MASTER'S THESIS | LUND UNIVERSITY 2017

General Methods for the Generation of Seamless Procedural Cities

Tobias Elinder

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2017-30

Lunds Tekniska Högskola
Datateknik

General Methods for the Generation of Seamless Procedural Cities

Tobias Elinder

Contents

Abstract **1**

Acknowledgements **3**

1 Introduction **5**

1.1 Motivation and Objectives 5

1.2 Statement of Originality 6

2 Background Theory **7**

2.1 Introduction 7

2.2 Practical Outline 8

2.3 Scope of thesis 8

2.4 Theory 9

2.4.1 Related Work & Approach 9

2.4.2 Reference cities 11

3 Algorithm **13**

3.1 Road Generation 15

3.1.1 Input 15

3.1.2 Output . . . 15
3.1.3 Description . . . 16
3.1.4 Heatmap . . . 16
3.1.5 Plot Polygons . . . 17
3.2 Plot Generation . . . 17
3.2.1 Input . . . 17
3.2.2 Output . . . 17
3.2.3 Description . . . 17
3.3 Sidewalk Generation . . . 18
3.3.1 Input . . . 18
3.3.2 Output . . . 18
3.3.3 Description . . . 18
3.4 Building Generation . . . 18
3.4.1 Input . . . 18
3.4.2 Output . . . 19
3.4.3 Description . . . 19
3.5 Apartment Generation . . . 20
3.5.1 Input . . . 20
3.5.2 Output . . . 20
3.5.3 Description . . . 20
3.6 Furniture Placement . . . 22
3.6.1 Input . . . 22
3.6.2 Output . . . 22
3.6.3 Description . . . 22

4 Implementation **23**

4.1 General . . . 23

4.2 Road Generation . . . 23

4.3 Noise . . . 24

4.3.1 Perlin-noise . . . 24

4.3.2 Heatmap . . . 25

4.4 Polygon Splitting . . . 25

4.5 UV-mapping . . . 26

4.6 Generating plot polygons from road network . . . 27

4.7 Building height . . . 27

4.8 Building shapes . . . 29

4.9 Mesh Placement . . . 30

4.10 Performance . . . 30

5 Results **33**

5.1 City Structure . . . 34

5.2 Apartment Structure . . . 37

5.3 Windows . . . 37

5.4 Furnishing . . . 38

5.5 Comparison to Reference cities . . . 38

5.5.1 Recreating Manhattan . . . 38

6 Conclusion **43**

6.1 Summary of Thesis Achievements . . . 43

6.2 Returning to Reference Cities . . . 44

6.3 Applications 44

6.4 Future Work 45

6.4.1 Improvements 45

6.4.2 Unexplored areas 46

6.5 Final Thoughts 47

.1 Appendix A 48

.1 Appendix B 54

Bibliography **54**

List of Figures

3.1 Main structure diagram 14

4.1 Sample output from the single-banded Perlin function 25

4.2 Sample user-created heatmap 25

4.3 Calculating $general\_modifier$ for $noise\_mp = 1$, x is a random stochastic variable between 0 and 1 . 28

4.4 Calculating $general\_modifier$ for $noise\_mp = 0.5$, x is a random stochastic variable between 0 and 1 29

5.1 Screenshot at ground level 34

5.2 Screenshot from inside a meeting room in an office 35

5.3 Heatmaps 35

5.4 Resulting cities 36

5.5 Changing turn rates 36

5.6 Noise based cities 36

5.7 Simple Apartments 37

5.8 Exteriors 38

5.9 Apartment subdivision for a simple small building floor 39

5.10 Apartment subdivision for a larger and somewhat more complicated building floor 39

5.11 A kitchen with furniture 40

5.12 Real Cities 40

5.13 Procedurally Generated Cities 40

5.15 Generated city, type = Grid like 41

1 Apartment with room requirements 49

2 Apartment after one split 50

3 Apartment after two splits, Bedroom placed 51

4 Apartment after three splits, Bathroom placed 52

5 Finished apartment, all rooms placed 53

6 Roads with lines placed on both sides, direction is reversed for opposing sides 54

7 Lines are split whenever intersecting the middle of a road, recursively creating new lines 55

8 Lines are checked for collisions against each other, intersection points are saved as polygon points. Each line keeps track of potential parent line and child line. 56

9 In every polygon where the last point has the first point as child, a complete polygon and is formed and saved. In this limited example, two polygons are created 57

Abstract

Procedural generation as a concept is as old as computer graphics. It is usually defined as a method for creating data algorithmically as opposed to manually. Work in this area often revolves around noise functions and large degrees of randomness, which works well for chaotic structures and for generating natural environments, but it does not always suit more sophisticated and coherent environments. This thesis proposes some new approaches for procedural generation, specifically on the subject of cities. Many of the solutions discussed are applicable in other areas of procedural generation where order and coherency is important as well. Work on procedural cities has been done before, but the scope has usually been limited to one or a few aspects of the city. For example the general structure of the city and exteriors [1], building shapes [5], the room layout [2], or the interior of a single room [9]. This project uses several of these ideas, combines them into a composite whole, and adds things like different building types (office, apartment), different rooms (bathroom, living room etc), and further specifics.

This thesis will discuss the different methods employed as well as their inherent strengths and weaknesses.

Acknowledgements

I would like to express sincere thanks to:

Professor Michael Doggett, my supervisor during this project and a great support in every regard, who provided valuable comments and suggestions.

Liselotte Schäfer Elinder, for proofreading.

Family and friends, for support, opinions, and constructive criticism.

Chapter 1

Introduction

1.1 Motivation and Objectives

Before getting invested in computer graphics, my main area of interest within computer science was AI and more specifically applied problem-solving algorithms. This manifested itself in a couple of side projects with game AI:s and similar subjects.

After some learning on my own and some university courses I started looking into simple procedural generation. I came to realize the width of the subject, as well as how sprawling and multifaceted the term is.

I knew that I wanted to do something quite independently and went to my former professor in graphics Michael Doggett to talk about possible subjects. We ended up talking about procedural generation, and more specifically cities. Procedural cities is nothing new, but other efforts have been limited in scope in a way that we wanted to overcome. We talked about different concepts for making cities ground-up with interiors for the buildings and ended up with the goal document for the project.

In essence the vision of the project was to achieve a procedurally generated city without "illusions", meaning cities where there are no walls textured with details that are only superficial. A window on the outside of any building should correspond to a window looking out of a room inside of that building. This demand for coherence proved to be the project's biggest challenge.

1.2 Statement of Originality

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyones copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

Chapter 2

Background Theory

2.1 Introduction

Procedural generation of environments is a very interesting field because of its immense potential for size and unpredictability. Its primary uses are games and movies with huge vivid worlds. Procedural generation is perhaps an area that seems easier on the surface than it is to actually work with. The constant weighing of order against chaos in order for environments to appear plausible but not too predictable is something everyone who has tried his/her hand on procedural generation should recognize. It makes every decision a careful planning act.

Lately, increased computer performance has allowed larger environments to be stored in memory, arguably reducing the need for procedural generation in certain areas. However the cost of creating content in modern games has also increased by an order of magnitude, which in contrast makes procedural generation more appealing, so there are valid arguments to be made both for and against this approach. The game No Man's Sky[11] managed to generate a lot of interest in its procedurally generated terrain, but turned out a disappointment, possibly hurting the publics general perception of procedural generation. Another possible deterrent against wide-scale adoption of procedural generation could be the streamlining of the process with which virtual worlds are created today, often in engines that work well only with static pre-determined meshes and environments. The biggest problem of all with procedural generation however is simple; it's not simple. On a small scale, anything created by procedural algorithms can be created easier without it.

Cities are large and very complex, making them in theory a suitable but difficult target for procedural generation. A good example of where this could be used in practice is the GTA series[13], these games span huge worlds and

large cities, which is all handcrafted, requiring a huge effort. Maybe including procedural generation in some areas could reduce costs and improve the final products.

Work has been done on cities, city layout and creation of individual building previously. There have also been some attempts at generating rooms from building shapes. This project attempts to combine some of the ideas from these previous projects.

2.2 Practical Outline

More specifically, procedurally generating environments in this case refers to the automatic placement of polygons and static meshes. A polygon is a two-dimensional shape that can span up an arbitrary number of points, combining very many of these polygons at different angles forms the basis for most 3D environments. Static meshes are models where the polygons are already set in relation to each other, for example in the form of a chair. These polygons and static meshes are what everything in the project boils down to. All the abstractions discussed simply aim to provide a way to decide the properties of these polygons and static meshes.

This project was implemented in Unreal Engine 4. Most of the content is pure C++, and could probably have been implemented without the help of a game-engine. There are a couple of benefits with using an existing engine however, the biggest being the reduced amount of code needed to get the project going, as well as performance. Placing many polygons and static meshes into the world quickly becomes very computationally intensive, Unreal Engine (as well as other game-engines) is very good at optimizing and reducing the workload by for example removing polygons that can't be seen. For presentation purposes Unreal Engine also offers textures, materials and lighting which makes the scene look a lot better. One reason for choosing this engine was that the author was familiar with it.

2.3 Scope of thesis

Since the expressed goal of the thesis is the development of new methods, and the refinement of older ones, the exact scope remained somewhat scalable through a large part of the project. The key parts have remained the same though, which has been finding methods for a complete top-down generation of cities, from the general city structure down to specific furniture placement. After this functionality was achieved on a basic level, focus was instead shifted towards the generalization of the methods used, and widening the scope for the different methods.

In order to reduce the complexity of the problem, some specific constraints for the project were decided early on. These were:

- The whole city is situated on a completely flat surface.
- The city structure can be generated before everything else.
- There should be four types of interiors: office, living, shop, and restaurant. These environments do not need to be very detailed, but their implementations should follow a scalable and interchangeable system that can easily be expanded.
- The stairwells and elevators in buildings are allowed to follow simple pre-determined patterns.
- Apartments and rooms should be defined only two-dimensionally.

A few key questions were asked ahead of this project, which will be addressed in this thesis:

- Is it possible to generate the whole city procedurally? If not, why?
- At which level of the process is it necessary to fall back to predetermined objects and structures?
- Will it be possible to find general algorithms that cover a wide variety of cities? Or will they have to be tailor made to specific types of cities and buildings?
- In terms of performance and scalability, are these algorithms feasible on a larger scale?

2.4 Theory

2.4.1 Related Work & Approach

Even though much potentially falls under the umbrella-term that is procedural generation, in many cases different projects have very little in common. Procedural generation within different domains face very different challenges, and are usually solved in very different ways. Generating content and goals for games procedurally can for example result in goals being impossible to achieve. Scenarios like this has to either be checked post-generation and fixed, or such conditions must be fundamentally impossible, achieving this can be very hard however. In our case, for the cities and for procedural generation attempting some sort of realism in general, a big problem is implausible environments that don't quite make sense logically, we will return to this subject in later chapters.

