Generating realistic city boundaries using two-dimensional Perlin noise

Steven Wijgerse

February 12, 2007

University of Twente
Cluster: Human Media Interaction (HMI)
Department of Electrical Engineering, Mathematics and Computer Science (EEMCS)

Generating realistic city boundaries using two-dimensional Perlin noise

Graduation thesis for the Doctoraal *program,*
Computer Science

by Steven Wijgerse,
student number 9706496

February 12, 2007

Graduation Committee
dr. J. Zwiers
dr. M. Poel
prof.dr.ir. A. Nijholt
ir. F. Kuijper (TNO Defence, Security and Safety)

University of Twente
Cluster: Human Media Interaction (HMI)
Department of Electrical Engineering, Mathematics and Computer Science (EEMCS)

Abstract

Currently, during the creation of a simulator that uses Virtual Reality, 3D content creation is by far the most time consuming step, of which a large part is done by hand. This is no different for the creation of virtual urban environments. In order to speed up this process, city generation systems are used.
At re-lion, an overall design was specified in order to implement such a system, of which the first step is to automatically create realistic city boundaries. Within the scope of a research project for the University of Twente, an algorithm is proposed for this first step.
This algorithm makes use of two-dimensional Perlin noise to fill a grid with random values. After applying a transformation function, to ensure a minimum amount of clustering, and a threshold mechanism to the grid, the hull of the resulting shape is converted to a vector representation. This result can be used as a city boundary in other parts of the specified city generation system.
In order to test these results for realism, two quantifiable properties of realistic city boundaries are specified. The first property is the fractality of the boundary, as well as a fractal dimension between 1.18 and 1.28. The second property is the newly defined minimum *enclosed area ratio*, which is the relation between the enclosed area and the maximum enclosed area for an equal maximum diameter. The minimum enclosed area ratio of a realistic city boundary is estimated at 6.35%.
An experiment is described, in which a large amount of results, obtained from different sets of input, of the proposed algorithm are tested for fractal dimension and enclosed area ratio. By analyzing the results of this experiment, optimal ranges of input values are found, for which 73.0% of all generated shapes fulfill the given definition of a realistic city boundary. Furthermore, it is concluded that it is currently not possible to find a correlation between the input values and the fractal dimensions of the output.

Contents

Part I

Theory

Chapter 1

Introduction

1.1 Situation

Virtual Reality is the name for a technology that allows people to interact with a simulated environment. Although usually done through image and sound, currently simulations also exist that track human motion, and provide tactile feedback. Through the on-going increase in computing power, and recently in specialized graphics hardware, very detailed simulations are possible these days.
It is the goal of re-lion, a company based in Enschede, The Netherlands, to create training and simulation software solutions for use in the health-care and defense sectors. Within re-lion a range of VR product has been developed over the past years, including 3D visualization software (Lumo) and a system for easy creation of distributed physical simulations and visualization set-ups (Lumo Scenario[1]). The use of this software makes it possible to create an operational simulator, such as the Scooter Simulator[2], within a relatively short time span.
Building a virtual reality simulation includes steps such as defining a hardware set-up, and creating necessary software connections to this hardware. Content has to created, which includes the virtual world and its dynamics representation. Furthermore, the hardware and software needs to be installed, tested, and maintained.
Currently, content creation is by far the most time consuming of these steps. Both the visual objects as well as the dynamics representation are usually created by 3D designers, using software packages such as Autodesk 3ds Max[3] and Autodesk Maya[4]. As a last step, in the case of a Lumo Scenario simulator, the separate 3D objects are combined into a 3D virtual world using another product, Lumo Builder.
Within re-lion, an improvement to this system has been developed over the last years, by creating a library of frequently used objects, that can be placed in a virtual world at will. These *building blocks* include objects such as trees, buildings, benches, road signs, etcetera. Although this method saves a large amount of time, the placement of the objects still has to be done by hand, and even if a large library is used repetition is often easily visible. Another improvement is the automatic generation of visual and physical road representations from a set of spline definitions. This used to be a very labor intensive process when done by hand. Together, these pieces of software form a Builder package, with the aid of which city landscapes can be produced by defining the road network as a set of splines, and adding buildings in between.

[1] http://www.lumoscenario.com
[2] http://www.re-lion.com/nonjs.php?sm=145
[3] http://usa.autodesk.com/adsk/servlet/index?id=5659302&siteID=123112
[4] http://usa.autodesk.com/adsk/servlet/index?siteID=123112&id=7635018

1.2 Generating city content automatically

To speed up the process described above even further, re-lion is in need of a system that can fully automate parts of the content creation step. Since the creation of city landscapes takes a large amount of time, the system should focus on this particular type of content. Summed up, the desired system has the following characteristics:

- Automate city content creation
- Should provide some form of realism
- Integrates into Lumo Builder product
- Little amount of input, simple parameters
- Extensible through scripting
- Easy-to-use
- Fast

At re-lion, an overall design has been developed to implement the above features. This design divides the generation of a city into four separate parts. The first step generates the boundaries of the city. The result of this step is a vector representation of a closed shape, enclosing the area belonging to the city.
The second step of the system defines the main transport axes through the city. A transport axis is a line through the city where transportation is fast, easy, and relatively cheap. This, more abstract, concept of transport axes should not be confused with real transportation routes such as highways. Other options such as an important railway, a subway system, or a canal or river also qualify.
In the next step, the available land enclosed by these boundaries is categorized with class-information such as *residential area* and *recreational area*, as well as information about density. In the final step the actual content placement is done, adding details such as smaller roads and actual houses, trees, etcetera.
Detailed information about this system in planning can be found in Section 2.3.2.

1.3 Research project

A research project was defined to contribute to the system described before. This project focuses on the first step of city generation, realistic city boundaries. To this end it was chosen to propose an algorithm using Perlin noise, because of positive experiences of the author in previous attempts to create natural patterns. Since a city boundary spreads in at least two dimensions, the algorithm also makes use of a two-dimensional version of Perlin noise.

1.3.1 Project goal

As mentioned before only one part of the whole of city content generation was addressed. The goal of this project is to answer the following research question:

> *How can two-dimensional Perlin noise be used for automatic generation of realistic city boundaries within easy-to-use city content creation systems?*

This research question encompasses the following phrases:

- **Two-dimensional Perlin noise** - Perlin noise is a form of random data with a natural coherency. It is often used in computer science to create movement, shapes and behavior that is supposed to look realistic or natural. Perlin noise can be created in any desired amount of dimensions, two-dimensional being closest to the nature of a city boundary map.
- **Realistic** - As a requirement of re-lion, it must be proven that the city boundaries created feature a certain level of realism. In order to reach this goal, it needs to be defined what *realism* means for city boundaries.
- **City boundaries** - A city is largely defined by the city limits, the border between the ground that belongs to the urban area and the ground that does not. As such, city boundaries can be seen as a closed shape.
- **Easy-to-use** - The goals that have been set by re-lion for this system are that it can be used by anyone with some form of higher eduction. Since most people are not well-informed on city structures, user preferences should be self-explanatory and kept to a minimum.
- **Within city content creation systems** - It is important to review how automatic generation of city boundaries fits into the process or workflow of a city generator. This not only applies to the aforementioned design of re-lion, but also to existing city generators.

1.3.2 Research questions

Throughout this project, the following sub-questions are used as a guideline. These questions were formulated by analyzing which pieces of information are needed to answer the central research question.

1. How does a self-contained hull creation algorithm fit into full city generation systems?
   (a) What methods of city generation are used?
2. What defines a realistic shape of a city?
   (a) What are general geometric properties of a city?
   (b) Do cities have a fractal boundary?
      i. What defines a fractal?
      ii. How can fractal dimension be calculated?
3. How does Perlin noise work?

(a) What is input for Perlin noise?

i. What is a good random number generator to feed Perlin noise with?

(b) What is the output of Perlin noise?

4. Is Perlin noise a form of fractal?

5. Can the proposed algorithm be used to generate a realistic city boundary fast?

(a) Do the generated shapes fulfill the earlier "what defines a realistic shape of a city?" question?

i. Do the generated shapes fulfill the earlier "what are general geometric properties of a city?" question?

ii. Is the form fractal according to the earlier found definition?

(b) What parameters affect the result of the algorithm?

(c) How do these parameters affect the result of the algorithm?

i. Is the output stable for the same input?

1.3.3 Methodology and structure

In order to be able to find a satisfactory answer to the overall research question and the related sub-questions, this work follows a twofold approach, comprising a literature study on the one hand, as well as empirical research on the other hand. The empirical part consists of an experiment testing a proposed algorithm by comparing its results with values found in the literature study.
First, in Chapter 2, the results of the literature study are presented. In this chapter a description of fractals is given and some theory regarding the shape of cities laid out. To answer question 1a, an overview is given of some existing city generators and their properties. After this a description of Perlin noise is given and a short introduction to the input for Perlin noise: Random number generators. The chapter ends by summarizing the answers to research questions 1 through 4.
In the theory chapter (Chapter 3) a city boundary generation method is introduced that uses two-dimensional Perlin noise. It is described how a vector representation of a city boundary is created from an input stream of random data.
Part two of this thesis describes the experiment that was carried out to answer research questions 5 through 5(c)i. Its first chapter defines the methods used to gather results and the parameter settings of the two tests that were run. The next chapter covers the design of the software that was used to run the experiment. Finally, Chapter 6 shows and evaluates the results that were obtained.
In part three, as a last step, conclusions are drawn from the result evaluation. This is done by answering the main research question through use of the answers on the sub-questions. This thesis is then closed off with some recommendations for an implementation of this method into a city generator, and some pointers for future research.

Chapter 2

Literature

This chapter contains the results of the literature study of this project. Research was done on five different topics that are all addressed in the aforementioned research questions. Every section first contains a detailed description of one of these topics and closes with an overview of the particular research questions it answers.

2.1 Fractals

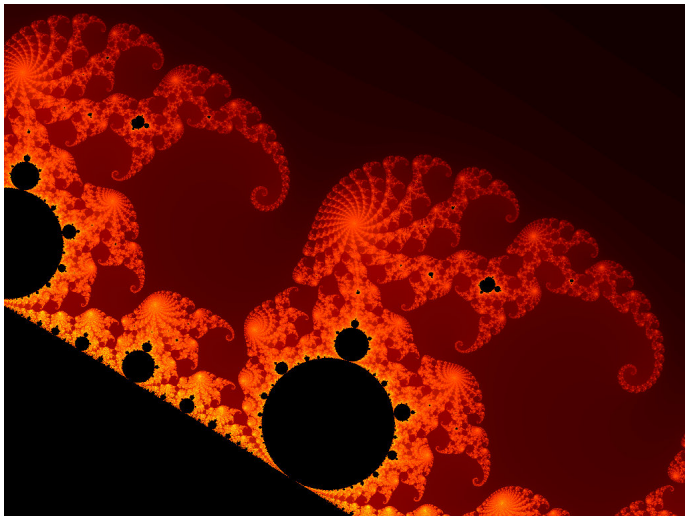


Figure 2.1: A picture of the Mandelbrot set

Although the first fractal forms were already described long ago[1], The word *fractal* was first introduced by Benoît B. Mandelbrot to describe shapes that are independent of scale. This means that they look similar independent of the scale they are viewed on[8]. According to multiple sources, fractal forms have the following properties:

- **Self-similar** - The self-similarity as described above can be seen in Figure 2.1[2]. One large black circle is visible here, with smaller black circles around it. At this level of discretization it is still visible that one of these smaller circles is surrounded by other, smaller, black circles. In the Mandelbrot set this will keep on happening independent of the level of *zoom* of the picture.

