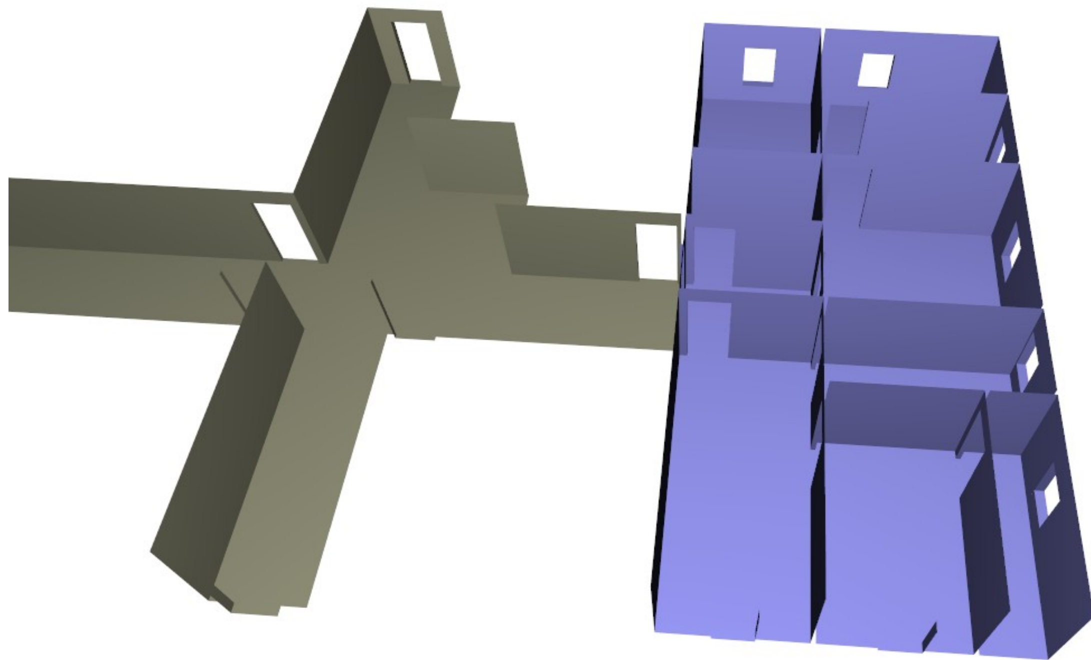
CHALMERS

Procedural Generation of Indoor Environments



ALEXANDER DAHL

LARS RINDE

Master's Thesis

CHALMERS UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering
Division of Computer Engineering
Göteborg 2008

All rights reserved. This publication is protected by law in accordance with “Lagen om Upphovsrätt, 1960:729”. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior permission of the authors.

© Alexander Dahl, Lars Rinde, Göteborg 2008.

Abstract

During the last few decades, computer games and simulations have grown in complexity almost as fast as the capabilities of hardware have increased. Because of this, there is an ever greater need for large-scale 3D environments and content to fill it, but it is also becoming unfeasible to manually create all this content. This means that methods for procedural content creation will need to take a larger role as demands on the size and richness of virtual worlds continue to rise.

This Master Thesis describes the development of a new algorithm for procedural generation of interior spaces, with the goal of providing a solid foundation for future deployment in real-time applications such as games. The developed algorithm uses an hierarchical subdivision process which supports an on-demand creation of data, which means only regions that are visible need to be generated. Additionally, because of independence between subdivided spaces, later stages of the algorithm can be efficiently parallelized.

Further, a prototype implementation of the core mechanics of this algorithm, in the form of an interactive interior generator, is described. This implementation is focused on the generation of apartment buildings. The report is concluded with a look ahead at some related areas of particular interest.

Sammanfattning

De senaste årtiondena har inneburit en kraftig ökning i komplexitet för datorspel och andra simuleringar som har varit nästan lika snabb som förbättringar av hårdvarans kapacitet. På grund av detta finns det ett ständigt växande behov av storskaliga 3D-miljöer och innehåll för att fylla dem, men det blir också mer ohållbart att producera allt detta innehåll för hand. Detta innebär att metoder för att procedurellt skapa innehåll måste ta en större del av ansvaret när kraven på virtuella världars storlek och deras innehållsrikedom fortsätter att öka.

I denna rapport beskrivs utvecklingen av en helt ny algoritm för att generera procedurella inomhusmiljöer, med det framtida målet att skapa en solid grund för integrering i realtidssystem som t.e.x. spel. Den hierarkiska uppdelningsprocess som används ger automatiskt stöd för att generera data vid behov, dvs möjlighet att bara bygga de regioner som är synliga. Det faktum att underregioner är oberoende gör det även möjligt att effektivt parallellisera senare delar av algoritmen.

Utöver detta beskrivs även den implementation av algoritmen som gjordes i form av en interaktiv interiörgenerator. Fokus för denna applikation ligger på generering av bostadsbyggnader med lägenheter. Rapporten avslutas med en genomgång av relaterade ämnen som är speciellt intressanta för framtida studier.

Acknowledgments

We would like to thank Gustav Taxén, our supervisor at Avalanche, for giving us the opportunity to study this interesting field, and for providing comments and suggestions during our work; Ulf Assarsson, our examinator at Chalmers; Pablo Carranza at KTH Architecture, for providing useful suggestions when we started out. We would also like to thank our fellow Master Thesis writers at Avalanche: Jenny, Magnus, Erik F, Rickard, Joel and Erik S. Finally, a big 'thank you!' to our respective relatives (Astrid Ewerlöf, Kerstin Laurell, Karla Ramsbäck and Gunnar Löfgreen) who provided accommodations during our work on the thesis in Stockholm.

Contents

1 Introduction **3**
1.1 Background . . . 3
1.2 Purpose . . . 3
1.3 Scope . . . 4
1.4 Method . . . 4

2 Theory **5**
2.1 Related Work . . . 5
2.2 Architectural Theory . . . 5
2.2.1 Space Syntax . . . 6
2.2.2 A Pattern Language . . . 6
2.2.3 S-Spaces . . . 8
2.3 Voronoi Diagrams . . . 8

3 Developed Algorithm **13**
3.1 Overview . . . 13
3.2 Input Data . . . 14
3.3 Building Skeleton . . . 14
3.4 Transition Area . . . 15
3.4.1 Vertical access areas . . . 15
3.5 Regions . . . 19
3.6 Sub-regions (Apartments) . . . 19
3.7 Room Walls . . . 21
3.8 Room Type Allocation and Entrypoints . . . 23

4 Prototype Implementation **29**
4.1 Initial Considerations . . . 29
4.2 Input Data . . . 29
4.3 User Interface . . . 30
4.4 Script Interface . . . 30
4.5 Results . . . 30

5 Discussion **33**

6 Further Development **35**
6.1 Improvements . . . 35
6.1.1 Rules and parameters . . . 35
6.1.2 Randomization . . . 35
6.1.3 Performance . . . 36

6.2 Future Work 36
6.2.1 Integration with building generators 36
6.2.2 Generalization to cover other building types 36
6.2.3 Generation of auxiliary data 36
6.2.4 Automatic placement of furniture and decorative items . 37
6.2.5 Integration with level design tools 37
6.2.6 Movement and space in simulated versus real interior environments 37
6.2.7 Automatic texturing of interiors 37

Bibliography **39**

A Prototype Design **41**

B Additional Results **45**

Definitions

Building skeleton A structure derived from the outer walls, representing the general shape of the building.

Concave corner A corner that is concave when seen from the exterior, that is a polygon corner where the interior angle is larger than 180 degrees.

Convex polygon A polygon without any concave corners.

Corridor A transition area that allows access to different areas of a building floor. In this thesis, this always refers to the main corridor in the house, and never to corridors inside apartments.

Depth Used here to describe how far inside a structure a space is located, where distance is measured in the number of spaces traversed rather than in meters.

Region A region is generally used to describe an abstract space (an area which will be further subdivided into smaller regions or finished rooms).

Space A space can be a region, a room, or any other area in a building. The boundary of a space always defines a simple polygon.

S-spaces Spaces created by taking a building outline and creating walls at appropriate places, mainly between windows, out from corners etc. (Peponis et al. [1997]).

S-space skeleton A representation of the s-space walls that is used as a snap grid.

Simple polygon A polygon encompassing a continuous area without holes, and with no edges crossing another edge in the polygon.

Transition Area An area which is used primarily for transportation between rooms.

Vertical Access Area An area in a floor plan which correspond to a vertical access route, such as a staircase or elevator.

Voronoi diagram A space partitioning system using a set of seeds, and associating every point in the space with the closest seed.

Wall skeleton A wall skeleton is a representation of the walls separating the rooms of a region.

Chapter 1

Introduction

1.1 Background

The need for large-scale realistic 3D environments has been rising rapidly as computer games and simulations have become more and more complex. Because this complexity often means both more detail (such as higher resolution textures and advanced pixel shaders) and more content, artists and leveldesigners have recieved a correspondingly higher workload. The result is that time and effort that could have been spent on items such as important structures and areas is now spread over all the content in a virtual environment.

1.2 Purpose

The purpose of this Master Thesis is to investigate ways of procedurally generating interior environments in order to offload work from artists and level designers. Implementing a system for interior generation will allow virtually infinite variation in game or simulation environments, without requiring more work hours from content creators. The focus for the thesis has been four-fold:

- Creation of an algorithm for the generation of interiors for multi-story buildings.
- Creation of an application showcasing this algorithm, allowing a user to generate the interior of buildings with various shapes.
- Investigate the possibility of applying descriptions of desired outcomes using the architectural theory Space Syntax. This should ideally lead to an interface where a user can specify how interior regions should be *experienced*, rather than how they should be layed out.
- Explore which areas are most interesting to explore in future projects building on the work presented here.

1.3 Scope

Due to the complexity of the problem a number of limitations were introduced during the planning phase in order to reduce the scope of the project to a manageable size. The following main limitations were set:

- Only consider 'flat' floors so that the problem can be initially worked on in two dimensional space. This means floors cannot have rooms that cover several stories (with the exception of fixed-size vertical access areas).
- Limiting the type of structures to residential buildings, ie focusing on generating interiors which contain only apartments.
- Limiting the shape of buildings to simple polygons without holes.
- Limiting the shape of buildings to have similar or linearly changing thickness throughout (ie no hourglass shapes) to allow for a single transition area (corridor network).
- Limiting the input buildings to those that need a corridor

1.4 Method

The following steps were followed to achieve the goals set out for this Master Thesis:

- Study existing literature on Space Syntax and other relevant architectural theories (described in chapter 2).
- Design a high-level algorithm for generating interiors based on the results of the literature review (chapter 3).
- Design and implement (possibly several variations of) each sub-component of the interior generation algorithm (chapter 4).
- Evaluate results and compare to existing techniques (chapter 5).
- Produce a list of recommended further research and extensions/modifications to the produced algorithms based on the findings of the evaluation (chapter 6).

Chapter 2

Theory

2.1 Related Work

While different techniques for procedural content generation have been utilized in games for decades (e.g. Elite [Braben and Bell, 1984]), attempts to generate large-scale man-made 3D environments have just recently started. Despite the current activity in this area, and the seeming abundance of projects with focus on generation of cities [Flack et al., 2001, Greuter et al., 2003, Parish and Müller, 2001] and buildings [Birch et al., 2001, Brenner, 2000, Hahn et al., 2006, Larive and Gaildrat, 2006, Laycock and Day, 2003, Müller et al., 2006, Wonka et al., 2003], the collected body of work on generation of interior spaces is surprisingly small [Hahn et al., 2006, Martin, 2005, Noel, 2003]. Some older ventures into the area focus on algorithms for computer-aided design, such as Galle [Galle, 1981].