Finding literature that easily fits into the work flow of the project turned out quite hard. The fact that there is so much information available about certain aspects (e.g. city structure) and so little about other (e.g. interiors) made things complicated. As an introduction to several of the problems inherent to procedural cities, a survey by George Kelly and Hugh McCabe [8] did a good job of presenting it simply and was a good place to start.

George and Kelly also created the project citygen[10], this project combines manual and automatic work. It does this by using procedural algorithms more as tools rather than a foundation in order to give the user more control in creating the city. This project does not handle interiors.

Another interesting, contemporary project is Esri CityEngine[12]. This is a very sophisticated program that also seeks to combine manual and automatic labor to create prototype cities. These cities are used primarily for urban planning. CityEngine uses procedural modeling heavily via its custom grammars; CGA, which can be written by the users. This project does however also not handle interiors in any way.

The single most valuable literature available for this project, and was the seminal paper by Parish and Müller which outlines a method of generating road placement and city structure procedurally [1]. The method uses L-Systems[1] and has since its inception been refined into a much simpler and more effective version by Sean Barrett [3] that uses a priority queue[2] instead, originally in a blog post. This refined version is the method the road generation in this project was based on.

Modeling of buildings procedurally is a problem which becomes much harder when coherence of exteriors and interiors, as discussed earlier, has to be achieved. Good work on procedural exteriors (without interiors) exists [5], but it was decided early on that imposing the conditions of the exterior onto interiors would put too many limitations on the interiors. Because of this, it was decided that the exterior facade should be generated by the room generator itself, in a way generating buildings from the inside - out, this approach made it hard to use previous work at all for the building modeling. The chosen approach has some obvious drawbacks as well as some large benefits. The main drawback is performance, in order to render the outside of a building the inside has to be generated first. A large benefit however is that this allows interiors to "make more sense", adding windows where it's logical, more like in real buildings. It also alleviates some of the burden already put on interior generation since rooms have to be generated from arbitrary shapes.

Regarding interiors of buildings, there was actually a surprising lack of information, and although there have been projects done in this area [2] [4], none seemed to work well within this context, or the ideas proposed weren't simple enough to fit with this work flow. Although Dahl & Rinde's work [2] seemed to have produced good

[1] A recursive grammar system first used to mathematically describe plants.
[2] An array that is always sorted where items can be placed and removed efficiently.

results (especially regarding corridor placement), it seemed hard to combine with what we wanted to achieve. An important part of this project was to create an interior-generating system which allowed easily modifiable room and apartment - descriptions, based on user-specified rooms and apartments. It was not obvious how one could combine these goals with previous work. In the end a completely new approach was settled upon. This approach starts with a description of rooms that should exist within the apartment, and recursively splits the apartment into smaller and smaller pieces, assigning rooms to fitting parts. This is a simple and fast algorithm that gives a very natural look at first glance, but does have some weaknesses, which will be discussed later.

All in all, it was clear that large parts of the project would have to be built from the ground, but with good inspiration from previous work.

2.4.2 Reference cities

In order to have something to work towards, a few cities were chosen as reference cities. The patterns identified and used in the project were found in the following cities.

- New York (specifically Manhattan)
- Central Los Angeles
- Urban Malmö

Results were compared to these reference cities in order to find improvements. Their layout and defining characteristics have had a major impact on the final program. New York and Los Angeles have a lot more in common with each other than with Malmö which has resulted in a program that is best suited for dense cities.

Chapter 3

Algorithm

The logical structure that is responsible for building the city is based on a very modular top-down approach. The city is built one step at a time, beginning with the most fundamental and then adding more and more detail. The top-down approach means that information flows only downward. This was decided because of the increased clarity and decreased complexity it brings. It does however mean that some limitations are imposed. The top-down approach means that every step has to operate within certain limits and according to certain requirements. Having steps that are well defined in their constraints, their expected input, and their expected output also means that they can be used independently outside of the current program context. This top-down approach is strictly logical, and does not imply anything about where geometry is actually generated in the program. For the specific steps in this structure, see 3.1.

Roads are generated before anything else. It can be argued that this is not always correct because roads in many (especially older) cities are younger than the oldest houses. But since one step should be finished before starting with another step in the generation (in order to follow the strict approach discussed earlier), roads were deemed most suited to start. It is hard to even conceive a program that does not start with generating the roads. Looking at previous work, roads seem to always have been the starting point.

As a result of the road network, a number of plots defined by the empty areas in between the roads are created. It can be discussed whether using these areas is sufficient. In real life less dense cities buildings are placed simply on the side of the road, and not in between them. Ultimately it was decided however that finding plots in between roads was the approach that worked best for our goals. These polygonal areas are independently handled one by one in the next step, plot generation.

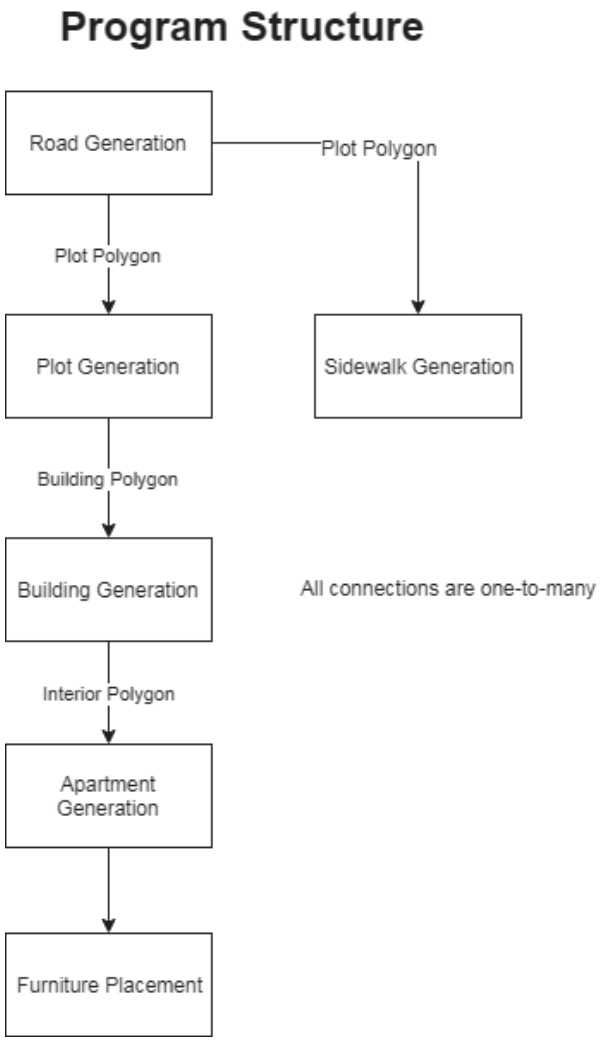


Figure 3.1: Main structure diagram

The plot generator decides what to do with each plot, each of these plot polygons can be subdivided according to the method discussed in 4.4. This (among other things) results in a number of building plots, these are handled in the next step, building generation. Buildings are created floor by floor using the stairwell, the sides of the house, and central corridors to determine polygonal areas to use in the next step, apartment generation. Apartment generation, similarly to the plot generation step, starts with a polygon and subdivides it further, now into single rooms. Taking these room polygons as input, the furniture placement steps determines location of specific meshes and potentially adds additional polygons to the room.

Another way to look at the goals of the algorithm is as a declarative interface for the user, the user defines the sizes and properties of buildings, and what should be included in apartments and rooms, and the algorithm tries to accommodate all of the wishes at once. This is easy for building size height, but gets more complicated when it's

about fitting user-specified rooms together in an arbitrary space.

This chapter will expand on each step discussed in this brief introduction.

For clarity, when discussing the different steps a common format will be used, with input defined as either obligatory or optional. Input being obligatory means that it is central to the algorithm and cannot be omitted, optional means that these parameters will be set to default values when omitted. It should be noted that in most steps there are more parameters that could potentially be tweaked, only the most important ones are included.

3.1 Road Generation

3.1.1 Input

- Obligatory:
  - Size of road network: The total number of roads to place.
- Optional:
  - Custom population heatmap: An image provided by the user showing the population heatmap, replaces the default noise-based population heatmap.
  - Road Length: The in-engine length of each individual road piece.
  - Maximum main road turning rate: The maximum possible change in rotation from the previous road for a newly placed forward-oriented main road.
  - Maximum secondary road turning rate: The maximum possible change in rotation from the previous road for a newly placed forward-oriented secondary road.
  - Main road advantage: In order for the big ”main” roads to be built first, with smaller roads around them, main roads are given a higher priority than smaller roads, the user can define the specific advantage themselves, a higher advantage yields more and longer big roads with less clustered cities.

3.1.2 Output

- An array of road lines with corresponding thickness describing the road network.
- An array of polygons describing the plots between the placed roads.

3.1.3 Description

Parish and Müller created a framework for a road generation algorithm [1], which was later refined [3]. This refined version formed the basis for the road generation used in this project. The basic idea of this algorithm is to place one "road piece" at a time. Using a priority queue, whenever a road is placed, the potential extensions of that road are placed as well, each given a priority[1] depending on the determined value of the road, (to see how this value is calculated see 4.2). Whenever a road is actually placed it might also need fitting to attach with other existing roads.

Using this technique the main road is given slightly higher priority, and smaller roads are given lower, meaning main city roads are placed before the smaller ones. This makes the small roads adapt to the larger roads instead of vice versa, creating a more natural looking road network.

3.1.4 Heatmap

Positive Heatmap

This project uses a general heatmap which is either generated via simplex noise or provided by a user, which defines what areas are particularly interesting. Taking this into consideration when giving each road its priority, it results in more neat and predictable city structures by giving the road generation certain intuitive global goals. The heatmap is grayscale and each position in the map corresponds to a single value.

Negative Heatmap

In addition to this general heatmap, this project uses a "negative" heatmap that is relevant for the area covered by main roads. Basically, when a large road is placed, each new large road within a certain distance will have lower priority than they would otherwise have. This attempts to make sure that the main roads do not go around in circles around certain points. It also artificially emulates the reduced need of a main road close to an area where there already is one close by.

[1] The priority decides the position in the priority queue

3.1.5 Plot Polygons

Plot shapes are found by placing one line to the left and one to the right of each road. These lines are checked against the roads to see whether they intersect a road, if they do, lines are recursively cut into smaller lines. After this all lines are checked for collision with the other lines, potentially forming polygons when several lines intersect. This approach is simple on paper but a lot of work remains to be done after intersections are found. More information on the method used is discussed in Section 4.6 as well as in Appendix B.

3.2 Plot Generation

3.2.1 Input

- Obligatory
  - An array of polygons describing the plots between the placed roads.

3.2.2 Output

- An array of polygons describing the shapes of the specific houses that shall be built.
- An array of polygons not suitable for buildings but for simple areas that should still be textured and detailed.

