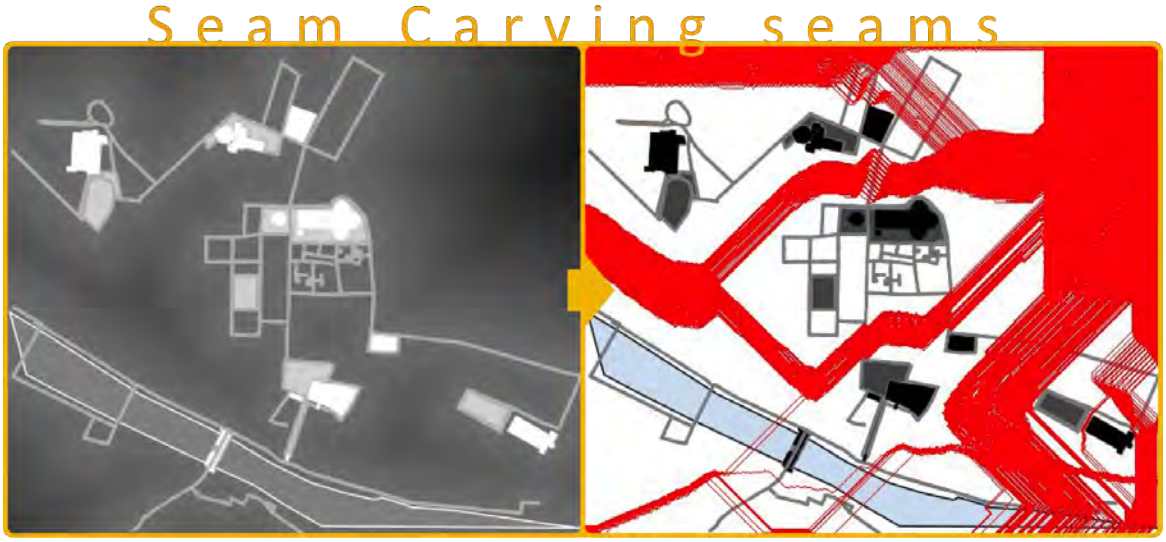

Figure 41: Firenze center: Seam Carving seams to be removed from the *skeleton* city image. *Left*: The energy function of Firenze's center *importance skeleton* city image. *Right*: The Firenze center *importance skeleton* city image with all seams that can be removed in red. This case shows 100 vertical and 100 horizontal seams.

4.5.4.3 *Seam Carving removing seams*

After finding each seam, we remove it from the image. The user can decide how many seams to remove, in both vertical and horizontal directions. Hence, the designer can get as a result a city with required aspect ratio. To remove a seam means to delete all the pixels that form the seam path, and to move all the pixels by one pixel, on the right of the seam, when a vertical seam is removed. Similarly for the upper pixels when the removed seam is a horizontal one. When moving all those pixels to occupy the seam pixel space, we also move the extra data matrix cells correspondingly, such that we continue having coherent information of pixel color and extra data.

The resulting raster image is the shrunk image. We can appreciate the city image reduction in Figure 42 between the original area in the left column and a shrunk area in the right column. Note how all the important places are kept and they are closer together, while keeping their relative positions (Figure 42 *Bottom*, green arrows). However, this image does not look entirely correct, because those streets crossed by many seams have been cut, resulting in inconsistent orientations. In Figure 42 *Top-right* and *Bottom-right*, red circles show this case. This will not be a problem when translating this result to the *City* structure because of its vectorial nature. A street segment in the raster image is a set of pixels drawing the line between the two end points, while in the vectorial space it is represented by the two end points and nothing else in between.

4.5.5 *Shrinking post-process*

After running the Seam Carving operator over the raster city image, we have to translate this result back to vectorial space, getting a new *City* structure, the final *shrunk* city.

During the Seam Carving process, when removing the seam, we update accordingly the extra data matrix. So we have now an original *skeleton* city image and a shrunk *skeleton* city image, but also an original-size extra data matrix and a shrunk-size extra data matrix. This is the extra data explained in Section 4.5.3.

Obviously, in the shrunk extra data matrix, the pixel coordinate values are wrong: Any pixel moved by a seam passing through a lower longitude or latitude keeps its original pixel position coordinates. On the other hand, the pixels containing city nodes that have been moved have the *City* node identifier, but its coordinates should be updated: The ones defined in the original *skeleton* city are not appropriate any more. To provide new coordinates to all *shrunk* city nodes, we process all the nodes that extra data cell has. In this process, we decrease the original node coordinates by the number of pixels it has been moved times a pixel side length. Therefore, considering $P = (i, j)$ the pixel with at least one node matrix position,

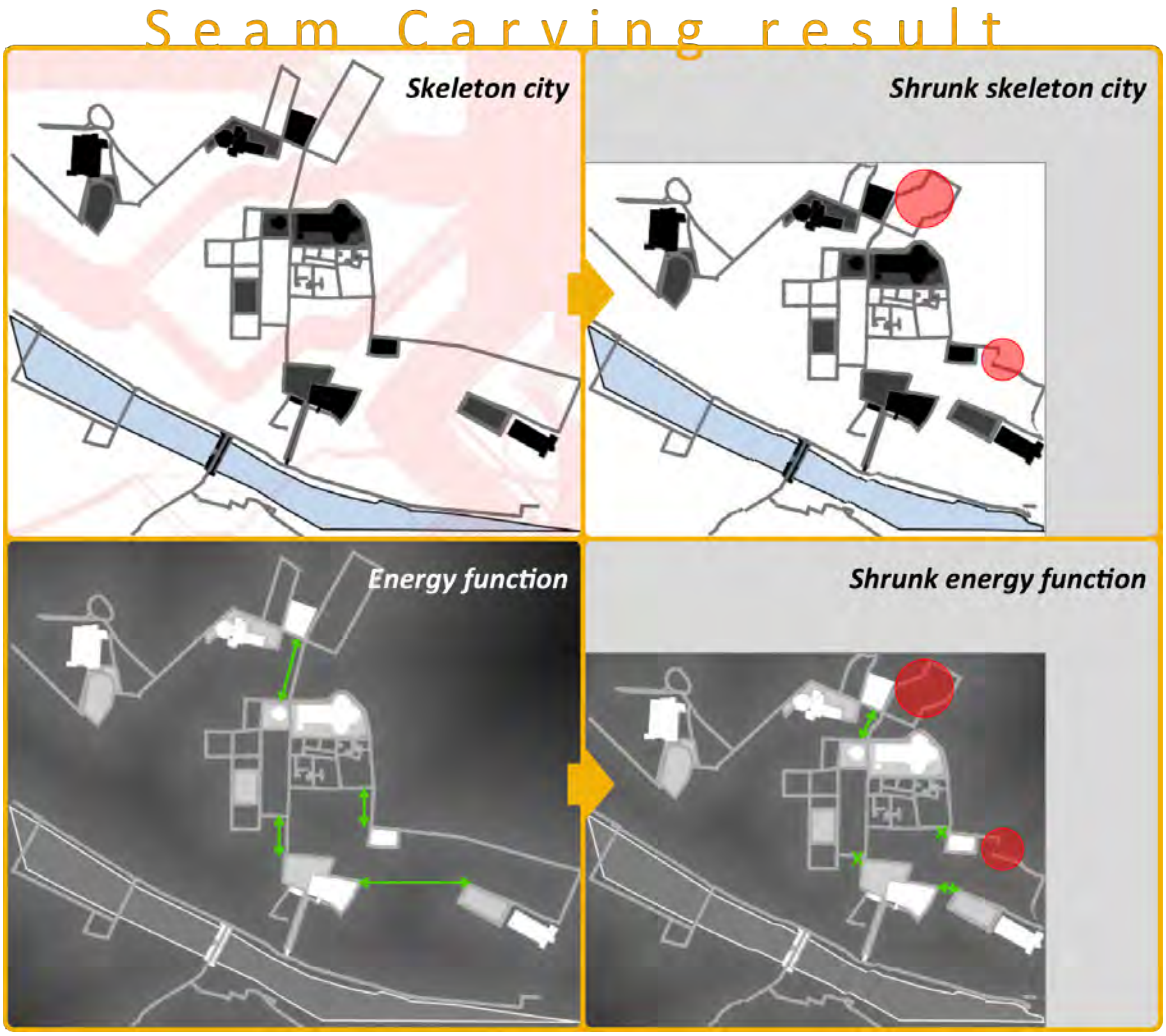

Figure 42: Firenze center: *Skeleton* image shrinking by Seam Carving reduction. *Top*: *Importance skeleton* city images. *Bottom*: The respective energy functions with green arrows to indicate the shortened distances between important places. *Left*: The original size of the *skeleton* city image. *Right*: The shrunk *skeleton* city image after removing 100 vertical and 100 horizontal seams with disturbing streets in red circles. *Top-left* image has seams removed in the background to facilitate the understanding of the resulting *top-right* image.

$P' = (i', j')$ the same pixel after the Seam Carving process changed its position, c_{size} the matrix cell side size in real word measures, $N = (x, y)$ the original city node coordinates and $N' = (x', y')$ the same node in *shrunk* city, we can relate them by:

$$x' = x - c_{size} \cdot (i' - i)$$

$$y' = y - c_{size} \cdot (j' - j)$$

Note that some nodes can be removed during the Seam Carving process (but not intersection street nodes). These are not frozen pixels with nodes in them. This leaves the possibility of experiencing some street deformation. Removing one

street node can make the street lose curvature, so this could result in a simplification of its shape. This effect is undesired, because in fact we are getting a simplified city. Anyhow, it can easily be corrected by changing all the street nodes to be frozen pixels.

With the new node coordinates we have to create the new *shrunk* city. It adds all the nodes with new coordinates (x', y'). If a node has no new coordinates it means that the node has been removed. In this case, we update all the ways using this node, removing the node from the node list. See the resulting *shrunk* city in Figure 43. Note that those streets deformed and unconnected in the raster image become connected and stretched because of the vectorial nature of the *City* structure.

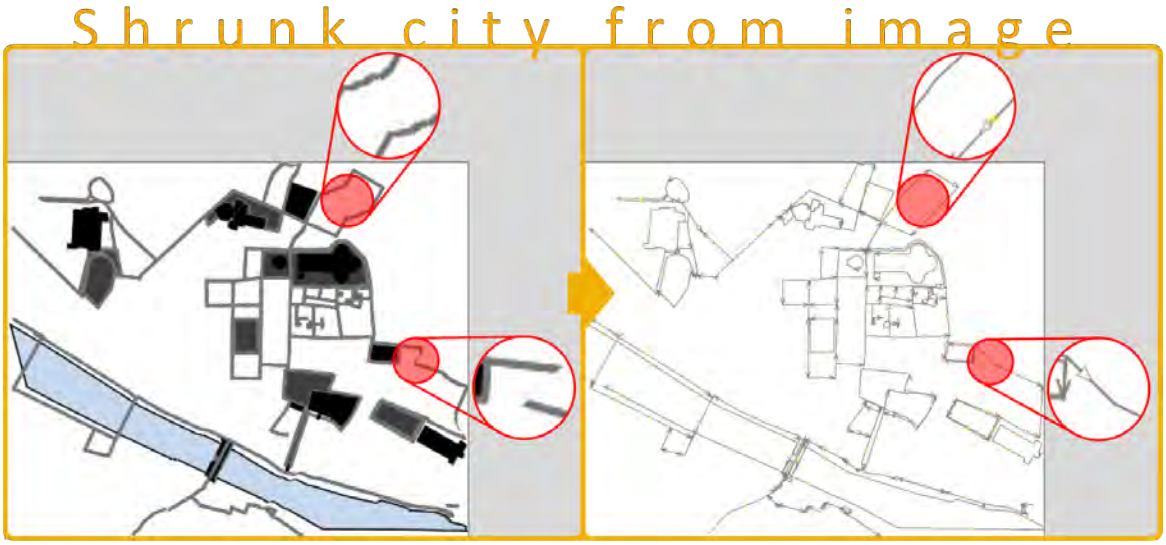

Figure 43: Firenze center: Shrunk image transformation to a *shrunk* city street map. *Left*: The raster image after running the Seam Carving process. *Right*: The street map of the *City* model. Zoomed-in areas show the change from raster to vectorial space, resulting in a satisfying vectorial street map.

Nonetheless, depending on the structure of the city map, the non-intersecting nodes in a street, and because of the Seam Carving nature, even after the conversion of the raster image to *City* structure, we can get streets with undesired extra curvature. See as an example the resulting *shrunk* Barcelona in Figure 44 *Middle*. Many streets have sinuosities that were not there in the original map. Two examples are marked with red circles in the images. In these cases, we run a stretching process. We stretch all the street segments that were straight in the original city but not in the *shrunk* city. We keep the streets having curvature on the original city without experiencing any stretching process. The resulting city map after the stretching process has a better appearance than the one after the Seam Carving process, although it has lost rectangular-shaped intersections, which is an important feature of the Barcelona Eixample. Even so, this characteristic can be recov-

ered by designer editing (see Figure 50) or by applying the process modification proposed in Section 4.6.1 (see Figure 51).

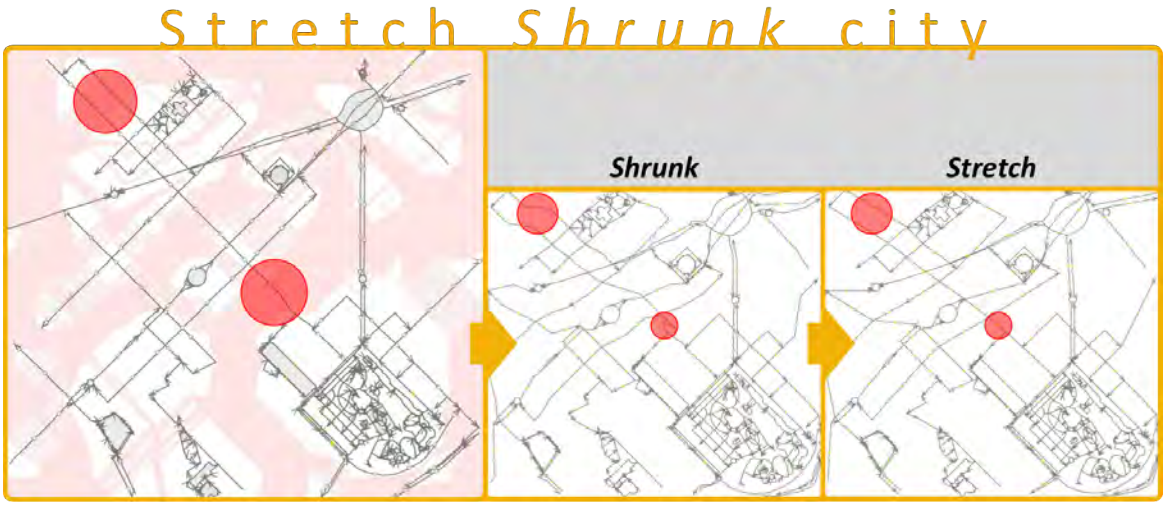

Figure 44: Barcelona: *Shrunk* city and stretching map. *Left*: The *skeleton* Barcelona Eixample map with seams to remove (150 × 150) in the background. *Middle*: The *shrunk* city map. See the undesired sinuosity of the streets in the red circles. *Right*: The *shrunk* city after the stretching process.