The previously cited work in interior generation have several constraints which limits their usefulness for general applications. Noel (2003) creates rooms within a rectangular boundary using a simplistic space splitting scheme, followed by adjustment of the resulting walls. Hahn et al (2006) also focus on rectangular floor plans, but use a more sophisticated splitting approach. Their results are also limited to generating rooms with the same depth, which is unusual in most houses. Martin (2005) takes the reverse approach, in that he generates the room layout and the builds the exterior from that. Also, his algorithm only support single-story houses.

2.2 Architectural Theory

The main problem with many publications within architectural theory is their lack of scientific approach. Many of the papers and articles are simply compilations of ideas from a group of individuals, or even a single author's opinion on a subject. This of course makes it hard to extract anything but very general guidelines when implementing an architecture generation system. Still, there are specific areas that show promise in this regard, and one of these is the collection of work on the subject of Space Syntax [Hillier and Hanson, 1984, Hillier, 2007]. The widely known (particularly in software development circles) *A Pattern Language*[Alexander et al., 1977] was also consulted.

2.2.1 Space Syntax

Space Syntax [Hillier and Hanson, 1984, Hillier, 2007] is a collection of methods for representing and analyzing the layout of architectural structures, from city planning to interior floorplans. The part that is interesting for this thesis is the analysis of the accessibility graph of a building.

The accessibility graph of a building is a graph with all rooms and other areas (such as corridors and staircases) as the nodes, with the connecting doors as connections between the nodes. Analysis of this graph in real houses yield some general rules for the placement of specific rooms. For instance, private rooms such as bedrooms are usually placed at the deepest points of the graph, bathrooms are placed somewhat out of the way and public rooms such as living rooms are placed with easy access to every other part of the apartment.

Figure 2.1 shows example room configurations for a building, with the nodes of the graphs (called justified graphs by Hillier, since they can be constructed from the point of view of any space within the configuration) representing rooms, and connections representing doorways [Hillier, 2007]. Hillier also identifies four types of nodes (spaces) in these graphs; a-type nodes, which are dead-end spaces (only one connection), b-type, which are nodes with two connections that connect the root area of the graph with a tree sub-structure, c-type, which lie on a single ring of nodes, and d-type nodes, which lie on two or more rings. These types are interesting in that they can be used to determine which type of room a space can be used for. For example, a-type spaces are generally private rooms, since they have only one entrance and thus the only movement through them is with the purpose of entering, leaving, or working in the room, not passing through.

2.2.2 A Pattern Language

In A Pattern Language [Alexander et al., 1977], Alexander et al have compiled a collection of design patterns, which are descriptions of how to best approach certain situations in architecture. It is interesting to note that some patterns correspond to findings in Space Syntax analysis, such as the Intimacy Gradient pattern which suggest a placement of rooms according to their privacy level. This means that bedrooms should be placed 'deeper' in an apartment than a living room or hallway, if possible. Different kinds of rooms in an office building can be arranged in a similar manner, with reception area, meeting rooms, and private offices placed at increasing depth.

However, many patterns in A Pattern Language are more specific to a particular culture and function (mainly North American homes), and thus are not directly applicable in a system attempting general building or interior generation. Also, in A Pattern Language, Alexander et al attempts to describe *best practices*, which are not often followed in built architecture. This can make it counter-productive to implement a system following these patterns, as the results would more resemble an ideal spatial configuration (according to the authors) than an actual building.

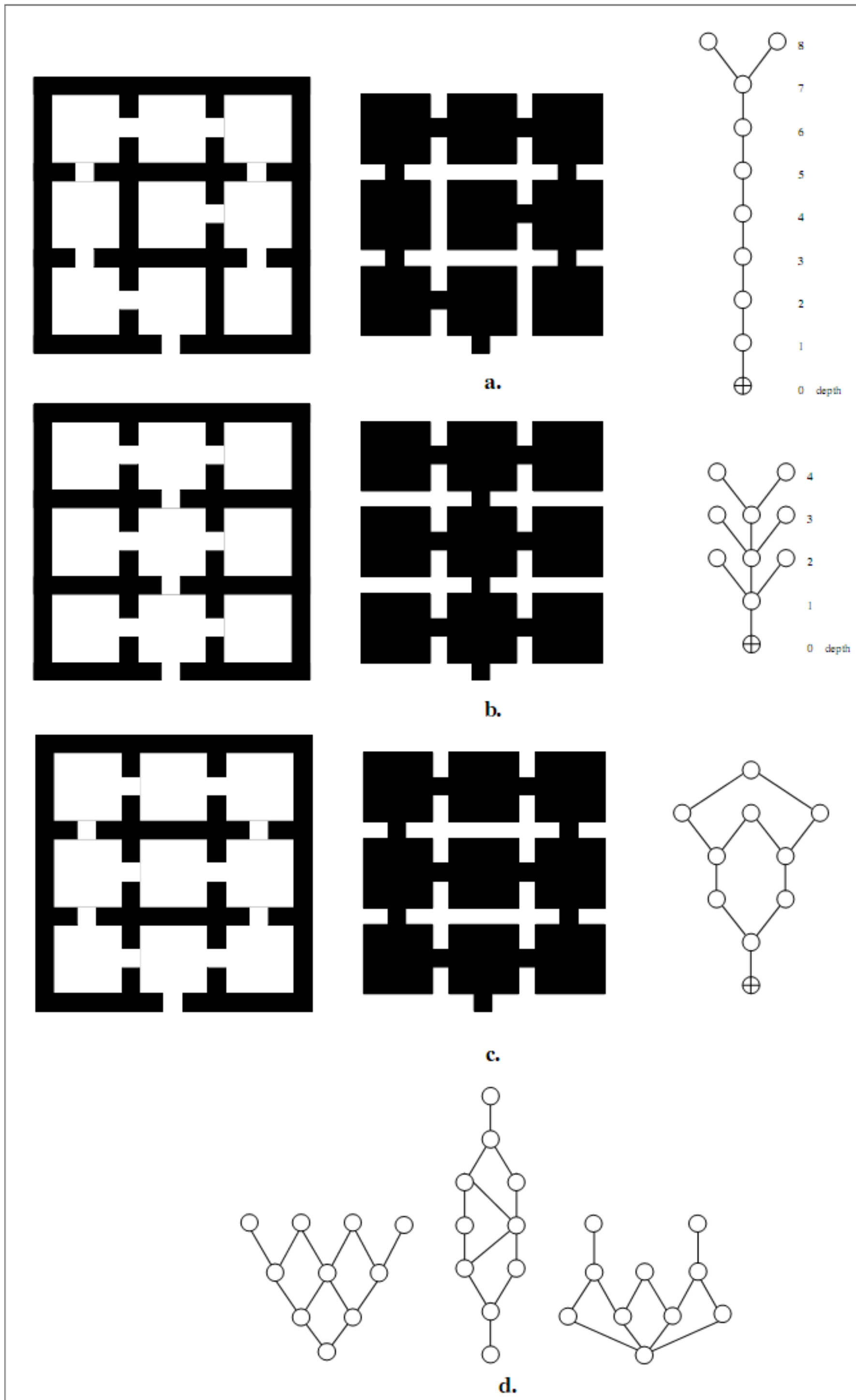


Figure 2.1: Example of different room configurations and their justified graphs.

2.2.3 S-Spaces

In their study of interior wall placement, Peponis et al (1997) found that interior building walls usually lie along lines defined by features of the outer walls and inner load-bearing walls. Such features include windows, doors and corners. If lines are drawn perpendicular to the wall from all corners as well as from the walls between windows and doors, the result is a line network creating areas called *S-Spaces*. Note that if the walls are at right angles, the lines drawn from the corners will usually coincide with another wall (see figure 2.2a). If the walls are not at right angles, there will sometimes be S-Space lines that lie at a very low angle to a wall, making them unsuitable for wall placement (see figure 2.2 b).

The S-Space line network will thus represent natural positions for placement of interior walls. In this thesis, a slight extension of this approach is used to ensure that the walls are placed in positions that feel realistic. The extension that is used means that any wall that is too long and not divided by an S-Space line will be divided by an extra line. This is to avoid the unwanted situation where all the S-Space lines go in the same direction in an apartment (see figure 2.3).

2.3 Voronoi Diagrams

The basic idea of a voronoi diagram is that you have a set of seeds (positions, possibly associated with weights) on a plane and you want to partition the plane into *voronoi cells*, with one cell per seed. The cell associated with a specific seed is defined as all points for which the distance to the seed is less than the distance to all other seeds, as measured by some definition of distance. Another way to describe it is that the demarkation lines (the *cell walls*) lie along lines with equal distance to the two closest seeds. In the basic case, distance D is measured as the standard Euclidian distance (figure 2.4 a). With Δx and Δy being the distance along the x- and y-axis respectively between a location on the plane and a seed, Euclidean distance is defined as:

$$D = \sqrt{\Delta x^2 + \Delta y^2}$$

This generates a pattern of straight voronoi cell walls. It is, however, possible to define distance in a number of different ways, some of the common ones being Manhattan distance (figure 2.4 b),

$$D = \Delta x + \Delta y$$

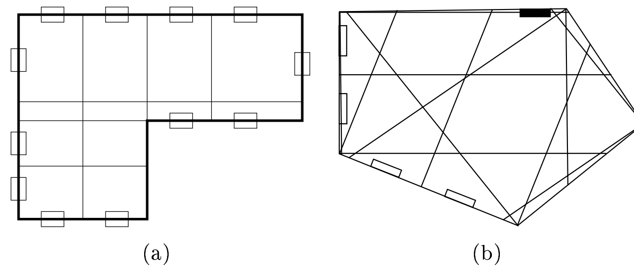
multiplicatively weighted Euclidean distance (figure 2.4 c) with a different weight k_i for each seed i (if all weights k_i are the same, this is the same as the standard Euclidean distance),

$$D = k_i\sqrt{\Delta x^2 + \Delta y^2}$$

and the supremum metric (figure 2.4 d),

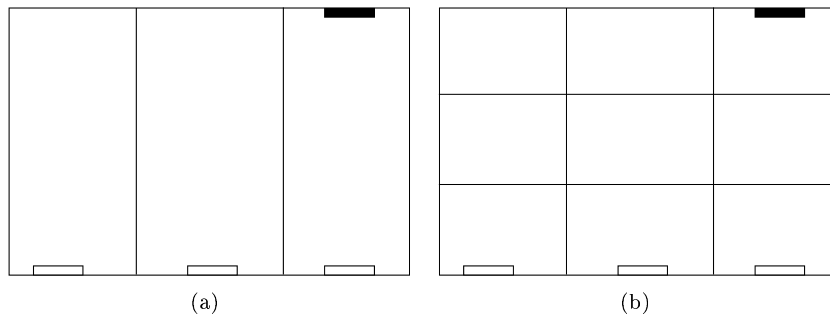
$$D = \begin{cases} \Delta x, & \Delta x > \Delta y \\ \Delta y, & \Delta x \leq \Delta y \end{cases}$$

each one giving quite different patterns for the same set of seeds. It is even possible to define distance in ways so that some seeds will not, in fact, lie inside their corresponding voronoi cells. It is also quite possible to extend the theory to three or more dimensions. For the purposes of this thesis, however, it is only necessary to consider the Euclidean definitions of distance (weighted and non-weighted), and two dimensions is all that is required. Areas where voronoi structures arise in nature include such wildly different things as soap bubbles, spider webs, cell growth and galaxy cluster formations.