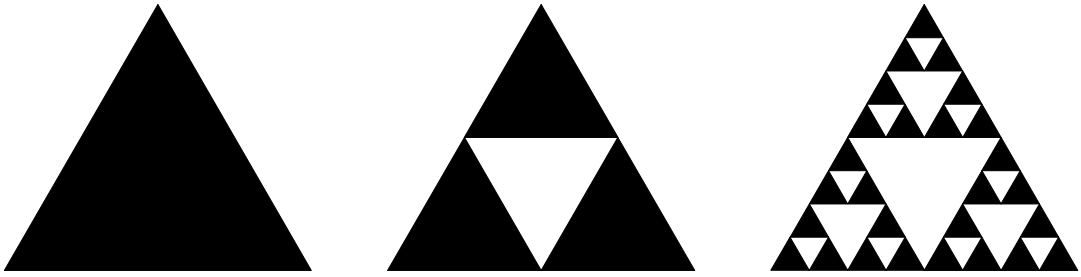
[1] `http://en.wikipedia.org/wiki/Fractal`

[2] This picture was released into the public domain and was taken from `http://en.wikipedia.org/wiki/Image:Mandelpart2.jpg`

- **Initiator** - Fractals have an initiator, which is the shape used to describe the starting point of the fractal. In the example of the *Sierpinsky gasket* in the next section, this is a simple triangle. Other regularly used forms include lines and squares.
- **Generator** - Fractals have a generator, which is a function that describes the transformation of a shape into another shape. To obtain the shape of the fractal, this function is applied recursively to the initiator.
- **Hierarchical** - Because of the generator function, fractals are highly hierarchical. The initiator defines the top of the hierarchy, after which every iteration of the generator function defines a new level in the hierarchy.
- **Irregular** - Most fractals, as for instance the Mandelbrot set shown in Figure 2.1, are complex enough to display a certain level of irregularity. On the other hand, a fractal definition that complies to the first four properties in this list does not necessarily need to look irregular, as can be seen later on in Figure 2.4.
- **Natural appearance** - Because fractals are for some part irregular forms, but also show a regularity in that the iterations are self-similar, they are often used to create virtual plants, landscapes, trees, mountains, coastlines and other elements of nature. However, this is not a quantifiable property.

2.1.1 Example: Sierpinsky gasket

A good example of a fractal definition and the resulting shape is the *Sierpinsky gasket*. It starts with a simple triangle as its initiator. The generator function takes the original triangle as its input, and outputs three smaller triangles that exactly fit in the original shape. The hierarchy now consists of a top level with one triangle and a second level with three smaller triangles. In the next iteration each of the three smaller triangles is transformed by the generator function into three even smaller triangles, leading to a new level in the hierarchy with nine items.
This process can be repeated another time, which results in the shape shown in Figure 2.2(c). When zooming in on the shape, the generator function can be applied infinitely, resulting in an image that will always look similar.



(a) Initiator

(b) Generator

(c) The Sierpinsky gasket after three iterations of the generator function

Figure 2.2: The Sierpinsky gasket

2.1.2 Fractal dimension

Two dimension definitions are usually used to describe the concept of fractal dimension, *Hausdorff dimension* [13] and *similarity dimension*. Mostly it is claimed that the Hausdorff dimension is used,

after which a description of the similarity dimension is given, probably because of the complexity of the former. In this section the concept of similarity dimension is described, both because of its understandability and the fact that it fits the self-similarity property of fractals. The description and formulas in this subsection are derived from publications [1], [8] and [13].
As self-similarity demands, every iteration of the generator delivers the same shape, scaled and rotated, and multiple times. We define r to be the scale factor that is used to create the parts of the new set in each iteration. N is defined as the amount of these scaled parts that make up the new set. In the Sierpinsky example above this amounts to the values $r = \frac{1}{2}$ and $N = 3$.
A fractal starts with its initiator. In the first iteration of applying the generator, the multiplier for the amount of shapes N_1 equals the main multiplier N. This also applies to the second iteration, so the total multiplier N_2 for this level is defined as

$$N_2 = N_1 \cdot N = N \cdot N = N^2 \tag{2.1}$$

It can easily be seen that after the k^{th} iteration this becomes

$$N_k = N_{k-1} \cdot N = N^k \tag{2.2}$$

For the scaling factor r there is an analogous description: During the first application of the generator function, the parts of the set are scaled by a factor r, so the total scaling after one iteration r_1 is the same as main scale factor r. In the second iteration the scaling factor is applied again, so

$$r_2 = r_1 \cdot r = r \cdot r = r^2 \tag{2.3}$$

Also here it can be easily derived that after the k^{th} iteration we have a total scaling factor

$$r_k = r_{k-1} \cdot r = r^k \tag{2.4}$$

Fractal dimension D is defined as the relation between N_k and r_k. This (exponential) relation can be found by solving

$$N_k = (r_k)^{-D} \tag{2.5}$$

This can be solved using a logarithm:

$$\ln N_k = -D \cdot \ln r_k \tag{2.6}$$

$$D = -\frac{\ln N_k}{\ln r_k} \tag{2.7}$$

Using Equations 2.2 and 2.4 we get

$$D = -\frac{\ln(N^k)}{\ln(r^k)} \tag{2.8}$$

$$D = -\frac{\ln N}{\ln r} \tag{2.9}$$

In the Sierpinsky example above where $r = \frac{1}{2}$ and $N = 3$, using Equation 2.9 leads to a fractal dimension of $D \approx 1.585$.

(a) Initiator (b) Generator (c) The fractal after three iterations

Figure 2.3: A fractal that spreads in one-dimensional Euclidean space

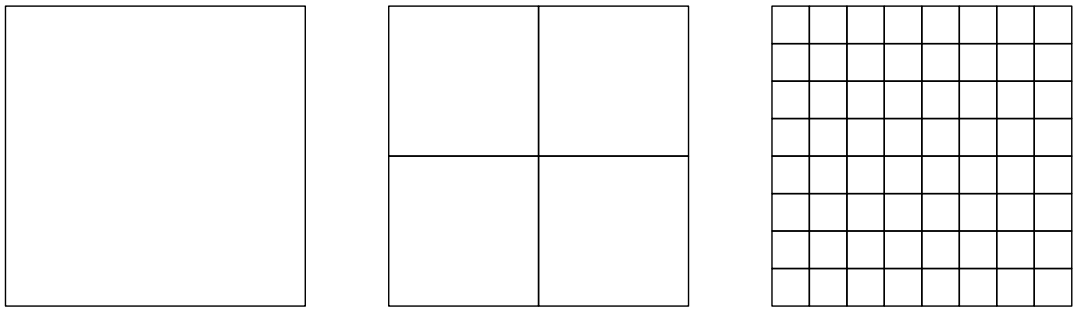
2.1.3 Meaning of dimension

There are many views on the meaning of fractal dimension. Often it is mentioned that it denotes how *rough* the shape is [8]. It is also described that it is a form of dimension that can be a fraction because certain shapes *obviously have a dimension higher than one (a line), but lower than two (a plane)* [1], of which the Sierpinsky gasket would be an example.
To illustrate the connection to Euclidean dimension, this section shows two example fractal definitions. The fractal shown in Figure 2.3 is defined within one-dimensional Euclidean space. It has a simple line as its initiator, and the generator divides the line into two equal parts. In terms of the last section this means that the generator introduces $N = 2$ self-similar shapes scaled by $r = \frac{1}{2}$. According to Equation 2.9 this leads to a fractal dimension of

$$D = -\frac{\ln 2}{\ln \frac{1}{2}} = -\frac{\ln 2}{\ln 2^{-1}} = \frac{\ln 2}{\ln 2} = 1 \tag{2.10}$$

In contrast, the fractal of Figure 2.4 uses two-dimensional Euclidean space. The initiator is a simple square, and the generator divides the square into four equal smaller squares. In this case the generator introduces $N = 4$ self-similar shapes, but the scale factor stays the same at $r = \frac{1}{2}$. This leads to a fractal dimension of

$$D = -\frac{\ln 4}{\ln \frac{1}{2}} = -\frac{\ln 2^2}{\ln 2^{-1}} = 2\frac{\ln 2}{\ln 2} = 2 \tag{2.11}$$



(a) Initiator (b) Generator (c) The fractal after three iterations

Figure 2.4: A fractal that spreads in two-dimensional Euclidean space

A good description of fractal dimension is[3] that it is *a statistical quantity that gives an indication of how completely a fractal appears to fill space, as one zooms down to finer and finer scales.* The Sierpinsky gasket obviously does not fill two-dimensional space, since parts are left open, and thus it is has a fractal dimension smaller than two.

[3] `http://en.wikipedia.org/wiki/Fractal_dimension`

2.1.4 Estimating fractal dimension

The fractals discussed in the last sections are all mathematical descriptions of self-similar shapes, which makes it possible to calculate fractal dimension. When such a mathematical description is not available an estimation has to be made. Examples of such a case would be to determine the dimension of a coastline or of existing urban boundaries.
One method of making such an estimation is the *Structured walk* [1] method. This method approximates the original shape by a sequence of connected line segments of equal length. The length of these segments or *chords*, represented by r, defines the amount of segments needed to traverse the original shape, represented by N. N_0 and N_1 correspond to chord lengths r_0 and r_1. If a step from r_0 to r_1 is taken, dimension can be defined analogously to Section 2.1.2:

$$\frac{N_1}{N_0} = \left(\frac{r_1}{r_0}\right)^{-D} \tag{2.12}$$

To simplify[4] the equation for later use, r_1 can be expressed related to r_0 as $q_1 = \frac{r_1}{r_0}$. Written differently, the equation now reads

$$N_1 = N_0 {q_1}^{-D} \tag{2.13}$$

To find out if the approximated shape is in fact fractal, this must not only be true for N_1, but for all $N(r)$, being a function of the segment length. In the same manner as above, q is defined as $q = r/r_0$.

$$N(r) = N_0 q^{-D} \tag{2.14}$$

To simplify D from this equation, a logarithm can be used:

$$\ln N(r) = \ln N_0 - D \ln q \tag{2.15}$$

Since $\ln N_0$ is a constant, D can now be determined from a set of matching Nr and r (or q) pairs using

$$D = \frac{\ln N_0 - \ln N(r)}{\ln q} \tag{2.16}$$

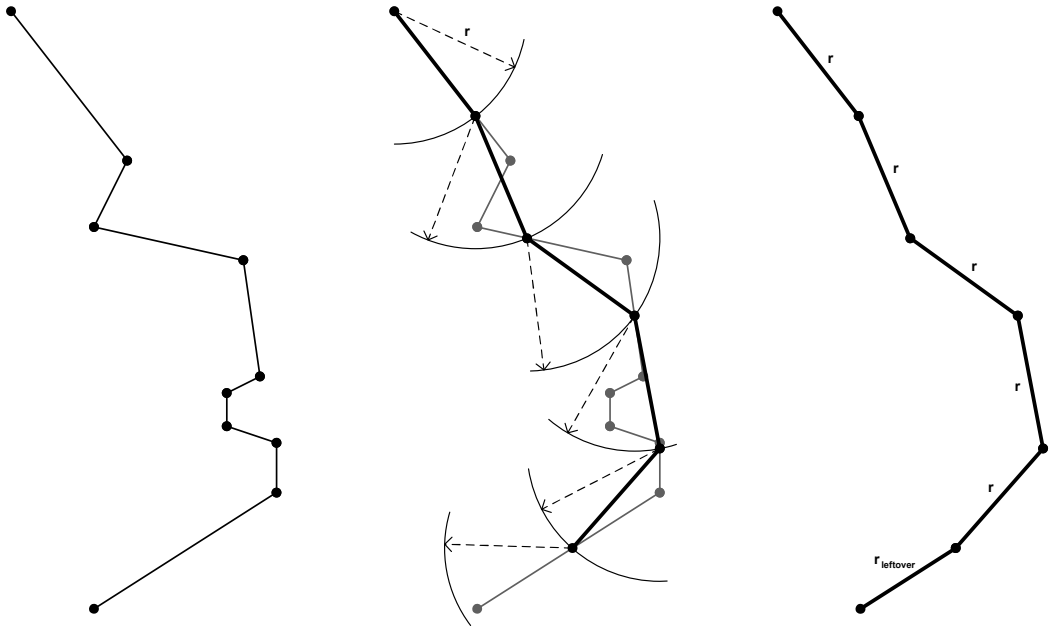
Algorithm

An example of the input and result of the *Structured walk* algorithm is shown in Figure 2.5. To obtain the values of N for different sizes of r the original shape is approximated using the following steps:

1. Define the current position as the first vertex of the dataset.
2. Write the current position to the output buffer.
3. Draw a circle around the current position with radius r.
4. Search for the first chord in the input after the current position that has an intersection with the circle.
5. Make this intersection the current position.
6. Resume with step 2.

This algorithm continues until no further unused input chords can be found that intersect with the circle. Unless the last chord exactly closed the gap to the beginning point, there is an open piece left over. This gap is closed with a chord of length smaller than r.

[4] In [1] this step is not taken. This leads to trouble simplifying the equation later on, which is why the authors decide that ${N_0 r_0}^{-D}$ is constant. This is odd at least, since D is the variable that is sought.