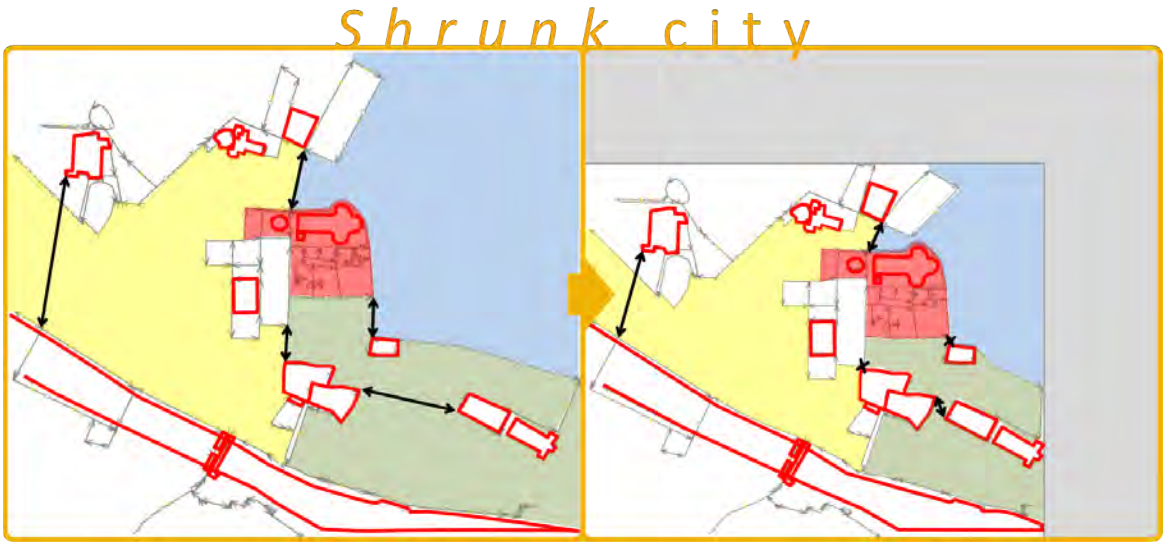

Figure 45: Firenze center: Shrinking the *Skeleton* map. *Left*: The original size Firenze center *importance skeleton* city, with important places in red. Note the colored (yellow, blue and green) areas without important places in them, and the black arrows for some distances between important places. *Right*: The *shrunk* city. Note how unimportant areas are smaller, and important places are closer together and mostly maintained their relative positions.

Summarize *shrunk* city:
The resulting *shrunk* city is smaller in comparison with the original size *skeleton* city. Nevertheless, it brings closer together the important places and with their sizes and shapes untouched (see Figure 45). We have been illustrating step-by-step the example of the Firenze center map. If we run the shrinking process to this city map with a shrinking factor $shF = 0.15$ and removing 100 vertical and 100 horizontal seams, we get the resulting *shrunk* city shown in Figure 45. The *shrunk* city is a city with its unimportant areas decreased and distances between important places reduced (see yellow, blue and green colored areas and black arrows in Figure 45). In the same image the important places are colored in red.

4.6 RESULTS

We have implemented a pipeline of city operations that takes an original city map with a set of important places and streets, and converts them to a smaller city with a similar distribution, retaining its essence. After the shrinking process, the designer gets the *shrunk* city. This city has only the elements (streets and other places) selected to be on the *skeleton* city. This final city is a generalization of the original city with smaller surface areas, keeping its important places closer together. From this proposal of a smaller city, the designer can start editing it to get the desired final city (see Figure 46).

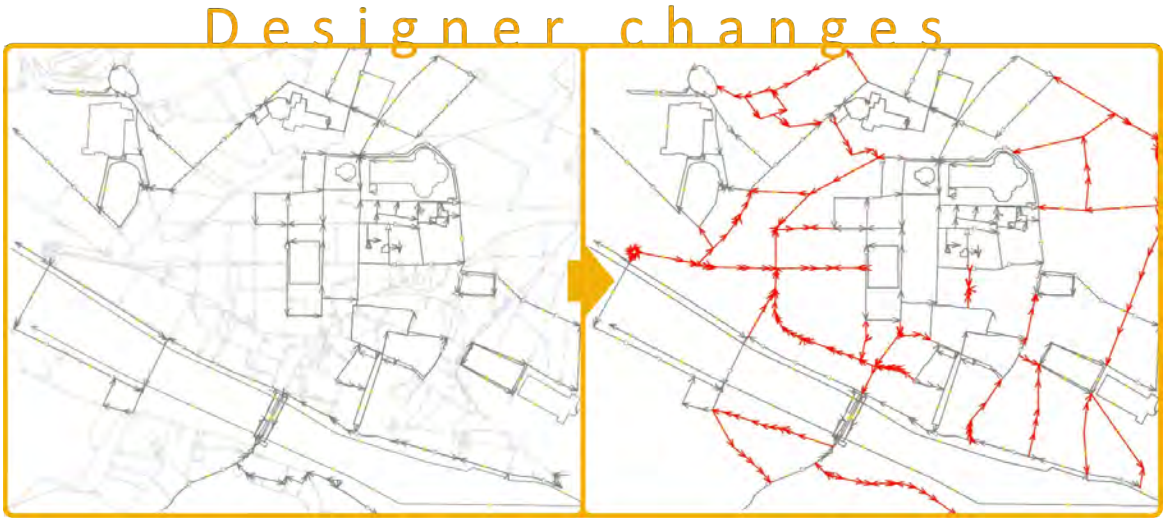

Figure 46: Firenze center: The *Shrunk* city designer edition. *Left*: *Shrunk* city and all full city streets experiencing the same transformation due to the shrinking process, in lighter gray. *Right*: The red streets are streets that a designer decided to add or modify. The added ones are inspired by full city streets.

The designer can then add new streets, remove undesired ones (or street segments) or edit a street by changing its shape. The streets added can be based on the original streets not included in the *skeleton* city, so we provide them to the designer as can be seen in the left image of Figure 46, in lighter gray. Note that all streets experience the same transformation due to the shrinking process with the *skeleton* city energy function. The streets already included in the *shrunk* city, can also be edited to change their shapes. See an example of this modification on the center bottom street of the figure. In the left image, the street has a shape like a stair while in the right it becomes smoother, in red.

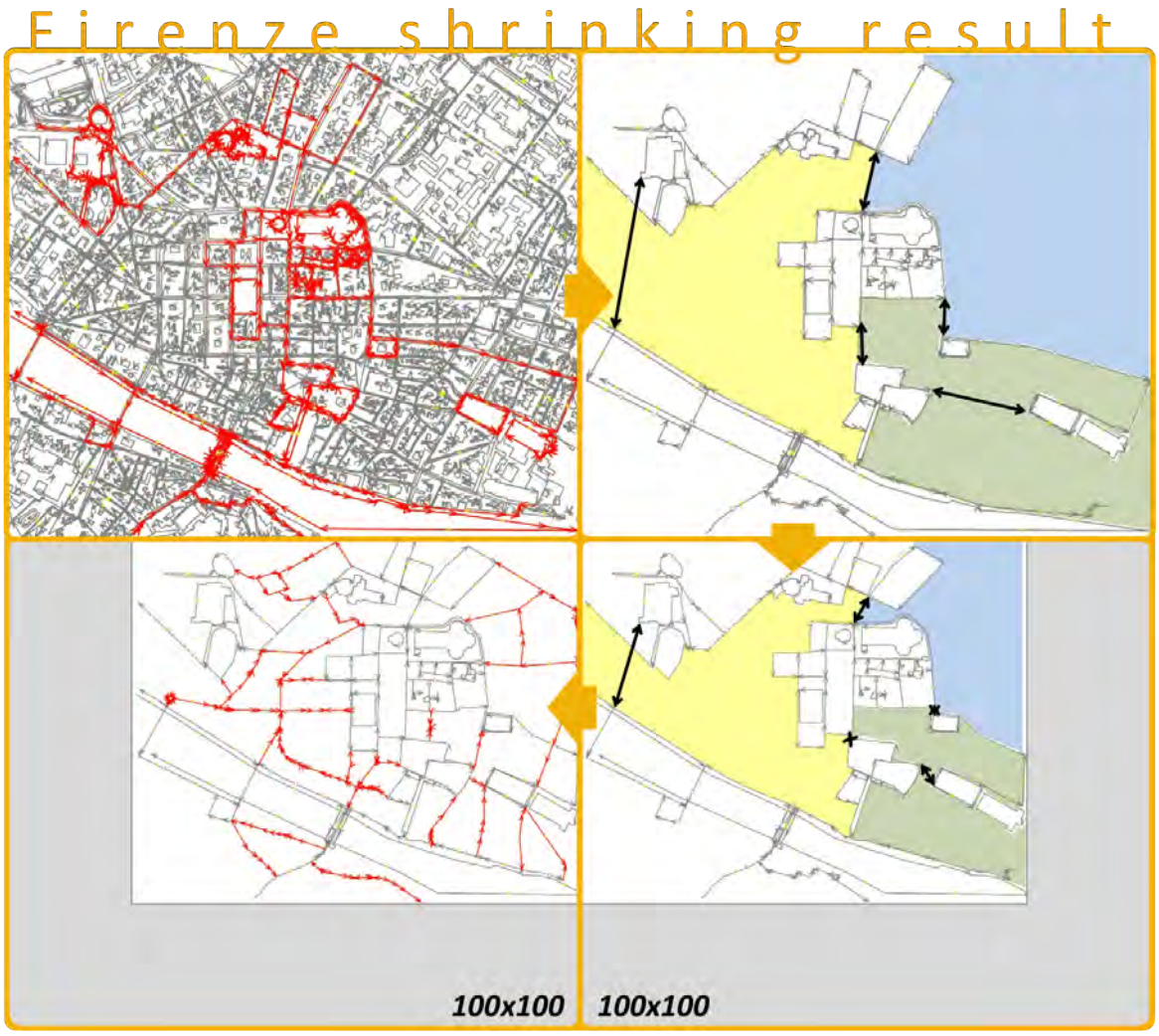

Figure 47: Firenze center: Shrinking process final result. *Top-left*: The original Firenze center *importance skeleton* city, with important places in red. *Top-right*: The colored neighborhood areas (yellow, blue and green) representing areas with no important places within them. *Bottom-right*: The *shrunk* city, after applying the shrinking process. Note how unimportant areas are smaller and important places are closer together while respecting their relative positions.

Figure 47 summarizes our shrinking process. Note how the resulting city has a smaller area and maintain the most important places but placing them in closer positions. Figure 47 *Top-left* is the full original city with the *skeleton* selected elements highlighted in red. And the *Bottom-left* the final city after being shrunk with the designer's modifications in red. The less important area of the city has been removed. In Figure 47 *Top-right* and *Bottom-right*, we can see the unimportant areas (colored in yellow, blue and green) and how they have been reduced in the bottom one. By doing this shrinking process, important places become closer together, like it is illustrated by the shortened black arrows in Figure 47 *Bottom-right*.

This process can be parameterized by the user with different values if the default ones do not result in a satisfying *shrunk* city. We show below the list of these tunable parameters (sorted in the order of their use in the pipeline process):

PROXIMITY STREET THRESHOLD: A real value to limit the maximum distance that a street could be from an important place to be added as an indirect important street.
Default: $\mathtt{proxStrThreshold} = 0.00033$

STREET WEIGHTS: A dictionary for each kind of city way importance weight.
Default: $\mathtt{highwayWeights_d} = \{\mathtt{motorway} = 3.,\ \mathtt{primary} = 2.5,\ \mathtt{secondary} = 2.0,\ \mathtt{tertiary} = 1.5,\ \mathtt{residential} = 1.0\}$

SHRINKING FACTOR: A real value $\mathtt{shF} \in [0.0, 1.0]$ which specifies the fraction of streets to keep when creating the original *importance skeleton* city. It is the most important parameter in our implementation.
Default: $\mathtt{shF} = 0.2$

IMPORTANCE | DISTANCE BALANCE: A real value $\mathtt{impDistBalance} \in [0.0, 1.0]$. It specifies the weight that has the distance or the street importance on the $A^\star$ algorithm to make the city connected. A value of 0.0 means just importance criteria, while 1.0 means the whole weight applied to the distance criterion.
Default: $\mathtt{impDistBalance} = 0.5$

REMOVE UNIMPORTANT CLUSTERS: A boolean to decide if the algorithm should keep unconnected clusters with no specified important places or remove them. This is processed when creating the *skeleton* city.
Default: $\mathtt{noImpClusters} = \mathtt{True}$

CONNECTING DEGREE: An integer value to define the maximum number of way segments to use when trying to connect dead-end nodes.
Default: $\mathtt{m-degree} = 3$

IMAGE SIZE: An integer to specify the maximum size, in pixels, of the image to run the Seam Carving algorithm on (it will be used as height or width depending on the aspect ratio of the *City* bounding box).
Default: $\mathtt{imageSize} = 500$

GRAYSCALE | DISTANCEFIELD BALANCE: A real value to specify the image blending balance $\texttt{grayDistBalance} \in [0.0, 1.0]$ between the gray scale pixel values (important elements) and the distance field pixel values (distance to important elements).
Default: $\texttt{grayDistBalance} = 0.5$