(a) (b)

Figure 2.2: Example of S-Space networks



(a) (b)

Figure 2.3: An S-Space skeleton without the added lines (a) and with the added lines (b)

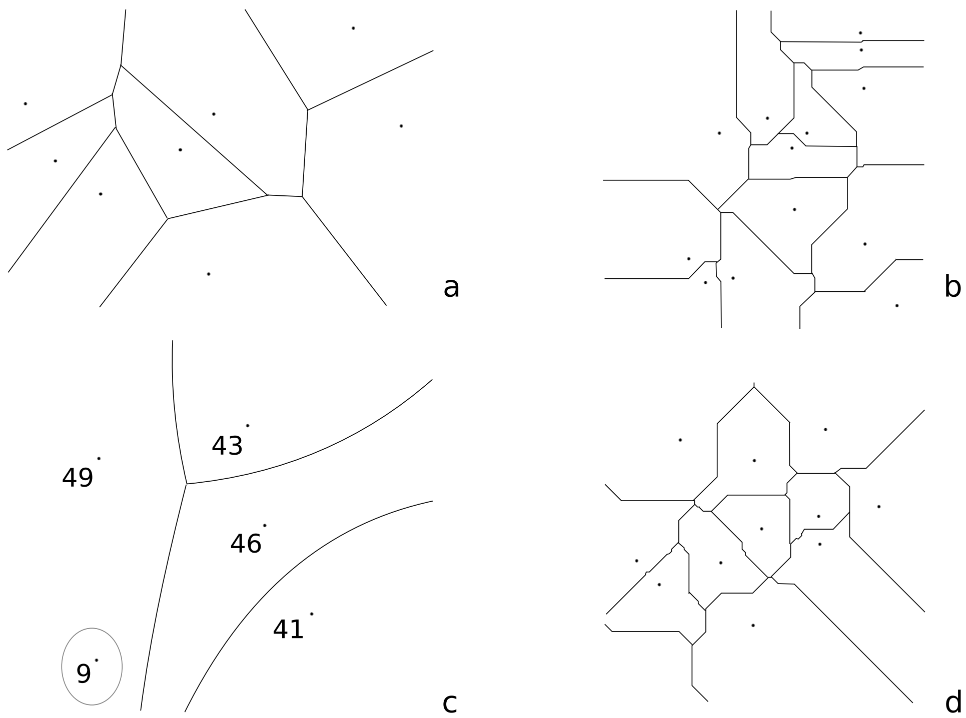


Figure 2.4:
Examples of Voronoi diagrams: a) Regular Euclidean distance, b) Manhattan distance, c) Multiplicatively weighted (numbers represent weights), d) Supremum metric

Chapter 3

Developed Algorithm

3.1 Overview

The different stages of the algorithm developed are designed to be modular, so that for example different subdivision algorithms can be used in different buildings, or even hierarchically at different levels of the same building. Figure 3.1 shows a high-level overview of how the algorithm operates. The following stages make up the algorithm:

Input Data: This is the initial information that is passed to the algorithm from the previous stage (be it hand-made or generated).

Parameters: Parameters are used to control the result of the algorithm in variety of ways, ranging from corridor width to which types of regions and rooms the building should contain.

Building Skeleton: The building skeleton is generated from the input data to facilitate analysis of the building shape, as well as construction of the internal transitions areas (such as corridors).

Transition Area: After the skeleton has been built the corridors within the building are placed. This is the Transition Area used to gain access to all parts of the interior.

Regions: The regions making up the maximum continuous areas are created from the input data and the created transition area, and used for further sub-region division.

Sub-Regions: Sub-regions are created from the maximal regions and represent the boundaries of a collection of rooms (in this case apartments).

Room Walls: The room walls are built using a hybrid weighted pseudo-Voronoi/S-Space algorithm, where the cell boundaries of a Voronoi diagram are used to select suitable S-Space edges for walls.

Rooms: The rooms are built from the room walls. During this process a graph of potential neighbouring rooms is also constructed, which is then used to allocate room types (from the given parameters) and place doors between some of the rooms.

Geometry: In the final step of the algorithm, the spatial hierarchy constructed is traversed and walls and entrypoints used to construct the 3D mesh of the interior.

3.2 Input Data

One of the requirements for the algorithm is that it should take an already existing exterior as input, rather than creating the exterior as part of the process. This is a large difference to some of the cited previous work, and it is motivated by the fact that the existing city-generation engines produce hollow building shells, which need only be 'filled' with an interior if the viewing position is inside or close by. Further, the creation of an empty exterior does not require extra data and processing time, which would be required if the exterior was created from the interior.

As no generated buildings were available to extract the required data from directly, assumptions had to be made about the input data that could be expected. Representing a building exterior requires at least the following data for each floor:

- The outer wall, or shape of the structure (represented by a polygon).
- The positions and sizes of any windows and doors.
- The height of the floor (which is required when constructing geometry).
- The thickness of the outer wall.

Because of the lack of input data specification, the decision was made to use only this minimum data set.

Additionally, the algorithm can be modified through a number of parameters, such as region types, room types, and corridor width.

3.3 Building Skeleton

The first part of the algorithm produces a skeleton for the outside wall polygon of the building. The basic idea for creating a skeleton is to push all walls inwards at a constant rate and create a skeleton edge where two walls meet [Felkel and Obdrzalek, 1998]. This results in a structure that lets the algorithm analyze the shape of the building and determine if there should be inner corridors in the house and where they should be placed (see figure 3.5).

The actual algorithm which is used for creating this skeleton is a slightly modified version of the one used by Felkel and Obdrzalek (1998). Their algorithm loops over the vertices whereas the one described here loop over the skeleton edges. This means they have to treat different types of skeleton intersections differently, while the algorithm used here is able to treat all intersections the same way. However, their algorithm can handle polygons with "holes", which the one used here cannot. This functionality was not required for this project, so the decision was made to use the method that was simpler to implement. The simplified algorithm for creating the building skeleton is as follows:

1. From each corner of the house, create a skeleton edge at equal angles to both corresponding outer walls (see figure 3.2). Each skeleton edge is associated with the two walls defining its direction.

2. Find all intersections between skeleton edges that share an associated wall and lie inside the building (see figure 3.3). The intersection is associated with the shared wall.

3. From those intersections, choose the one closest to its associated wall. Create a new skeleton edge originating from this intersection, corresponding to the two walls that the parent skeleton edges did not have in common (see figure 3.4).

4. Repeat steps 2 and 3 until only one skeleton edge remain that has not intersected another skeleton edge. The last edge will simply be the root of the skeleton tree. It will not have any corresponding walls, and so will be of length zero. The root will be placed at the deepest point of the house in the sense that it is the point that is the furthest away from any outer wall (see figure 3.5).

5. Attach all doors to the skeleton by creating skeleton edges perpendicular to the doors and attaching these to the closest intersection point with the skeleton (see figure 3.6).

3.4 Transition Area

Looking at the skeleton, it is simple to determine if there is a need for inner corridors in the current building. If the distance from the root node to the closest wall is big enough, corridors are needed to minimize the number of rooms without windows.

If corridors are needed, the skeleton tree is traversed recursively starting at the root, placing corridors along the edges of the skeleton if:

- the skeleton edge is far enough from the closest wall (see figure 3.7 a)
- there is a door attached to the skeleton edge (see figure 3.7 b),

or

- the recursive traversal returns corridors from skeleton edges further down in the skeleton tree.

This yields a corridor network with access to all parts of the building as well as the outside (see figure 3.8).

3.4.1 Vertical access areas

After the corridor wall has been built, and if the current floor is the entry level floor, vertical access areas (VAA) are added. These are vertical volumes of space which are used to reach higher floors, such as staircases and elevator shafts. The placement of these is done to allow access to floors other than the entry level. For each door in the exterior wall:

1. Starting at the door, find the first skeleton segment in the corridor that is long enough to place a VAA at. This is a potential position for placement of a VAA.

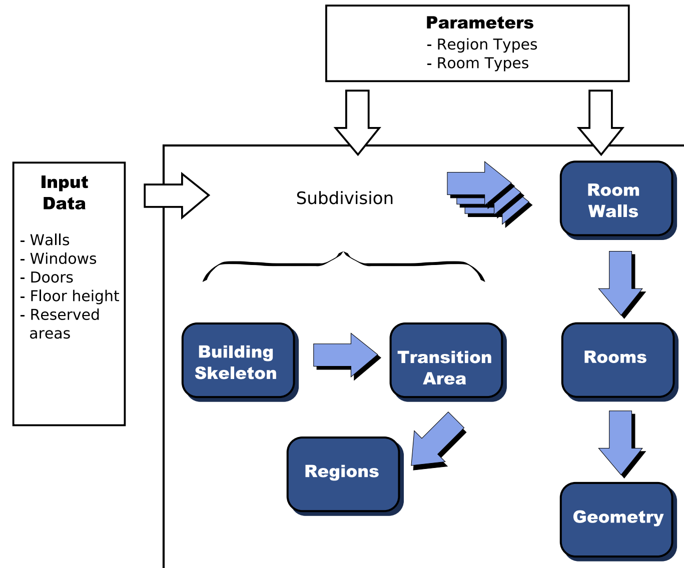


Figure 3.1: Algorithm overview: Input data is used to create a subdivided interior space, the regions of which can then be further divided or have room walls inserted. Following this, the actual rooms are built and allocated types according to the provided parameters, and finally the geometry is constructed.

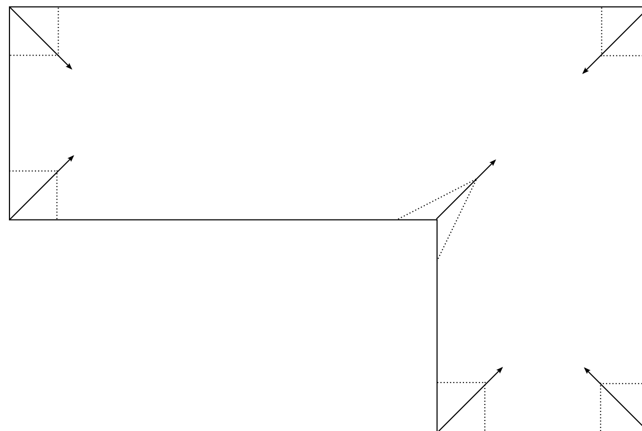


Figure 3.2: Starting skeleton edges with associated walls marked.

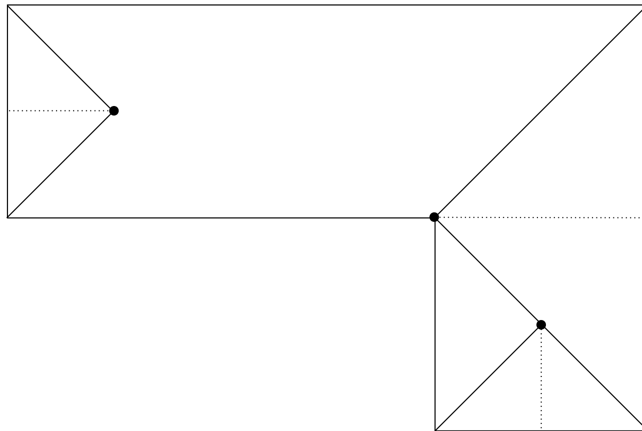


Figure 3.3: Skeleton edge intersections with associated walls marked.