(a) The input dataset

(b) The circles with radius r and the chords that are formed

(c) The output dataset with five chords of length r and one of length $r_{leftover}$

Figure 2.5: The structured walk method

When the algorithm is finished there are $N-1$ chords of length r and 1 leftover chord with length $r_{leftover} < r$. The formula $(Nr + r_{leftover})/N$ delivers the average output chord length. To rule out any coincidental *good* or *bad* fits, where N would be much higher or lower by coincidence, the algorithm can be started once at every vertex in the input dataset. The averages of these results are then taken as the correct values for N and r.

2.1.5 Summary

To summarize this section, the following research questions from the introduction can now be answered:

- **2(b)i. What defines a fractal?** - Fractals can be defined by describing their qualitative properties, as they are laid out in the beginning of this section: Natural appearance, self-similarity, the use of an initiator and generator, and the facts that they are hierarchical and irregular. A quantitative way to define a fractal is the notion of *fractal dimension*; In a true, mathematical, fractal this dimension is constant, independent of scale.
- **2(b)ii. How can fractal dimension be calculated?** - The fractal dimension of an irregular shape can be calculated using the *Structured walk* method described in this section.

2.2 Realism in city boundaries

As was mentioned in the introduction, generated city boundaries should have a certain level of realism. Since it is a subjective quality, it is hard to define quantifiable properties of realism in city boundaries. To enable automatic testing for realism, two properties that can be measured were chosen. It is assumed that, for use within the context of this research project, a high enough level of realism is attained when the generated city boundaries satisfy these properties.

2.2.1 Fractal properties

As explained in the last section the word fractal was introduced by Benoît B. Mandelbrot. He published a paper in 1967 describing the problem of measuring the length of the coastline of Britain. According to his paper, this length depends on the size of the *stick* used to carry out the measurement. When the stick gets smaller, more detail in the coastline is picked up, and the total length will get bigger. This specific example is shown in Figure 2.6[5]. This method has evolved into the *Structured walk* method as described in the last section.

Figure 2.6: Measuring the coastline of Britain

Research such as this has also been done with city boundaries. The authors of [1] have done *Structured walk* calculations on datasets of existing cities, and found that these usually show a fractal dimension between 1.18 and 1.28. With this, they have developed a quantitative method to classify city boundaries.

2.2.2 Enclosed area

When a city boundary is generated randomly by a computer algorithm, it is not enough to measure its fractal dimension, and assume that it qualifies as a realistic city boundary only because it lies between 1.18 and 1.28. A reason for this is that it is possible for a boundary with a certain fractal dimension to have an area of almost zero. A simple example of this situation is when a fractal line with a correct fractal dimension is duplicated and put directly next to the original. When the endings are then connected it would be a closed shape. This extreme case will have an area of zero, while still fulfilling the fractal dimension requirement.
Therefore another, geometric, property of city boundaries is addressed, the enclosed area ratio. Since cities offer room to its citizens, this means that these city boundaries enclose an area. The enclosed area ratio is defined as a function of the maximum diameter that can be found within the boundary

[5] Image taken from `http://tinyurl.com/noeld`. The image was released under the terms of the GNU Free Documentation License

shape, and represents the ratio of the real enclosed area with respect to the maximum enclosed area for that diameter:

$$r_{encl} = \frac{A}{A_{max}} \tag{2.17}$$

The maximum possible enclosed area is easily defined as the area of a circle:

$$A_{max} = \frac{1}{4}\pi d^2 \tag{2.18}$$

The minimum for the enclosed area ratio can be obtained by, for a constant value of d, taking the lowest value possible for A, and the highest value possible for A_{max}. For the minimum possible enclosed area A_{min} an estimation has been made, by defining a village of 200 meters wide times 4 kilometers long. This leads to a minimum area of

$$A_{min} = 0.2\ km \cdot 4\ km = 0.8\ km^2 \tag{2.19}$$

For this case the maximum possible diameter is $d = 4.005\ km$, which leads to a maximum enclosed area of $A_{max} \approx 12.597\ km^2$. Therefore

$$r_{min\_encl} = \frac{A_{min}}{A_{max}} \approx \frac{0.8\ km^2}{12.597\ km^2} \approx 0.0635 = 6.35\% \tag{2.20}$$

2.2.3 Summary

To summarize this section, the following research questions from the introduction can now be answered:

- **2a. What are general geometric properties of a city?** - An estimation has been made for the minimum *enclosed area ratio*, the relation between enclosed area and maximum enclosed area for equal diameter. This estimation results in a value of $r_{min\_encl} \approx 0.0635 = 6.35\%$.
- **2b. Do cities have a fractal boundary?** - Yes, cities have a fractal boundary with a fractal dimension between 1.18 and 1.28.

2.3 City generators

This section is the result of a search for existing city generators. First, an overview of some common properties of city generators will be given. Then some existing systems are described in terms of their compatibility with the proposed method of generating realistic city boundaries as a separate step.

2.3.1 Properties of city generators

To create an overview of the city generators described in this section, a few common properties are described here.

Offline / online generation

The available systems can roughly be divided into two categories based on offline or online generation. A system that uses offline generation creates the entire virtual world, in this case the entire virtual city, before it is needed for display or simulation, hence offline. An advantage of this type of system is that cities can be edited by hand before display. A disadvantage of these systems is that, since they are not designed to produce data directly when the system needs it, city generation takes more processing time. This can be either by design, i.e. an algorithm that is too slow for online use, or because of the fact that the implementation has not been optimized for online use. Other disadvantages are needed disk space - including saving times, loading times and memory usage - and the fact that the entire city needs to be created when the application starts, because it cannot be automatically enlarged on demand.
Online systems can be more flexible than offline systems, by creating parts of the virtual world at latest at the moment they are needed by the visualization or simulation system. Loading times are mostly lower, because at application start only the parts need to be loaded or generated that are directly visible to the user. Of course, there is no possibility for an author to control the layout of the city, because it does not exist yet at application distribution.
Some systems are a hybrid of offline and online generation, when they for instance create the streets and allotment online, but fill them with previously designed building blocks.

Urban realism

The systems described can easily be divided into two groups by the fact if they use any form of realistic properties to build up a city. Two of these properties were introduced in Section 2.2. Furthermore, in real cities there are certain rules that function both on a bigger scale, eg. the fact that concentric circles seem to form around one or more central districts, and on a smaller scale, such as the fact that in American cities houses are often placed in *blocks*, forming a strict grid structure of roads. More about these patterns on a bigger scale can be read in Appendix B.

Growth models (iterative) or non-iterative models

Most of the available systems implement generation of a virtual city at a certain cross-section in time. A city boundary is generated on an allocated piece of land, which is then divided into lots and filled with a road system. These systems assume that by the generation models that are used, the city that is created will look as though it grew over time, as existing cities (with a few exceptions) naturally did.
Another type of system will not only look at this one desired cross-section in time, but simulate natural growth of a city over the years before that. These cities change by cumulative adding and removal of buildings and roads, often reusing formerly occupied space. These changes occur at the most basic levels, but when simulated successfully the same urban patterns will emerge as in non-iterative systems. An example of an iterative system is the one proposed by re-lion as described in Section 2.3.2.

Generation steps and order

As described in Section 1.2 there are four phases of city generation that are interesting in this research project. Of course, not every system described necessarily uses all four phases. Also, these phases can be ordered differently in the available system descriptions. The distinguished phases are:

- Boundary creation
- Main transport axes
- Land use classification
- Details, smaller roads, placement of objects

2.3.2 Planned system at re-lion

At re-lion, a design was made for an offline city generator, that creates urban structures with a certain level of realism. This system follows the four steps as described in Section 1.2. The design of the city boundary step is to be defined further within the scope of this research project.
Two of the other steps have already been planned in more detail. More detailed descriptions of these steps are provided here.

Land use classification

An iterative model was designed that incorporates items from *urban ecology*. From the beginning of the 20th century, models of urban social structure have been developed. These models are focussed on the socioeconomic patterns in cities, for instance ethnic communities. Although most of these models have been developed a long time ago and have received serious criticism, they are still used today as a basis for explaining urban processes. More information on these models of urban ecology can be found in Appendix B.
In the iterative model entities will be instantiated belonging to one of the five social classes. In the spirit of the model of Burgess, these classes will take up a different amount of space, for simplicity expressed in square meters, and compete with each other for the available space. The universal item that can be competed with will be money, in the form of rent that they can afford.
To accommodate the need for space for these entities, the city within its boundaries will be divided into cells. These cells should not be viewed as a city block surrounded by streets, but as cells on a more abstract level. Every cell will occupy an equal amount of square meters of direct land use, but the amount of square meters available will be defined by an extra property, the average amount of floors that the buildings there have.
Because this is an iterative model, entities need an incentive to move through the city. This is accomplished with a fitness function, defined separately for every class. These fitness functions include the urban ecology factors such as the pull factor of living near a commercial center, near a transport axis and living amongst people of the same SES[6], but also include the price paid per square meter. When the outcome of an entities fitness function, due to changes in its environment, sinks too low, the entity will look for any other cell in the city, based on the same fitness function. When more space is needed within a cell the average amount of floors can be raised to create more effective square meters.
Demand for a certain cell will regulate the ground prices. As in reality, this should accomplish that cells where large amounts of square meters are empty, the ground prices will drop. Cells where demand is high will raise their prices, only making it possible for higher spenders, such as commercial activities, to acquire room. It is also possible to add more floors to the cell, dividing the higher ground price over more square meters, making them more affordable again.
Some restraints will be put into place to avoid odd layouts. As in reality, the average amount of floors per cell cannot change rapidly. Furthermore construction costs per effective square meter

[6]SES: Social Economic Status.

rise exponentially when buildings are higher than 20 floors[7] which makes buildings with 300 floors unaffordable to any entity in the system.

Transport axes

Transport axes are needed for the functioning of the urban ecology models, so they need to be generated before the ecology models can be simulated. To create these, a slightly adapted version of the highway creation method in the *City engine* of Pascal Mueller[8] will be used. In this system, described in Section 2.3.5, highways are used to interconnect highly populated areas in the city, defined beforehand.
The locations of these higher populated can be defined randomly or specified by the user in the form of an image map. According to the description of Hoyt and common bid-rent theories, the internal transport axes will also radially be connected to points outside of the city, to create a realistic pattern.
The more abstract concept of transport axes described here should not be confused with real highways. A transport axis is a line through the city where transportation is fast, easy and cheap. This can indeed be a highway, but other options such as an important railway, a subway system, or a canal or river also qualify.

Summary

- System: offline
- Urban realism: uses both realistic city boundaries as well as realistic land-use patterns
- Iterative growth process
- Steps:
  - Boundary creation
  - Main transport axes
  - Land use classification
  - Details, road network and placement of objects

2.3.3 eSCAPE project

Within the eSCAPE project, running until the year 2000 at the university of Manchester, two projects have been carried out that were focussed on the automatic creation of urban landscapes. These were the *City Generator* of Eduardo Hidalgo and the *Virtual CityScape* project.
The *City Generator* supplies an editor to enable a user to lay out a street map, after which the pieces of land in between are automatically filled with buildings. Because the road maps are created by hand, they are proportionally sophisticated to the amount of work that is put in by the user. The filling of the open land is done with a small set of high buildings, which makes it impossible to create the feeling of a smaller city. Little attention has been paid to the filling of the ground between the buildings and the roads. The result is a city without any form of realism. [12]
In the *Virtual CityScape* project the road network is created automatically in a fashion similar to the method of Stefan Greuter, which will be discussed in Section 2.3.4. The available virtual surface is handled as a grid of which the cells are categorized as infrastructure or building area. Compared to the *City Generator* this saves a lot of work and the results shown have substantially more detail. It is unclear whether the buildings are generated automatically at the time of placement or chosen from a library of handmade objects. [22]

[7] Source: The World Housing Encyclopedia, Earthquake Engineering Research Institute. `http://www.world-housing.net`

[8] `http://www.vision.ee.ethz.ch/~pmueller/`

Usability

Neither of both projects is very well documented. However, it is obvious that both work with a very basic ground division algorithm, after which buildings are filled in. Since both are in essence less advanced versions of the system that Stefan Greuter is using, they will not be discussed here any further. It should be noted that, in contrary to his system, these systems both work in an offline fashion.