SEAMS TO REMOVE: A two-integer tuple to specify the number of horizontal and vertical seams to remove. Considering the image size and the seams to remove, we can easily define the area of the reduction of the city area. (These values would also be possible to be defined height and width fractions.)
Default: $\texttt{imageSizeReduction} = (100, 100)$

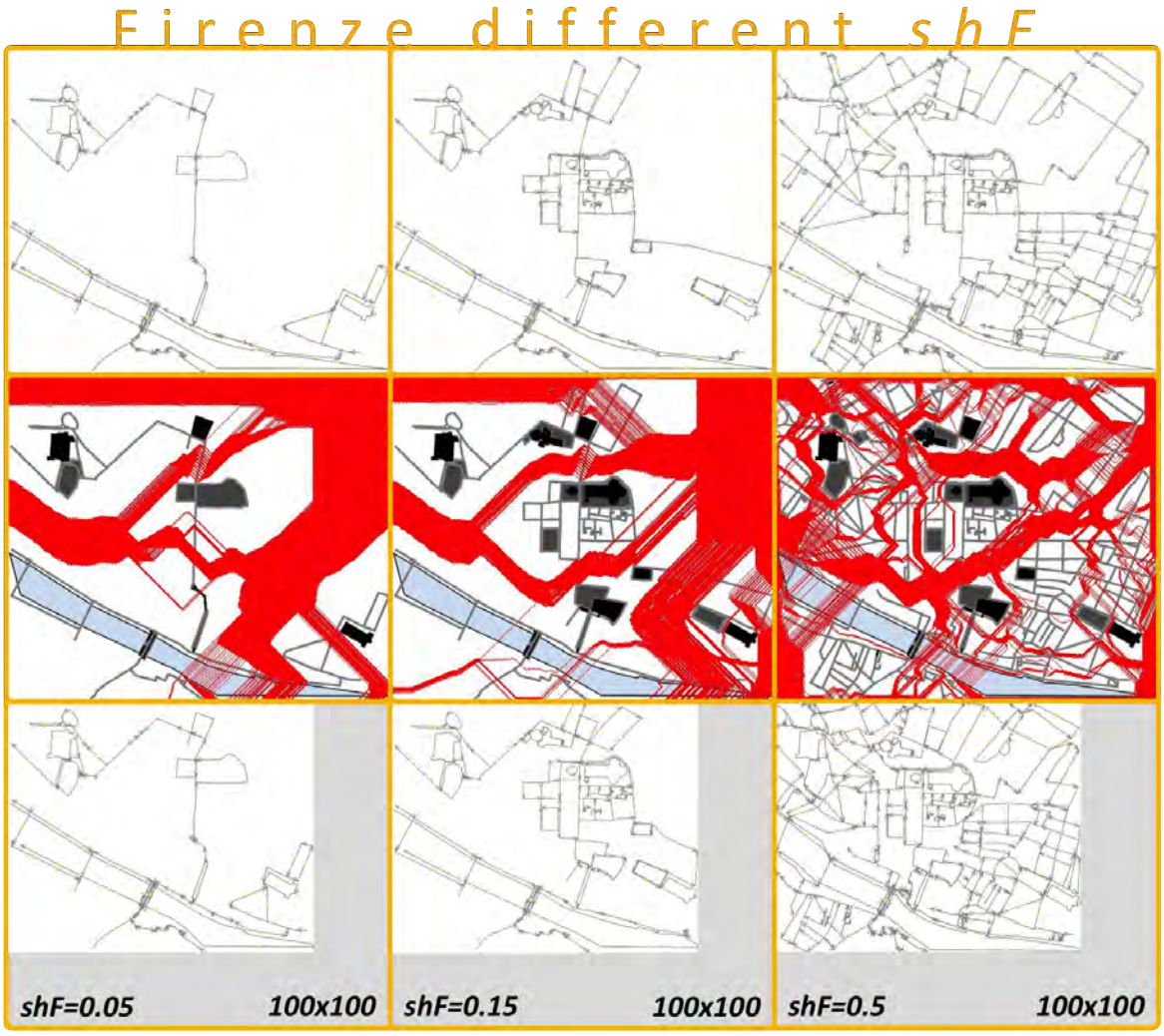

Figure 48: Firenze center: Shrinking results by changing *shF* (shrinking factor). From left to right, using smaller to larger numbers of streets. *Top*: Original size *skeleton* map. *Middle*:The 100 × 100 seams removed. *Bottom*: *Skeleton* shrunk map.

In Figure 48, we can see different *shrunk* cities by changing the shrinking factor ($shF = 0.05$, $shF = 0.15$ and $shF = 0.5$) and keeping the same number of seams to remove (100×100). In Figure 49, we can see the different *shrunk* cities by changing the number of seams to remove ($\#seams = 50 \times 50$, $\#seams = 100 \times 100$ and $\#seams = 200 \times 150$) and keeping a fixed shrinking factor $shF = 0.15$.

Figure 49: Firenze center: Shrinking results by changing the number of seams to remove. From left to right, removing fewer to more seams. Hence the resulting *shrunk* city area decreases from left to right. Note how the more seams we remove, the more deformed the streets become.

Another example in Figure 50, changing the city processed, shows the Barcelona Eixample neighborhood map. In Figure 50 *Top-left*, the important places are highlighted: In blue, interesting places like *Sagrada Família*, *Plaça Catalunya*, *Catedral de Barcelona*, *Torre Agbar*, *Passeig Lluís Companys* and bullring *La Monumental*. In green the *Parc de la Ciutadella* and *Zoològic* area. And finally, in red, important streets and squares like *Rambla de Catalunya*, *Gran Via de les Corts Catalanes*, *Diagonal*, *Meridiana* or *Plaça de les Glòries*. In *Top-right* image we can see the seams removed to shrink the Barcelona map. The Figure 50 bottom images are the shrunk city map before and after the designer's work. In the *Bottom-left*, we can see some originally straight streets with their resulting curvatures, which makes the shrunk city lose the characteristic regular blocks of the Barcelona Eixample neighborhood. We discuss this issue in the following paragraphs. The designer has the responsibility to recover this city feature by moving the blocks and adding new streets, like in Figure 50 *Bottom-left*.

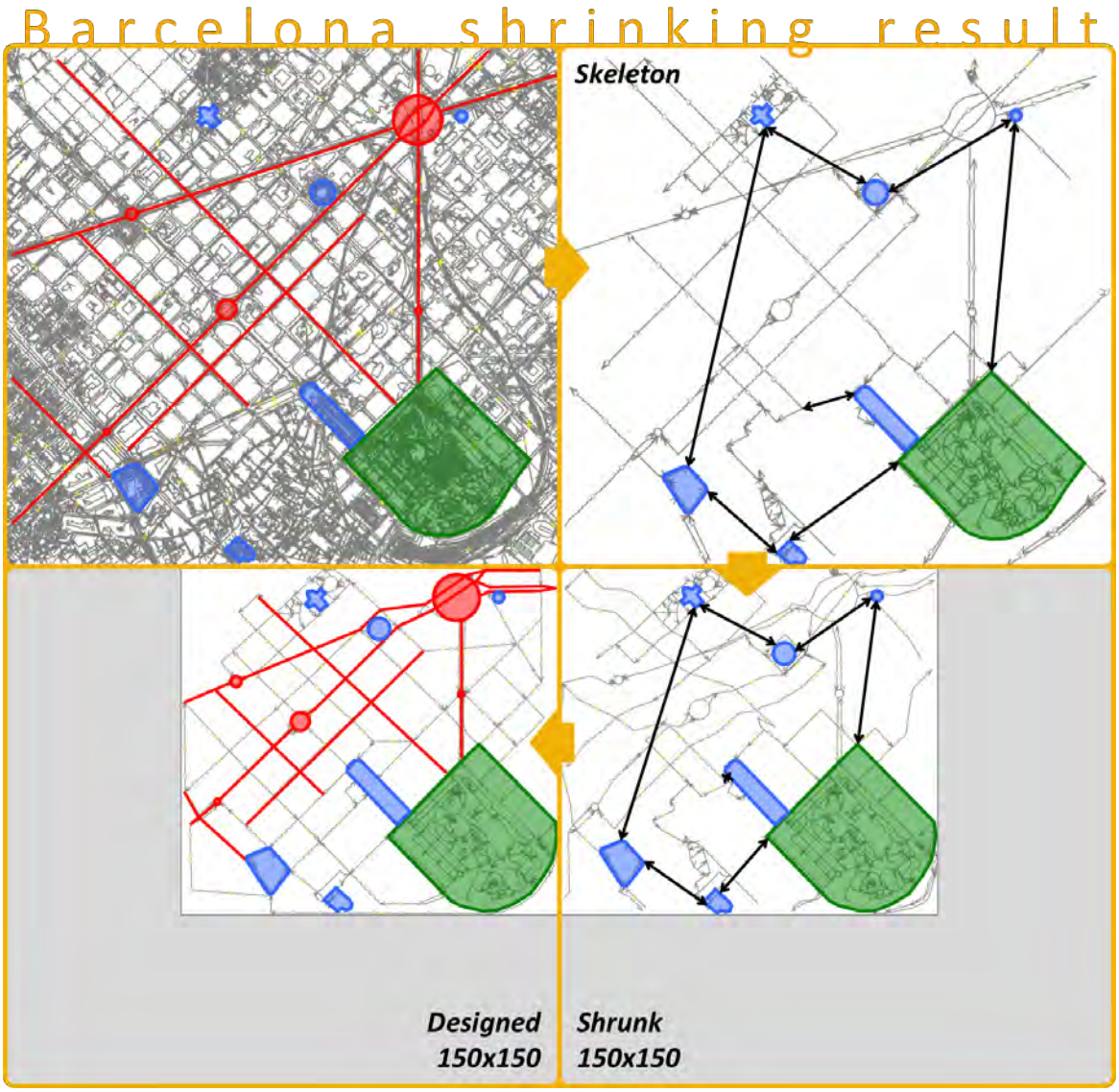

Figure 50: Barcelona Eixample: Shrinking process final result. *Top-left*: The original city with important places highlighted like *Sagrada Família*, *Plaça Catalunya* or *Catedral de Barcelona* in blue; the *Parc de la Ciutadella* and *Zoològic* area in green; and some of the most important streets and squares like *Rambla de Catalunya* or *Diagonal* in red. *Top-right*: The *skeleton* with black arrows to compare distances between important places. *Bottom-right*: The *shrunk* city removing 150×150 seams shown with shorter black arrows. *Bottom-left*: The designed *shrunk* city after the stretching process and the designer's work.

The timings of the overall process are in Table 2. Note that our Seam Carving implementation is rather naive. The computational cost of the operator to find a seam is linear $O(N)$ in the original paper [7], and the cost is also linear with the number of seams to process $O(S)$ for S seams to process. We have an implementation of order $O(N \times S)$. Therefore, our achieved timings are of a few minutes to run out Seam Carving (depending on the number of seams to run), should reduce

to just a few seconds. Avidan and Shamir[7] report that an image of 500×400 pixels (size used for our examples) can be reduced to a 100×100 pixels in 2.2 seconds. The reason why such a big difference between their timings and ours is that we have implemented the dynamic programming algorithm in a naive way and using Python, which is an interpreted programming language (always slower than a compiled language).

In Table 2 we can see the time spent in each shrinking step. These numbers have been obtained with different parameters and for different cities. The times are in seconds and the executions ran on a Mac BookPro 2011 with OS X 10.10 and processor Intel Core i7 2.66GHz. All the code is run on the CPU, no GPU is used in our implementation.

The first and main conclusion we can draw from the table is that the most expensive step is the shrinking process itself, i.e., the sum of columns *step8Pre*, *step8SC* and *step8Pos*. To get the *skeleton* city from the original city is quite fast, around 20 seconds. The cost of the shrinking process is divided between the time of processing the extra data matrix and running the Seam Carving algorithm. The fast timings on the city operations are thanks to the *City* library structure and tools. See this library details in Sections 5.1 and 5.2. The next conclusion to make is that, when finding the indirect important streets, those streets near to a non-street important place, require a longer time if the important places have more nodes and a higher density of nodes in that area of the city. See in Table 2 how Barcelona executions take much more time than Firenze executions in *step1*. Also, when the first selection of streets to add to the *skeleton* city has more unconnected clusters, the connecting clusters step takes more time. The case of Barcelona takes more time to connect them than the Firenze case, see highlighted cells in *step3&4*. Another nice conclusion is that, for higher *shF* values, the time to create the *skeleton* city image is higher, while the time to process the extra data matrix is smaller. The reason for these two facts is basically that the *skeleton* city has more elements selected as important. In one case, to create the *skeleton* image, the increasing time is because the tool that translates from .osm file to .svg file needs to process more elements. In the other case, to process the extra data matrix, the decreasing time is because, when processing the distance field with the flood fill algorithm from each pixel, the distance to the nearest city element is smaller. We ran a flood fill algorithm from each pixel, visiting pixel neighbors while searching for an important element. Because of the higher density of important elements, all pixels stop their search before because they find an important place earlier. See *step7* and *step8Pre* columns for Firenze with $shF = 0.5$ in comparison with Firenze with $shF = 0.05$ and $shF = 0.15$. The last conclusion we want to draw is that the running time of the Seam Carving is longer when removing more seams, which is logical. Note how, when increasing the number of seams to remove, the time also increases in the highlighted cells of *step8SC* column.