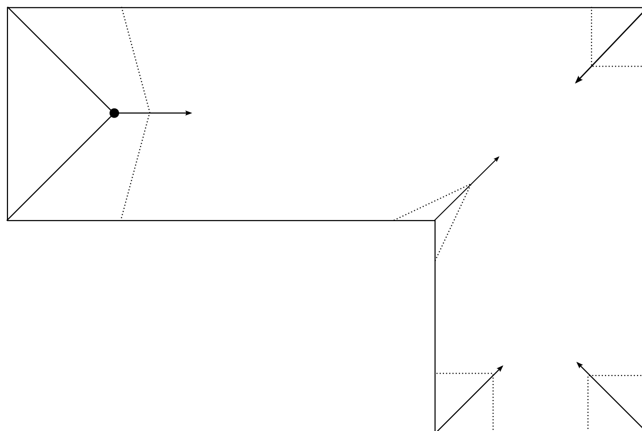


Figure 3.4: New skeleton edge created, with associated walls marked.

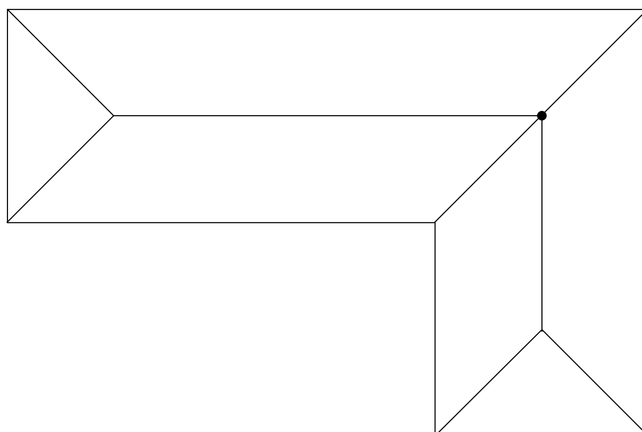


Figure 3.5: Skeleton with root marked (before doors have been attached).

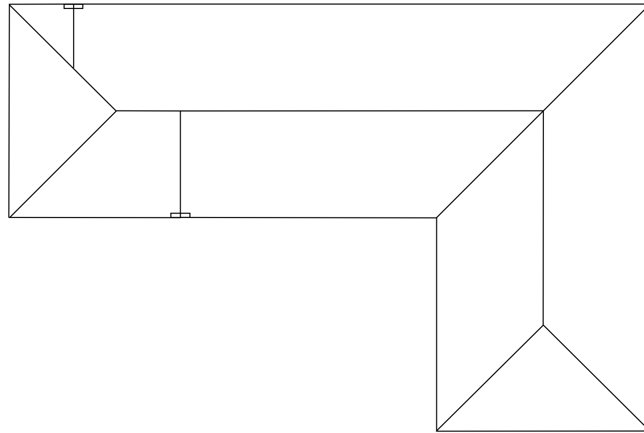


Figure 3.6: Skeleton with doors attached.

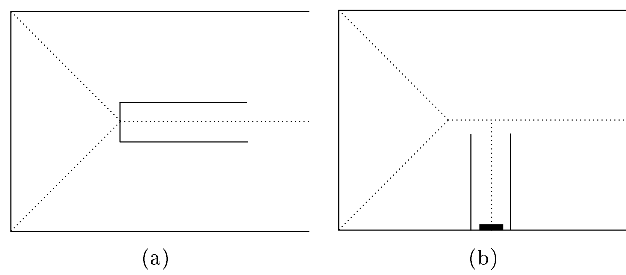


Figure 3.7: Corridor creation due to being far enough from the walls (a) and a door being present (b).

2. If the potential position is too close to another VAA, continue with next door.

3. Add a VAA at the current position.

4. Merge all VAAs with the current corridor (see figure 3.9).

3.5 Regions

After the creation of corridors, the remaining space is split into maximum connected areas, each with a continuous boundary. These are the largest regions which can be populated with rooms or further sub-regions, and are the spaces made up from the initial interior space split by the generated corridors. The entry-level floors in figure 3.10 a-c result in several separate regions each. As can be seen, the number of regions correspond to the number of doors on entry-level floors. Other floors (figure 3.10 d) require an artificial split wall to be added in order to get a maximal region without holes.

The algorithm for creating maximal regions is executed as follows:

1. Create a list of intersections between the transition area and outer wall. If there are no intersections, a splitting wall is inserted, making the minimum number of intersections two.

2. Create a new region.

3. Start on an external wall segment with intersections, and add a wall segment part from the last intersection to the end point.

4. Continue adding wall segments from the outer wall until encountering the next intersection.

5. Start on the corridor wall segment which intersected at the last intersection, and add segments of the corridor wall until returning to the first point in the region wall.

6. Repeat steps 2-5 until all regions have been built (the number of regions is the number of intersections divided by 2, since each region is defined by two intersection points between the outside wall and the corridor wall).

3.6 Sub-regions (Apartments)

When the regions have been created, they are further subdivided into sub-regions (in this case apartments). For apartments, it is important to make sure that each one has at least one window, windowless apartments being very rare.

The algorithm which is used for creating sub-regions is as follows:

1. Create a wall from between the first two windows to the corridor. The windows are ordered counterclockwise around the wall of the region.

2. If the created sub-region is too small, do step 1 again, going one window forwards.

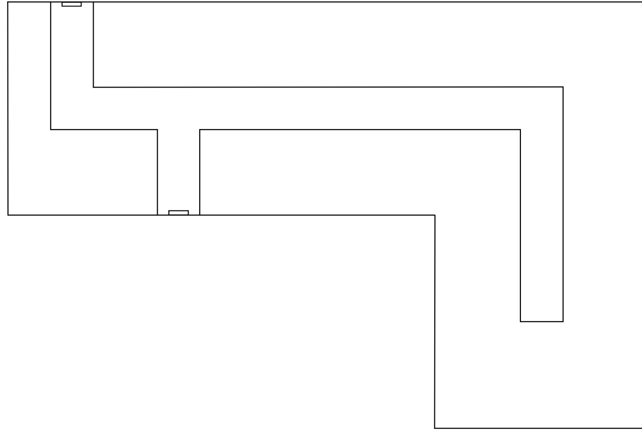


Figure 3.8: Corridor before staircase attachment

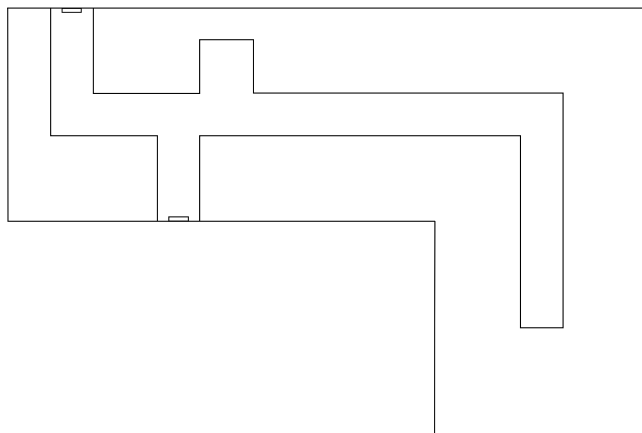


Figure 3.9: Corridor after staircase attachment

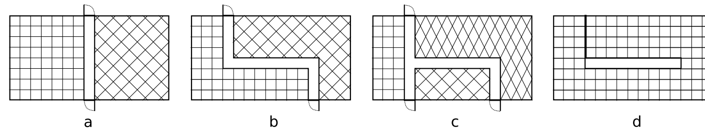


Figure 3.10: Region division for different floors: a) a single corridor segment between two opposite doors create two regions, b) a more complex corridor, but still two regions, c) three doors separate the interior space into three regions, d) a floor without doors results in a single region

3. If this creates a large enough sub-region, leave it as it is.

4. Continue adding sub-regions this way until all the space has been assigned to a sub-region (see figure 3.12).

Of course, there are also some special cases to take care of:

- Make sure sub-regions around corridor ends ("endcap" sub-regions) doesn't create walls along the entire length of the corridor (see figure 3.13).

- If the last sub-region of a region is too small, merge it with the previous sub-region created.

- If the whole region is too small, create an apartment out of the whole region.

- If the region contains no windows, no apartment will be created.

When the apartments have been created, a door is placed between each apartment and the corridor. These doors cannot be placed earlier, since it is not known where the apartments are which would make it hard to ensure that each apartment actually gets a door to the corridor. This means that the sub-region creation has to be done, even if only a view of the corridor is required, since doors need to be correctly placed in the corridor.

In the algorithm presented here, doors can be placed on corridor walls at either end of the wall or in the middle of the wall, to account for a reasonable amount of different possible positions without having to handle too much data (see figure 3.11). For each such position, the distance from that position to each window is summed up. The door is then placed at the position with the highest such sum. This usually leads to a door far away from all windows, which is good since hallways are perfectly acceptable even without windows.

3.7 Room Walls

With sub-regions defined, the actual rooms within them should be created. To do this, a two-part algorithm is used. The first part involves creating a pseudo-voronoi diagram. The problem here lies in the fact that a non-weighted voronoi diagram does not account for different room sizes. However, any true weighted voronoi diagram will have curved lines between the voronoi cells, and that is far more difficult to handle than straight lines. A non-weighted voronoi diagram

on the other hand would not allow the specification of different weights for the seeds, which would lead to very similarly sized rooms regardless of purpose.

The method used here creates an approximate weighted voronoi diagram with straight cell walls. This is accomplished in the following way:

1. Create seeds at each window and the apartment door with weights according to desired room sizes.

2. Create as many more seeds as necessary, again with weights according to desired room sizes (see figure 3.15). In most cases, one room will be created for each seed. The exception is when a room becomes too small, in which case it will be merged with a larger room.

3. Repeat steps 4-9 for each seed, disregarding seeds that has already got a voronoi cell defined. Use the polygon defined by the sub-region walls as the starting area to be divided.

4. Choose the seed that is the furthest from all the other seeds. This is to ensure that when the voronoi region for the chosen seed is removed, the remaining area is continuous. To find the wanted seed, sum up the squared distance between a seed and all other seeds. Choose the seed with the highest such sum.

5. For each other seed, find the weighted point between it and the current seed. This point is found at a distance $D = \frac{k_1}{k_1+k_2}\sqrt{\Delta x^2 + \Delta y^2}$ from the current seed, where k_1 is the weight of the current seed, k_2 is the weight of the other seed, and Δx and Δy is the distance between the seeds along the x-axis and the y-axis, respectively.

6. Create a cutoff line at a right angle to the line between the two seeds (see figure 3.16). The cutoff line should pass through the weighted point form step 4.

7. For each cutoff line, cut away the part of the area that is on the other side of the cutoff line from the seed (see figure 3.17). This will create a voronoi region for the seed.

8. Add the walls of the voronoi region to the voronoi diagram.

9. Remove the voronoi region from the area to be divided, using only the remainder for later iterations.

This creates a voronoiesque diagram (see figure 3.18) that is somewhat dependent on the order in which the seeds are considered, but this is not really a problem, since it is not critical that the voronoi diagram is exactly correct. This is because the voronoi diagram will only be used as a starting approximation of where the final walls should be. It is necessary to make sure that the seeds are traversed in the same order every time, or the same house would not look the same if created again. On the other hand, this is done anyway, since seeds are chosen in a predictable way so that the remainder is a simple polygon.

The voronoiesque diagram has straight, but not well-aligned, cell walls, which brings us to the second part of the room creation algorithm.

The voronoi cell walls should now be aligned so that they are at mainly right angles to the apartment walls (or at least close to right angles). Rooms with difficult to use areas (for instance long, very narrow passages created by walls being too close to each other) and too many corners should also be avoided. Most rooms should have four corners (usually giving a roughly rectangular room), but L-shaped rooms with 6 corners are perfectly acceptable, as long as the extra part of the room isn't too small. This is usually not a problem in a house where the outer walls are at right angles to begin with, but when this isn't the case, some lenience is required when deciding what shapes are acceptable.