3.2.3 Description

Plots are created from the shapes in between the roads generated by the road placement algorithm, meaning that the shape of the plots are fundamentally dependent on the road generation. The plots themselves then go through a number of decisions deciding how they will be handled. The alternatives are:

- A simple plot with some texture and decorative meshes (for tiny plots).
- A single building (for small to medium sized plots).
- A block of several buildings and plots, resulting from a subdivision of the original plot (for medium-large sized plots and upwards).
- A textured area with houses placed iteratively as long as there is room at random. (for very large sized plots and upwards).

The exact cutoffs are not important, there is also randomness involved in selection of appropriate type. This is done to smooth out the overlaps between the different types of plots, so as to not make it too obvious at which point a plot becomes a certain type.

3.3 Sidewalk Generation

3.3.1 Input

- Obligatory
  - An array of polygons describing the plots between the placed roads.
- Optional
  - Sidewalk width

3.3.2 Output

- Polygons describing the sidewalks that are to be placed in the world.

3.3.3 Description

The sidewalk generation is fairly simple, it places sidewalks all around the plots generated in the road generation step, optionally with user-defined width. The exact polygons describing the sidewalk and sidewalk edges are generated here.

Besides the polygon generation, meshes specific to the sidewalk are also placed in this step, such as street lamps. The placement of these meshes follow pre-determined patterns and are not as interesting as other mesh-placement algorithms. Every sidewalk has a certain chance to spawn meshes of a certain type independently of other sidewalks.

3.4 Building Generation

3.4.1 Input

- Obligatory

  - A polygon describing the shape of the house at ground level.
  - A description of which sides are allowed to contain windows.
  - A description of which sides are allowed to contain entrances.

- Optional
  - Minimum/Maximum number of floors.
  - Height of a floor.
  - The number of times the polygon has a chance of being reduced at random (for example by cutting of an edge, increasing this number results in more irregular shapes and vice versa).

3.4.2 Output

- An array of polygons describing the shapes and locations of apartments to be populated in the building

3.4.3 Description

The building generation takes the polygon generated by the plot generation and optionally modifies it at random to make it more interesting. These changes are strictly reductive since the position and shapes of other houses and the other areas belonging to the plot are out of scope. The building generation is done individually and independently floor by floor, always remembering the shape of the building from the previous floor as well as which sides may contain windows. More about the building generation can be found in section 4.8.

Algorithm 1 Pseudo-code for building creation

```
1: for each floor do:
2:     potentially modify building shape again, affecting all floors this level and above, strictly reductive modification.
3:     determine polygon describing the center stairway/elevator.
4:     place corridors in each direction outwards from it.
5:     determine apartments resulting from the intersection of center polygon, corridors, and sides of house, tell the apartment which sides may contain windows (i.e. which walls face outwards from the building).
6:     for all determined apartments do
7:         if apartments is too large then
8:             recursively split apartment until it's within a reasonable size limit.
```

3.5 Apartment Generation

3.5.1 Input

- Obligatory
  - An array of polygons describing the shapes and locations of apartments to be populated in the building with corresponding descriptions of which sides have doors, which have windows, which sides are exterior walls, and of which type the apartment is (office, living, store, restaurant).
  - Blueprint for the different kinds of rooms, with size specifications.
  - A description of which sides contain entrances.

3.5.2 Output

- An array of polygons describing the shapes of individual rooms, with corresponding descriptions of which sides have doors, which sides have windows, which sides are exterior walls (walls towards the outside of the building) and of which type the room is (bedroom, bathroom etc.).

3.5.3 Description

The apartment generation is perhaps the most interesting step, because of how much it affects the final result both directly and indirectly (through determining the outer walls and windows of a building), as well as how varied results it produces. The fact that the apartments generation affects the look of the buildings is very important to keep in mind. Designing the algorithm, the two main goals were flexibility and performance.

This implementation is based on the idea that a good way to generate natural shapes of rooms is found by splitting existing polygons in parts as opposed to for example fitting predetermined polygons together. All rooms are created by splitting polygons continuously. A side benefit of this approach is that it makes it possible to make sure that all rooms remain reachable from all other rooms without doing any path finding or graph traversal. The reason why unreachable rooms are avoided is the fact that every time a room is split, both of the newly created rooms keep their connections to rooms the previous one was already connected to. This means that connections between rooms are never lost, which in turn means that as long as the original main room polygon has an entrance outwards, there will be a path from that entrance to every room created. It's also a strictly iterative approach of time complexity n (where n is the total area of the apartment), making it scale well with increased size of apartments.

An aspect that becomes a challenge with this approach is the fact that all rooms are created equal. Some specific rooms in real apartments are unlikely to act as junctions or be in the immediate way between other rooms. It is for example unlikely that it would be necessary to go through a bathroom to get to the living room in a real apartment. This problem can be handled by more careful splitting of rooms. In this implementation every room type blueprint contains a boolean attribute that describes whether it works as a junction or not. When splitting a room which is too big, the algorithm is attempting to make the room with the least connections a room without this attribute. This is not a solution to the problem but it makes it much less prevalent.

When creating apartments, an essential function used very frequently is the function splitting of rooms in half. This is an approximative method that is described in the figures in Appendix A and described in more detail in section 4.4.

Essential terms:

- Needed rooms: Room blueprints that need to exist in the final apartment, for living apartments this might include a bathroom and a bedroom.

- Optional rooms: "Filler" rooms, an array of room blueprints that can be placed in case there is space leftover, this array can be traversed multiple times as long as there is leftover space. For living apartments this might include a bathroom and a bedroom.

Algorithm 2 Pseudo-code for the room placement

```
array emptyRooms
array finishedRooms
Add apartment polygon to emptyRooms
for blueprint in needed rooms do:
    if polygon of suitable size exists in emptyRooms then:
        remove polygon from emptyRooms
        add polygon to finishedRooms with same type as blueprint
    else if all polygons were too big then
        Polygon p = first polygon in emptyRooms
        remove p from emptyRooms
        while p is too big do
            split p in half
            add one half to emptyRooms, keep the other as p
        add p to finishedRooms with same type as blueprint
```

Repeat for blueprints in optional rooms, and continue the algorithm while there are rooms left in emptyRooms.

3.6 Furniture Placement

3.6.1 Input

- Obligatory
  - A polygon describing the shape and location of the room, with corresponding description of which sides have entrances and which sides have windows.
- Optional
  - Alternative meshes for each object.

3.6.2 Output

- An array of meshes to be placed in the world.

3.6.3 Description

Furniture placement is done very simply, the goal being to find generalized and simple algorithms that cover a lot of possible use cases. To create these a couple of real life apartments were studied, primarily living and office apartments. It was found that most furniture placement in real life is done based on a few different geometrical rules, see section 4.9.

Using different combinations of these possibilities all of the rooms in the city are generated. Each room type is allowed to specify which furniture should be placed and according to which pattern, in any combination. This has proven to be a solution that, while not covering all possible rooms, provides a wide variety of possibilities. This solution is very scalable, building on the current concepts with more rooms and apartments requires very little additional code.

The current implementation is quite small, because extending the current system with more rooms is not an interesting part of the project from a research point of view. Defining more rooms with more meshes however would go a long way towards making the city more diverse and interesting.

Chapter 4

Implementation

4.1 General

The program is implemented in Unreal Engine 4, polygons are rendered using the Runtime Mesh Component plugin, and static meshes are placed as instanced static meshes. The shaders in the engine and similar effects have been left untouched, and only minor details regarding textures and materials have been modified.

Polygons are triangulated and modified with the small but very adequate library polypartition[1].

All of the static meshes used were either made for this project specifically or used free assets found online.

4.2 Road Generation

As mentioned earlier, the road generation is based on the improved version [3] of Parish and Müller's work[1]. This version uses the priority queue. The comparison metric for the queue is a mix of a few different things. New road pieces enter the priority queue when a road from the priority queue is placed. When placed, a road attempts to place a new road piece in front and some attempt to place road pieces perpendicular to itself. The exact angles of these roads are calculated by testing which directions yields the best priority, within a defined maximum angle. For a certain $maximum\_change\_intensity$ a road forward could have a rotation anywhere between $previous\_segment - maximum\_change\_intensity$ and $previous\_segment + maximum\_change\_intensity$. This means that higher values for $maximum\_change\_intensity$ yields more curves and a value of 0 gives a strict grid network.

[1] https://github.com/ivanfratric/polypartition

The value of a road is calculated as

$$noise\_at\_point(road\_middle) + detriment$$

where a lower value gives higher priority. The term *detriment* is only relevant for larger main roads, and is then calculated as

$$\sum_{n=0}^{num\_roads} \frac{detriment\_range - distance(road, roads[n])}{detriment\_range} * detriment_impact$$

where *roads* are the main roads already placed within *detriment_range* and $detriment_impact$ is the magnitude of the detriment.

This term *detriment* corresponds to a sample from the negative heatmap discussed earlier. This implementation is not very optimized, and could be made faster by creating an actual image and then picking the samples from there, however that would cost a little bit of accuracy.

For the generalized road placement algorithm, see [3].

4.3 Noise

The noise should be thought of as a pre-determined approximation of population density over different areas. It is used in the road generation allowing it to follow global goals, as the individual road placements should in some way follow a pattern. There are two distinct modes for generating this noise-map, either via a Perlin-noise function or via a user-provided image. Generally, providing a user-defined heatmap gives better looking results.

Code wise, this is handled via a singleton-interface which any class can access to get noise values for specific coordinates, this singleton is defined in the initial part of the program as either based on a heatmap or noise.

4.3.1 Perlin-noise

There are several ways to use Perlin-noise. It was decided that the simplest single-banded perlin noise would be used for the city since lower complexity noise would be easier to work with and debug.

Figure 4.1: Sample output from the single-banded Perlin function

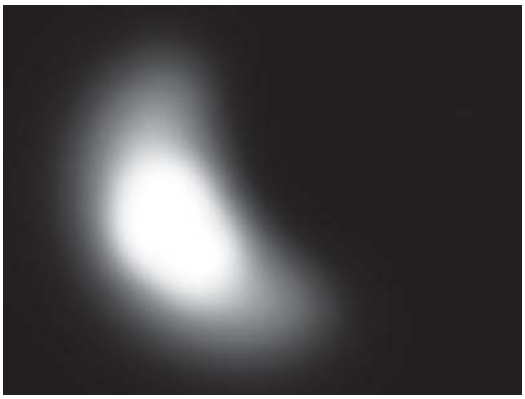


Figure 4.2: Sample user-created heatmap

4.3.2 Heatmap

The user-provided heatmap is simply a gray scale image, the program works with any image but works best with gradients, as that allows roads to find logical directions to build further roads, as otherwise the road generator has a hard time "climbing hills" towards the greatest population densities. See section 4.2 for details.