Steps definition	
step1	Add indirect important streets 4.4.1
step2	Select important streets 4.4.2
step3&4	Connect clusters (and remove no important ones) 4.4.3
step5	Connect fringe paths 4.4.4
step6	Add non-street important places to *skeleton* city 4.4.5
step7	Create *skeleton* city image 4.5.2
step8Pre	Shrinking pre-process, extra data matrix 4.5.3
step8SC	Shrinking Seam Carving execution 4.5.4
step8Pos	Shrinking post-process, new *shrunk* city coordinates 4.5.5

step1	step2	step3&4	step5	step6	step7	step8Pre	step8SC	step8Pos
Firenze center ($shF = 0.15$, $\#seams = 50 \times 50$)**:**								
15.8	0.2	3.4	1.2	0.0	4.6	304.5	201.1	0.6
Firenze center ($shF = 0.15$, $\#seams = 100 \times 100$)**:**								
16.5	0.2	3.3	1.2	0.0	4.5	295.6	359.4	0.7
Firenze center ($shF = 0.15$, $\#seams = 150 \times 150$)**:**								
15.3	0.2	3.2	1.1	0.0	4.4	301.0	448.6	0.6
Firenze center ($shF = 0.15$, $\#seams = 200 \times 150$)**:**								
15.9	0.2	3.6	1.3	0.0	4.4	290.3	489.9	0.4
Firenze center ($shF = 0.05$, $\#seams = 100 \times 100$)**:**								
15.9	0.1	3.6	1.1	0.0	3.1	331.4	342.0	0.5
Firenze center ($shF = 0.15$, $\#seams = 100 \times 100$)**:**								
16.5	0.2	3.3	1.2	0.0	4.5	295.6	359.4	0.7
Firenze center ($shF = 0.5$, $\#seams = 100 \times 100$)**:**								
15.4	0.3	7.7	1.8	0.0	7.1	73.2	355.8	1.0
Barcelona Eixample ($shF = 0.2$, $\#seams = 50 \times 50$)**:**								
61.2	0.7	25.9	2.5	0.0	8.2	151.3	231.0	1.5
Barcelona Eixample ($shF = 0.2$, $\#seams = 150 \times 150$)**:**								
63.3	0.6	26.6	2.5	0.0	8.2	157.8	507.9	1.1
Barcelona Eixample ($shF = 0.2$, $\#seams = 200 \times 200$)**:**								
63.0	0.7	26.0	2.4	0.0	8.2	151.1	590.6	0.8

Table 2: Shrinking process step timings in seconds.

4.6.1 *Shrinking process discussion*

We want to organize our discussion around five different topics: the freedom of the designer to get different results from the same original city, the possible generalization of the shrinking criteria, the behavior of the process for cities with very regular street network, and the addition of unimportant streets on the *shrunk* city large empty areas.

- **Designer's freedom:**
  We give the designer the opportunity of achieving the desired final city in two ways. First of all, he can tune the parameters to get the result as similar as the expected final city (*shF*, number of seams to remove, connecting fringe paths pass degree, etc.). But we also provide the unimportant streets –shrunk too– to fill large areas without any street, a part from the option of modifying and removing streets from the resulting process, as shown in Figure 46.

- **Seam Carving generalization:**
  We explained in Section 4.5.4.1 how we calculate the energy function to run the Seam Carving operator over the city. This energy function can be implemented in many ways, adjusting the energy of the pixels according to each case. Moreover, we have added an extra data matrix that can be used to get as much information as desired for each pixel. Hence, we can think of any energy function for different purposes and shrink the city for any criteria we can imagine. In this way, we can control more our algorithm to froze all street nodes (not only the intersections) or all the surfaces of the important areas (instead of only the elements inside it). Even more, we can define any new specialized criteria for resizing the city by considering, for example, to keep only the most significant commercial streets, the most populated districts, etc. Also, we can consider that the translation from vectorial space to raster space can be generalized. Any kind of graph or data can be mapped over the raster image. Then, giving a desired information to each pixel can create a new specific energy function to run the Seam Carving on.

- **Strong pattern street networks:**
  As seen in the Barcelona Eixample results, cities with strong street patterns can lose a bit of their shapes even after the stretching process. Examples can be squared blocks (perpendicular intersections), radial and circular streets, etc. In these cases, the nature of the vertical and horizontal seams of the Seam Carving operator can give undesired results because street intersections can change angles. For the squared block pattern, this issue is specially problematic when the streets have a diagonal orientation with respect to the longitude and latitude axes. A way to solve it is to rotate the whole *skeleton* city before running Seam Carving, to have the more streets as possible in vertical and horizontal directions, run the operator and then rotate again the resulting *shrunk* city. See in Figure 51 how the straight streets (in green) re-

main straight after shrinking when they are parallel to longitude or latitude axis, while the streets at 45° have an undesired curvature (red circles).

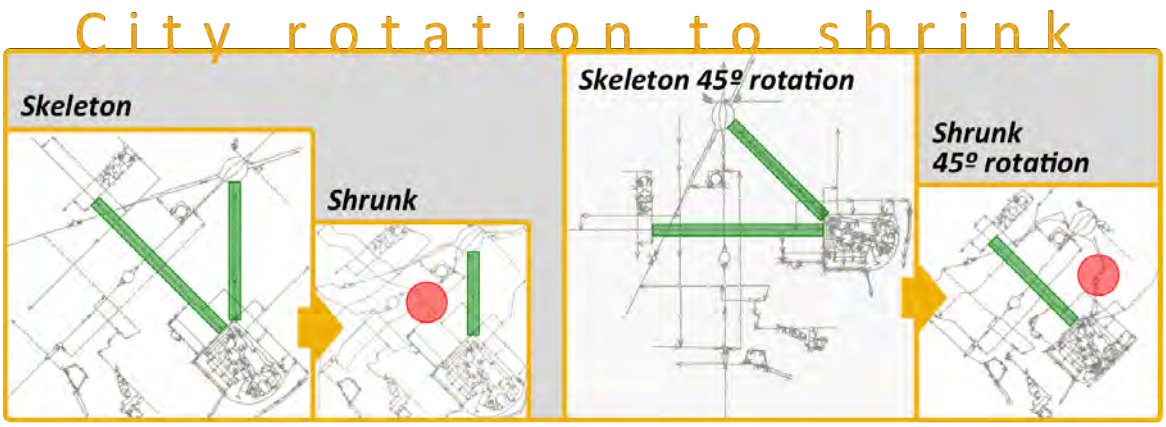

Figure 51: Barcelona Eixample: Shrinking after a 45° rotation. *Left*: The *skeleton* and *shrunk* city maps keeping node coordinates. *Right*: The *skeleton* city after a rotation of 45° to all the nodes, and the resulting *shrunk* city rotated back to its original coordinates, −45°. Red circles show straight streets being deformed because of the Seam Carving.

- **Filling unimportant empty areas:**
  The resulting *shrunk* city leaves large areas without important places. We can consider there areas as big blocks or small neighborhoods. This is because we are shrinking the *skeleton* city with only important places and streets in mind. Filling these areas is mainly a work for the designer even though this is routine work. We provide them with all the shrunk city streets, and they can select the ones to be added and modified. For this, we introduced two approaches to get the shrunk unimportant streets. The first idea was using a cage-based deformation algorithm (Mean Value Coordinates) [30]. For this, we find the cages with the *skeleton* city streets and their bounding box minimal closed circuits. Then, for all those cages that were too large, we get the same cage on the *shrunk* city. Having the transformation of the cage, we apply the same transformation to the inner nodes. This first approach did not give us the expected results because cages and Seam Carving are different deformation systems. Some street nodes fell out of the *shrunk* cage. Seeing that this was not a good idea we decided to apply exactly the same Seam Carving deformation to the whole full streets map. All node coordinates are added to the extra data matrix explained in Section 4.5.3. The nodes of the unimportant streets are not frozen, so they can be removed if a seam goes thought them. This changed its shape, but if we decided previously to consider this street as an unimportant one, its shape is not transcendental to the resulting city. Note that we did not run a Seam Carving over the energy function of the full map, we only ranit with the *skeleton* city map energy function, and the seams removed in this process are applied for both, *skeleton* and full

map node coordinates. Trying to fill automatically these areas is not an easy task, but it would be a nice future work. We point out how to explore this issue in the future work in Section 7.2.

- **Raster image operator in vectorial urban graph:**
  It might be strange to turn from a graph vectorial space to an image raster space to run the Seam Carving, and turn back to the graph again after the shrinking process is finished. But this change of space is to take benefit of the Seam Carving algorithm, that provides us a good way to do the guided resizing of the city that interest us. Combining both spaces is also done by Emilien et al. [25] to take profit of the Seam Carving algorithm. The way of changing from one space to the other is explained in depth in Section 4.5.3. Aliaga et al. [4], explicitly say that Avidan and Shamir image-processing algorithms (Seam Carving), enables image resizing and retargeting. But this algorithm is not suitable for images with highly structured information. They also argue that it is not an appropriate algorithm for urban layouts because they consist of building contours, road networks, and local details that can not be treated the same way as a patch of grass or sand. Important boundaries, angular relationships, and logical connectivity must be maintained, and tiles should not be deformed. Despite this, in our case, it can be used, precisely because the empty unimportant areas in the *skeleton* can be considered like large patches of grass, sand or any other unstructured elements. We have to consider any element in these areas as unimportant and it will not appear in the resulting *shrunk* city. The only affectation is the changing of the angles of the intersecting streets and the relations between important places, but in general, a little angle distortion does not change the essence and the feeling of being in the target city.

5

SYSTEM: THE *CITYLIB* LIBRARY

To carry on the shrinking process, we have implemented a data structure to store cities and apply operations on them. This chapter summarizes this work and explains the foundations over which we have developed the shrinking process. The chapter is divided in three sections: the *city* type (Section 5.1), some specific *city* tools (Section 5.2) and the explanation of the city operation pipeline used to chain together the Shrinking process steps (Section 5.3). In the *city* type section, we explain the data structures implemented to store the OSM files and discuss the reason why we implement them in this way. We introduce also how we manage the node and way atomic operations. In the *city* tools section, we explain some more sophisticated tools implemented to be run over a *city* structure, tools that support the shrinking process. In the section on *city* operation pipeline, we present the implementation of a generic and modular way of concatenating city operations to make a *city* object experience changes from one state to another.

5.1 *city* TYPE

To manage the shrinking process, we had to implement a library to store the input urban data (the OSM data) and a set of methods and tools to manage this data. We call this library *CityLib*. *CityLib* is an efficient structure for common network operations implemented in *Python*. It can be divided in three main parts: parsers, the *city* class itself, and tools. We have implemented a parser to read an .osm file and load it as a class called *city*, and the other way around, a parser to store a *city* object into an .osm file. Other parsers can be implemented to load and store any other urban data sources as a *city* object. The *city* class, like any other class in Object-Oriented programming, has its structure (a set of fields) but also its behavior (a set of methods). *City* methods are basic and atomic node and way operations with an optimized computational cost, thanks to the internal *city* structure that we will present in the next subsections. Finally, the *CityLib* library has also some tools which are more expensive or sophisticated algorithms. All these algorithms can be applied to a *city* object. Most of them are implementations of well known graph theory algorithms.

5.1.1 *City bare data (nodes & ways)*

Most GIS data sources provide their data as a set of nodes and ways. A *node* is a geo-positioned point over the earth surface. Its position is defined by a tuple with values for latitude, longitude, and sometimes height above the sea level. A *way* is a list of nodes drawing a path over the earth surface. These are basic and common data that GIS has. Nonetheless, each geographic data source can have extra meta-data associated to each node or way. In the case of OSM, their XML-based format adds a set of tags to each node or way to define its characteristics. See more information about this format in Section 4.2.

We implemented a parser to load the data from an .osm file and store its nodes, ways and meta-data into a *city* object. We store this data without filtering any kind of information. We store everything: city streets, buildings, gardens, electric lines, vegetation or whatever that the parsed GIS file provides in node and way elements. The way to store this data is the following:

- **Nodes:** A dictionary with all the nodes of the city. The dictionary key is the unique node identifier.
- **Ways:** A dictionary with all the ways of the city. The dictionary key is again the unique way identifier.
- **Bounding box:** A four-value tuple delimiting the city in a rectangular area `(minLat, minLon, maxLat, maxLon)`.
- ***StreetNWAdmin***: The *StreetNWAdmin* (street network administrator) is an object that encapsulates different representations (or structures) of the city

street network (not all the city ways) and it manages which of them to use depending on the query. See Section 5.1.2.

A *city* node stores all its information, which is: the unique identifier, the coordinates (latitude and longitude) and the meta-data. The meta-data, in case of the OSM data, is the XML node element header attributes and also internal tags. All the meta-data is stored as tuples with a key and a value, for example `traffic_sign:city_limit`. In the same manner, a way stores all its information: the unique identifier, the list of nodes that make it up and the meta-data. In the case of the ways, the meta-data is again the XML way element header attributes and internal tags. Also for the ways, each meta-data information is a tuple with a key and a value, like for example, `highway:residential`.

5.1.2 *Street structure manager*

The *streetNWAdmin* is a class that manages the different data structures in which we store all the street ways. Let us emphasize that these data structures only store the street network of the city; that is, in the OSM data, all those *city* ways with the `highway` tag. To improve the speed of any operation on the city streets, we have implemented different data structures to store this information, the street network of the city. Each data structure is more appropriate for some operations and less for some others. We want to keep transparent to *CityLib* users which structure is used to implement which *city* operation method. For this reason, we call always city methods to the *city* object and it is the *streetNWAdmin* who has the responsibility to call the most efficient data structure, the one which implements that operation in the optimal way. We can see the schematic class diagram in Figure 52, observing that all class methods referring on the city street network are in the *streetNWAdmin* class, but each one implemented only in one of the specific data structures, the most appropriate for that algorithm, getting the best computational cost possible.

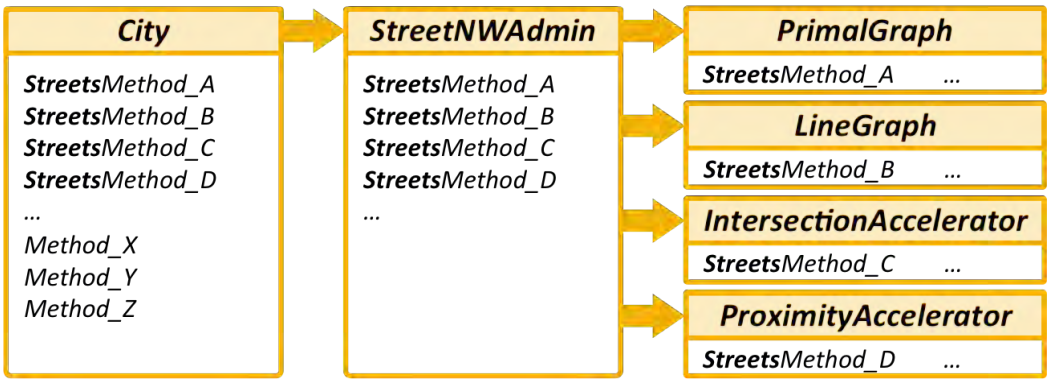

Figure 52: Street structures manager class diagram.

We have four different data structures storing in a different way the same city street network:

DIRECT GRAPH: This is a direct graph, which stores a node for each city street node and stores an edge for each way segment delimited by two consecutive street nodes.