To accomplish this alignment, an S-Space wall skeleton is created. To make sure the resolution is high enough, some extra walls are added to the S-Space wall skeleton where there are have too large gaps. From the S-Space skeleton, walls are selected that correspond as well as possible to the voronoi cell walls. More specifically, this is done in the following manner (see figure 3.19):

1. For each voronoi cell wall, find all combinations of wall segments from the S-Space skeleton on which you could project the entire voronoi cell wall. A combination in this case is a set of connected wall segments in the same direction.

2. If there are several combinations that are at an angle lower than a specific threshold, choose from among them the one that lies closest to the voronoi cell wall.

3. Otherwise, choose the combination that is at the least angle to the voronoi cell wall.

This algorithm gives us rooms with walls placed in realistic places, while still maintaining much of the weights from the voronoi diagram (see figure 3.20).

3.8 Room Type Allocation and Entrypoints

When the room walls are created for an apartment, they are used to build the individual rooms. These are then connected to each other through entrypoints (doors), and allocated a room type starting with a room which is connected to the transition area. Entrypoints between rooms are placed in a way so that the useable space in each room is kept high [Alexander et al., 1977] (see figure 3.21).

When constructing rooms from the wall skeleton, a list of neighbours is added to each room. This way, a complete graph with all potential doorways between rooms is built as part of the process. This graph can then be used to classify rooms according to their depth and number of potential access points, using the a,b,c,d-types defined by Hillier (see section 2.2.1). Rooms are then given a type from those given as parameters, for apartments these can be hallway, kitchen, living room, etc.

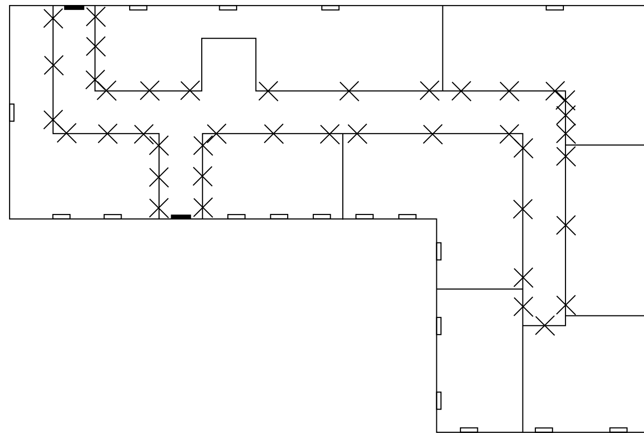


Figure 3.11: Sub-regions with possible apartment door positions marked

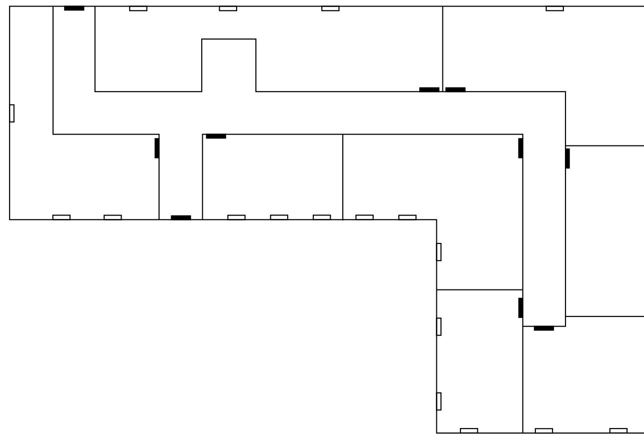


Figure 3.12: House with apartments created

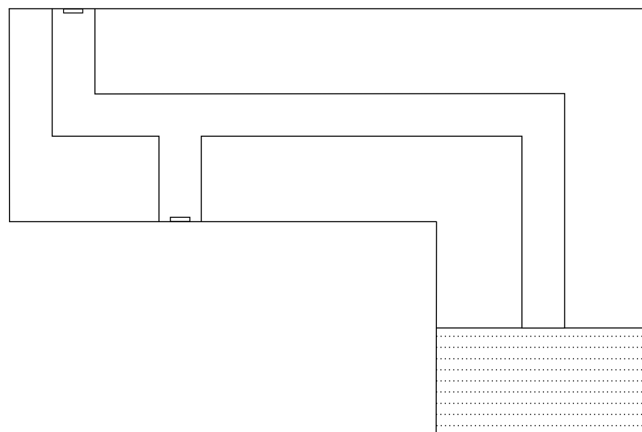


Figure 3.13: Illustration of “endcap” apartment

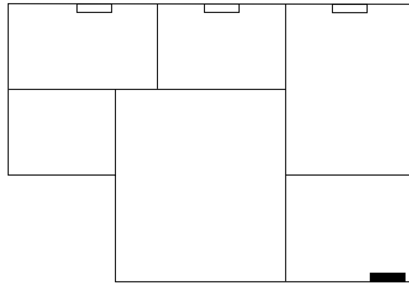


Figure 3.14: Placement of walls between rooms

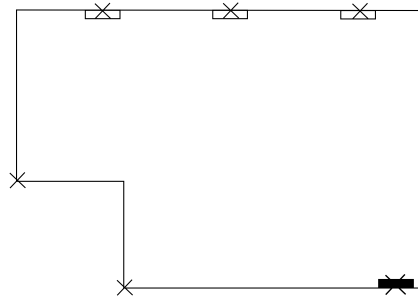


Figure 3.15: Sample apartment with voronoi seeds marked by X.

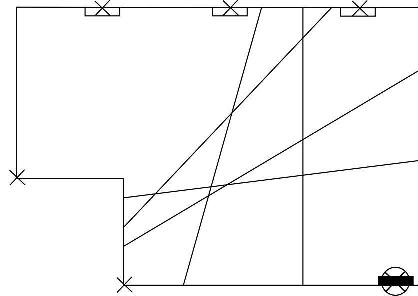


Figure 3.16: Voronoi cutoff lines for first cutoff step. Current seed marked by a ring.

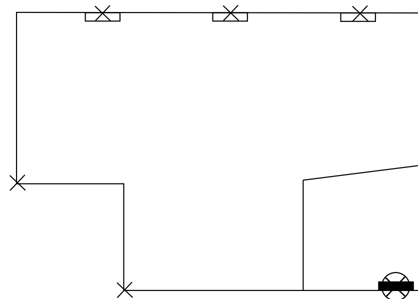


Figure 3.17: First voronoi region created.

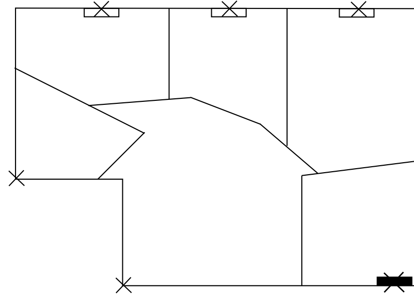


Figure 3.18: All voronoi regions created.

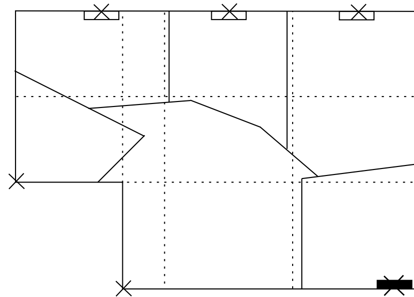


Figure 3.19: Voronoi regions and S-Space skeleton superimposed, with S-Space skeleton as dashed lines.

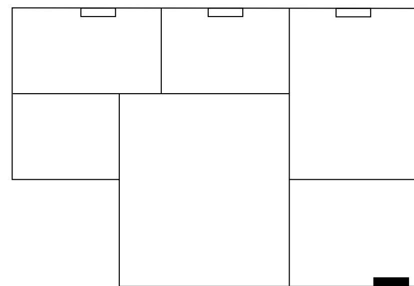


Figure 3.20: The completed apartment.

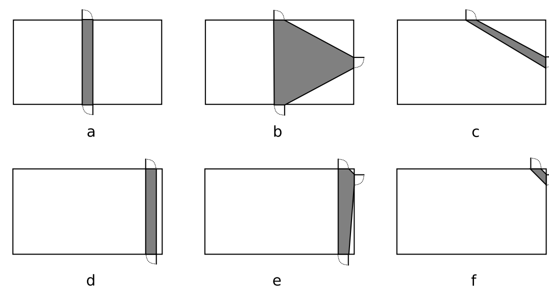


Figure 3.21: Illustration of useable area for different door configurations: The gray regions are areas which lie on the route between two doors, and is thus impractical to use for anything other than movement.
In the first row: a) two doors placed in the middle of opposite walls cause a division of the room into two halves, b) three doors placed in the middle of three different walls cause hard-to-furnish triangular areas, c) two doors on neighbouring walls cause a cutoff of an unnecessarily large area.
Second row: d-f) show how moving the doors in the rooms of a-c) can drastically increase useable space.

Chapter 4

Prototype Implementation

4.1 Initial Considerations

To get an environment suitable for fast implementation and testing of the algorithm, the choice was made to use C++ in conjunction with Lua [Ierusalimschy et al., 2007], which is a scripting language used in a wide range of applications. This enabled testing of changes without restarting the main application, which made the turnaround time very short. The main disadvantages with this were that performance was probably considerably lower than it would have been if the whole algorithm had been implemented purely in C++, and that there was no way of stress testing the algorithm under any kind of real circumstances. Another disadvantage is that if the algorithm will ever actually be used, it will have to be reimplemented from scratch.

An additional goal with the prototype was to allow a graphical representation of the results of the implemented algorithm, as this makes it easier to spot flaws and create visual presentations. As this was a one time implementation that would not require extensive updates, the graphical portion of the application was implemented in C++.

Details of the application design are available in Appendix A.

4.2 Input Data

The input data (as described above) is parsed from XML files containing simple text-based descriptions of the exterior wall, windows and doors, and other data such as floor height and corridor width.

A similar approach is also used for algorithm parameters, which include room and region type definitions. These are organized so that a region type contains a list of names of room types that are valid within such a region. They also contain information about which of these rooms can be used as entrypoint room for that region type, since a region typically has a specific hallway or similar space.

4.3 User Interface

The user interface for the prototype application has two modes (or views):

- First, a two-dimensional view of the generated floor plan, with options to show or hide various layers and loading new building files. Here, the user may also check any information about the generation process in an information panel, as well as click on apartments to generate the rooms inside them (figure 4.1).
- Second, a three-dimensional view of the generated geometry where the user can navigate using keyboard and mouse (figure 4.2).

The user can switch between these modes at any time.

4.4 Script Interface

The prototype application uses the same stages described in chapter3, but divided into two parts:

- The initial stages (parsing input data, creating transition areas and regions) are automatically executed when the user loads a new file.
- Creation of rooms and steps after this are done as a response to user interaction. This allows the user to decide when to generate an apartment.

4.5 Results

The algorithm as it stands works reasonably well as long as the building exterior is not too irregular. If the outer walls are at right angles to each other, the result is usually quite realistic. This may be due to the fact that the algorithm is not limited to rectangular rooms, as well as to the use of voronoi diagrams to get weighted room sizes, which appears to be a unique solution.

It is possible to generate the floor plan of any given apartment without generating the floor plans of any other apartment, which means that you don't have to process any more than you absolutely have to - just generate the apartments the player actually enters. It is also possible to have the algorithm disregard certain areas inside the building. This is mainly useful for hand-designed areas (perhaps story related content in a specific apartment) and stairwells on upper floors (since the stairwells are generated on the ground floor and simply propagated through the other floors).