4.4 Polygon Splitting

A central function to many of the program is the creation of plots, buildings and rooms by splitting, or subdivision, of polygons into smaller parts. In all of these instances the subdivision algorithm is the same, and looks like the pseudo-code in Algorithm 3.

Algorithm 3 Pseudo-code Polygon Subdivision

```
1: Polygon originalPolygon
2: longestEdge = getLongestEdge(originalPolygon)
3: middle = middle(longestEdge)
4: otherEnd = middle + (Rotate90degrees(longestEdge))
5: otherEdge = find closest edge in originalPolygon that intersects with edge(middle, otherEnd)
6: firstEdge = first_of(longestEdge, otherEdge)
7: lastEdge = last_of(longestEdge, otherEdge)
8: edgesBetween = [all edges between firstEdge and lastEdge]
9: otherPolygon = edgesBetween + edge(middle, otherEnd)
10: originalPolygon = originalPolygon − edgesBetween + edge(middle, otherEnd)
```

This algorithm reduces the original polygon, adds a new edge where the previous was removed, and creates and returns a pointer to a new polygon. In practice this is a lot of code because of the different properties belonging to the polygon that has to be updated. For example in rooms, it is important to keep track of which walls allow for windows.

4.5 UV-mapping

A problem that arose early when starting to place procedurally generated polygons in the world was how to correctly find good UV-coordinates on the polygons[2]. Fortunately, there is a simple and for the most part adequate way of determining good UV:s.

Since every polygon exists only on a 2D-plane, this plane and each tangent's coordinates on this plane can be determined from three points via basic linear algebra.

In practice it looks as follows 0.

Algorithm 4 Pseudo-code for UV-Mapping

```
1: Polygon pol
2: e1 = normalize(pol[1] − pol[0])
3: normal = e1 × (pol[2] − pol[0]);
4: e2 = normalize(e1 × n);
5: origin = pol[0]
6: for point in pol do:
7:     point_x_coordinate = e1 · (point − origin)
8:     point_y_coordinate = e2 · (point − origin)
```

This works well if these coordinates are found before triangulation of the polygons, since that way the correct coordinates in relation to other vertices in the same polygon remain. It does not work as well for lots of tiny

[2]Coordinates for textures

polygons. This would work somewhat after triangulation for one triangle at a time but only be correct internally and not externally for each separate triangle since seams between triangles would be very obvious.

4.6 Generating plot polygons from road network

Generating plot polygons from the road network got a lot more complex than expected and hoped. The basic idea is to draw two lines, one on each side of every road, and then build polygons from intersections between these lines. There are however, a lot of cases for these intersections to consider. One important thing to note is that these lines extend a little bit beyond the length of the road itself, in order to be able to check for intersections with crossing lines.

Before doing any intersection checks against other lines, collision tests must be done against the roads, if a road connects to the road the line was based on. To solve this, lines are recursively split for every intersection with a road. Remaining after this step is a list of lines at least twice as many as the number of roads not intersecting any road.

After this, intersection checks are performed for each line against the other lines, continuously storing incomplete polygons via linked structures and building onto them with new lines. Each line can contain a pointer to another line that connects to its front, and a pointer to one line that connects to its back, thus creating a bidirectional graph. Important to note is that what is stored and what will also become the polygon are not the exact points belonging to the lines, but rather the intersection points between lines.

After these structures are established, it is possible to traverse these pointers by starting with a line and going forwards or backwards until a node is encountered that has already been found, then a polygon can be created from the path traveled.

For a visual example of how this algorithm works, refer to Appendix B.

4.7 Building height

A surprisingly troubling area was getting the algorithm deciding building height to work decently without making it overly complex. A basic uniform distribution of heights within a range is not suitable, since in real cities most buildings fall in a lower range of heights, while some buildings are much taller. Because of this some sort of exponential distribution had to be used.

Remembering how the noise in the program is meant to emulate population density (section 4.3), it seems reasonable that this should also influence the height of the buildings at different locations.

There are a lot of ways to calculate building height, the version that ended up in the program was the following. Taking $minFloors$, $maxFloors$ and $noise\_h\_inf$ as parameters.

$$noise\_mp = ((1 - noise\_h\_inf) + (noise(house\_center) * noise\_h\_inf)$$

$$general\_modifier = \frac{-\ln(rand\_num\_range(e^{-4} + (1 - noise\_mp), 1) * noise\_mp}{4}$$

$$diff = maxFloors - minFloors$$

This means that $noise\_mp$ uses a linear interpolation between 1 and $1 - noise\_h\_inf$ and $general\_modifier$ uses inverse transform sampling with exponential distribution.

The final range for $modifier$ of [0..1]. $noise\_h\_inf$ is a constant in range [0..1] defined by the user. It determines the influence the noise in the current spot should have on the height of a building. A value of 1 makes noise extremely important for the height of buildings while a value of 0 makes the distributions for all buildings the same independently of noise of current location. This value of $noise\_mp$ affects the final height in two major ways. It is used internally to calculate $general\_modifier$ and is also multiplied with it afterwards. $noise\_mp$ defines what range of the exponential distribution should be used. This is done to emulate how larger buildings in cities also have been found to have much greater variance in height. See 4.3 and 4.4.

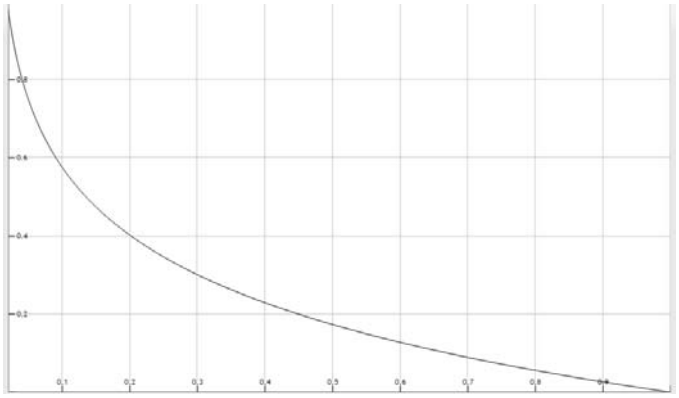


Figure 4.3: Calculating $general\_modifier$ for $noise\_mp = 1$, x is a random stochastic variable between 0 and 1

Final height is calculated according to:

$$height = minFloors + diff * general\_modifier$$

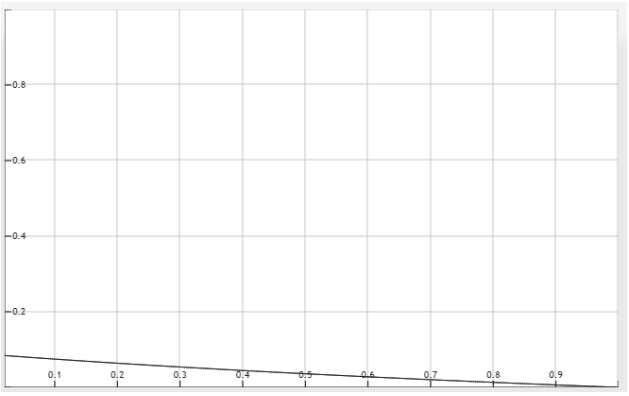


Figure 4.4: Calculating $general\_modifier$ for $noise\_mp = 0.5$, x is a random stochastic variable between 0 and 1

To see what this looks like applied on a whole city, see for example 5.15.

4.8 Building shapes

The concept of small functions applied iteratively is central to building shape generation. Every building receives a polygon from plot generation to base the building on. This polygon is then modified, potentially several times. Possible modifications are:

- Moving an edge inwards.
- Reducing size in every direction, "shrinking" the polygon.
- Removing a corner by removing a vertex and adding two new vertices on the middles of what used to be the edges connecting first the vertex.

These changes have certain chances to be applied at every pass, and the number of passes depends on user input, the resulting modified polygon forms the base or ground level shape of the building.

A stairway/elevator area is randomly selected somewhere inside the house polygon. This stairwell, as well as the four paths of 90 degree increments out from the stairwell, and the sides of the house, forms the boundaries for the apartments.

For every level of the building there is a chance that the building shape is modified again, via one of the aforementioned methods. This does not affect floors below; these modifications add onto each other. These changes are strictly reductive, since the outside of the original house polygon is outside of scope for this part of the algorithm.

4.9 Mesh Placement

There are a couple of steps in the program where static meshes are placed in the scene, primarily in the furniture placement, but also in sidewalk generation, plot decoration and house generation.

These different placements have some things in common since the strategies used for mesh placement are shared throughout the program. Originally looking at real interiors, a few strategies for how furniture is generally placed were found, These strategies were then expanded to include patterns observed for objects placed outside of buildings. Made as simple as possible, these strategies were:

1. Placing an object along a random wall or a side of the polygon.
2. Placing an object in the middle of the polygon.
3. Placing objects in rows inside the polygon.
4. Placing objects at a random spot within the polygon.

In the program, these methods also take a list of blocking polygons as parameter. Polygons describing doorways are included in this list of blocking polygons, as well as earlier placed objects. This means that these methods can be used iteratively in any combination with each other. In this implementation the collision boxes for the different meshes are taken directly from the collision bounding box automatically generated by Unreal Engine 4 from the imported meshes.

For furniture placement, types 1, 2 & 3 are used. For outdoor vegetation, type 4 is used. For roof decoration, type 3 & 4 are used. Sidewalk decoration is not done by any sophisticated algorithm; meshes are simply placed repeatedly with certain intervals.

4.10 Performance

Even though performance considerations are not as important for this thesis compared to the ideas it explores, performance is extremely important in any practical application that might use these concepts.

A significant amount of thought has been put into performance for this project, an innate benefit of recursive algorithms and algorithms with very limited scope is how they often reach a low time complexity.

In addition to these implicit design considerations, there have been some very explicit parts of the program added just to increase performance. The program provides two different ways to generate the city, the simplest alternative is to generate everything at once. Another alternative is to generate interiors for buildings when the camera is close, and de-allocate interiors when the camera is further away. This is the default mode since it allows much larger cities using the same amount of memory. It is important to note however that basic room calculations have to be done even when interiors are not rendered, since these calculations decide window placements, which have to be coherent with their locations at the outside of the building.

Performance considerations also include threading. Threading was implemented in order to be able to generate parts of the city at runtime (although it also increases performance even if everything is generated at once), so that parts could be generated and de-allocated dynamically, saving a lot of memory. The threaded part of the program more specifically is the house generation. Everything related to a building can be calculated on other threads. One problem however is that it is not currently allowed in UE4 to place objects and polygons into the world on any other thread than the main game thread. This means that there is some stutter if doing runtime placement of meshes and polygons.

Chapter 5

Results

This chapter will shows the different aspects of the program in their respective contexts.

Figure 5.1: Screenshot at ground level

5.1 City Structure

Through parameters many of the characteristics of the generated city can be decided by the user. A very influential parameter is the population heatmap, or the lack thereof (see 4.2). To see the dramatic effect the population heatmap has on the final result, refer to the heatmaps pictured in 5.3 and the corresponding city shapes in 5.4.