LINE GRAPH: This is the line graph of the previous graph. Each node is a street and each segment represents the intersection of the two nodes that are connected. Therefore, a line graph edge represents the point in which two streets intersect.

INTERSECTION ACCELERATOR: This is a dictionary with a city street intersection node as key and the list of intersecting streets at that node as values.

PROXIMITY ACCELERATOR: This is a regular grid splitting the city area in a regular grid. In each grid cell we list the nodes whose coordinates fall within that cell boundaries.

In Figure 53 we can see a simple city example with its respective four representations: *directGraph*, *lineGraph*, *intersectionAcc* and *proximityAcc*. All the node identifiers are numbers and way identifiers are letters. The intersection accelerator dictionary has the intersection node keys in yellow background and all its intersecting ways are listed in its value. Finally, we can see that in the proximity accelerator grid there are three negative node identifiers in red. These are extra nodes whose existence and utility are explained in Section 5.1.6.

5.1.3 *Direct graph*

The first data structure that the *streetNWAdmin* manages is the *directGraph*. This class is a graph structure where each node is an OSM node and each edge is a straight OSM way fragment that joins two consecutive OSM nodes. It is the data structure with the most similar visual representation of the street network. It does not consider the direction of the streets.

This structure is ideal for neighboring node operations and routing algorithms. Moving from one node to any other node of the graph is like traveling over the street network itself.

5.1.4 *Line graph*

The second data structure that we implemented to store the street network information is the *lineGraph*. As its name indicates, this class is a graph structure representing the line graph of the street network direct graph. In the previous work Section 2.4, we talked about graph theory definitions of direct (or primal) graph, line (or vertex-to-edge dual) graph and dual graph. The *lineGraph* is a graph where each node is an OSM way and each edge is an intersection OSM node between two OSM ways.

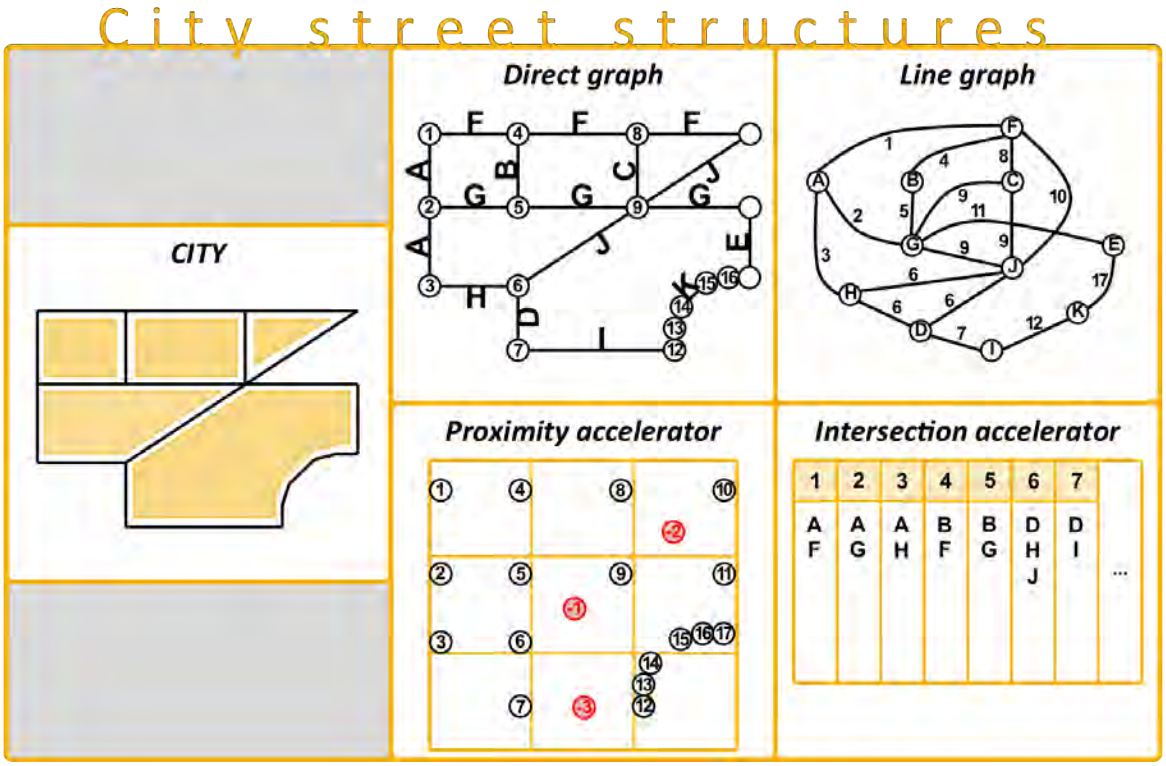

Figure 53: Example of city street network structures. *Left*: The city example with black streets and orange block areas. *Top*: Direct graph and it respective line graph. *Bottom*: the two accelerator structures for intersection and proximity operations.

This structure is ideal to find, given any street, all the streets crossing it. This is because all graph nodes adjacent to a graph node (a street) are the streets traversing this node (street). This structure is not the best and fastest one to compute all the streets crossing in a concrete intersection: Note that, for each intersecting OSM node, the number of *lineGraph* edges with this node associated is the factorial of the number of OSM ways intersecting minus one. Therefore, if four streets intersect in the same node, we will have $3! = 6$ edges in the *lineGraph* associated with this node value. We can see in Figure 53 how the direct graph intersection node `6`, with intersecting streets `D`, `H` and `J`, is tagged in three line graph edges connecting `D`, `H` and `J` nodes. The same happens in Figure 4 with primal (or direct) graph node `2` and ways `A`, `C` and `D`.

5.1.5 *Intersection accelerator*

We have added another class named *intersectionAccelerator* which is a classical dictionary with an OSM intersection node as key and the list of OSM street ways intersecting it. Note that this class is a good complement of the previous one, grouping the line graph edges with the same intersection node. This is the most efficient structure to get all streets crossing an intersection point. It has order $O(1)$ access,

because with the intersection node identifier we can get all the streets intersecting there in constant time.

5.1.6 *Proximity accelerator*

Finally, the *proximityAccelerator* class is a grid representing the city surface area delimited by the *city* bounding box. It is a regular grid with squared cells. All OSM street nodes are grouped in those cells. Each cell of the grid contains a list of city street node identifiers. The nodes placed in a specific cell have their coordinates (latitude and longitude) within the respective cell of the city area. This regular grid partition could also be implemented in a quad-tree to avoid empty cells in a non-uniform distribution of city nodes.

This data structure is more appropriate for operations related on proximity, for example the nearest node or nearest street to a given point. With this structure, we can find a point nearest to any node or way by only analyzing those nodes or ways belonging to the neighboring cells within a given radius.

There is one situation that can lead to a false positive when searching the nearest street of a given point. This case is illustrated in Figure 54, where a straight way has two consecutive nodes between them. In such a case, and because we store only the nodes in its grid cell, it can happen that none of the tested cells contain one of the two nodes delimiting the actual nearest street segment. This situation can be solved in two ways. The first one is by adding information about the streets that each cell contains, which can add a certain burden to the computations. The second one, and the one we have implemented, is by adding extra nodes dividing regularly those street segments that are too long. These extra nodes have a negative-value identifier to differentiate them from the original OSM city nodes. These nodes are shown in red in Figure 54. This figure shows, at inset *(1)*, a simple city street network with its nodes (black circles) placed in the proximity accelerator grid, and a given point P (green cross). Then, the figure illustrates how the algorithm would get the nearest street from point P without extra nodes (insets *2* and *3*) and with them (insets *4* and *5*). At inset *(2)*, we can see in green, the cells that the algorithm will process to find the nearest street. At inset *(3)*, we show in blue the nodes inside the processed cells and their adjacent street segments. We can see that the nearest one is not detected (the segment in red). It also shows, with dotted lines, the distances from P to the street segments, and with a thicker line the one found. If we do the same, but adding the extra nodes to the long segments, as shown in inset *(4)* with red circles, then we process more nodes and the resulting nearest street is the correct one, as its shown it inset *(5)*. In fact, we could use a simple 2D Digital Differential Analyzer (DDA) algorithm for scan-conversion of line segments to identify the cells that a segment traverses, and to generate only one extra node per cell that had no nodes from this line segment.

We can find detailed explanations of all the classes, fields and methods of the *city* type in the technical report of Pueyo et al. [56].

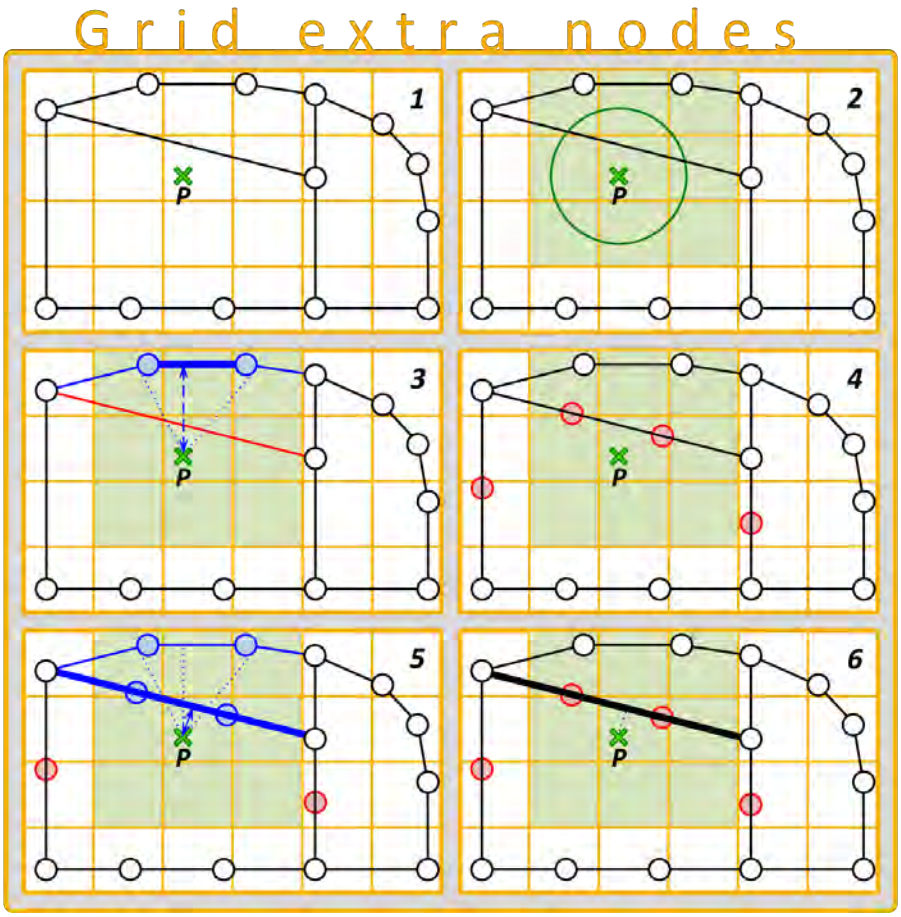

Figure 54: Example of city proximity grid with extra nodes. *(1)* Proximity grid, street network nodes and target point P. *(2)* Cells to visit to find the nearest street. *(3)* In blue, the nodes in the processed cells and their adjacent segments (the nearest one with a thicker line) and in red, a segment not processed. *(4)* Extra nodes (in red) added to long segments. *(5)* The same situation as in *(3)*, but now considering the extra nodes. *(6)* The nearest street found is the correct one.

5.2 *city* TOOLS

Beyond the basic methods of the class *city*, the *CityLib* also has a set of tools that can be used. Some of these tools are algorithms from graph theory that can be used over a city street network. Here is a list of the tools already implemented in the *CityLib Tools* namespace:

- **Astar**: Explained in depth in Section 5.2.1.
- **BoundCity**: A new rectangular street is added to the street network, creating a street like a bounding box.
- **ConvexHullCity**: A new street is added corresponding to the street network convex hull.
- **CleanFringePaths**: Removes all city dead-end street segments.
- **ClosedMinimalCircuits**: Gets a set of street network closed minimal circuits (closed polygons) which can be considered as city blocks.
- **FloodFillClusters**: Explained in depth in Section 5.2.2.
- **RotateCity**: Rotate all city streets by a certain angle counter-clockwise with the center of rotation of the centroid of the city bounding box.
- **TranslateCity**: Translate all city streets by a certain distance along a given direction.

The *Tools* namespace can obviously be expanded with the implementation of more tools. In the following sub-sections we present some of the most useful we have implemented and used in the shrinking process steps.

5.2.1 $A^\star$

One of the most useful tools implemented is an A* (*A-star*) algorithm in the class called *Astar*. A* is a search algorithm widely used in path-finding and graph-traversal problems. It finds the minimum-cost path from a given initial node to a goal node. The cost of going from one node to an adjacent one is defined by a function. The two most simple cost functions probably are the Euclidean distance and a constant value (e.g., 1) between two adjacent nodes. Depending on the cost function, the resulting minimum-cost path to go from the same origin to the same goal nodes in the same street network can change. We can see in Figure 55 how the resulting green paths are not the same when using different cost functions.

We decided to implement an A* also as generic as possible. To achieve this goal, we specify to the algorithm if we want to consider the street network as a directed or undirected graph. Thus, we can find the minimum-cost path considering the street traffic direction or not, depending on our interests. Moreover, we can specify to the *Astar* object the cost function to be used when computing the cost to go from one node to the other by using lambda expressions (i.e., functions that are not bound to a name, also known as anonymous functions). Hence, this class will process an A* algorithm over the city street network with the input parameters:

- The city itself.
- A list of tuples with node pairs to find their minimum-cost path (*originNode*, *goalNode*).
- *(Optional)* A boolean to specify if the street network should be considered as a directed graph.
- *(Optional)* A cost function. Defined out of this class, this function should at least have two parameters with the origin and goal nodes. If the user does not specify any specific cost function, the default one is the Euclidean distance between two adjacent nodes.

After creating the class, the algorithm is run and we get as result a list with a tuple for each input pair (*originNode*, *goalNode*). Each of these tuples has two elements:

- List of nodes defining the minimum-cost path to go from the origin to the goal nodes.
- Cost value of the resulting path.