The algorithm sometimes return undesirable floor plans when the outer walls are not at right angles to each other (see figure 4.3). This is due to the fact that some of the S-Spaces have the same angles as the outside walls. One of the post-processing steps tries to clean up such irregularities as much as possible, but there are cases where it does not help. Further testing and tuning is required to minimize this problem.

Additional results are shown in Appendix B.

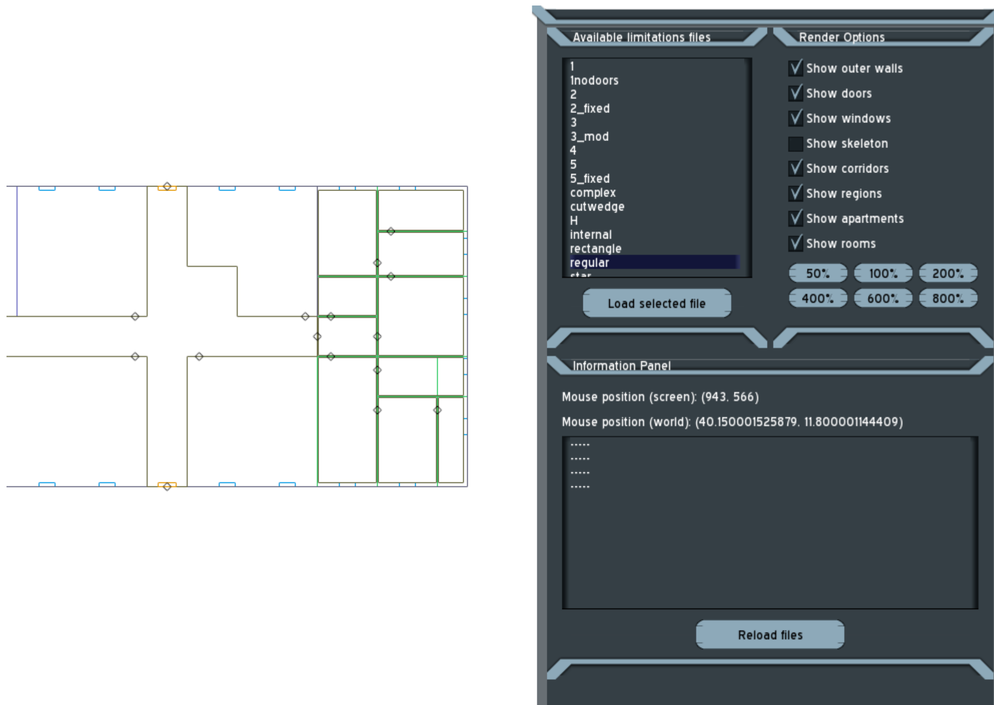


Figure 4.1: Prototype application user interface (floor plan mode)

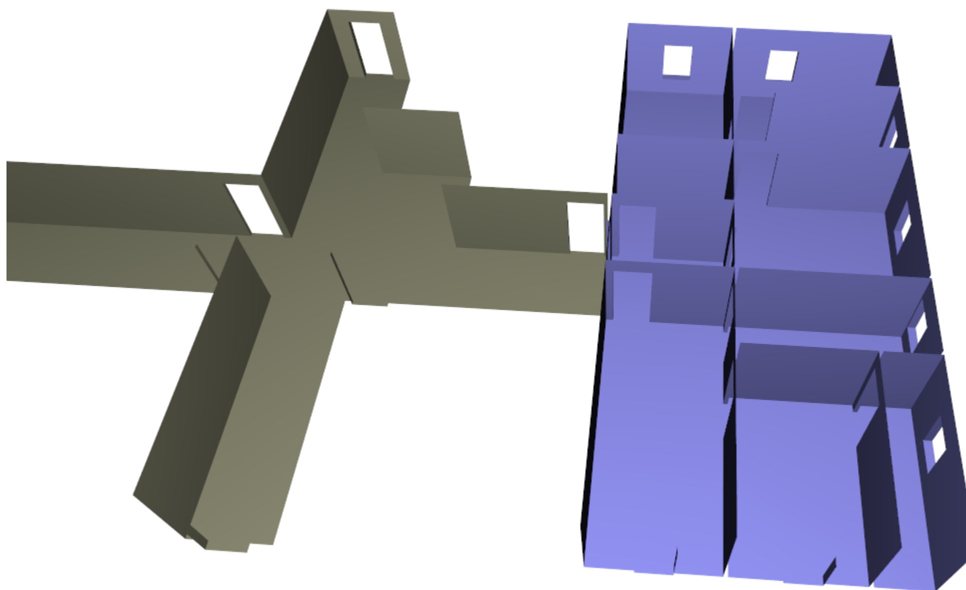


Figure 4.2: Prototype application user interface (walkthrough mode)

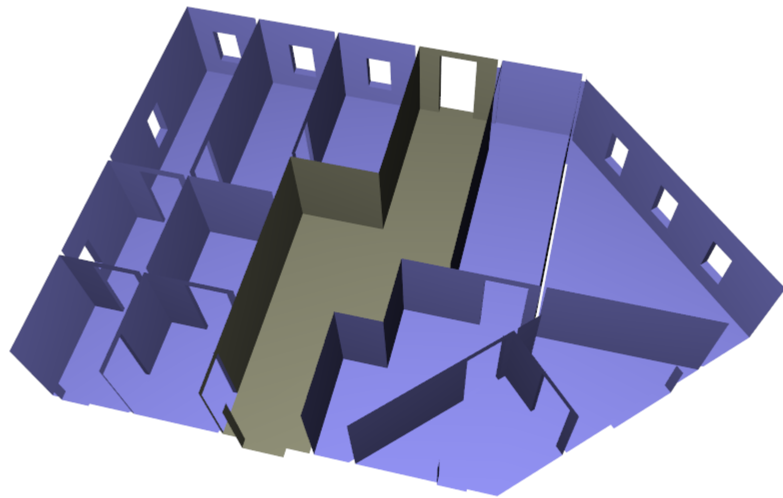


Figure 4.3: Example of building with large angles between wall segments

Chapter 5

Discussion

Looking at the work that has been made earlier in this field, there have been close to nothing like this thesis made before. That is to say, most other researchers have either worked with severely more limited computing power (due to computers being slower during the 1980s than in 2007), or they tackled the problem from another point of view. Bearing this in mind, we had very few actual algorithms to build on. What we had was a lot of architectural analysis and theory. Usually, though, this kind of theory isn't very specific, but is rather formulated in general terms, so applying it to our problem was very much a question of interpretation. All of this meant that we had to design our own algorithms pretty much from scratch, which of course meant a few missteps along the way, along with quite a lot of work spent on algorithms that simply did not work. The end result, though, looks (at least to our eyes) at least as good and usually better than every other automatically generated floor plan we have seen, especially when it comes to houses where the outer walls are not at right angles to each other.

There is, of course, a lot of room for improvement. A lot of tuning has to be done for the specific application the algorithm is going to be used with, and it is probably possible to create more post-processing algorithms, especially for houses with walls at non-right angles. What we have is a working starting point for a more complete algorithm, which was the main goal from the beginning of the project. We concentrated on residential houses that are large enough to need inner corridors, since we considered that to be one of the hardest cases. The reason to choose a hard case rather than to start with simple cases was that a general solution that works in a hard case usually works in simpler cases as well, and we wanted to make sure that the solution was viable in as many cases as possible.

Most of the algorithm will probably work just as well for other types of buildings with the only necessary change being some different parameters specified. However, some parts, notably the region generation and apartment placement, may need to be exchanged. To facilitate swapping out of parts of the algorithm, we designed it in a modular fashion. It should be quite easy to simply exchange the parts you need and keep the rest.

One of the requirements for a floor plan generator is that it should be possible to configure for a wide array of situations. For instance, you might want to have a house with really small rooms and narrow corridors, or you might want a

house with a lot of large open office space. Our algorithm does have some of that configurability. It is possible to specify the least acceptable sizes of apartments as well as wall segments. By changing the room requirements and limitations, you can also make sure there aren't any rooms that are too small. However, it is hard to input any absolute limitations due to the multi-step algorithm and the limits of the S-Space skeleton.

There is also the question of performance. Creating an algorithm that will work in a real-time environment have been a consideration right from the start. That said, it was never the goal of this thesis work to create the actual real time application, but rather to make sure the algorithm we did create wouldn't take too much time. Using Lua didn't allow for precise performance testing, but we assumed that as long as the Lua version wasn't too slow, performance when converted to C++ probably wouldn't be a problem. There are also quite a lot of performance issues due to a lot of rewritten code - parts of the algorithm had to be changed again and again, leading to suboptimal solutions in a number of places. These issues can probably be overcome quite easily when transferring the whole algorithm to C++, since it will then be possible to plan the code a lot better in advance.

Chapter 6

Further Development

6.1 Improvements

The algorithm is not, and was never intended to be, the end solution to the problem of generating interior spaces. It should be viewed as a starting point for future work rather than a finished product. With this in mind, there are a number of things that need improvements. The underlying algorithm is solid, but the exact details of the implementation may need some changes. Possible improvements include:

- Choose only S-Space walls from concave corners, or at least only from corners with an angle larger than some value. This would alleviate the issue with wedge-shaped rooms.
- Select S-Space segments that lie along the *main direction* of the apartment (this could for example be defined by the most common direction of the apartment walls). This could also alleviate the problem with wedge-shaped rooms, but defining the main direction may not be an easy task.
- Prune the building skeleton before creating the corridors so it only contains edges along which corridors should actually be created. This would simplify and speed up the corridor construction, as well as the creation of auxiliary data.

6.1.1 Rules and parameters

The prototype that resulted from this thesis uses a number of parameters for modifying the results of the algorithm, but these are on a relatively low level. Future development of high-level parameters as well as a higher level interface for adjusting the structure of generated spaces would be beneficial.

6.1.2 Randomization

In addition to extended parameters, a more extensive use of seeded randomization would allow for a much higher degree of variation, especially in the case of many similar exteriors.

6.1.3 Performance

Since the current prototype was implemented in a mix of C++ and Lua code, actual performance will be improved in a pure C++ implementation. Certain algorithms can also be improved at the design level (such as graph traversal in room wall construction). Some of the sub-algorithms used can also be replaced with more efficient but more complex versions (such as the triangulation algorithm used for constructing floor and ceiling geometry).

6.2 Future Work

Part of our work on generation of interiors was to compile a list of interesting areas for further study. There are a large number of paths that can be followed from this initial work, but the following are those we have deemed would provide the most valuable additions.

6.2.1 Integration with building generators

A number of issues can be avoided by considering the following when integrating the interior generation with a system creating building exteriors:

- Outer door positioning depending on house shape: Placing outer doors in the middle of a short wall, or at the end of a wall with a sharp angle could make it impossible to generate a realistic (or even navigable) interior.

- Doors should be placed at least at a minimum distance (which corresponds to the outer wall thickness plus half the corridor width) from a corner to avoid interior walls being too thin or going outside the exterior wall.

- Staircase placement generated from outer window placement: If the exterior has half-story offset windows as typically used in staircases, stairs should be placed there instead of in the building interior.

6.2.2 Generalization to cover other building types

The scope of this thesis was limited to a particular building type in order to restrict the problem to a manageable size. Future work in generalizing our algorithms would be very valuable for any applications where varied building types are required.

6.2.3 Generation of auxiliary data

One of the largest positive aspects of procedural interiors as we see it is the possibility of simple and efficient generation of additional data as part of the construction process. This can include pathfinding graphs, occlusion culling information, or zone division for lighting algorithms.