2.3.4 Stefan Greuter

The goal of Stefan Greuter is to create a system that is truly online and, in his words, therefore infinite. In order to do so, he has combined the city generation part with a rendering engine that asks the generator to fill in the part that is currently viewed by the user. This cone shaped area is called the view frustum. The building types that are within the frustum are identified by a 32-bit integer key. [9]
The generator uses a caching mechanism to serve the renderer the three-dimensional geometry belonging to every key. If the key is not in the cache, this means the geometry has either never been generated yet or been erased from the cache to free up space for other objects. If this is the case, the generator part will (re)create the building belonging to the key, using the key, being the the only input, as its random seed[9]. This way one 32 bit key completely defines a building type. An LRU[10] system is used to free up space in the caching mechanism when it is needed for buildings that are currently within the view frustum.
The amount of memory needed to store a whole city is reduced even further by the fact that the actual keys also do not need to be stored by the system, since they are calculated from the coordinates of their respective cells, together with an overall *city seed*. This way, to store a whole city only the city seed needs to be saved, the rest can be recreated at any time.

Building generator

In the spirit of a truly online system, everything in the landscape is created at runtime. To accomplish this the system includes a building generator to create a building from the aforementioned key. The first step is to define the shape of the roof of the building by using random geometric shapes. While traveling down extra shapes are randomly added to this base shape, in the end creating a ground floor that is the same as or bigger than the roof. [10]

Summary

- System: online
- Urban realism: no, a simple grid pattern is used
- Non-iterative
- Steps: The system skips all phases and goes straight to the filling in of the details.

2.3.5 City Engine

The *City Engine* described by Yoav I H Parish is a commercially available tool with impressive visual results. A drawback to the system is that, compared to the other systems described here, much information needs to be supplied by the user. This needs to be done in the form of image maps, which are not easily produced for most everyday users.
After the user supplies an image map of population density, the system follows four major steps for the automated content creation.

[9] Details about random seeds and their use are described in Section 2.5
[10] Least Recently Used

Road generation using L-systems

The road generation is, according to its author, driven by an extended L-system. An L-system is a parallel string rewriting mechanism based on a set of production rules. The rewriting rules of the L-system used in the City Engine are kept very basic, but include calls to externally defined functions, that determine whether or not a certain rule can be applied. In short, this actually amounts to data based recursion, where a road in the system can generate another road, which is checked against the growing database of roads already defined according to a set of rules. Recursion for a certain road stops when the set of constraints prohibits any road to still be split off from it.
Two types of constraints are used, global and local ones. The local constraints check if a new road intersects another road already in the system, which is not allowed, and rearranges a new road when it ends close to another road or crossing so that it connects.
The global constraints make sure that the city follows certain super-imposed patterns. The published paper of the system [18] shows, amongst others, rules based on concentric circles (*Paris rule*) and rectangular blocks (*New York rule*). The global constraints also produce highways between densely populated districts and normal streets in the areas between those highways.

Subdivide areas between roads to define allotments

The areas between the streets are recursively divided into smaller lots by creating simple geometrical forms. This is done until a user supplied threshold for lot size is reached. Lots that are not connected to any streets are removed, which leads to small empty spaces *behind* houses.
Area categorization can be done by the user by supplying an image map. This categorization is used by the next phase to determine the building type.

Generate buildings

Building shapes are generated in a fashion that reminds of the method of Stefan Greuter, described in the last section. The difference is that the system of Greuter starts with a small geometric roof plan, to which more and more forms are added when descending through the building, whereas this system starts with a rectangular block and cuts more and more pieces of that away while ascending through the building, essentially giving similar results.

Convert to input for renderer

As a last step all data created in earlier phases is converted to a format which can be used by the renderer for the final visualization step. In this phase textures for the buildings are also created from tileable façade textures.

Summary

- System: offline
- Urban realism: uses realistic patterns both on bigger and smaller scales.
- Non-iterative
- Steps:
  - Boundary creation is covered implicitly by the user specifying a population density map: the city ends where population density reaches zero
  - After this the road generator first creates the main transport axes, then smaller roads are filled in.
  - The space between the roads is then divided into lots, but not classified by the system. This can be done by the user by supplying an image map.

– The final placement of the buildings is obtained from population density and categorization from the image maps supplied by the user.

2.3.6 Binary Worlds - Descensor Engine

The *Descensor Engine* of the company *Binary Worlds* is advertised on their website as a commercial product. The site displays screenshots, movies and even downloadable product demonstrations. However, it has been tried on more than one occasion to contact the company about more detailed product information or a pricing scheme, without success.
It is claimed by the company that: *Descensor generated models follow urban and architectural common rules, for that reason they look so realistic*[11]. However, without information about the rules used, it is not possible to check the validity of this claim.
The visual results can be considered to be at least as good the demonstration programs of Stefan Greuter. But because of the lack of reaction on e-mails and the fact that the website is dated 2003, it seems that the product or even the company has been discontinued. Therefore, the Descensor engine is left out of the scope of this project.

2.3.7 Rama

In [3] the author claims that his *Rama*[12] generator creates virtual worlds fully procedurally, including cities. However, from the paper it is obvious that the city creation is heavily dependent on predefined road networks from Google Earth. One of the cities is for instance modeled after Manhattan. To fill up the spaces between the road network the generator uses the same building construction technique as Stefan Greuter and the City Engine.
Because this generator does not bring anything new in relation to this thesis it is not discussed here any further.

2.3.8 Summary

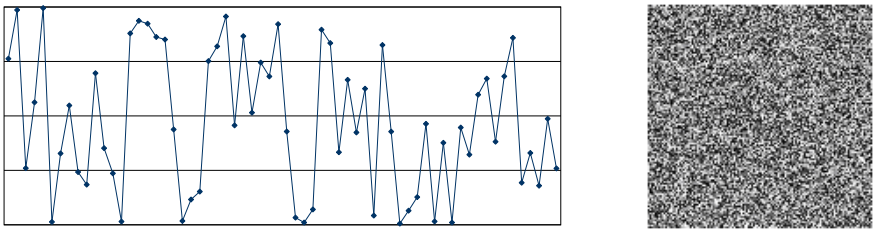
Next to the planned city generator of re-lion, two other interesting generators were described in this section. The system of Stefan Greuter uses some very interesting concepts, like his building generator and the random seeding system, which can certainly be built upon by the system of re-lion. However, the main idea of his generator is to generate city structure in an online fashion, which in this case also means that none of the same four steps are used, that were defined in Section 2.3.1. Because of the fact that his cities are theoretically infinite of size, there is no need or possibility to include a separate boundary generation algorithm. Therefore, his system is not very interesting within the narrow scope of this research project.
The other interesting generator is the City Engine, as described in Section 2.3.5. This system is in many ways very similar to the planned generator of re-lion, given the fact that they both use offline generation, and a certain form of urban realism. However, the most interesting feature for this research project is its compatibility with a separate boundary generation algorithm. At the moment this task is left to the user, who has to supply them in the form of an image map. It is very well possible to replace this user input by a city boundary generated by the algorithm proposed in this thesis.
The following research questions from the introduction can now be answered:

- **1a. What methods of city generation are used?** - In this section, an overview has been given of online and offline generators, and growth models versus non-iterative models. Furthermore, it has been shown that city generators arrange the creation steps in different ways.
- **1. How does a self-contained hull creation algorithm fit into full city generation systems?** - The one similar city generator that was reviewed, the City Engine described in Section 2.3.5, can very well use a previously created city hull as input for the rest of the process.

[11] `http://www.binaryworlds.com/products.html`
[12] A fictional world within a spaceship described by author A. C. Clarke.

2.4 Perlin Noise

Perlin noise was invented by Ken Perlin to simulate natural form and behavior. In the real world, hardly any object is completely motionless, or can be described by a pure geometric form. To enhance a simulation in such a way that it looks more natural, random movements or patterns can be added to the geometric ones.
The problem with normal random functions is that they only incorporate one maximum amplitude, and a wavelength of roughly two to four sample points, as can be seen in Figure 2.7(a). The amplitude and wavelength can of course be linearly transformed by multiplication and interpolation respectively, but it still is hard to use these values for natural behavior such as movement.



(a) One-dimensional (b) Two-dimensional

Figure 2.7: Fully random noise

For instance, the moving of a branch of a tree in the wind will have a movement with small amplitude and small wavelength, jitter hardly visible from farther away, but also movement with higher amplitude and higher wavelength, which defines the characteristic movement of a branch. Also, a height map[13] displays objects with high amplitude and long wavelength such as hills, but also objects with low amplitude and short wavelength, eg. rocks and sand.
Perlin noise combines multiple layers of these different mutations of random values, to create a more natural result. Starting with a layer with high amplitude and high wavelength, then adding an arbitrary amount of layers each with lower amplitude, but therefore also lower wavelength.

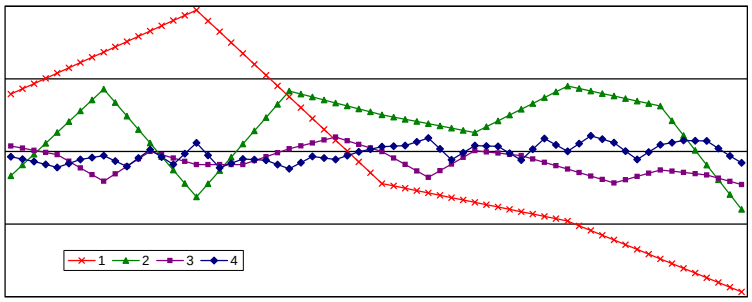


Figure 2.8: The separate layers

Figure 2.8 shows four separate random lines. Line 1 represents the starting layer, with a minimum wavelength of 32 sample points, and with amplitude 4. Line 2 has a minimum wavelength of 16 sample points and amplitude 2, line C minimum wavelength of 8 sample points and amplitude 1 and finally line 4 has minimum wavelength of 4 sample points and amplitude 0.5.
All lines were produced using linear interpolation. To smooth results, cosine or bicubic interpolation can be used, but the effects on multi-layer (6 and up) Perlin noise are very small, while the amount of needed processor power rises.

[13] A two-dimensional map of a piece of land where the value at a sample point defines the height of the land

When the above lines are added up, the result is the image as shown below. It is clearly visible that line 1 roughly determines the general direction of the noise line, although the small dents of line D are also noticeable. The result is a line that has the possibility to use the full amplitude range of line 1, while it can also have jitter with small wavelength. This result is shown in Figure 2.9.

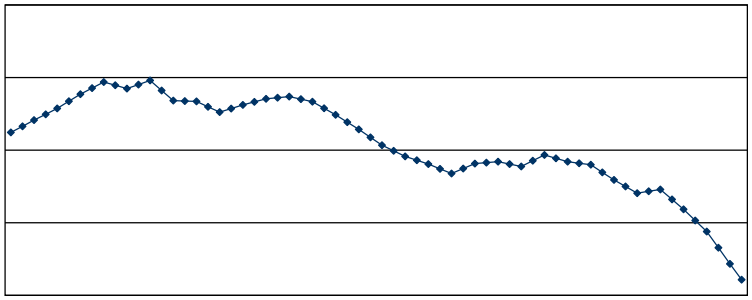


Figure 2.9: All layers combined

2.4.1 Function description

Perlin noise can be seen as a function of an amount of one, two or more parameters. One-dimensional Perlin noise is usually a function of time, the parameters of two-dimensional are mostly used as x and y axes of images or textures. A function description of the one-dimensional case would be defined as

$$P(x) \rightarrow A\frac{f(x,0) + f(x,1) + f(x,2) + \dots}{p(0) + p(1) + p(2) + \dots} \tag{2.21}$$

Where

$$f(x,k) \rightarrow p(k)R_k(\frac{x}{k+1}) \tag{2.22}$$

and

$$p(k) \rightarrow (\frac{1}{2})^{k+1} \tag{2.23}$$

A is the maximum amplitude and R_k are random generators. When N depicts the total amount of layers used, $P(x)$ can be written as:

$$P(x) \rightarrow A\frac{\sum_{k=0}^{N} f(x,k)}{\sum_{k=0}^{N} p(k)} \tag{2.24}$$

$p(x)$ here functions as the persistence function. It defines the maximum amplitude for every layer of the Perlin noise. At the same time, the minimum wavelength for every layer is defined by the $\frac{x}{k+1}$ part of $f(x,k)$. For bigger values of k, $\frac{x}{k+1}$ becomes smaller, resulting in a larger wavelength.