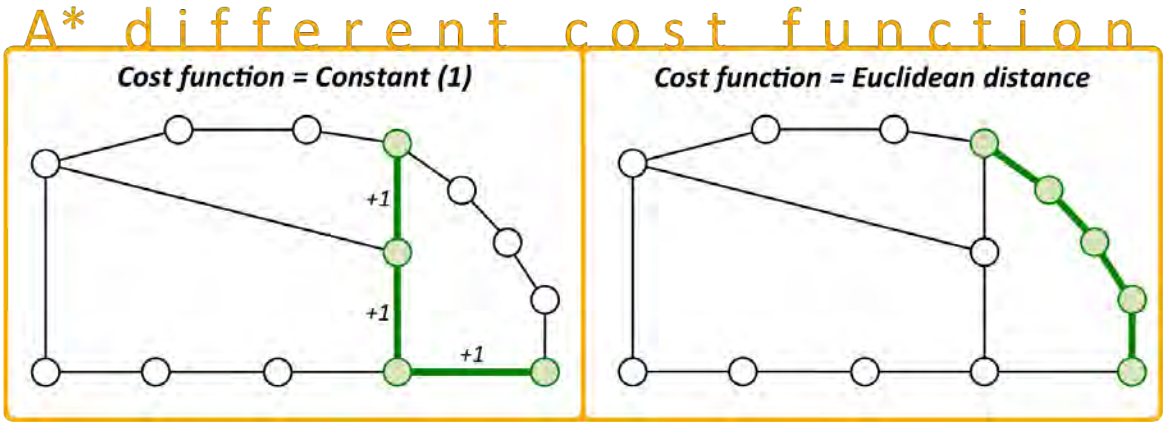

Figure 55: Example of A* algorithm using different cost functions. *Left*: The cost function is a constant value (1) to go from one node to an adjacent one. The minimum-cost path (in green) is 3 and going through the right most curve street costs 4. *Right*: Using the Euclidean distance between two adjacent nodes, the resulting minimum-cost path (in green) is different than using a constant value as cost function.

The possibility of specifying the cost function to the A* algorithm is very powerful, because it lets the user define his own cost function, as complex as needed, without implementing anything else than a method with the desired cost function. The lambda expression is the method processing the cost function and the origin and goal nodes. This cost function can also have extra parameters.

To clarify how it works, let us take as an example the function used in Section 4.4.3. That function balances (with an α value) the importance of the street that connects two consecutive nodes (imp_{AB}), and the Euclidean distance (d_{AB}) between them to connect unconnected clusters by the most important streets:

$$dist_{AB} = \alpha \cdot \frac{1}{imp_{AB}} + (1 - \alpha) \cdot d_{AB}$$

For this case, we have to specify not only the A and B nodes and the city itself, but also the α value `balance` and a list with the street importances `streetsImp_sl`. We defined the following method to process the previous function:

```
appearWeightDist(A,B,city,streetsImp_sl,balance)
```

Then, the following lambda expression for any origin node x and goal node y computes the cost function:

```
costF = lambda x,y: self.appearWeightDist(x,y,city,streetsImp_sl,balance)
```

Finally, we create the *Astar* class specifying with the pair of orign and goal nodes (`nodeClusterA`, `nodeClusterB`), a boolean to consider the graph as directed graph or not (`directed`), and the cost function (`costF`):

```
astar = Astar(city,[(nodeClusterA,nodeClusterB)],directed,costF)
```

5.2.2 *Flood fill clusters*

Another useful tool that we have implemented is the *FloodFillClusters* class. This class implements a *Flood Fill* algorithm to detect the different clusters in a city street network not totally connected (actually only possible after experiencing other changes making the city connectivity crashed). This is a well known algorithm that determines the area connected to a given node in a multi-dimensional array. It is very common to apply it on an image to fill a particular bounded area with a given color. In this case, it is also known as a *Boundary Fill*. We have implement it by considering the nodes connected by a street segment.

When we apply this algorithm over a *city*, it returns a list with all the disconnected clusters of the city. Each cluster is a list of the connected street ways. We can see in Figure 56 an example of a disconnected city with each output cluster in a different color.

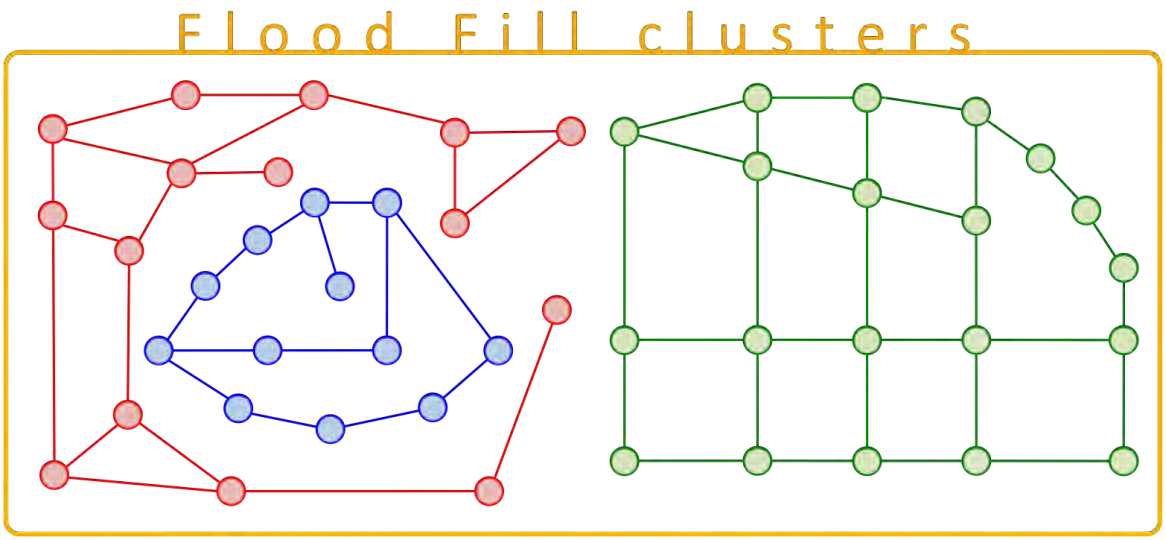

Figure 56: Example of three disconnected city clusters detected by a flood fill algorithm. The three detected clusters are colored differently: red, blue and green.

5.3 *city* OPERATIONS PIPELINE

As we mentioned in Chapter 4, the shrinking process is a concatenation of operations on the city structure. The output of one step is the input of the following step. In each step, we take a *city*, apply a set of changes to it and get a different *city* as output. Actually, it is the same *city* object but with certain changes applied. This could be considered as a generalized work flow applied to city evolutions. For this reason, we decided to implement a generic solution to ease the *CityLib* library users define new city operations and to sequence them without problems. We called this system ^{CT}Ops (City Operations).

A city is an urban structure that can experience many changes. We want to get a way to simulate these changes. The problem is that a city is a complex concept to represent and operate with. We created the *city* type explained in Section 5.1. To apply a complex modification to a city, it is recommendable to divide the task in a sequence of operations, each one doing smaller changes, following a divide-and-conquer approach. Our goal is to have a system able to apply those changes to a *city* object, changing its urban state. An urban state is a snapshot of a particular city structure at a certain moment in time. A city operator is a modular algorithm applied to a *city* street network to make it evolve from one state to another.

Ideally, we want a *CityLib* with many operators that could be applied to a *city* object in any arbitrary order, each of them resulting in a new city state. Each one of these operators should work as a black box that takes a city state and, after applying a set of changes, returns another city state as output. In other words it changes the *city* data stored. The users of *CityLib* can use any of the operators we provide, but also expand the operator library by implementing their own ones to fulfill other city changes.

When a user wants to simulate a complex city evolution, he can select an ordered set of operators from the library. Basically, the *CTOps* system works as an operator pipeline. One city experience the changes of an operator, giving as result a different urban state of the same city. This second state experiences the changes of another operator, giving as result a third urban state, and so on. In Figure 57 we show a pipeline which illustrates a generic example of a possible city evolution applying two operators.

Figure 57: Example of a city evolution pipeline: City operators and states.

The operators are always applied to the city in sequential order. The sequence of operators determines the final result. The set of operators used, and also their order, will strongly condition the final city. Therefore, users should design the operators pipeline consciously to get a desired goal city state. A city going through a given chain of operators and a second one going thorough a chain of different operators, different operator order or different operator parameters, will result in different final urban states. Figure 58 illustrates an example where changing the order of two operators in a pipeline applied over the same original city, results in two very different cities. Imagine that the first operator, let us call it A, shortens as mush as possible all vertical streets. And the second operator, operator B, fills horizontal streets empty spaces with random buildings. We can see in Figure 58 the different resulting urban states after applying these two operators in a different order, $A \oplus B$ and $B \oplus A$.

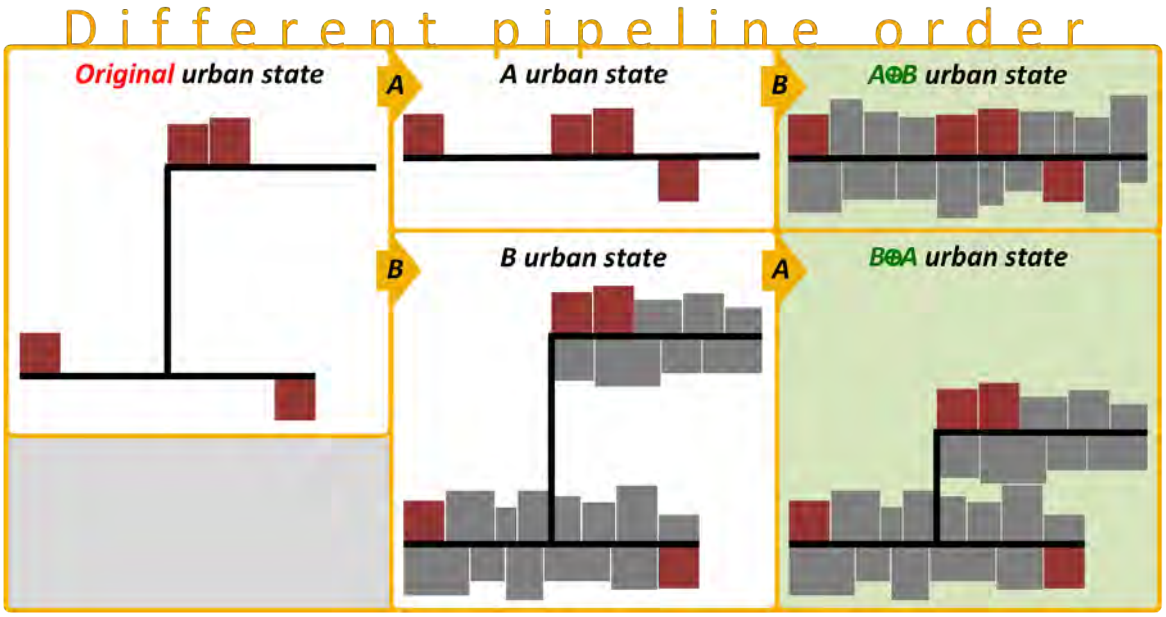

Figure 58: Operator pipeline results changing the operator's order. *Left*: The original urban state. *Middle*: Urban state after applying one operator. *Right*: Final pipeline urban state. *Top-Right*: Result when applying first operator A and then B (chain $A \oplus B$). *Bottom-Right*: Result when applying first operator B and then A (chain $B \oplus A$).

In our implementation, an operator could also be a list of simpler basic operators, while continues being considered an operator, keeping the same behavior of basic operators along explained. The way we apply basic operators within larger operator is important, too. Continuing with the example of Figure 58, we can implement a complex operator AB and another one BA, that both apply basic operators A and B in the specified order. These complex operators will return different resulting urban states, as shown in the figure. Moreover, we can have a complex operator applying two (or more) simpler operators, plus extra processing before, between or after the basic operators are applied. Using the same naming convention as in this paragraph, we could have for example an operator AB&more. We can see a set of different operators, simple and complex, combining with other operators in Figure 59.

One more thing we want to put emphasis on is that, for the same operator applied to the same input *city*, we can get different results depending on the input parameters. Imagine for example an operator that creates buildings on both sides of a street if it finds empty areas. The surface occupied by each building can be parametrized to get fewer large buildings or more smaller ones as a result. Obviously, the resulting urban state will be different depending on the value of the parameter that the user decides to use.

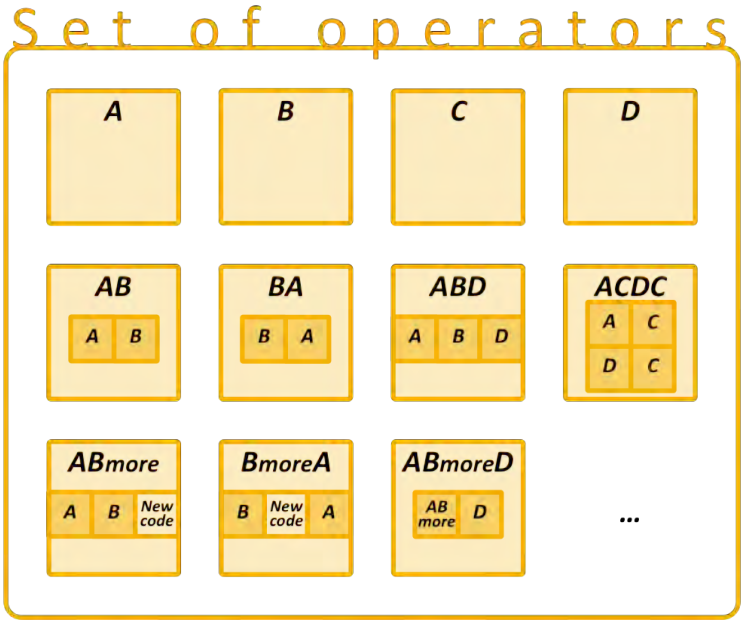

Figure 59: Examples of operator sets: simple operators and complex operators built by combining others. In the top row we see basic operators. The rest of the operators are complex, with their code calling other operators and, eventually also adding new source code.

Therefore, with the ^{CT}Ops system we have designed a modular way of applying complex operations over *city* objects. Let us summarize what ^{CT}Ops system provides to the *CityLib* users:

- A set of complex city operators.
- An easy and modular way to simulate city evolution. Users only need to design a chain of city operators to run the operator pipeline and apply the desired transformation to the city.
- A clear and flexible way of writing new operators:
  - By combining them.
  - By overwriting them.
  - By creating new ones.