6.2.4 Automatic placement of furniture and decorative items

A large part of the experience of an interior space is how it is furnished and decorated. A project looking at how to select (or even generate) appropriate furniture and then place it in a natural way would be invaluable to a complete procedural generation of interiors. A number of things that are likely to be of particular interest are:

- Using the generated room layouts to place furniture and other items in a realistic-looking manner.
- Using the useable areas of a room (the areas not required for movement) defined by doors as a valid area for placing furniture.
- Consider the tradeoff between the realism of individual furnishing and performance gain of a small number of furniture types.
- Using model synthesis approaches to generate varying furniture [Merrell, 2007].

6.2.5 Integration with level design tools

Integration of a generalized algorithm with existing level design and modeling applications would potentially provide a large time-saving feature for content creation, as level designers can get a starting point for an entire building at the touch of a button.

6.2.6 Movement and space in simulated versus real interior environments

Another interesting area which is related to achieving quality results with any interior generation algorithm is the study of how movement differ between real-world and virtual buildings. Questions of particular interest are:

- Do virtual environments generally require more space due to constraints in movement control?
- How are space requirements affected by camera perspective? Compare, for example, interior spaces in existing third person (e.g. Max Payne) and first person (e.g. Half-Life) games.

Recent work of interest include articles in the field of Spatial Cognition [BremenUniversity, 2008] and on Space Syntax.

6.2.7 Automatic texturing of interiors

Applying textures to the generated interior in a convincing way is important to get convincing spaces. Also, studying how textures and color schemes affect viewers will be vital in order to achieve the goal of high-level descriptions of space properties. Areas of interest include:

- Choosing or generating textures based on the type of building and desired mood of a region.
- Consulting studies on the effects of properties such as color, brightness and patterns on how a space is experienced.

Bibliography

Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

P. J. Birch, S. P. Browne, V. J. Jennings, A. M. Day, and D. B. Arnold. Rapid procedural-modelling of architectural structures. In *VAST '01: Proceedings of the 2001 conference on Virtual reality, archeology, and cultural heritage*, pages 187–196. ACM, 2001. doi: http://doi.acm.org/10.1145/584993.585023.

David Braben and Ian Bell. Elite. `http://www.reference.com/browse/wiki/Elite_(video_game)`, 1984. URL `http://www.reference.com/browse/wiki/Elite_(video_game)`. Accessed on December 19, 2007.

BremenUniversity. The spatial cognition website. `http://www.spatial-cognition.de/`, 2008. URL `http://www.spatial-cognition.de/`. Accessed on January 11, 2008.

C. Brenner. Towards fully automatic generation of city models. *International Archives of Photogrammetry and Remote Sensing*, 33, Part B3:85–92, July 16-23 2000.

Petr Felkel and Stepan Obdrzalek. Straight skeleton implementation. In László Szirmay Kalos, editor, *14th Spring Conference on Computer Graphics*, pages 210–218, 1998.

P. A. Flack, J. Willmott, S. P. Browne, D. B. Arnold, and A. M. Day. Scene assembly for large scale urban reconstructions. In *VAST '01: Proceedings of the 2001 conference on Virtual reality, archeology, and cultural heritage*, pages 227–234. ACM, 2001. doi: http://doi.acm.org/10.1145/584993.585029.

Per Galle. An algorithm for exhaustive generation of building floor plans. *Communications of the ACM*, 24(12):813–825, December 1981. doi: http://doi.acm.org/10.1145/358800.358804.

Stefan Greuter, Jeremy Parker, Nigel Stewart, and Geoff Leach. Real-time procedural generation of 'pseudo infinite' cities. In *GRAPHITE '03: Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 87–ff, New York, NY, USA, 2003. ACM, ACM Press. ISBN 1581135785. doi: 10.1145/604471.604490. URL `http://portal.acm.org/citation.cfm?id=604490`.

Evan Hahn, Prosenjit Bose, and Anthony Whitehead. Persistent realtime building interior generation. In *Sandbox Symposium 2006*, pages 179–186. ACM, 2006.

Bill Hillier. *Space is the Machine: A Configurational Theory of Architecture*. Space Syntax, 2007 electronic edition, 2007.

Bill Hillier and Julienne Hanson. *The Social Logic of Space*. Cambridge University Press, 1984.

Roberto Ierusalimschy, Waldemar Celes, and Luiz Henrique de Figueiredo. The lua scripting language. `http://www.lua.org`, 2007. URL `http://www.lua.org`. Accessed on January 11, 2008.

Mathieu Larive and Veronique Gaildrat. Wall grammar for building generation. In *GRAPHITE '06: Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia*, pages 429–437. ACM, 2006. doi: http://doi.acm.org/10.1145/1174429.1174501.

R. G. Laycock and A. M. Day. Automatically generating large urban environments based on the footprint data of buildings. In *SM '03: Proceedings of the eighth ACM symposium on Solid modeling and applications*, pages 346–351. ACM, 2003. doi: http://doi.acm.org/10.1145/781606.781663.

Jess Martin. Algorithmic beauty of buildings - methods for procedural building generation. Master's thesis, Trinity University, 2005. Computer Science Honors Theses.

Paul Merrell. Example-based model synthesis. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 105–112, New York, NY, USA, 2007. ACM. doi: http://doi.acm.org/10.1145/1230100.1230119.

Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. In *ACM SIGRAPH 2006*. ACM, 2006.

John Noel. Dynamic building plan generation. Master's thesis, University of Sheffield, 2003. Undergraduate project Dissertation, Department of Computer Science.

Yoav I. H. Parish and Pascal Müller. Procedural modeling of cities. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 301–308, New York, NY, USA, 2001. ACM. ISBN 1-58113-374-X. doi: http://doi.acm.org/10.1145/383259.383292.

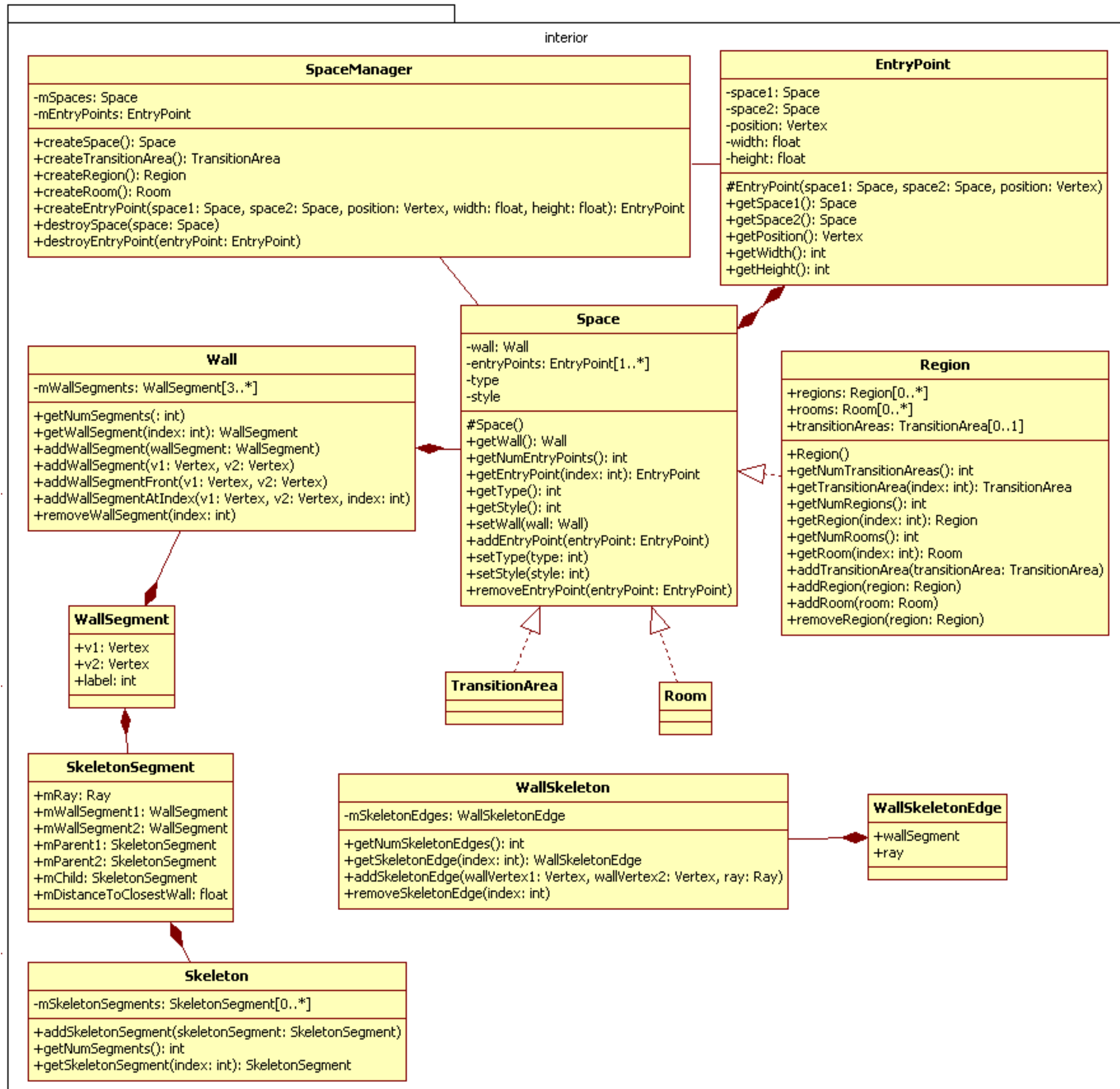
John Peponis, Jean Wineman, Mahbub Rashid, S Kim, and Sonit Bafna. On the description of shape and spatial configuration inside buildings. In *Space Syntax First International Symposium*, 1997.

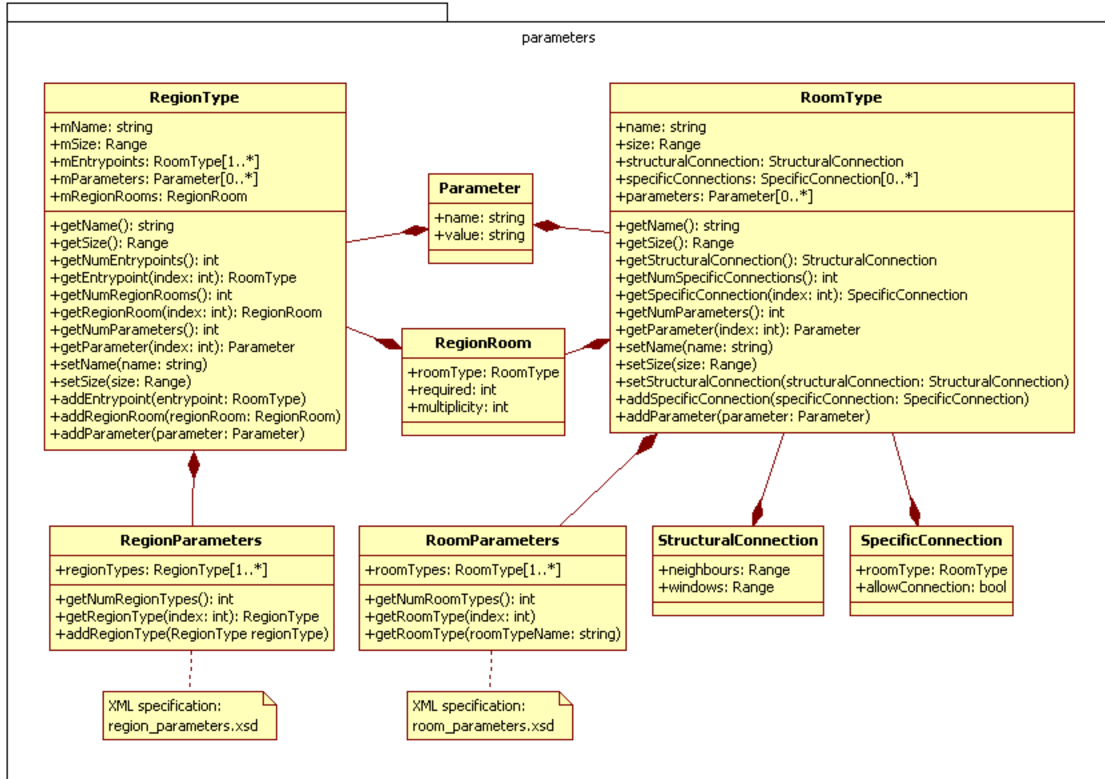
Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. Instant architecture. *ACM Transactions on Graphics*, 22(4):669–677, july 2003. URL `http://artis.inrialpes.fr/Publications/2003/WWSR03`. Proceeding.