2.4.2 Implementation types

In the functions above, the R_k are referred to as random generators. Random number generators used in computers are of a discrete type, in that they deliver a random number each time they are called. For the function $f(x,k)$ to be continuously defined, R_k also has to be continuous, so of a fractional type. This can be accomplished by defining R_k as follows:

$$R_k(x) \rightarrow D_k(x) \quad \text{Where } x \in \mathbb{N} \tag{2.25}$$
$$R_k(x) \rightarrow Intp(R_k(\lfloor x \rfloor), R_k(\lfloor x \rfloor + 1), \{x\}) \quad \text{Where } x \in \mathbb{R} \tag{2.26}$$

Where $\{x\}$ is the fractional part of x, $\lfloor x \rfloor$ the integer part of x, and $D_k(x)$ a set of discrete random generators. Different interpolation methods will not be discussed here, since they hardly have any effect on Perlin noise with multiple ($n > 4$) layers. This will not be proven here. Linear interpolation is the fastest interpolation method, and is therefore the best candidate for performance applications. To create an implementation of the $R_k(x)$ function, there are two different approaches:

1. Use a random generator as a function of its random seed. In the $D_k(x)$ notation, x then represents the random seed. With every call to $R_k(x)$, call $D_k(x)$ twice and interpolate between the return values.
2. With the first call to $R_k(x)$, fill an array in memory with the results of multiple calls to $D_k(x)$. With every consecutive call to $R_k(x)$, take the correct two values from the array and interpolate.

The first method requires an insight in the way the random generator used works internally, because the random seed cannot be initialized with an arbitrary value such as the system timer. Another drawback is that it is very CPU intensive, because for every interpolated value the random generator has to be called twice.
The second method can be used with any random generator and any seed initialization, because it needs to be called only once for every possible integer value of x to fill the array. In this process the function input value of x does not matter. Also, because of this, the random generator does not have to be called again for every interpolated value, reducing the amount of CPU time needed. A drawback of this method is the amount of memory needed for storing the array. Another problem is that this method cannot (easily) be used for continuous applications.

2.4.3 Fractal properties of two-dimensional Perlin noise

Perlin noise is in multiple sources on the Internet referred to as *fractal noise*. To illustrate the concept of Perlin noise being fractal, it will be discussed using the qualitative properties of fractals as described in Section 2.1:

- **Self-similar** - Every new layer of Perlin noise introduces a similar set of data on a smaller scale. As with fractals, this process can be repeated semi-infinitely in practice, but infinitely in theory.
- **Initiator** - For two-dimensional Perlin noise the square that will be filled can be seen as the generator. This square has a random initial *color* value.
- **Generator** - Analogously to the description of the initiator, the generator divides every quad into four quads of equal size, each with a random number added to the base value of the starting quad.
- **Irregular** - Perlin noise is normally used to create irregular shapes or behavior such as clouds, idle facial muscle movement or dirty textures. It is exactly this kind of *natural irregularity* that is often tried to accomplish with the use of fractals.

The initiator and generator can be illustrated using the *fractal* displayed in Figure 2.4, with different values for square.

2.4.4 Summary

To summarize this section, the following research questions from the introduction can now be answered:

- **3. How does Perlin noise work?** - Perlin noise uses layers of random values of a different scale that are added together to create a noisy shape. This can be done in any dimension that is needed.
- **3a. What is input for Perlin noise?** - Random values from a random number generator, the number of the start layer and the amount of layers that should be added together.
- **3(a)i. What is a good random number generator to feed Perlin noise with?** - The next section of this thesis answers this question.
- **3b. What is the output of Perlin noise?** - A shape of the requested dimension. This shape is smoother than simple random output, because of the property that neighboring values are related.
- **4. Is Perlin noise a form of fractal?** - Yes, according to Section 2.4.3.

2.5 (Pseudo) Random number generators

Since the proposed method of generating city boundaries uses Perlin noise, all results of this project depend on random numbers. To give some insight in the functioning of random number generators, a few properties of the most common ones will be discussed here. This section will not go into detail, the references can be checked for that. Any lists presented here will only be comparative, and are not claimed to be exhaustive.

2.5.1 About

When a system should be able to generate multiple instances of different output using the same input set, the system cannot be deterministic[14]. To accomplish this behavior in a deterministic algorithm, extra input is mixed into the system in the form of random[15] numbers.
There are three main categories of random number generators (RNGs). These are:

- Real random number generators
- RNGs for cryptographic use
- Pseudo random number generators (PRNGs)

The first two will be discussed here shortly, before moving on to a more detailed description of the only suitable type for the applications in this thesis: PRNGs.

Entropy

The word entropy[16] is often used in random number theory in two different ways. First of all, as suggested by the definition in the dictionary, it is a measure for the *amount of randomness* in a stream of data. A higher value means that the stream is less predictable. Secondly, it is used to describe unpredictable events that can be captured into discrete data, making it input to a system. In this case, entropy is often used as the seed of a PRNG or to keep a random sequence cryptographically secure (see below).

2.5.2 Real random numbers

Real or true[17] random numbers can be defined by the fact that they cannot be predicted. Production methods typically involve a form of sampling analog sources. Post-processing has to be done to filter out the influence of events that cause regular patterns. Commonly used methods are measuring time between clicks of a Geiger counter, atmospheric noise from a radio or just background noise. [11] Measuring entropy from user input devices (mouse or keyboard) is also possible, but that has to be heavily compensated for the fact that user input always contains patterns.
For cryptographic use it is important that a random sequence cannot be replicated from the same source by others. For instance, if the high-order bits of the intensity of sunlight are used, it would be possible for somebody in the same building to generate exactly the same sequence. Hard disk timings and network packets timings are a possibility, but one has to be sure that this data is not also available to other people, eg. a network administrator or another user of the same computer system. [4]

[14] Deterministic: Describes a system whose time evolution can be predicted exactly. Source: The Free On-line Dictionary of Computing, ©1993-2005 Denis Howe

[15] Random: Having no specific pattern, purpose, or objective: random movements. Source: The American Heritage Dictionary of the English Language, Fourth Edition, 2000 by Houghton Mifflin Company.

[16] Entropy: A measure of the disorder of a system. The entropy of a system is related to the amount of information it contains. A highly ordered system can be described using fewer bits of information than a disordered one. Source: The Free On-line Dictionary of Computing, ©1993-2005 Denis Howe

[17] A long discussion can be held about the use of the words *real* or *true*. This subject will not be touched here, because the author of this thesis is not a philosopher.

2.5.3 For cryptographic use

In encryption systems RNGs are used to generate session keys or initialization vectors for temporary encryption, but also for keys that will be used for an extensive period of time. RNGs for cryptographic use have the task to be random enough to make it impossible for an attacker to figure out what the previous, current or next random output was or is going to be. For example, if a session key is produced by a PRNG (see below) that was initialized with a random seed made up of the amount of milliseconds since the beginning of the year, an attacker that can trace the starting time back with the precision of a minute will only need to try at maximum 60.000 possible encryption keys, instead of the usual 2^{256}. Also, if the data encrypted with the output of a PRNG with an insecure algorithm, this makes it easier to launch an attack against the encrypted data stream.
These typical problems can be solved by using a stronger PRNG algorithm (eg. the Blum-Blum-Shub generator[18]) or by periodically reseeding the PRNG with mixed entropy pools such as the high resolution system timer, as done by Fortuna[19]. These solutions immediately illustrate that these RNGs are slower than most PRNGs. Collecting entropy from different system pools is resource intensive work and slows down the generating process. The Blum-Blum-Shub generator uses a slow algorithm for every iteration, and only uses one bit of data from each iteration. Cryptographic security is thus a trade-off against performance.

2.5.4 PRNGs

PRNGs are, as is any computer algorithm, deterministic systems. They use a *random seed*, usually an integer number but any form of data will do, as their input to initialize the internal state, which determines the next random number. Every time a new random number is output, the internal state is updated.
Because a PRNG only has access to a finite memory state, after outputting enough numbers it will eventually visit the state it was initialized in. Since all following numbers are determined by the current state, the output will repeat itself from then on. The period, or cycle length, of a PRNG is the amount of numbers a PRNG can output before repeating itself[20].

Linear congruential generators

The system supplied rand() function is usually a linear congruential generator (LCG). This type of random generator is a simple function of its internal state, that is defined as

$$I_{j+1} \equiv aI_j + c \pmod{m} \tag{2.27}$$

m is called the modulus, and a and c are positive integers called the multiplier and the increment respectively. This is a very simple form of PRNG, and there are some major weaknesses in the general concept of LCG's [20]. However, the biggest problem is usually the implementation that is used in the compiler, eg. poorly chosen values for a and c or bugs in the implementation. The *Minimal Standard* of Park and Miller, with $a = 7^5 = 16807$, $m = 2^{31} - 1 = 2147483647$ and $c = 0$ has been proven to be a good PRNG.

2.5.5 Comparison and practice

For the algorithms described in this thesis random number generators are needed. A few properties and possibilities were discussed in the previous sections. It is agreed on by the sources that the standard C++ rand() function should not be used, because the chances on something being wrong with it are considered to be high.
The suitable candidates presented in [20] are reported to have cycle lengths between 10^9 and $2.3{\cdot}10^{18}$. Apparently science has progressed since 1988, because currently other fast PRNGs are available with

[18] `http://www.win.tue.nl/~asidoren/bbs.pdf`
[19] `http://en.wikipedia.org/wiki/Fortuna_(PRNG)`
[20] Source: `http://en.wikipedia.org/wiki/Pseudorandom_number_generator`

huge cycle lengths. Examples are Mother-of-all, with a cycle length of $3 \cdot 10^{47}$ [7], and Mersenne twister, with a cycle length of 10^{6001} [16].
Since the sophistication and unpredictability of special generators for cryptographic use is not needed for this project, they do not make good candidates, because of their lower speed. Although many sources propose their own PRNG candidate, they all refer to Mersenne Twister as an *excellent* random number generator in terms of entropy, period (cycles) and uniformity of distribution. Because of these excellent reviews, an implementation of Mersenne Twister has been used throughout this project.

Bad ideas

The following practices should be avoided at all times when making use of random numbers:

- Using a very complex algorithm without thorough analysis. Simpler algorithms are easier to analyze and can therefore be proven to be strong [5].
- Using the system supplied rand() function. These have a long history of being weak. [20]
- Using a good random generator with a bad random seed. Sometimes initializing the PRNG is forgotten at all, which usually makes it output the same set of numbers every time it is used.
- Obtaining a small random number by applying a mod operation to a possible large random number. In case smaller random numbers are needed, always use the highest order bits.

Chapter 3

City boundary creation algorithm

In order to automatically create realistic city hulls using two-dimensional Perlin noise, an algorithm is proposed in this chapter. This algorithm first fills a two-dimensional grid with Perlin noise values. This type of grid can be considered a greyscale bitmap, and can also be displayed as such. To ensure a minimum amount of clustering in the bitmap, a transformation function is applied that emphasizes the values near the center of the grid more than values near the outside. This method increases the probability that there is at least one coherent island left over after the next step, the application of a threshold.
All values in the grid lower than a certain threshold value are removed, leaving a shape consisting of the values higher than the threshold. Since a city *boundary* is requested by the project goal, the boundary of the shape is extracted from the bitmap representation. The result of the algorithm is an array of two-dimensional position vectors, describing a closed shape.
These steps will be described in more detail in this chapter, including a description of the input parameters of every step.

3.1 Fill with two-dimensional Perlin noise

The algorithm starts with an empty two-dimensional square grid, which is then filled with Perlin noise, as described in Section 2.4. When this grid is converted to a greyscale bitmap, it looks similar to the image that is displayed in Figure 3.1. This image was created at a size of 512 x 512 pixels, but Perlin noise can easily be used to fill images of other sizes.

Figure 3.1: Example of 2-dimensional Perlin noise

Parameters

For this step the following parameters are defined:

- ***Starting* layer** - The layer in the noise with the highest relative amplitude and the biggest wavelength.
- **Layer count** - The amount of layers used in the Perlin noise. It is expected that higher layer counts will give images with more detail.

3.2 Transformation function

As can be seen from the example in Figure 3.1 Perlin noise, although far less than the simple two-dimensional noise from Figure 2.7(b), still shows erratic behavior. When the example figure would be used as input for the threshold step in this algorithm, the result would look something like can be seen in Figure 3.2.