We want to make the ^{CT}Ops operators independent of each other, but following a common contract to allow the replacement of any of them in a designed pipeline without undesired consequences. For this purpose, we have defined an interface (`I_CityOperator`) which defines a contract that all operators implemented should respect. The main method is the `eval` method, which is the one that executes the operator algorithm. This method always gets as input a *city* object and a dictionary of parameters. The `eval` result is the input *city* with a new urban state. Hence, this method is where the city will experience a set of changes that

takes it from the input state to the new output state. See the class diagram of Figure 60.Note how operators `CityOperator_A`, `CityOperator_B`, `CityOperator_C` and `CityOperator_ABmore` all implement the interface `I_CityOperator`. Also note that `CityOperator_ABmore` uses inside it the basic operators `CityOperator_A` and `CityOperator_B`, like we have previously explained.

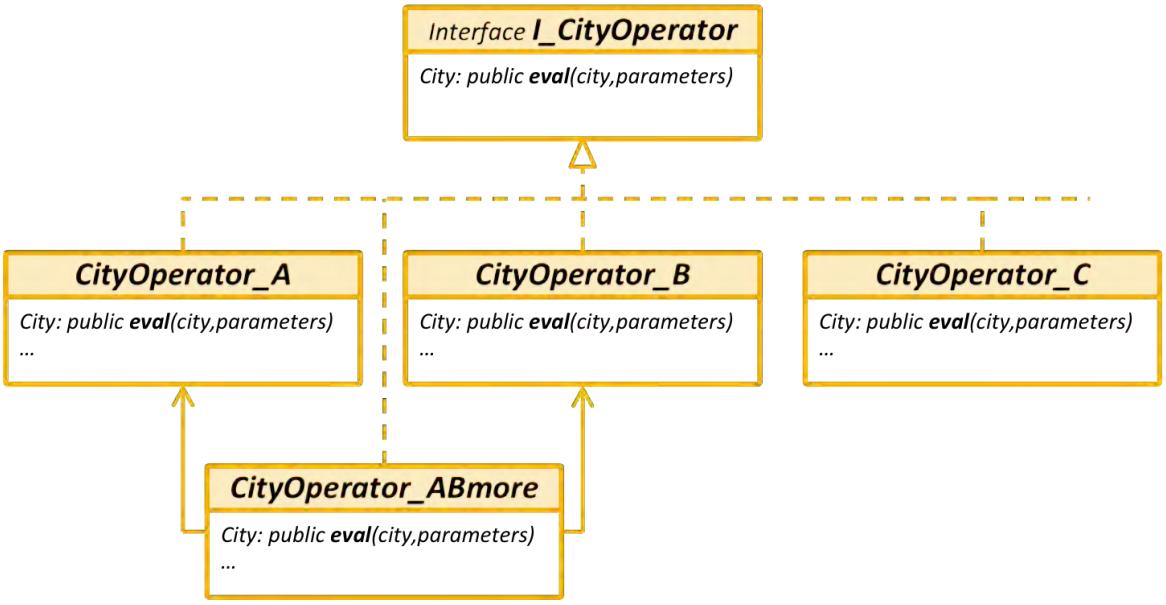

Figure 60: City operator class diagram. All the operators implement `I_CityOperator`. `CityOperator_ABmore` has a dependency with operators `CityOperator_A` and `CityOperator_B`, because it uses them inside its implementation.

Part III

CONCLUSIONS AND FUTURE WORK

A conclusion is simply the place where you got tired of thinking.

Dan Chaon, *Stay Awake*

6

CONCLUSIONS AND CONTRIBUTIONS

In this chapter, to present the conclusions that we can take from the work done in this thesis. We review all the work done and results obtained for each chapter of this thesis.

6.1 CONCLUSIONS

In this thesis we have presented two new processes for the urban modeling community: one to structure cadastral data in a hierarchical structure of blocks and buildings, and one to shrink real city street networks in a guided way to keep the essence of the original city. Moreover, we have presented the implementation of *CityLib*, a library that defines efficient data structures to store city models and a system to apply, in a flexible way, different complex operators to simulate the evolution of a city.

6.1.1 *Structuring conclusions*

We have developed a mechanism for semi-automatically getting robust structured data able to represent city blocks, buildings and any interesting urban structure from potentially badly structured legacy cadastral urban data. We have designed and implemented methods for processing and correcting the existing errors and ambiguities. Then we have developed algorithms for the accurate structuring of the layout data in blocks and buildings. The resulting structured city consists of a list of blocks, each one with its well-defined outline and a list of building elements and courtyards. The whole process only needs the user to identify the useful layers and set two thresholds. In this way, we consider that we have made cadastral data reusable, a valuable source of information that often is neglected because of its many errors. See Figures 23 and 25 as representative images of the processes' results.

Also, after getting our structured data, we have been able to generate 3D urban models representing actual buildings of an existing city. Throughout the process we assume we are provided with an input file with different layers for block and building data, but our method remains robust even with a single mixed layer with all architectural data clustered together.

6.1.2 *Shrinking conclusions*

We have designed an automatic process to simplify real city street networks. This a process that nowadays is done by designers when creating city models for video games located in real world cities. For our algorithm input, we take a real city layout and a list of "protected" city elements: the most important places (landmarks), streets and areas. Then, the shrinking process reduces the original city area to the desired size. This is not a standard resizing, but a guided reduction. It removes the less important areas, those with less important elements, protecting the important elements and areas. The result is a smaller city street network with unaltered landmarks that retain their relative positions. Also, the mean of block areas is not reduced, only the number of blocks to go from one landmark to an other. Thus

we keep the essence of the city: If one walks, along this shrunk city, he should still have the feeling of being in the original one. Figure 47 illustrates a summary of the process and results. This process delivers a first draft of the final designed city. Our process is a first step from where the designer should continue working to get a desired city. The designer can add, remove or modify any street of the resulting city. In fact, we provide the first draft of the reduced city but also the set of not added to the *shrunk* city streets. These unimportant streets are provided with the corresponding deformation, the same that the *shrunk* city has experienced, to let the artist use them if he consider it necessary.

6.1.3 CityLib *conclusions*

There are many data structures to store city models. The street network can be considered as a graph, but again there are different kinds of graphs that can store the city layout, each one better suited for one or an other usage. We have analyzed efficient structures for different basic operations and implemented a full Python library to work with cities, the *CityLib*.

This library defines *city*, a data type to store a city. This type data structure is in fact a multi-structure for its street network (direct graph, line graph, intersection accelerator and proximity accelerator) that uses a different data structure depending on the algorithm to apply on the city. Even though, this multi-structure is transparent to the *city* model user. We consider that we have an efficient city data structure for any operation we want to do, taking advantage of each data structure, also providing some implemented common algorithms. We can see a representation of the different internal structures in Figure 53.

Furthermore, *CityLib* has a city evolution system implemented that consists in a set of city operations and a pipeline to apply them sequentially in a flexible way. The operator set can be expanded with new operators or by overwriting any one of them. The only condition to use new operators is respecting with a defined interface class. This system provides the users of the library a way to easily change a city state by applying a list of operators and then to change the chain of operators applied without problems. Figure 58 shows a city evolution after applying two pipelines with the same operators, but in reverse order.

6.2 CONTRIBUTIONS

As a result of this thesis, a first paper has already been published. and three more papers are in the process of being submitted soon to well-known Computer Graphics conferences or journals. I have also co-advised undergraduate final projects, which have been developed using the library *CityLib*.

PUBLICATIONS:

- **Structuring urban data.** Oriol Pueyo and Gustavo Patow. *The Visual Computer*, volume: 30, number: 2, pages: 159–172, publisher: Springer. February 2014. doi: 10.1007/s00371-013-0791-7 [57]
- **The CityLib system.** Oriol Pueyo and Gustavo Patow. *To be submitted*. [59]
- **City Shrinking.** Oriol Pueyo, Gustavo Patow and Michael Wimmer. *To be submitted*. [58]
- **An overview of generalisation techniques for street networks.** Oriol Pueyo, Xavier Pueyo and Gustavo Patow. *To be submitted*. [60]

CO-ADVISED PROJECTS:

- **Generació automàtica de mapes turístics a partir d'informació d'interès.** Jordi Escobar Avila. (ETIS) July 2012. Co-advisor: Gustavo Patow.
- **Sistema de gestió de camins gps i càlcul de recorreguts a mida.** Marc Rovira Alsina. (ETIG) September 2013. Co-advisor: Gustavo Patow.

SOFTWARE:

- ***CityLib*.** Python library to manage urban layouts.

7

FUTURE WORK

In this final chapter we discuss new avenues for future research, from each of the main contributions: structuring and shrinking, but also from a generic viewpoint. During a PhD, new ideas always come to our mind. Some of these ideas are closely connected to the work documented here and some are farther from it. In this chapter we present some of the ideas that we would like to consider for our research and work.

7.1 STRUCTURING FUTURE WORK

Several useful new avenues are open for further research. We see two clear directions to follow after our work done on structuring. One goes in the direction of improving our algorithms: we did continuous improvements on our algorithms during this project, but we continue thinking that we could still do better. The other direction considers our structuring algorithms as robust enough and proposes to continue development on the pipeline of the procedural generation of virtual real cities, looking for a nicer and richer render of the model with as much real information as possible. We present these different ideas in the following items:

- **Hierarchical automatic thresholds:**
  One important target is to find on how to "close" all blocks as polygons without losing any detail of the real block shapes. The main structuring process drawback is the one explained in Section 3.6: The user must provide a threshold value, if it is too large then some blocks are not closes, and contrarily, if it is too small, then some shape distortions are introduced. Finding the best option may require a few trials to get the best value. Anyhow, in most times a fixed threshold value for the whole city makes it impossible to detect all the building elements if we do not want to lose shape details. Hence, we would like to use a variable threshold with different values depending on the processing step (searching for blocks or for buildings) and also the nature, dimension and shape of the elements (blocks and buildings). Moreover, such a variable threshold value would be automatically adapted to suit to the element characteristics. This improvement would make the whole process fully automatic.

- **Improve 3D models:**
  We could also improve the quality of the 3D models generated from the structured data. The main goal of the structuring section is to give coherence to the city blocks and lots data, while the 3D model generation is a simple extrusion of building elements as a set of prisms. Without processing more DXF layers, we could make some improvements on the 3D model generation. The first one would be to place the buildings at different heights according to a terrain elevation map of the city (Digital Elevation Map). The next one would be to add overhangs on the building prims which have this characteristic. This last improvement has already been mentioned in the discussion paragraph of Section 3.5.3.1.

- **Add more urban elements from cadastral data:**
  Another way to continue our work with the city modeling process is by introducing more cadastral layers, and processing and structuring them to get a much more detailed city model. Some of the layers that cadastral data provides and could be processed are: urban environment, green lands, hydrology, traffic signs, etc.

- **Improve building facades:**
  We have not worked on realistic building models, because we have only accurate descriptions of the envelopes of the buildings. A nice way to get a more realistic 3D urban scene is to improve the building facades and roofs. But the data to achieve it is not readily available in our DXF input file. This could be done by applying textures to facades and roofs from real pictures of the city, or by reconstructing facades from range scan data or other vision-based techniques [37, 64, 67, 75].

7.2 SHRINKING FUTURE WORK

On the shrinking process, we achieved the goal of providing smaller street network inspired by real world cities. The design experts could use this first approach to continue modeling the final city. The first future challenge is to continue automatizing the process to finally get a smaller model of a real city with not only the street network but also buildings, vegetation and so on. But also on the street network shrinking process, there are a few aspects that can be improved and that will be interesting to continue investigating. In the following list, we want to introduce those different points of interest that we would like to continue working on:

- **Fill large areas automatically:**
  We would like to fill the resulting *shrunk* city large areas with a subset of the unimportant original streets. This is currently the first step that designers should do from the street network we provide, and we consider this task is candidate to be automated. We discuss how to deform the unimportant streets (those not included on the *skeleton* city) in Section 4.6.1, by using a cage-based algorithm (without achieving the expected result) and by applying the same Seam Carving guided deformation. Once we have those streets, our goal should be to select the ones that fill better each large area. Filling large areas in the best way means to add one by one the best candidate, unimportant, streets to the *shrunk* city, until we achieve the goal of getting block area mean equal to the one of the original city. To reach this city blocks area mean should also be considered the standard deviation, because we want to keep a small block areas variance.

- **Automatically calculate the number of seams to remove:**
  In our shrinking process, the user should give two basic parameters: the shF (shrinking factor) and the number of seams to remove in vertical and horizontal dimensions. The shF value is to determine how many of the streets should be considered as important. We would like to relate these parameters.

In this way, only with a single shrinking factor value, the algorithm could calculate the number of seams to be removed and thus the area reduction of the final *shrunk* city. It can look some simple step but it's a challenging point to investigate considering that one parameter defines how much we want to reduce the city area while the other defines how many city elements we consider important and hence how much respectable we are with the city characteristics.

- **Error calculation:**
  We would also like to find a way to get an error measure between the original and the *shrunk* city. There are different ways we would like to evaluate the differences between the input and output cities of this process. The first one could be to apply information-theory to compare both cities. Another one could be to compare a set of geometric characteristic values between both cities. Some example geometric characteristics are: angles between important places, important street deformations, important street intersection angles, etc. Getting values for this set of characteristics, we could then calculate the relative error between the original and *shrunk* cities.

7.3 OVERALL FUTURE WORK

In this thesis we have given structure to cadastral data, which is one of the most precise data we can get of urban environments, resulting in a city model hierarchically structured in blocks and buildings. We have also reduced a city street network in a guided way, keeping the important data unaffected, but removing the less important areas. In this shrinking process we get less blocks than in the original city. This means that a *shrunk* city block represents a set of the original city blocks. Combining the two previous results, bringing together structuring and shrinking processes, is a promising research line.

- **Merging buildings:**
  We would like to investigate how to represent a big set of buildings from different blocks into a smaller set of buildings for a unique block. Obviously, landmarks, points of interest and important buildings should be kept out of this process. The idea is to merge different blocks into a new one with similar global characteristics, trying to keep the essence of the real world city, like we did with the urban layout on the shrinking process. For procedural buildings, it would be an interesting way to investigate, to search how to find common procedural rules and patterns from different buildings and then trying to combine them, generating new similar buildings with similar, but not identical rules.