Appendix A

Prototype Design

This appendix contains class diagrams describing the main classes used in the prototype implementation.


interior
SpaceManager
-mSpaces: Space
-mEntryPoints: EntryPoint
+createSpace(): Space
+createTransitionArea(): TransitionArea
+createRegion(): Region
+createRoom(): Room
+createEntryPoint(space1: Space, space2: Space, position: Vertex, width: float, height: float): EntryPoint
+destroySpace(space: Space)
+destroyEntryPoint(entryPoint: EntryPoint)
EntryPoint
-space1: Space
-space2: Space
-position: Vertex
-width: float
-height: float
#EntryPoint(space1: Space, space2: Space, position: Vertex)
+getSpace1(): Space
+getSpace2(): Space
+getPosition(): Vertex
+getWidth(): int
+getHeight(): int
Space
-wall: Wall
-entryPoints: EntryPoint[1..*]
-type
-style
#Space()
+getWall(): Wall
+getNumEntryPoints(): int
+getEntryPoint(index: int): EntryPoint
+getType(): int
+getStyle(): int
+setWall(wall: Wall)
+addEntryPoint(entryPoint: EntryPoint)
+setType(type: int)
+setStyle(style: int)
+removeEntryPoint(entryPoint: EntryPoint)
Wall
-mWallSegments: WallSegment[3..*]
+getNumSegments(: int)
+getWallSegment(index: int): WallSegment
+addWallSegment(wallSegment: WallSegment)
+addWallSegment(v1: Vertex, v2: Vertex)
+addWallSegmentFront(v1: Vertex, v2: Vertex)
+addWallSegmentAtIndex(v1: Vertex, v2: Vertex, index: int)
+removeWallSegment(index: int)
Region
+regions: Region[0..*]
+rooms: Room[0..*]
+transitionAreas: TransitionArea[0..1]
+Region()
+getNumTransitionAreas(): int
+getTransitionArea(index: int): TransitionArea
+getNumRegions(): int
+getRegion(index: int): Region
+getNumRooms(): int
+getRoom(index: int): Room
+addTransitionArea(transitionArea: TransitionArea)
+addRegion(region: Region)
+addRoom(room: Room)
+removeRegion(region: Region)
WallSegment
+v1: Vertex
+v2: Vertex
+label: int
TransitionArea
Room
SkeletonSegment
+mRay: Ray
+mWallSegment1: WallSegment
+mWallSegment2: WallSegment
+mParent1: SkeletonSegment
+mParent2: SkeletonSegment
+mChild: SkeletonSegment
+mDistanceToClosestWall: float
WallSkeleton
-mSkeletonEdges: WallSkeletonEdge
+getNumSkeletonEdges(): int
+getSkeletonEdge(index: int): WallSkeletonEdge
+addSkeletonEdge(wallVertex1: Vertex, wallVertex2: Vertex, ray: Ray)
+removeSkeletonEdge(index: int)
WallSkeletonEdge
+wallSegment
+ray
Skeleton
-mSkeletonSegments: SkeletonSegment[0..*]
+addSkeletonSegment(skeletonSegment: SkeletonSegment)
+getNumSegments(): int
+getSkeletonSegment(index: int): SkeletonSegment


parameters
RegionType
+mName: string
+mSize: Range
+mEntrypoints: RoomType[1..*]
+mParameters: Parameter[0..*]
+mRegionRooms: RegionRoom
+getName(): string
+getSize(): Range
+getNumEntrypoints(): int
+getEntrypoint(index: int): RoomType
+getNumRegionRooms(): int
+getRegionRoom(index: int): RegionRoom
+getNumParameters(): int
+getParameter(index: int): Parameter
+setName(name: string)
+setSize(size: Range)
+addEntrypoint(entrypoint: RoomType)
+addRegionRoom(regionRoom: RegionRoom)
+addParameter(parameter: Parameter)
Parameter
+name: string
+value: string
RegionRoom
+roomType: RoomType
+required: int
+multiplicity: int
RoomType
+name: string
+size: Range
+structuralConnection: StructuralConnection
+specificConnections: SpecificConnection[0..*]
+parameters: Parameter[0..*]
+getName(): string
+getSize(): Range
+getStructuralConnection(): StructuralConnection
+getNumSpecificConnections(): int
+getSpecificConnection(index: int): SpecificConnection
+getNumParameters(): int
+getParameter(index: int): Parameter
+setName(name: string)
+setSize(size: Range)
+setStructuralConnection(structuralConnection: StructuralConnection)
+addSpecificConnection(specificConnection: SpecificConnection)
+addParameter(parameter: Parameter)
RegionParameters
+regionTypes: RegionType[1..*]
+getNumRegionTypes(): int
+getRegionType(index: int): RegionType
+addRegionType(RegionType regionType)
RoomParameters
+roomTypes: RoomType[1..*]
+getNumRoomTypes(): int
+getRoomType(index: int)
+getRoomType(roomTypeName: string)
StructuralConnection
+neighbours: Range
+windows: Range
SpecificConnection
+roomType: RoomType
+allowConnection: bool
XML specification:
region_parameters.xsd
XML specification:
room_parameters.xsd

Appendix B

Additional Results

This appendix contains a collection of examples illustrating the results of applying the interior generation algorithm to a set of buildings.

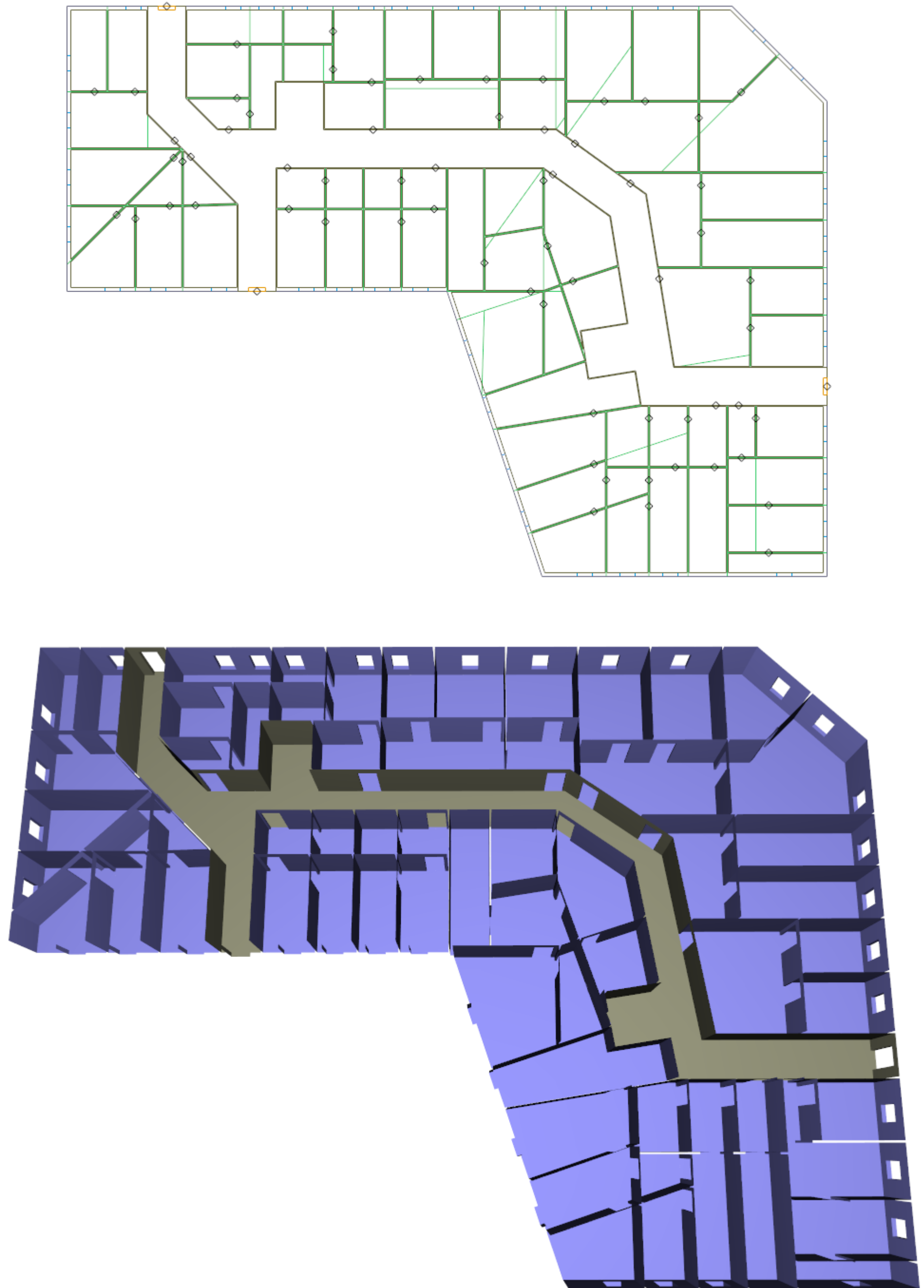
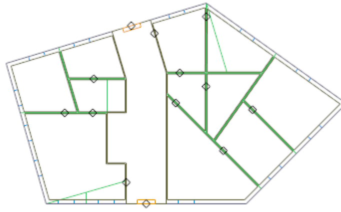


Figure B.1: Result of running the algorithm on one sample irregular building



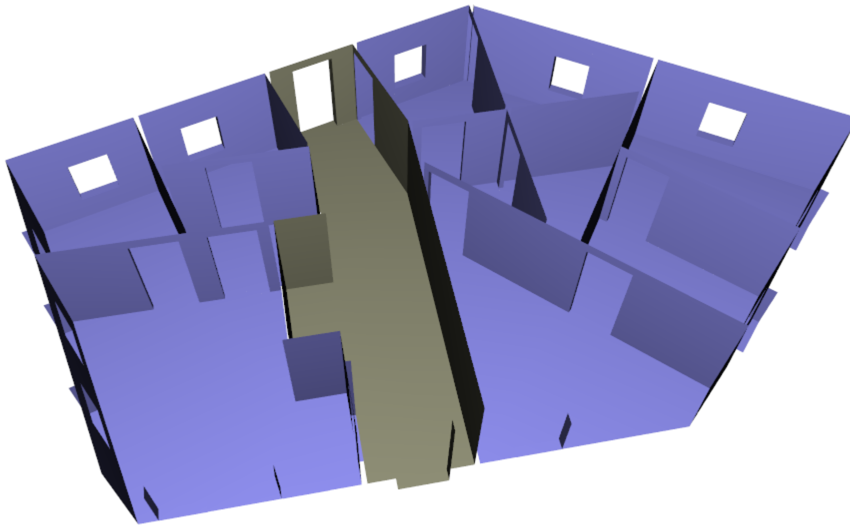


Figure B.2: Small building with only two doors, and with only one right angle