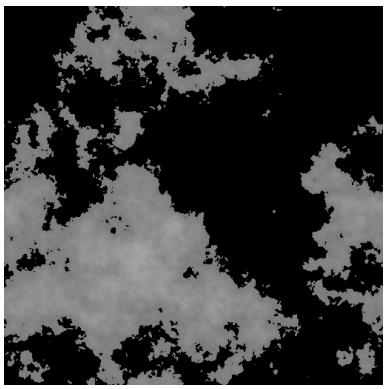
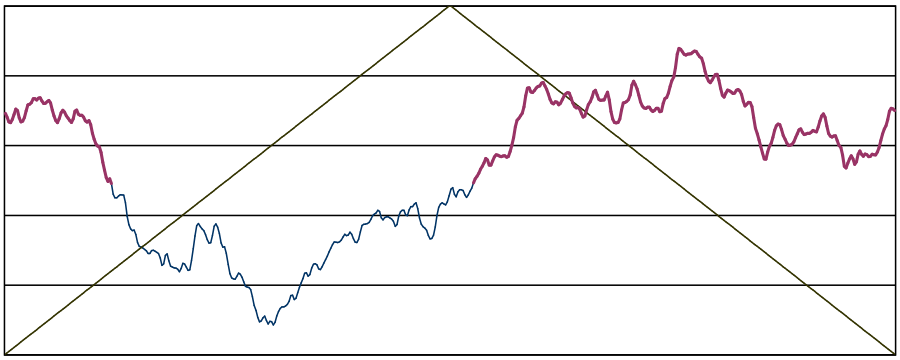


Figure 3.2: The same example with a threshold cut-off

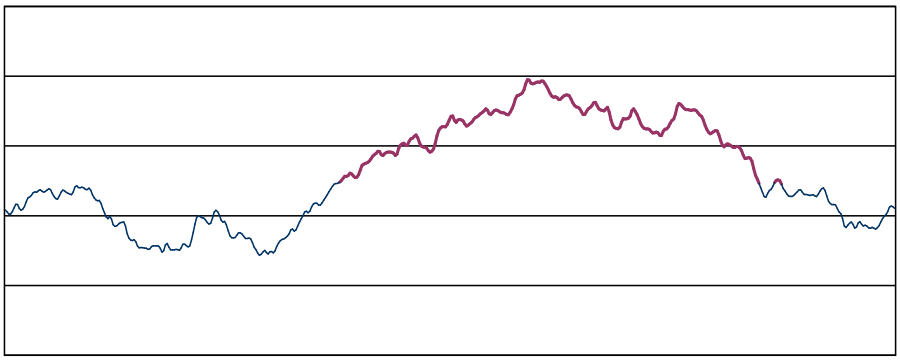
To ensure at least a minimal amount of grouping a transformation function is used. In this design a simple function is chosen that adds a slope value, related in a linear or polynomial way to the distance from the center of the bitmap. In case the results of the algorithm are not satisfactory, a more complex type of function can be chosen. Also, more complex functions could be the subject of further research.

$$O_{xy} = \frac{c_1(1 - {D_{xy}}^n) + c_2 I_{xy}}{c_1 + c_2} \tag{3.1}$$

where I_{xy} is the bitmap input value on coordinate (x, y), O_{xy} is the output value for the same coordinate and D_{xy} represents the distance of the coordinate to the center of the bitmap expressed in a range from zero to one. c_1 and c_2 are constants representing the weights that the Perlin noise and the slope function are multiplied by. Variable n makes the difference between a linear and a polynomial slope function. Proposed values for n are $n = 1$ and $n = 2$. It should be noted that choosing $c_1 = 0$ and $c_2 \neq 0$ gives $O_{xy} = I_{xy}$ meaning that the output is the same as the input.
The effect is shown in a one-dimensional variant in Figure 3.3. The first figure shows an example of Perlin noise, and the slope function for $n = 1$. The second figure shows the output of the transformation function, using $c_1 = 0.4$ and $c_2 = 0.6$. The violet part of both figures shows the part of the graph that is above an arbitrarily chosen threshold half-way on the vertical axis. In this example, a c_1 mix-in-value of 0.4 was enough to create a thresholded island in the middle of the figure instead of two islands against the rim of the figure. It is not guaranteed this value will work for all randomly generated figures, so it is considered input to this step.



(a) Perlin noise and linear transformation function



(b) Both functions combined ($c_1 = 0.4$ and $c_2 = 0.6$)

Figure 3.3: Perlin noise, transformation function and combination of the two

Parameters

For this step the following parameters are defined:

- **Transformation function shape** - The shape of the slope function, in this paragraph described by the variable n.
- **Transformation function mix-in relation** - The value of c_2 relative to c_1, determining the perseverance of the original Perlin noise when the transformation slope is mixed in.

3.3 Threshold the bitmap

After the image has been prepared in the last steps all values lower than a certain threshold will be removed from it. To do so, a threshold value lying between the minimum and maximum output values of the last step must be determined. The following things must be considered:
The threshold must be chosen higher than any pixel value that lies on the rim of the bitmap. This prevents any rim pixels from being included into the final image. If this would not be done, it would be possible that a whole stripe of pixels on the rim is included, creating a straight line in the output. An example of this situation can be seen in Figure 3.2. This situation would defy the purpose of the use of Perlin noise.
The threshold can easily be chosen by finding the highest value on the rim of the image and if necessary adding a small epsilon value. A drawback of this method is that if one of the highest values of the image is located on the rim, hardly any pixels will be left after applying the threshold. This problem is partly caused by the fact that the bitmaps used are squared and not round and partly

by choosing a low c_1 value. The extreme case of choosing the values $c_1 = 1$ and $c_2 = 0$ would solve the problem completely, because the highest values will always be in the center of the image. On the other hand it would also only produce perfect circles, and show nothing of the input Perlin noise. The conclusion can be drawn that there is a trade-off between a high Perlin noise influence on one side and preventing hard edges or low fill ratios on the other side. Several relations between c_1 and c_2 should be tested to find successful combinations.
With this reasoning in mind, four methods of determining a threshold are proposed:

- **Method 1** - Use the highest value that can be found on the rim of the image as the threshold, possibly in combination with an epsilon value. A drawback of this method is that the fill ratio of the image cannot be determined at all. It can be very low (less than 15%), as already discussed before, or even very high (more than 75%) in the coincidence that the higher values are all grouped to the inside of the image. This threshold value is later referenced as T_1.

- **Method 2** - Determine a threshold to end up with a certain fill ratio. Create a discrete array with pixel counts for different ranges in the bitmap values. For example, if the total range of the bitmap values is 0 to 255, an array with 256 integer values can be created, where the first value denotes the amount of bitmap values found between 0 and 1, the second between 1 and 2, and so on. To fill a certain amount of pixels, the values in the array are added up from the end to the beginning, until this sum is equal or higher than the requested value. The index in the array where this sum is reached determines the threshold for the filtering. A drawback of this approach is that the threshold needed to reach a certain fill ratio can be lower than the highest value found on the rim. This would mean that rim pixels are included in the image, delivering straight lines. This threshold value is later referenced as T_2.

- **Method 3** - To calculate T_3, the threshold value that is the outcome of this method, first calculate both T_1 and T_2. If $T_2 > T_1$ then $T_3 = T_2$. If $T_2 < T_1$ then $T_3 = T_1$. In short notation $T_3 = max(T_1, T_2)$. This solves the drawback of method 2 and one of the drawbacks of method 1 in that the fill ratio now has a maximum. The minimum fill ratio is still undetermined.

- **Method 4** - Calculate both T_1 and T_2. If $T_2 > T_1$ then $T_4 = T_2$. If $T_2 < T_1$ then start over from the beginning with a completely new bitmap of Perlin noise. If this happens multiple times in a row the settings of the parameters for the algorithm can be deemed unfeasible. This method solves all drawbacks of method 1 and method 2. It introduces a new drawback that when the ratio of unaccepted bitmaps to accepted bitmaps gets too high, one can wonder how much of the range of the original Perlin noise still is in the outcome of the algorithm, since so much is filtered out.

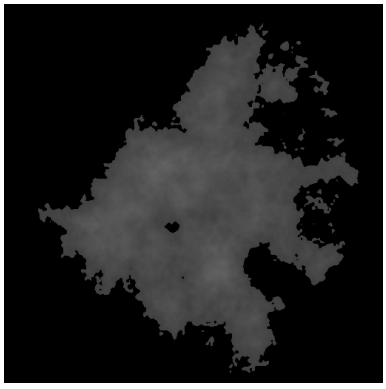


Figure 3.4: Example of Perlin noise with a transformation function and a cut-off threshold

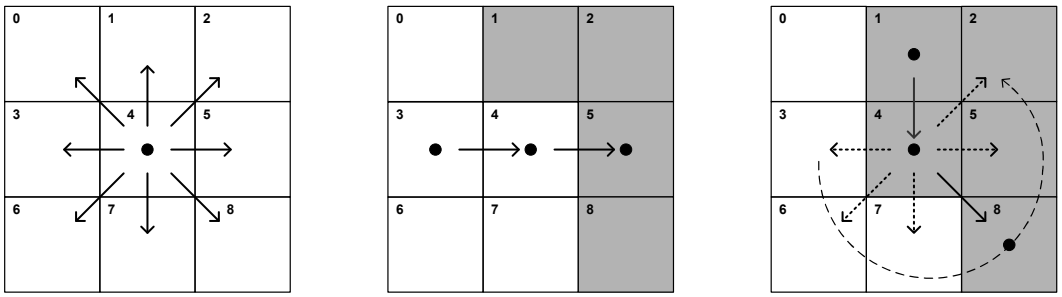
Parameters

For this step the following parameters are defined:

- **Method** - Which of the four above-mentioned methods will be used to determine a suitable threshold.
- **Minimum fill ratio** - The desired minimum fill ratio, in case of the use of methods 2 through 4.

3.4 Vector representation of the hull

The result of the last step is an image of the form that can be seen in Figure 3.4. Until now, this is still a bitmap, a discrete image form. Although information about the rim of the form is available in the bitmap, it still needs to be distilled to become the specification of a *boundary*, as requested by the project goal. To accomplish this, an algorithm is used that finds the rim of the shape in the bitmap, and traces this rim until a closed shape has been found. The result of this algorithm is an array of two-dimensional position vectors.
Internally, the algorithm makes use of a current position and a last direction variable. The current position variable tracks its position in the bitmap, the last direction variable tracks the direction the algorithm last moved in to get to its current position. Possible walking directions are the four orthogonal movements north, east, south and west as well as the four diagonal movements north-east, south-east, south-west and north-west. This is shown in Figure 3.5(a).



(a) The eight possible directions (b) The initialization of the algorithm (c) A next step of the algorithm

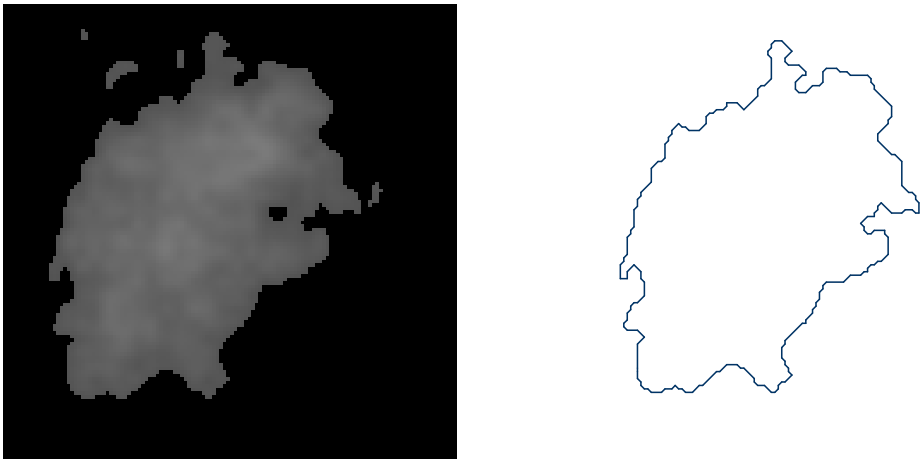
Figure 3.5: Hull vectorization algorithm

1. From the left side of the bitmap at an arbitrary height, find the first pixel that is filled. Add the coordinates of this pixel to the output array as the starting point, and make this coordinate the current position. This process is shown in Figure 3.5(b); The algorithm starts in cell 3, walks to cell 4, and finds the first filled cell in the form of cell 5.
2. Set the last direction to south-west.
3. Check the neighbors of the current position. Start with the direction two directions clockwise from the last direction (eg. south-west becomes south-east) and, if the pixel is not filled, try every next counter-clockwise direction until a filled pixel is found. Add the coordinates of this pixel to the output array and make this coordinate the current position. This process is shown in Figure 3.5(c); The last step started in cell 1, current position is now cell 4. The last direction was thus south, so the first cell tried is the one in direction west: Cell 3. Since this cell is not filled, respectively cells 6, 7, 8, 5 and 2 will be tried. The first filled cell found is cell 8.
4. Repeat step 3 until the current position is the same as the starting position. A closed walk hull has now been created.