Part IV

BIBLIOGRAPHY

I must say I find television very educational. The minute somebody turns it on, I go to the library and read a good book.

Groucho Marx

BIBLIOGRAPHY

[1] Ernest Adams. Designer's Notebook: The Role of Architecture in Videogames, 2008. URL http://www.gamasutra.com/view/feature/131352/designers_notebook_the_role_of_.php.

[2] M. Agrawala and C. Stolte. Rendering effective route maps: improving usability through generalization. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 241–249. ACM, 2001.

[3] D. G. Aliaga, P. A. Rosen, and D. R. Bekins. Style grammars for interactive visualization of architecture. *IEEE Transactions on Visualization and Computer Graphics*, 13:786–797, 2007.

[4] D. G. Aliaga, B. Beneš, C. A. Vanegas, and N. Andrysco. Interactive reconfiguration of urban layouts. *IEEE Computer Graphics and Applications*, 28(3): 38–47, 2008.

[5] D. G. Aliaga, C. A. Vanegas, and B. Beneš. Interactive example-based urban layout synthesis. *ACM Transactions on Graphics (SIGGRAPH Asia)*, 27(5):1–10, 2008.

[6] Autodesk. AutoCAD 2012 DXF Reference, 2011. URL http://images.autodesk.com/adsk/files/autocad_2012_pdf_dxf-reference_enu.pdf.

[7] S. Avidan and A. Shamir. Seam carving for content-aware image resizing. *ACM Transactions on Graphics (SIGGRAPH)*, 26(3):10, 2007.

[8] D. Bekins and D. G. Aliaga. Build-by-number: Rearranging the real world to visualize novel architectural spaces. In *IEEE Visualization 2005 (VIS 05)*, pages 143–150. IEEE, 2005.

[9] M. Birsak, P. Musialski, P. Wonka, and M. Wimmer. Automatic generation of tourist brochures. *Computer Graphics Forum (EUROGRAPHICS)*, 33(2):449–458, 2014.

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

[11] J. T. Bjørke. Generalization of road networks for mobile map services: An information theoretic approach. In *Proceedings International Cartographic Conference (ICA)*, 2003.

[12] J. T. Bjørke and E. Isaksen. Map generalization of road networks. Case study from Norwegian small scale maps. In *Proceedings XXII International Cartographic Conference*, 2005.

[13] CGAL Editorial Board. Computational geometry algorithms library, 2011. URL http://www.cgal.org.

[14] Mario Botsch, Leif Kobbelt, Mark Pauly, Pierre Alliez, and Bruno Levy. *Polygon Mesh Processing*. AK Peters, 2010.

[15] R. Chang, T. Butkiewicz, C. Ziemkiewicz, Z. Wartell, N. Pollard, and W. Ribarsky. Hierarchical simplification of city models to maintain urban legibility. In *ACM SIGGRAPH 2006 Sketches*. ACM, 2006.

[16] R. Chang, T. Butkiewicz, Ziemkiewicz C., Z. Wartell, N. Pollard, and W. Ribarsky. Legible simplification of textured urban models. *IEEE Computer Graphics and Applications*, 28(3):27–36, 2008.

[17] G. Chen, G. Esch, P. Wonka, P. Müller, and E. Zhang. Interactive procedural street modeling. *ACM Transactions on Graphics (SIGGRAPH)*, 27(3):103:1–103:10, 2008.

[18] European Community. Infrastructure for spatial information, 2007. URL http://inspire.jrc.ec.europa.eu.

[19] Visual Computing Lab. Italian National Research Council. Meshlab, 2011. URL http://meshlab.sourceforge.net.

[20] Ph. Dosch, K. Tombre, Ch. Ah-Soon, and G. Masini. A complete system for the analysis of architectural drawings. *IJDAR*, 3(2):102–116, 2000.

[21] D. H. Douglas and T.K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10(2): 112–122, 1973.

[22] A.J. Edwardes and W.A. Mackaness. Intelligent generalisation of urban road networks. In *Proceedings of GISRUK 2000 Conference*, pages 81–85, 2000.

[23] A.J. Edwardes and W.A. Mackaness. Road network simplification in urban areas. *Computers, Environment and Urban Systems*, 2005.

[24] A.J. Edwardes and N. Regnauld. Preserving the pattern of density in urban network simplification. In *Proceedings of GIScience 2000*, pages 104–105, 2000.

[25] A. Emilien, U. Vimont, M.-P. Cani, P. Poulin, and B. Beneš. Worldbrush: Interactive example-based synthesis of procedural virtual worlds. *ACM Transactions on Graphics (SIGGRAPH)*, 34(4):106:1–106:11, 2015.

[26] Esri. CityEngine, 2012. URL http://www.esri.com/software/cityengine.

[27] G. Fabritius, J. Krassnigg, L. Krecklau, Ch. Manthei, A. Hornung, M. Habbecke, and L. Kobbelt. City virtualization. In *Virtuelle und erweiterte Reality : 5. Workshop der GI-Fachgruppe VR/AR / Marco Schumann; Torsten Kuhlen (Hg.)*, Berichte aus der Informatik. Shaker, 2008.

[28] M. Feuchtwanger. Geographic logical database model requirements. In *Proceedings of AUTO-CARTO 9, 9th International Symposium on Computer Assisted Cartography, American Congress on Surveying and Mapping*, pages 599–609. American Society of Photogrammetry and Remote Sensing, 1989.

[29] D Flamanc, G Maillet, and H Jibrini. 3D city models: an operational approach using aerial images and cadastral maps. *International Archives of Photogrammetry Remote Sensing and Spatial Information Sciences*, 34(3/W8):53–58, 2003.

[30] Michael S. Floater. Mean value coordinates. *Computer Aided Geometric Design*, 20(1):19–27, 2003.

[31] OpenStreetMap Foundation. OpenStreetMap, 2011. URL http://wiki.openstreetmap.org.

[32] OpenStreetMap Foundation. Osmarender, 2012. URL http://wiki.openstreetmap.org/wiki/Osmarender.

[33] T. Ghawana and S. Zlatanova. Data consistency checks for building a 3D model: A case study of Technical University, Delft Campus, The Netherlands. *Geospatial World*, (4), 2010.

[34] F. Grabler, M. Agrawala, R. W. Sumner, and M. Pauly. Automatic generation of tourist maps. *ACM Transactions on Graphics (SIGGRAPH)*, 27(3):100:1–100:11, 2008.

[35] J. Hu, S. You, and U. Neumann. Approaches to large-scale urban modeling. *IEEE Computer Graphics and Applications*, 23(6):62–69, 2003.

[36] B. Jiang and C. Claramunt. A structural approach to the model generalization of an urban street network. *GeoInformatica*, 8(2):157–171, 2004.

[37] N. Jiang, P. Tan, and L.-F. Cheong. Symmetric architecture modeling with a single image. *ACM Transactions on Graphics (SIGGRAPH)*, 28(5):113, 2009.

[38] G. Kelly and H. McCabe. A survey of procedural techniques for city generation. *ITB Journal*, 14:87–130, 2006.

[39] H. Kolbe. CityGML, 2011. URL http://www.citygml.org.

[40] J. Kopf, M. Agrawala, D. Bargeron, D. Salesin, and M. Cohen. Automatic generation of destination maps. *ACM Transactions on Graphics (SIGGRAPH Asia)*, 29(6):158:1–158:12, 2010.

[41] R. Laurini and F. Milleret-Raffort. Topological reorganization of inconsistent geographical databases: A step towards their certification. *Computers and Graphics*, 18(6):803–813, 1994.

[42] R. Lewis and C. H. Séquin. Generation of 3D building models from 2d architectural plans. *Computer-Aided Design*, 30(10):765–779, 1998.

[43] M. Lipp, D. Scherzer, P. Wonka, and M. Wimmer. Interactive modeling of city layouts using layers of procedural content. *Computer Graphics Forum (EUROGRAPHICS)*, 30(2):345–354, 2011.

[44] K.R. Love. *Modeling Error in Geographic Information Systems*. PhD thesis, Virginia Polytechnic Institute and State University, December 2007.

[45] T. Lu, C.-L. Tai, F. Su, and S. Cai. A new recognition model for electronic architectural drawings. *Computer-Aided Design*, 37(10):1053–1069, 2005.

[46] K. Lynch. *The Image of the City*. MIT Press, 1960.

[47] W. A. Mackaness and K.M. Beard. Use of graph theory to support map generalization. *Cartography and Geographic Information Systems*, 20(4):210–221, 1993.

[48] S. S. Maraş, H. H. Maraş, B. Aktuğ, E. E. Maraş, and F. Yildiz. Topological error correction of GIS vector data. *International Journal of the Physical Sciences*, 5(5):476–583, 2010.

[49] A. Mas and G. Besuievsky. Automatic architectural 3D model generation with sunlight simulation. In *Proceedings of SIACG*, 2006.

[50] J.-C. Müller, J.-P. Lagrange, and R. Weibel, editors. *GIS and Generalisation: Methodology and Practice*. Taylor & Francis, 1995.

[51] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. Van Gool. Procedural modeling of buildings. *ACM Transactions on Graphics (SIGGRAPH)*, 25(3):614–623, 2006.

[52] Y. I. H. Parish and P. P. Müller. Procedural modeling of cities. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, pages 301–308. ACM, 2001.

[53] L. Plümer and G. Gröger. Achieving integrity in geographic information systems–maps and nested maps. *Geoinformatica*, 1(4):345–367, 1997.

[54] S. Porta, P. Crucitti, and V. Latora. The network analysis of urban streets: A dual approach. *Physica A: Statistical Mechanics and its Applications*, 369(2): 853–866, 2006.

[55] S. Porta, P. Crucitti, and V. Latora. The network analysis of urban streets: A primal approach. *Enviroment and Planning B, Planning and Design*, 33(5): 705–725, 2006.

[56] O. Pueyo and G. Patow. City library classes. Technical Report IMA12-03-RR, Universitat de Girona, Dept. of Informàtica Matemàtica Aplicada i Estadística, March 2012.

[57] O. Pueyo and G. Patow. Structuring urban data. *The Visual Computer*, 30(2): 159–172, 2014.

[58] O. Pueyo and G. Patow. City Shrinking. Technical Report IMAE15-03-RR, Universitat de Girona, Dept. of Informàtica Matemàtica Aplicada i Estadística, September 2015.

[59] O. Pueyo and G. Patow. The CityLib system. Technical Report IMAE15-01-RR, Universitat de Girona, Dept. of Informàtica Matemàtica Aplicada i Estadística, January 2015.

[60] O. Pueyo, X. Pueyo, and G. Patow. An overview of generalisation techniques for street networks. Technical Report IMAE15-02-RR, Universitat de Girona, Dept. of Informàtica Matemàtica Aplicada i Estadística, July 2015.

[61] H. Qu, H. Wang, W. Cui, Y. Wu, and M.-Y. Chan. Focus+ context route zooming and information overlay in 3D urban environments. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1547–1554, 2009.

[62] J.L. Raheja and U. Kumar. Properties based simplification of 2d urban area map object. *International Journal of Computer Science Issues (IJCSI)*, 7(5), 2010.

[63] J. Sae-Jung, X. Y. Chen, and D. M. Phuong. Error propagation modeling in GIS overlay. In *XXIst International Society for Photogrammetry and Remote Sensing(ISPRS) Congress - Technical Commission II*, pages 825–836, 2008.

[64] N. Snavely, R. Garg, S. M. Seitz, and R. Szeliski. Finding paths through the world's photos. In *ACM Transactions on Graphics (SIGGRAPH)*, volume 27, page 15. ACM, 2008.

[65] G. Stiny. Introduction to shape and shape grammars. *Environment and Planning B, Planning and Design*, 7(3):343–351, 1980.

[66] F. Taillandier. Automatic building reconstruction from cadastral maps and aerial images. In *Proceedings of ISPRS-CMRT05*, pages 105–110, 2005.

[67] Y. Tzur and A. Tal. Flexistickers: photogrammetric texture mapping using casual images. In *ACM Transactions on Graphics (SIGGRAPH)*, volume 28, page 45. ACM, 2009.

[68] T. Ubeda and M. J. Egenhofer. Topological error correcting in GIS. In *Proceedings of the 5th International Symposium on Advances in Spatial Databases*, SSD '97, pages 283–297. Springer-Verlag, 1997.

[69] C. A. Vanegas, D. G. Aliaga, B. Beneš, and P. Waddell. Interactive design of urban spaces using geometrical and behavioral modeling. *ACM Transactions on Graphics (SIGGRAPH Asia)*, 28(5):111:1–111:10, 2009.

[70] C. A. Vanegas, D. G. Aliaga, P. Wonka, P. Müller, P. Waddell, and Watson B. Modeling the appearance and behavior of urban spaces. *Computer Graphics Forum (EUROGRAPHICS'09, State of the Art Reports)*, 29(1):25–42, 2009.

[71] B. Watson, P. Müller, O. Veryovka, A. Fuller, P. Wonka, and C. Sexton. Procedural urban modeling in practice. *IEEE Computer Graphics and Applications*, 28(3):18–26, 2008.

[72] B. Weber, P. Müller, P. Wonka, and M. H. Gross. Interactive geometric simulation of 4d cities. *Computer Graphics Forum*, 28(2):481–492, 2009.

[73] E. Whiting, J. Ochsendorf, and F. Durand. Procedural modeling of structurally-sound masonry buildings. *ACM Transactions on Graphics (SIGGRAPH Asia)*, 28(5):112:1–112:9, 2009.

[74] P. Wonka, P. Müller, B. Watson, and A. Fuller. Urban design and procedural modeling. In *ACM SIGGRAPH 2007 Courses*, pages 229–229. ACM, 2007.

[75] J. Xiao, T. Fang, P. Zhao, M. Lhuillier, and L. Quan. Image-based street-side city modeling. In *ACM Transactions on Graphics*, volume 28, page 114. ACM, 2009.

[76] X. Yin, P. Wonka, and A. Razdan. Generating 3D building models from architectural drawings: A survey. *IEEE Computer Graphics and Applications*, 29(1): 20–30, 2009.

I might find somewhere where I can finish my book. I have thought of a nice ending for it:
'And he lived happily ever after to the end of his days.'

J.R.R. Tolkien, *The Lord of the Rings*