Regarding our reference cities the maximum allowed turn rate of a road is very important. It should be close to zero for smaller roads if the city should look like Manhattan (see 5.12a). But to achieve the look of "messier"/less grid-like cities like Malmö (see 5.12c) the turn rate should be higher.

Adjusting only the maximum road turning rate affects the city shape in major ways. To see how changing this variable can affect a city based on the same population heatmap see 5.5.

For clarity, three different parameter combinations were used for the road generation to generate the results in this

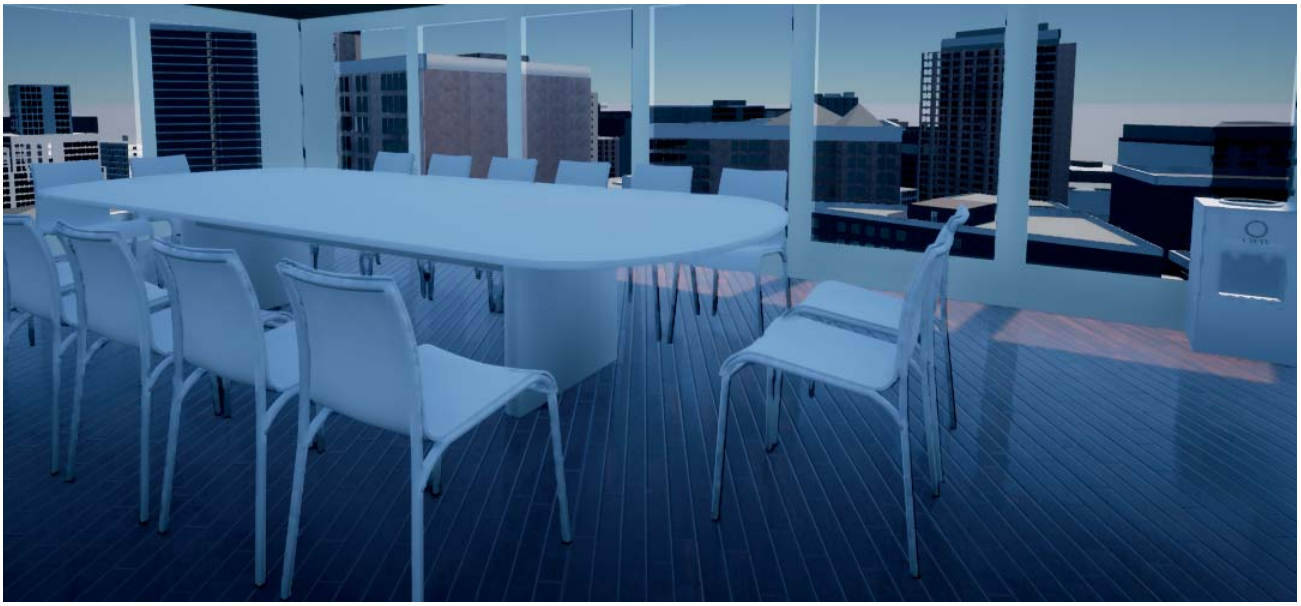
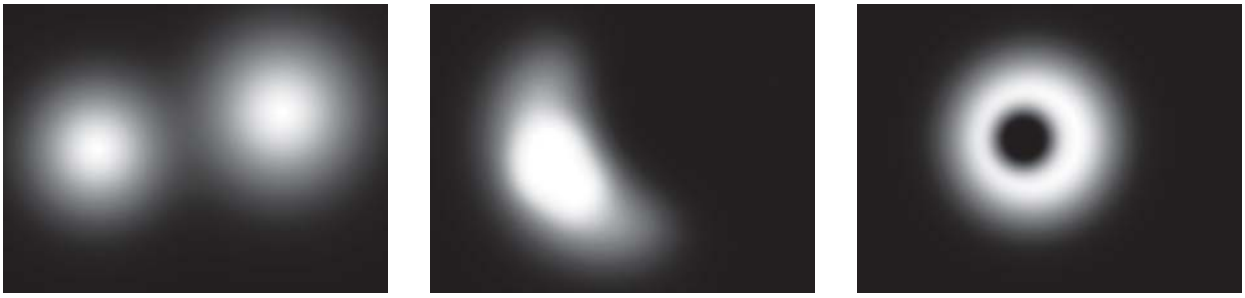


Figure 5.2: Screenshot from inside a meeting room in an office



(a) Heatmap 1 (b) Heatmap 2 (c) Heatmap 3

Figure 5.3: Heatmaps

chapter, found in table 5.1, these types are referenced in the attached images.

Type	Max main road turn rate (degrees)	Max secondary road turn rate (degrees)
Grid like	2	0
Normal	20	5
Chaotic	30	15

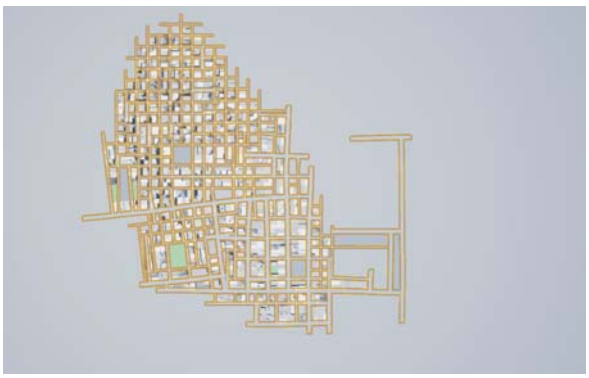
Creating the road network via Perlin noise produces quite different results, for examples, see 5.6. Perlin noise tends to produce more sprawling cities, a problem with using perlin noise is how the cities tend to be less centralized than one might want, occasionally forming separate clusters.

(a) City 1, type = Normal

(b) City 2, type = Normal

(c) City 3, type = Normal

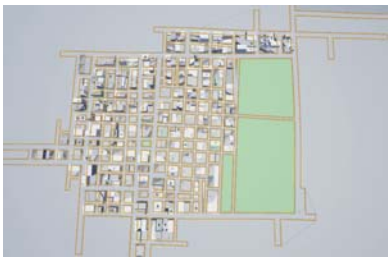
Figure 5.4: Resulting cities



(a) City 2, type = Grid like

(b) City 2, type = Chaotic

Figure 5.5: Changing turn rates

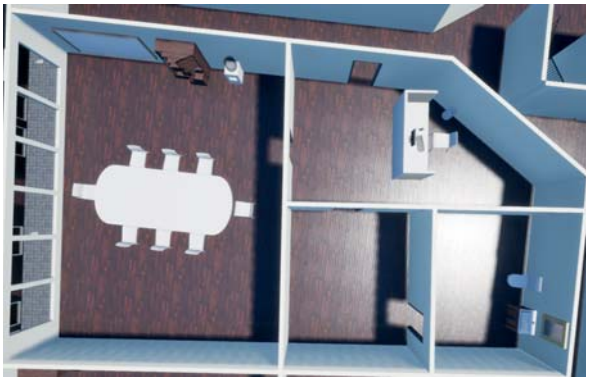


(a) Noise based city, type = Grid like

(b) Noise based city, type = normal

(c) Noise based city, type = Chaotic

Figure 5.6: Noise based cities



(a) An office apartment

(b) A living apartment

Figure 5.7: Simple Apartments

5.2 Apartment Structure

The apartment generation is capable of producing apartments of any size, but is best understood by looking at smaller examples, looking at the results in 5.7, it's very clear how the algorithm has worked to create these apartments step by step.

The office apartment 5.7a contains four rooms. This means that there have been three splits, since every split creates one additional room. The Apartment generation algorithm has begun by splitting the whole apartment into two pieces, assigning one of them the role of meeting room (the room to the left with the table). The right half has been subdivided again, a working room has been placed in the top part, then the bottom part resulting from this split has been split one final time, placing a toilet and an empty room.

The living apartment 5.7b also contains four rooms, meaning there has been three splits here as well, resulting in a living room, a kitchen, a toilet and a bedroom. The order in which these three splits were performed can easily be deduced.

5.3 Windows

Windows are handled simply, every apartment type provides a density and (possibly randomly defined) dimensions for windows, and for every exterior wall in a room that allows windows, a number of windows given by the formula $window\_density * wall\_length$ are fit evenly across the wall. For an example of windows in an office apartment see 5.8a.

Apartments on the ground level are handled a bit differently than apartments on other floors. They are allowed

(a) Windows on an office apartment

(b) Restaurant exteriors

Figure 5.8: Exteriors

to have entrances outwards, and can be decorated with awnings and simple signs, see 5.8b, the windows on the bottom floor need to accommodate these things.

For some more perspective on how the apartment generation algorithm works on a larger scale, see figures 5.9 and 5.10.

5.4 Furnishing

The way the mesh placement uses a few pre-determined patterns has been discussed in 4.9. Figure 5.11 shows how it can look in practice, this example with a kitchen is used because it shows several of the different patterns used at once.

The oven, fridge and kitchen-mesh are specified to be placed along a wall, so the algorithm has found suitable places alongside walls. The table is specified to be placed in the center of the room. The chairs are specified to be placed within regular intervals around another polygon (the table). The kettle is specified to be placed at a random point on another polygon (the table).

5.5 Comparison to Reference cities

5.5.1 Recreating Manhattan

Making a heatmap that somewhat approximates Manhattan allows us to compare the results of the algorithm to the real counterpart.