To make sure the algorithm picks up a form in the bitmap, the algorithm can be started at different positions on the rim of the bitmap. To find the biggest enclosed shape, select the resulting array with the highest amount of coordinates. It would also be possible to select all distinct resulting shapes and use these as an interacting set of city and suburbs for instance. Due to time restrictions this possibility is left out of the scope of this thesis, but should be investigated during any following project.

Parameters

Because this possibility is left out of scope, this step of the hull generation algorithm does not have any adjustable parameters.



(a) Input for the vectorization

(b) Vectorization result

Figure 3.6: Low resolution example of vectorization

3.5 Parameter summary

If parameters from all steps in the hull creation algorithm are taken together, the list shows the following items:

- **Perlin noise starting layer** - The layer in the generated Perlin noise with the highest relative amplitude and the biggest wavelength.
- **Perlin noise layer count** - The amount of layers used in the Perlin noise.
- **Transformation function shape** - The shape of the slope function, in this paragraph described by variable n.
- **Transformation function mix-in relation** - The perseverance of the original Perlin noise when the transformation slope is mixed in, as defined in Equation 3.1.
- **Threshold selection method** - Which of the four above-mentioned methods will be used to determine a suitable threshold.
- **Minimum fill ratio** - The desired minimum fill ratio, in case of the use of methods 2 through 4.

Part II

Experiment

Chapter 4

Experiment definition

In Section 2.2 two quantifiable properties of realistic city boundaries were defined. Also, an algorithm to generate city boundaries was proposed in Chapter 3. To test if these created boundaries fulfill the realism properties, an experiment was defined. During the rest of this thesis, the terms *valid* and *validity* will be used to indicate that a generated boundary conforms to the aforementioned realism properties.
Furthermore, it is desirable to generate city boundaries with a previously specified fractal dimension. It is possible to implement this in such a way, that city boundaries are generated and tested, until a correct one is found. However, the process of calculating fractal dimension can easily take a few seconds on older hardware, eg. the Athlon XP 2600+ that was used in the beginning of this project. In case for instance 30 results have to be checked before finding a correct one, processing time rises to approximately one minute. To overcome this problem, it is helpful to choose the input parameters in such a way, that it is possible to predict the fractal dimension of the output boundary. Therefore, another part of the experiment is to find a relation between the input parameters and the fractal dimension of the output.
The main setup of the experiment is to output multiple city boundaries for every possible set of input parameters. In order to determine whether a piece of output is valid, and to find a correlation between input an output, statistics are recorded and later processed for every test case.
Section 4.1 of this chapter explains how to test the first realism constraint, fractal dimension. After that, Section 4.2 gives an overview of the input parameters of the algorithm, and the value ranges that have been chosen. This section also explains how the second realism constraint, the minimum enclosed area ratio, is implicitly met. Section 4.3 then describes the input parameters used in the two tests that were run. Finally, Section 4.4 describes how the results of these tests were analyzed, and discusses the expected correlation behavior.

4.1 Calculating fractal dimension

The result of the hull generation algorithm is a vector array describing the border of a closed area. The fractal dimension of this hull can be estimated using the *Structured walk* method described in Section 2.1.4. In order to obtain reliable results, the algorithm is run once for every point on the hull, to rule out *good* or *bad* fits by starting point coincidence. Equation 2.16, that is used to calculate fractal dimension, is adjusted for discrete choices of r:

$$D_n = \frac{\ln N_0 - \ln N(r_n)}{\ln q_n} \tag{4.1}$$

where

$$q_n = \frac{r_n}{r_0} \tag{4.2}$$

The variables needed are:

- l_0 - The chord length that is taken as input for the first iteration of the algorithm.
- N_0 - The average amount of chords resulting from the first iteration of algorithm.
- r_0 - The average resulting chord length from the first iteration of the algorithm.
- $l_n = k^n \cdot l_0$ - The chord length that is taken as input for the n^{th} iteration. In this equation k is constant.
- N_n - The average resulting chord length from the n^{th} iteration of the algorithm.
- r_n - The average amount of chords resulting from the n^{th} iteration of the algorithm.

These variables are computed as follows:

l_0

To have a valid starting value for l_0 the method from [1] is used: The average chord length computed from the input set. The lengths of all input vectors are added up and divided by the amount of input vectors.

N_0

To calculate N_0 the Structured walk method is started once for every vector in the input set (amount V) with chord length l_0. Per vector this delivers an amount of chords $N_{0,x}$ with length l_0 and one leftover chord with length $r_{leftover,x} < l_0$, which is also counted. All these values are summed and divided by the amount of items in the input set, yielding

$$N_0 = \frac{\sum_{x=1}^{V}(N_{0,x} + 1)}{V} \tag{4.3}$$

r_0

To obtain the average resulting chord length the same data is used. Per input vector the Structured walk method delivers an average chord length

$$r_{0,x} = \frac{N_{0,x} l_0 + r_{leftover,x}}{N_{0,x} + 1} \tag{4.4}$$

For the total average chord length this leads to

$$r_0 = \frac{\sum_{x=1}^{V} r_{0,x}}{V} \tag{4.5}$$

$l_n = k^n \cdot l_0$

After l_0, N_0 and r_0 are known it is possible to start an iterating process calculating the fractal dimension of the shape at different scales. For every iteration an input value l_n is needed. These can be calculated using the constant k, which is defined analogously to the scaling constant r in Section 2.1.2. Since the average scale factor for 2D Perlin noise is 2, it is desirable to choose a power of 2 for k. A value of $k = 2^1 = 2$ is proposed, for extra detail a value of $k = 2^{\frac{1}{2}} = \sqrt{2}$ can be used.

N_n

To obtain the average resulting chord length from the n^{th} iteration of the algorithm the Structured walk is started again for every vector in the input set with an input chord length of l_n. As with N_0 this results in

$$N_n = \frac{\sum_{x=1}^{V} (N_{n,x} + 1)}{V} \tag{4.6}$$

r_n

The average amount of chords resulting from the n^{th} iteration of the algorithm is also calculated analogously to r_0:

$$r_{n,x} = \frac{N_{n,x} l_n + r_{leftover,x}}{N_{n,x} + 1} \tag{4.7}$$

$$r_n = \frac{\sum_{x=1}^{V} r_{n,x}}{V} \tag{4.8}$$

With the values of N_0 and r_0 and a list of pairs (N_n, r_n), fractal dimension can be calculated using Equation 4.1. An example of the results of these calculations is shown in Table 4.1. In this example the values $l_0 \approx 1,582$ and $k = \sqrt{2}$ are used. This results in values $r_0 \approx 1.581$ and $N_0 \approx 3520.590$, after which the rest of the table is filled in.
After these calculations, not one but 11 values for fractal dimension have been given for different scales. To create values that are comparable between different cases, a standard linear regression is performed between scale ($\ln r$) and fractal dimension (D_n). This relation will have the form

$$D_{reg} = a \cdot \ln r + b \tag{4.9}$$

where a and b are constants to be determined by the regression algorithm. For the example from the table above these values are $b \approx 1.204$ and $a \approx 0.020$. To visualize these results, D_n (blue) and D_{reg} (pink) are plotted in Figure 4.1

Comparable fractal dimension

In order to compare the fractal dimension between different test cases, it is necessary to define a constant measuring point on the $\ln r$ axis, i.e. at the same scale for every case. This constant value of $\ln r$, denoted R, defines the comparable fractal dimension D_R as $D_{reg}(\ln r = R)$.

n	r_n	N_n	q_n	$\ln N_n$	$\ln q$	D_n
0	1.581	3520.590				
1	2.236	2310.507	1.414	7.745	0.347	1.215
2	3.162	1508.597	2.000	7.319	0.693	1.223
3	4.471	968.564	2.828	6.876	1.040	1.241
4	6.321	623.730	3.998	6.436	1.386	1.249
5	8.936	403.482	5.652	6.000	1.732	1.251
6	12.628	260.259	7.987	5.562	2.078	1.254
7	17.841	162.776	11.285	5.092	2.423	1.268
8	25.187	105.066	15.931	4.655	2.768	1.269
9	35.535	67.754	22.476	4.216	3.112	1.269
10	50.035	42.599	31.648	3.752	3.455	1.278
11	70.271	25.572	44.447	3.241	3.794	1.298

Table 4.1: Example results of fractal dimension calculations

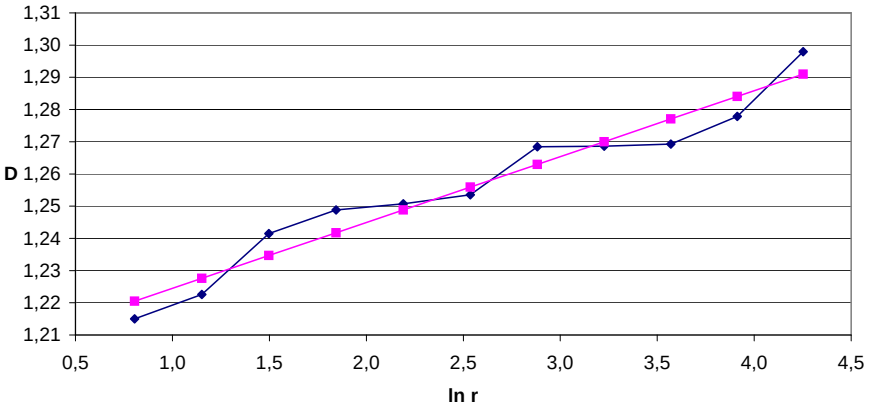


Figure 4.1: The values of D_n (blue diamonds) and D_{reg} (pink squares) as given in the example of this section, displayed in relation to $\ln r$ (horizontal axis)

This value R was obtained using the *what works* test as described in Section 4.3.1, by taking the average of all values for $\ln r_n$. This results in a value $R \approx 2.5696$, to be used throughout the rest of this report. For the example in this section this yields $D_R \approx a \cdot 2.5696 + b \approx 1.255$.

4.2 Test parameters

Tests were run with ranges of input values as elaborated in this section. These ranges were chosen due to limits on available processing power, and feasible amounts of time.

4.2.1 Image size

Images of 512 x 512 pixels were used to run these tests. This value was chosen because it gives a reasonable amount of detail and enough data to analyze, without needing the enormous overhead and necessary computing power of images of 1024 x 1024.

4.2.2 Perlin noise layers

Within images of 512 x 512 pixels layers numbered 0 to 9 are feasible. Layer 0 means that in X and Y direction there are 2^1 different pixels, an image with a resolution of 1 x 1. Layer 9 entails 2^9 pixels in both directions, representing a layer with 512 x 512 pixels. Obviously, it is useless to add layer 10, of 1024 x 1024 pixels, or higher, because the target image cannot contain this amount of detail. As a result, the maximum layer used is layer 9.
On the other end of the spectrum, layer 0 does not add anything to the resulting image. It only adds a single bias value to every pixel in the target image. The effect of this is canceled out by the way the threshold is determined. Layer 1, which consists of 2 x 2 different pixels, is also not very useful to the process. Basically, layer 1 influences the placement of the shape within the image. Thus, the first layer significant to this application of Perlin noise is layer 2.
The layer range has now been set between 2 and 9. Thus, the amount of different test cases for this parameter is $8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = 36$. In itself this is not a very high number. However, since this number is still multiplied by the ranges of the other test parameters, and every test case takes a significant amount of time to complete, it is desirable to keep this value as low as possible.
To partly resolve this problem, another constraint is added. Perlin noise does not behave like typical Perlin noise anymore when only few layers are used. For example, Perlin noise with one layer is basically just noise. It is proposed to only use noise with four layers or more, reducing the amount of different test cases for this parameter to $5 + 4 + 3 + 2 + 1 = 15$.
During the rest of this thesis, the term *Perlin set* is used to indicate a pair of a start layer and an end layer.