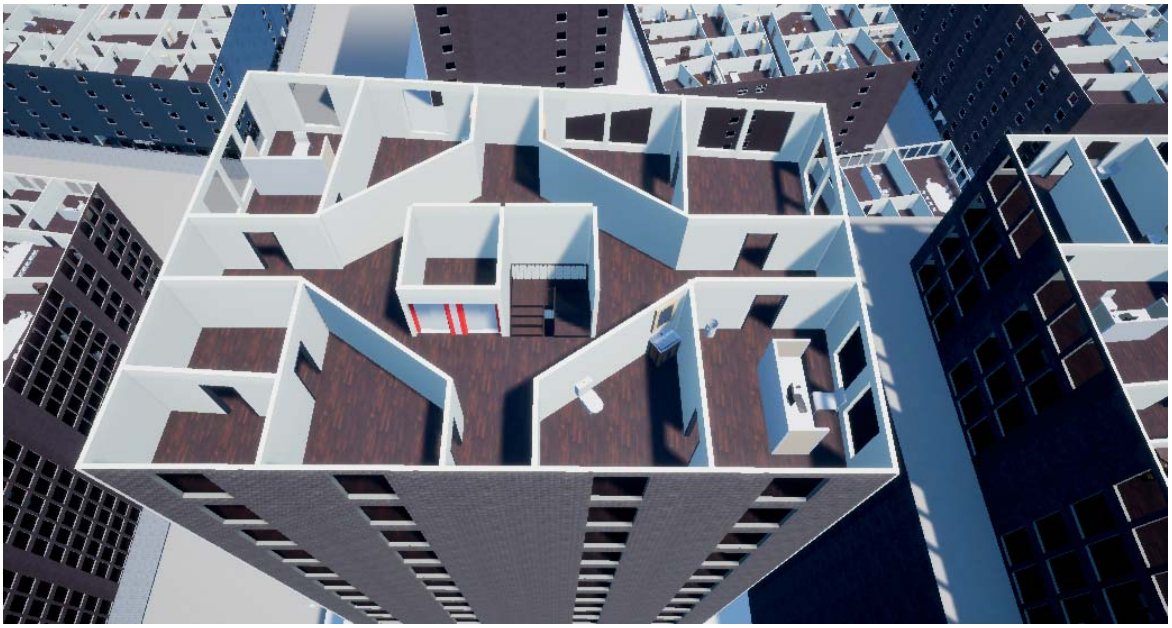


Figure 5.9: Apartment subdivision for a simple small building floor

Figure 5.10: Apartment subdivision for a larger and somewhat more complicated building floor

Figure 5.11: A kitchen with furniture

(a) Manhattan, NYC

(b) Los Angeles

(c) Malmö

Figure 5.12: Real Cities

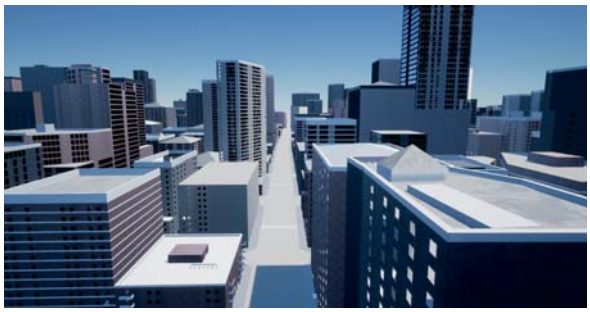
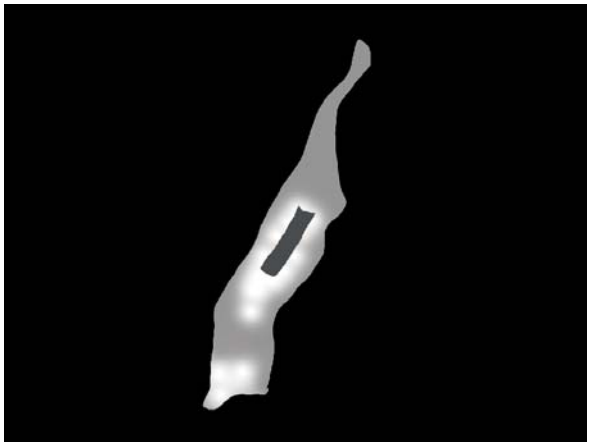


Figure 5.13: Procedurally Generated Cities

(a) Real manhattan



(b) Crudely Improvised heatmap

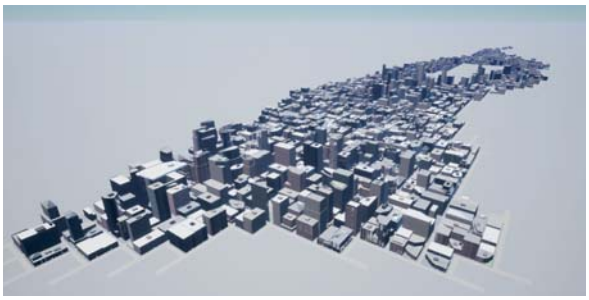


Figure 5.15: Generated city, type = Grid like

The scale of the procedural Manhattan is smaller, otherwise the accuracy is acceptable, with height distribution for different areas successfully approximating the distribution in real Manhattan. The only information the program has access to is the heatmap in 5.14b.

An aspect that the program does not handle currently is how large cities next to water (such as New York) tend to have roads along the sea shores, enclosing the city. This could be a nice addition to the program.

Chapter 6

Conclusion

6.1 Summary of Thesis Achievements

This project was successful in creating a prototype that achieves all the essential goals we set out to achieve. As a base of discussion, we'll return to the questions posed in the introduction:

- Is it at all possible to generate whole cities procedurally? If not, why? A: Yes, it's definitely possible.
- At which level of the generation does the project have to fall back to predetermined objects and meshes? A: The furniture and meshes placed in the world are predetermined objects, because of the sheer detail necessary in these objects, it's very hard to procedurally generate them in-engine, however it is possible that procedural techniques could be used when modeling them before importing them into the engine. Stairwells and elevator shafts are also pre-designed.
- Will it be possible to find general algorithms that cover a wide variety of cities? Or will they have to be tailor made to specific types of cities and buildings? A: The algorithms currently in place cover modern western urban cities. They do not support villas, gardens or older buildings, more algorithms and models are needed to cover these different types of environments. This also means that the algorithms are not as suited for the construction of less dense, older cities like Malmö as they are for cities like New York or Los Angeles. In terms of city layout, the system is based on previous work [1], and this system is very flexible and can cover a lot of different cities.
- In terms of performance and scalability, are these algorithms feasible on a much larger scale? A: Yes, although

some implementations currently are at worst n^2 or $n * m$, these are spatial algorithms that can be optimized by using quad trees or similar.

6.2 Returning to Reference Cities

As touched upon in the previous section, the algorithms produced work best for very dense cities where all space is occupied. Since the program does not currently support placing buildings outside of plots, or generating plots that are not completely surrounded by roads, it does not work as well with many of the areas in outskirts of larger cities or smaller cities, since buildings in these areas are often placed alongside roads, as opposed to in between.

This means that Manhattan and Los Angeles can be modeled satisfactorily whereas trying to create Malmö does not work as well within the current program. Improvements to allow for cities more similar to Malmö would include algorithms for placement of plots and buildings that are not entirely surrounded by roads. To model the older parts of Malmö with older (and very complicated) buildings more sophisticated building-generation algorithms are needed as well.

6.3 Applications

The most obvious use for procedural generation is in entertainment, specifically games and movies. Games today can offer much more than they did twenty years ago, but in sense of scale of environments, things have not changed that much. As discussed in the beginning, the true potential and allure of procedural generation in the context of physical environments is the sheer size that is possible to achieve, effectively bypassing the bottlenecks put in place by computer memory and human workload. In fact, to generate truly infinite worlds, there is no other choice than to utilize procedural algorithms. As of today, it's still much harder to implement environments procedurally than manually. On a small scale anything that can be done procedurally can just as easily or way easier be placed by a designer. Perhaps with more research in this area, this could change, or at least the difference could become less pronounced.

This project has tried to procedurally generate as much of its content as possible. In a more practical setting it might be more reasonable to make parts of an environment manually, and other parts procedurally. One could for example use the interior generation in combination with manually designed building-shells, since the apartment and room generation only needs the general shape of the current building floor as input. This allows for the "best of both worlds", where a designer can design what's really important manually, and use procedural generation to fill

out the details. Even simple procedurally generated interiors make a building much more interesting than a hollow one. Since a small if any sacrifice in memory is needed in order to do this, it seems like a good alternative.

6.4 Future Work

6.4.1 Improvements

Plot Generation

One good example of areas where the program might have gotten too complex is the plot generation. What began as a simple idea, placing lines along each road and checking for intersection, turned out much harder than expected, with a large amount of edge cases which had to be handled. A better approach might be saving the connections between roads generated in the road generation step and building a multi-directional graph based on the roads. Thereafter one could find polygons by for example starting in a node, walk along edges always turning left, and when (if) returning to the starting node, build a polygon based on the path.

Apartment Generation

Before apartments are placed, the corridors from the stairwell of the house are laid out. This is done in a very strict way, and could be made more interesting by making the stairwell placement and building layout more dynamic.

For the apartment generation itself, there exist a couple of problems, the severity of which depends highly on the use-case, originating in the recursive room design. The problems manifest themselves in two major weaknesses of the apartment generation. First, it's not possible to tell the algorithm where a certain room should be. This means that for example in a store it is not possible to specify that the main room with shelves should be in the front, next to the street, at least not for now. Stores and restaurants in the program are currently being treated as single-room apartments, so this weakness does not show.

The second weakness is what was discussed earlier, that although possible to influence, the algorithm cannot guarantee specific rooms not being junction points. Being a junction point means that the room contains at least two entrances, and could possibly be the only way between at least two other rooms. This is a problem with for example toilets, which should logically only have one entrance and definitely not be the passageway between other rooms.

It is unclear whether these problems can be completely solved without abandoning the simple recursive structure of apartment generation, which otherwise has a lot of benefits in how simple and yet powerful it is and how natural it looks.

Furniture Placement

Although the furniture placement as a concept is simple and suits its role fine, it deserves to be expanded upon. To further improve this step it seems logical to add more placement rules and simply more possible meshes. A very interesting next step could be creating and adding procedurally generated meshes that could fit the rooms even better, for example a kitchen that is expanded to fit a specific wall.

6.4.2 Unexplored areas

There are a lot of possible ways to improve and build on this project. When it comes to the different steps of the project, all of them deserve to be expanded and refined. Within the time limits of this thesis it was not possible to make every part of the program simple and manageable, but few if any parts of the program are complex in and of their selves. The complexity that exists is due to the fact that a lot of time was put into prototyping. This was intentional, but it did not allow for the level of polish that one might want in production-level work.

Apart from the project itself, there are a lot of areas outside of the scope of this project that would be very interesting to study in depth. This project does not cover the environment outside of the city, or hilly cities. Water, docks, and bridges are other natural extensions of this project, and hopefully these improvements could be combined with the project as it is right now without too many merging problems, since these parts should only affect the road generation step.

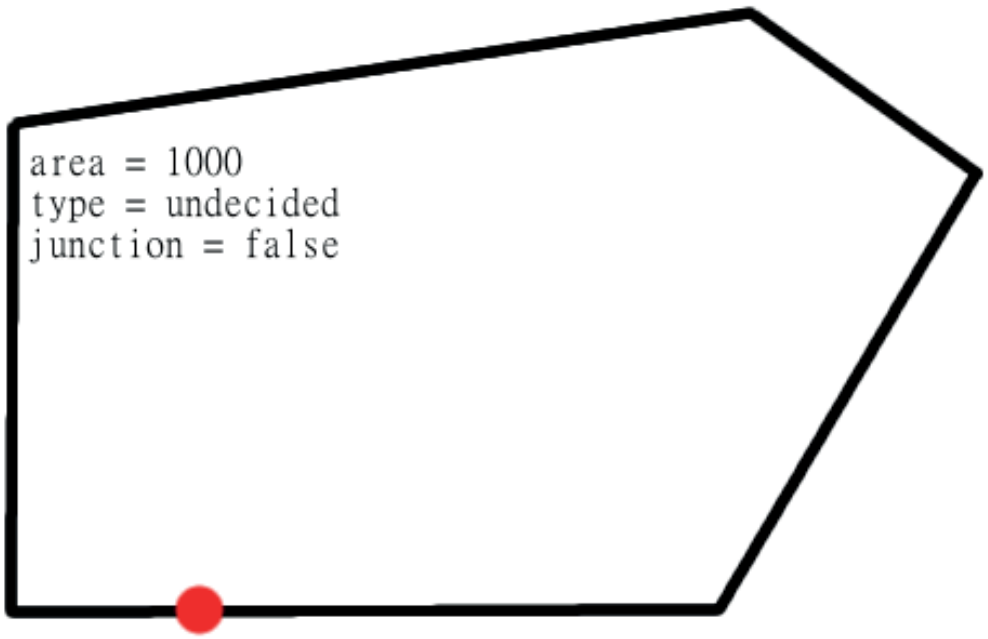
Vegetation in the project has remained simple, both in terms of how its placed randomly or predictably in certain spots, and in terms of the complexity of the meshes themselves. An interesting continuation in this area could be different ways of implementing procedural vegetation, perhaps as L-Systems[7].

Another area that could be interesting to look more into is weather. An interesting paper on this topic uses the environment at hand to calculate weather and clouds procedurally [6]. These kinds of modifications that are done strictly post-city generation should be easy to apply.

6.5 Final Thoughts

Though not without flaws, this project has shown that areas not always considered suitable for procedural generation, large-scale complicated coherent environments, can be tackled by procedural algorithms. Hopefully this is an area that can continue to develop in parallel with manual creation of environments as the subject of computer graphics continues to mature.

.1 Appendix A



Rooms to place:
Bedroom (200 < area < 400) avoid junction
Bathroom (100 < area < 200) avoid junction
Living Room (300 < area < 700) allow junction

Doorway

Figure 1: Apartment with room requirements

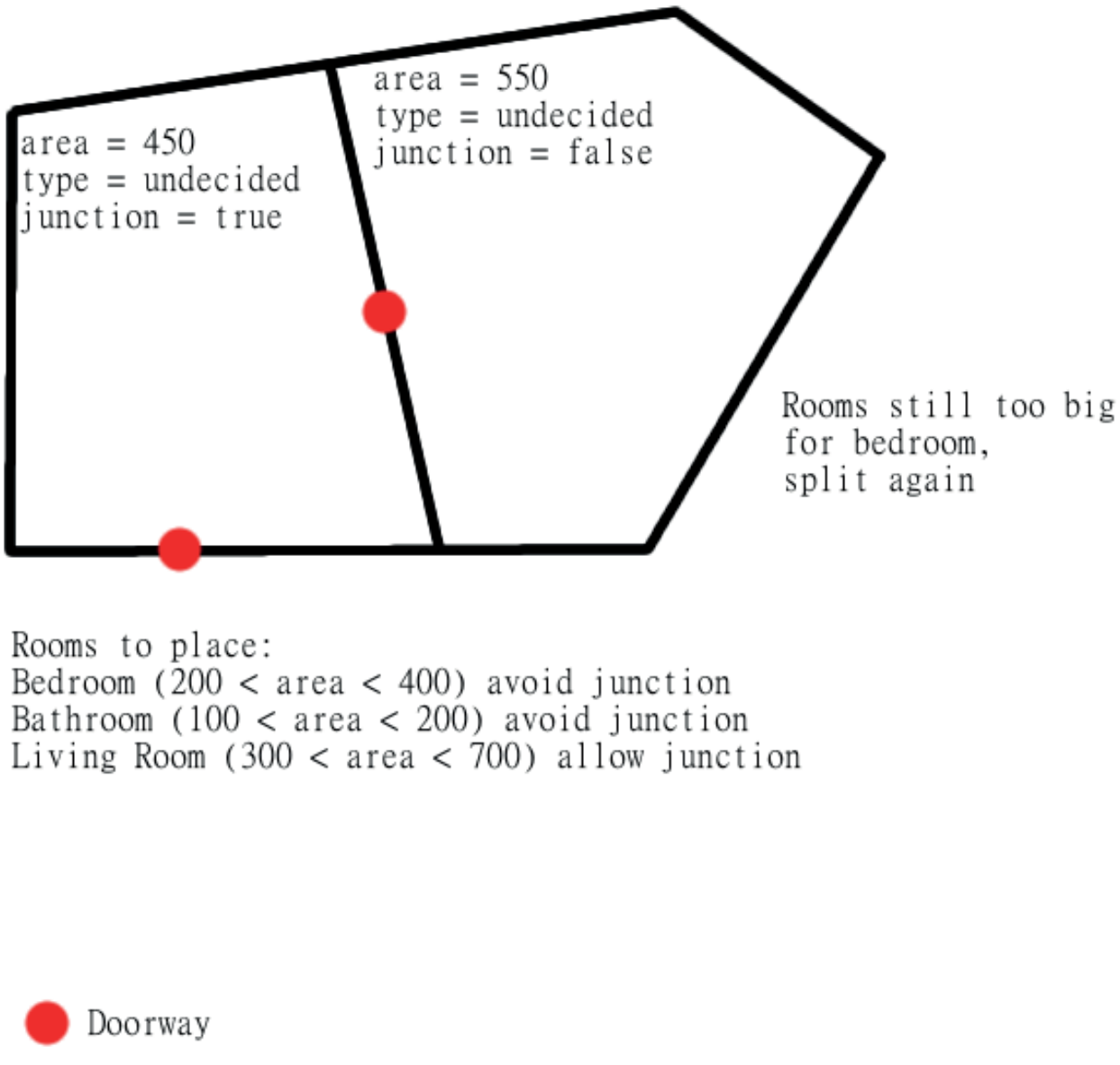


Figure 2: Apartment after one split

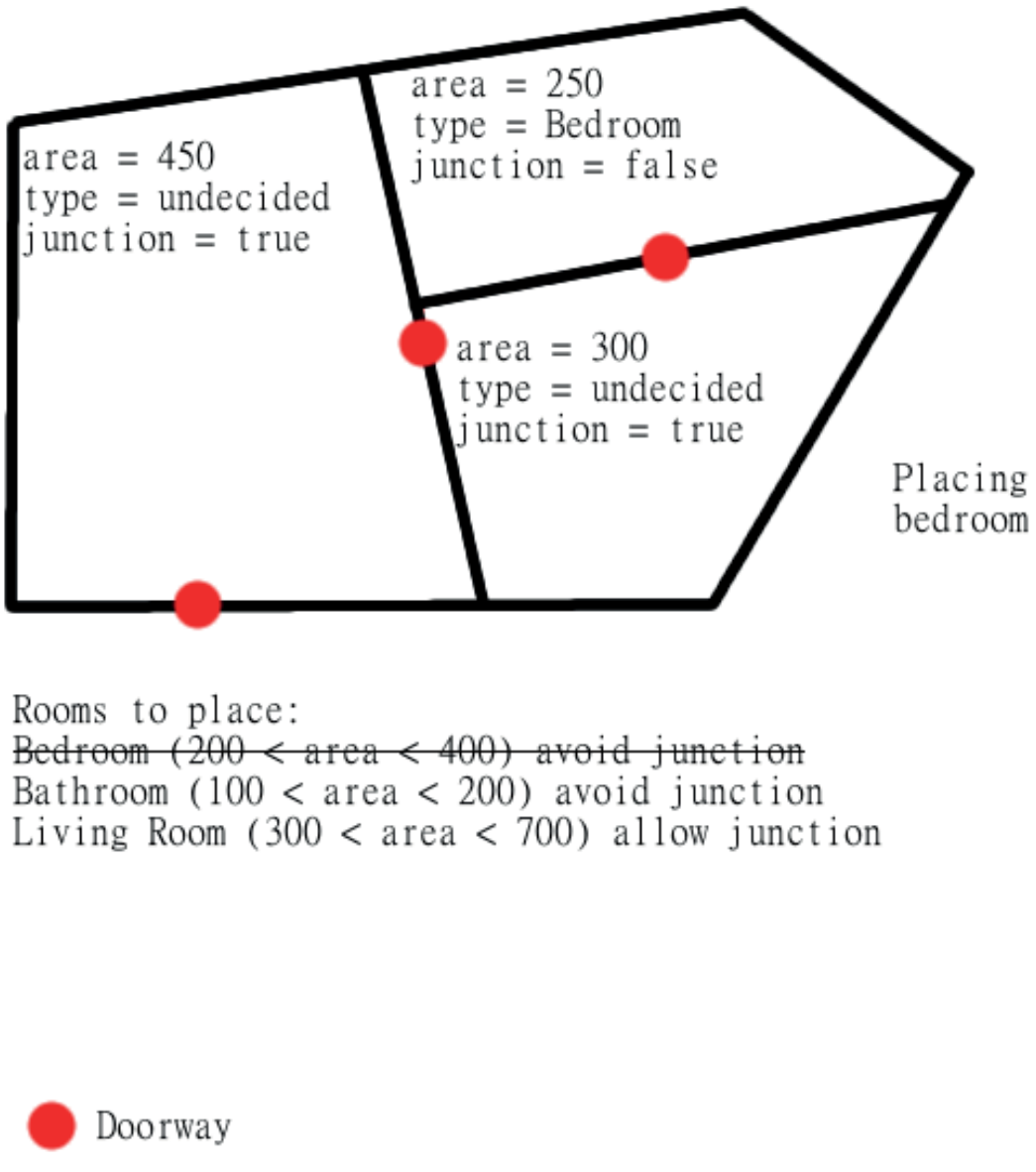


Figure 3: Apartment after two splits, Bedroom placed

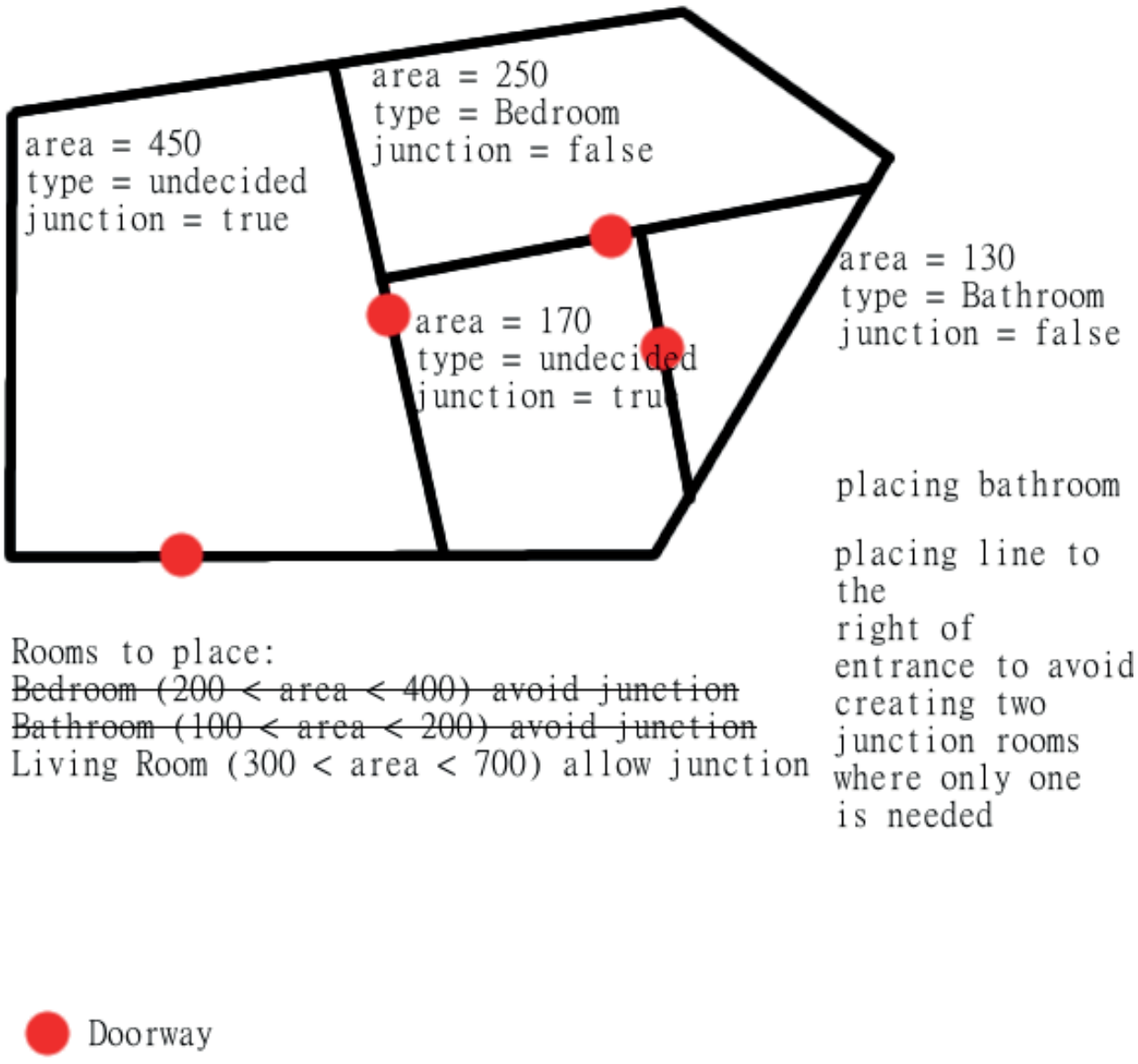


Figure 4: Apartment after three splits, Bathroom placed

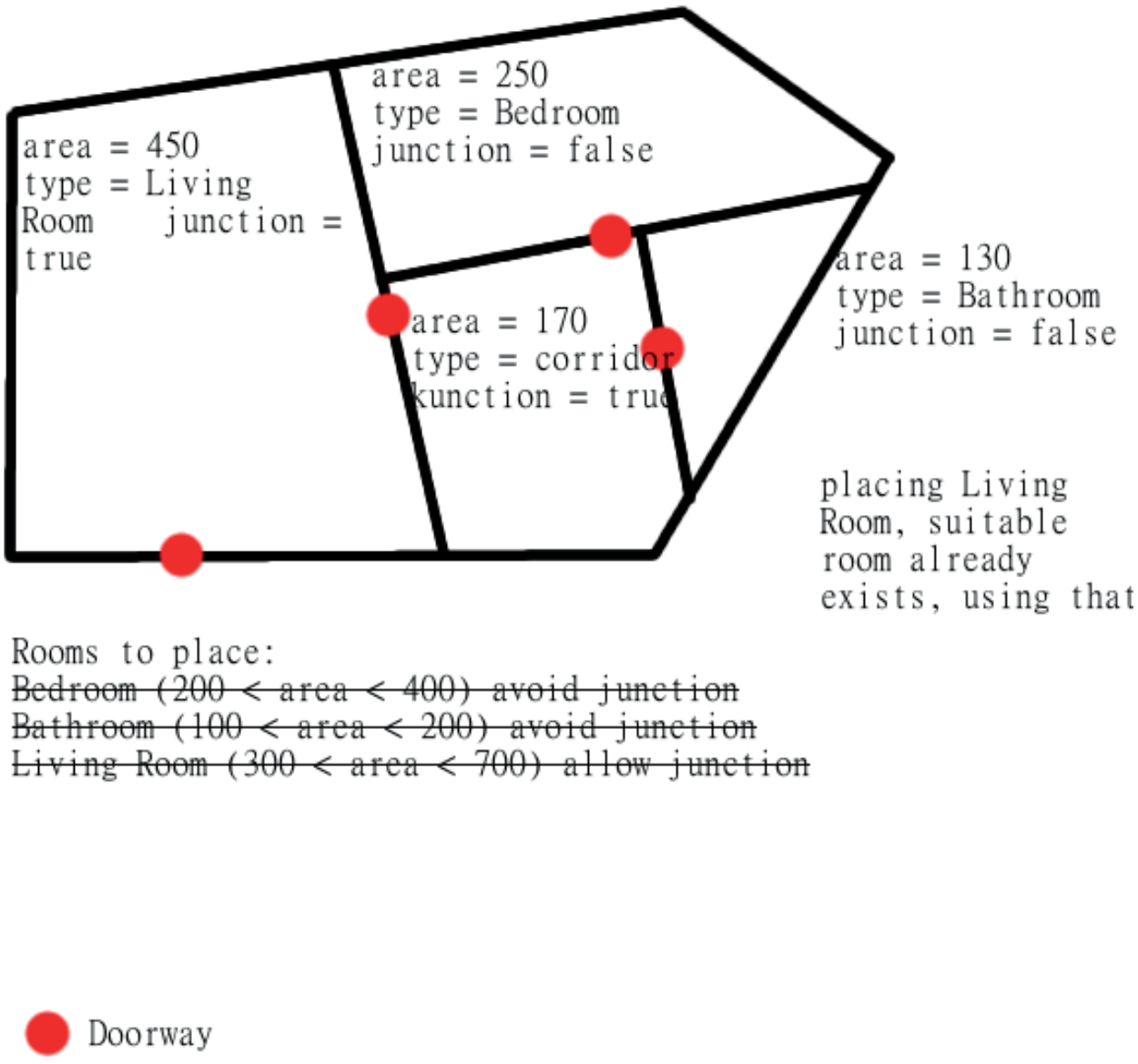


Figure 5: Finished apartment, all rooms placed

.1 Appendix B

Figure 6: Roads with lines placed on both sides, direction is reversed for opposing sides

Figure 7: Lines are split whenever intersecting the middle of a road, recursively creating new lines

Figure 8: Lines are checked for collisions against each other, intersection points are saved as polygon points. Each line keeps track of potential parent line and child line.

Figure 9: In every polygon where the last point has the first point as child, a complete polygon and is formed and saved. In this limited example, two polygons are created

Bibliography

[1] Pascal Mller, Yoav I H Parish. Procedural Modeling of Cities. ETH Zrich, Central Pictures, Switzerland, 2001.

[2] Alexander Dahl, Lars Rinde. Procedural Generation of Indoor Environments. Chalmers University of Technology, Gteborg 2008

[3] Sean Barrett. L-Systems Considered Harmful. 2007-2009

[4] Evan Hahn, Prosenjit Bose, Anthony Whitehead. Persistent Realtime Building Interior Generation. Carleton University, 2006

[5] Pascal Muller, Peter Wonka, Simon Haegler, Andreas Ulmer, Andreas Ulmer, Luc Van Gool. Procedural Modeling of Buildings. ETH Zurich, Arizona State University, K.U. Leuven 2006

[6] Ignacio Garcia-Dorado. Fast Weather Simulation for Inverse Procedural Design of 3D Urban Models. Purdue University, Google Research 2017

[7] Przemysaw Prusinkiewicz The Algorithmic Beauty of Plants 1990

[8] George Kelly, Hugh McCabe A Survey of Procedural Techniques for City Generation ITB, Dublin, Ireland 2006

[9] Paul Guerrero, Stefan Jeschke, Michael Wimmer, Peter Wonka Edit Propagation using Geometric Relationship Function s Vienna University of Technology, KAUST, IST Austria 2014

[10] George Kelly, Hugh McCabe Citygen: An Interactive System for Procedural City Generation Institute of Technology Blanchardstown, Ireland

[11] https://www.nomanssky.com/

[12] http://www.esri.com/software/cityengine

[13] https://www.rockstargames.com/grandtheftauto/

Programmer attempting to steal artists job

Tobias Elinder

LTH
tobiaselinder@gmail.com

Automatically generating environments is a very appealing concept. Instead of manually creating environments you can create rules which in turn create the content. This project has used this idea to build whole cities, including interiors, automatically (procedurally).

What are the actual problems that procedural generation solves? Well for one - it saves a lot of manual labor. You also don't have to store things in computer memory. Imagine creating a huge world manually, for example in a game. A huge map would occupy a lot of space on your hard drive, and if you make the world big enough it wouldn't fit at all! Anyone with experience playing computer games should be familiar with the GTA series, which boasts beautiful and intricate environments, but they would be even more interesting if the maps and cities could be made much bigger, this is where procedural generation shines. With procedural generation it's even possible to have infinite worlds, only generating areas when close, like a storyteller reacting to a person's choices by improvising new content on the fly. Procedural generation is also more dynamic than manual generation in that it's possible to tune parameters of the generator for very different results very quickly, without having to rebuild a massive world by hand.

This project has looked at how whole cities can be created with procedural algorithms, from the toilet in a small apartment to whole neighborhoods and structure of road networks. You'd think this was a well explored research area but it isn't. There have been a couple of projects working on procedural city structure and a few working on building interiors before, but few if any has ever tried to do everything simultaneously, which seems like a logical end goal. The vision for this project was a complete city without shallow facades, meaning that a window on a building far away should also lead in to a room when getting close enough.

What makes procedural cities/buildings so hard is the need for coherency balanced against unpredictability. Generating a mountain is easy, no coherency gives chaotic results, and nature is chaotic! Make an office with similar chaotic techniques and you'll end up with very unsafe work environments with cliffs in the middle of the office floor and windows behind bookshelves. Make the rules too strict and you'll end up with a sterile and predictable set of rectangular rooms where everything is always in the same location. To avoid these two extremes you have to find a middle ground where the environment is structured and coherent, but unpredictable. To do this I created a top-down method which generates roads first, then plots from road structures, then buildings from plot structures, the apartment and rooms from building structures, and so on. There is a lot going on beneath the surface, but the general algorithms themselves are actually quite simple. For a peek at the algorithms in action, see this demo.