4.2.3 Transformation function

Two parameters are defined for the transformation function. Variable n defining the form of the slope function has already been specified as $n = 1$ and $n = 2$ in earlier sections. As for the mix-in ratio of the transformation function, early tests showed that useful values for c_2 lie between 0.1 and 0.7. These are not discrete values, so this can theoretically lead to an infinite amount of test cases. A step size of 0.01 would already cause 61 test cases for one value of n, and thus 122 test cases for both values. To lower this large amount of test cases, first a test was done with a smaller subset, to see which values give reasonable results. The *what works* test will be described in more detail in the next section.

4.2.4 Threshold

This step possibly adds two parameters to the set of test cases. To reduce the exponential growth of test cases, only method 4, as described in Section 3.3, was chosen as threshold definition method.
In order to use method 4, a minimum fill ratio needs to be defined. A practical lower limit can be set at 20%, since below this value the filled mass is so small, that it would have been better to lower the image size instead. An upper limit can be set, but this is not strictly necessary because at higher values ($> 60\%$) most cases will fail because $T_2 < T_1$. This prevents the algorithm from executing

the Structured walk calculations, which are by far the most time-consuming of the tests. A practical upper limit will be set at 80%.
Since this parameter is not expected to greatly influence results, a step size of 4% is defined between tests. This means that minimum fill ratio will increase the amount of test cases by a factor $(80 - 20)/4 + 1 = 16$. However, it should be noted that due to the high fail ratio at higher values processing time will probably only increase with a factor 8.

Minimum enclosed area ratio

A lower limit of 20% for the minimum fill ratio is practical for yet another reason. As described in Section 2.2, there is a minimum of 6.35% for the relation between enclosed area and maximum possible enclosed area, ergo A/A_{max}. For an image of 512 x 512 pixels, the maximum possible diameter is $d = \sqrt{512^2 + 512^2} \approx 724$. According to Equation 2.18, $A_{max} = 1/4 \cdot \pi d^2 \approx 411775$. The minimum amount of pixels to fill is then $0.0635 \cdot 411775 \approx 26147$.
This is 10% of the total amount, $512^2 = 262144$, of available pixels. Even if some islands in the shape are missed by the vectorization algorithm, as described at the end of Section 3.4, a minimum fill ratio of 20% ensures that the second realism property is met.

4.3 Test definitions

The last section described the following values for the test case factors:

- Perlin noise layers: $P = 15$
- Transformation function: $B = 122$
- Threshold definition: $H = 16$

In order to analyze fail ratios and to rule out coincidences, a test with the same set of parameters should be repeated a significant amount of times. For the full test, an arbitrary amount of $M = 10$ times was chosen. Early tests have shown that each test needs about $T = 10$ seconds to complete. This amounts to an estimated processing time of $P \cdot B \cdot H \cdot M \cdot T = 15 \cdot 122 \cdot 16 \cdot 10 \cdot 10 = 2928000$ seconds, which equals more than 33 days. Since this is not practical, another test was run first to find useful combinations of parameter values.

4.3.1 *What works* test

To lower the total amount of test cases, an initial test run was first carried out with the following parameters:

- Perlin noise layers: To reduce the test case factor to $P = 5$ only the following test cases will be carried out:
  - Start layer = 2, end layer = 5
  - Start layer = 4, end layer = 7
  - Start layer = 6, end layer = 9
  - Start layer = 2, end layer = 9
  - Start layer = 3, end layer = 8
- Transformation function: This can be reduced to $B = 16 \cdot 2 = 32$ by increasing the step size to 0.04.
- Threshold definition: Because this a relatively high step size was already chosen for this parameter, this remains the same. For the amount of test cases this means a factor of $H = 16$.
- To reduce the amount of test cases, it is easy to lower the iteration count to $M = 4$ times.

The total amount of cases for this test is now $P \cdot B \cdot H \cdot M$ which is $5 \cdot 32 \cdot 16 \cdot 4 = 10240$. Since the higher half of the range of the minimum fill ratio is expected to fail before fractal dimension calculation, the estimated processing time for this test set is $10240/2 \cdot 10 = 51200$ seconds, which equals just over 14 hours.

4.3.2 Full test

As described in more detail in Section 4.4, for the definition of the full test the two most important results of the *what works test* are:

- **Perlin noise layers** - Only cases with end layers 8 and 9 gave valid results.
- **Processing time** - After optimizations as described in Section 5.3 the average processing time needed per test case was only around 0.35 seconds.

The first item reduces the amount of different Perlin noise layer specifications to $P = 9$. With the original amounts of $B = 122$, $H = 16$ and $M = 10$, this amounts to a total of 175680 different test cases. With the newly measured processing time this would come down to 61488 seconds, or just over 17 hours. Since this is computationally feasible, the full test was carried out with these parameters.

4.4 Evaluation method and expectations

As explained in the last sections, two tests were defined, both serving a different purpose. The setup of these tests will be explained here in more detail, after some more detailed guidelines have been given for the validity of a generated city hull.

4.4.1 Validity

Limits on dimension

As explained in Section 2.2 datasets of existing cities usually show a fractal dimension between 1.18 and 1.28. The first rule for a city hull being valid is thus defined as

$$1.18 < D_R < 1.28 \tag{4.10}$$

Constant dimension

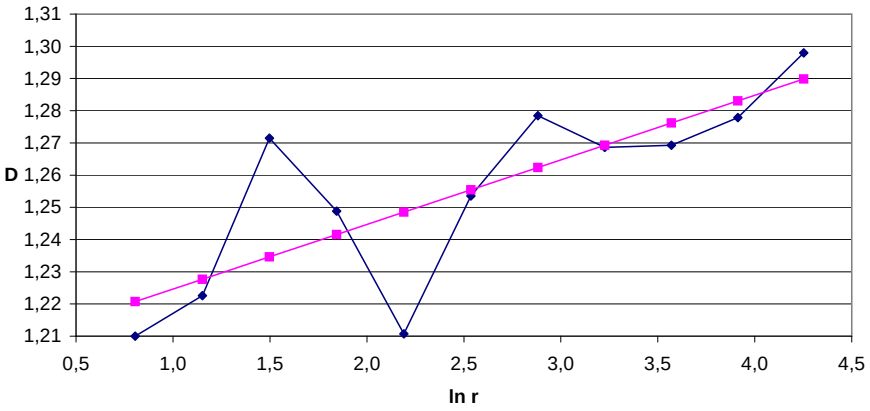
As mentioned in the summary of Section 2.1, true fractals have a constant dimension, independent of scale. Thus, a valid boundary must also have have a constant dimension over its entire range of scale. In Section 4.1, $\ln r$ represents scale, and the line $D_{reg} = a \cdot \ln r + b$ represents the estimation of the belonging fractal dimension. Since this is only an estimation, a maximum value is set for a instead of requiring that $a = 0$. This maximum absolute value is set at

$$a_{max} = \frac{1.28 - 1.18}{max(\ln r_n) - min(\ln r_n)} \tag{4.11}$$

to allow at maximum the exact distance between the upper and lower limit of the fractal dimension, respectively 1.28 and 1.18. In the *what works* test, the average of $max(\ln r_n) - min(\ln r_n)$ was 3.429, which gives $a_{max} \approx 0.02916$.

Standard deviation

When comparing Figure 4.4.1 to Figure 4.1, it can been that the same regression line D_{reg} corresponds to different point-sets. In particular, the point-set shown in Figure 4.1 is represented so poorly by the regression line that it is incorrect to draw conclusions from only that line.



In order to ensure accuracy of the regression line, a third validity rule is added. The standard deviation of the distance of each point to the line is calculated, and may not be larger than the

arbitrarily chosen value 0.02. If d_n is defined as the distance of each point to its corresponding point on the regression line, then the usual formula for standard deviation is defined as

$$\sqrt{\frac{\sum (d_n - \overline{d_n})^2}{N}} \tag{4.12}$$

where $\overline{d_n}$ is the average of all distances and N is the amount of values. This formula is simplified by the fact that the regression formula used ensures that $\overline{d_n} = 0$, resulting in

$$\sqrt{\frac{\sum {d_n}^2}{N}} \tag{4.13}$$

For the example shown in Figure 4.4.1 the standard deviation is 0.024 and thus higher than 0.02, while it is only 0.006 in the earlier example.

4.4.2 *What works* test

As mentioned before, the *what works* test was carried out to lower the execution time needed for the full test by excluding some input parameters. Also, its results provided some insight in the way the results of the full test should be processed.
The results were evaluated by first calculating the validity of every hull in the result set. If a hull was not valid it was thrown away. Then the input parameters of the remaining hulls were analyzed to see if there were certain parameters or ranges thereof that did not deliver any valid hulls. Beforehand it was expected that there would be no valid results for higher values of the minimum fill ratio, and for short spans between the start and end Perlin layers, eg. start layer is 6 and end layer is 9.

4.4.3 Full test

It is still remotely possible to analyze the results of the *what works* test by browsing through them by hand. This is definitely not a possibility anymore for the 175680 results of the full test, so all checks have to be statistical. The analysis of the results of the full test consist of two parts. Firstly, different summaries of the amounts of valid results for certain input parameters are constructed. Secondly, correlations are sought between arrays of input parameters and arrays of a and D_R, as referenced in Equation 4.9.

Summaries of valid results

- **Validity count per Perlin set** - Valid results are counted per set of start and end layer. Beforehand, it was expected that the sets with a small amount of layers would produce only a limited amount of valid results. To keep the results manageable, Perlin sets with less valid results than 5 percent of the total possible amount, 19520 per set, are considered unusable and are removed from the results.

- **Count per transformation slope form** - Since it was not expected that the two possible values for this parameter, $n = 1$ and $n = 2$, would influence the fractal dimension directly, it is only checked whether the amount of valid results is approximately equal for both values.

- **Per minimum fill ratio** - It was expected that a high minimum fill ratio would increase the probability that $T1 < T2$ in the threshold calculations of method 4. For this reason, it is interesting to know at what values this happens in praxis. The value that was expected beforehand lies between 0.4 and 0.5.

- **Per transform mix-in relation** - It was hard to make an accurate prediction about the influence of the mix-in relations, c_1 and c_2, on the amount of valid results. If c_2 is very low, the round shape of the transform function will be of most influence, delivering low detail and thus a too

low fractal dimension to be valid. On the other hand, with high values of c_2 there are two problems, the fact that without much influence of the transform function hardly any coherent area will be found and that the dimension might become too high to be valid. Therefore, it was expected that the c_2 value with the highest amount of valid results lies approximately half-way, so at 0.5.

Correlation with D_R

As explained in the beginning of this chapter, it is most interesting to predict the fractal dimension of the output when only knowing the values of the input parameters. Therefore another test on the generated results is to find correlations between the values of the input parameters and the output D_R and a. The following two methods are used to accomplish this.

- **Correlation graph** - For every input parameter named below a two-dimensional graph was plotted to see if the dots form a pattern. This gives a very good indication of a possible correlation in the data. Also, even when a high correlation coefficient is found, a highly irregular graph can prove that there is in fact no real correlation.
- **Correlation coefficient** - A fairly standard method is used to calculate a correlation coefficient, using the formula $C(X, Y) = \sum(x - \overline{x})(y - \overline{y})/\sqrt{\sum(x - \overline{x})^2 \sum(y - \overline{y})^2}$, where X and Y are the two sets of numbers that might be correlated. The following explanations are used for the correlation values. Negative values mean that there is the same amount of correlation as for its absolute value, the only difference is that if X increases, Y decreases. Correlation coefficients are interpreted as follows:
  - **0 .. 0.25** - Not correlated.
  - **0.25 .. 0.5** - There might be some correlation, but there is also a good chance that this a merely a coincidence.
  - **0.5 .. 0.75** - There is some correlation.
  - **0.75 .. 1** - The two series are highly correlated.

The following items were tried to correlate with D_R and a:

- **Perlin set** - This is the only input parameter that was really expected to show a correlation with both D_R and a. The presence of higher layers should also raise the total fractal dimension. Furthermore, the distribution of layers was expected to influence a as follows: A high amount of layers results in a value close to $a = 0$, whereas a small amount of layers yields either a negative or a positive value for a, depending on their location.
- **Transform mix-in relation** - This is tried for the whole result set at once, as well as with the results of separate Perlin sets. A correlation was not expected.
- **Minimum fill ratio** - A correlation was not expected here.

Chapter 5

Software design

To carry out the tests described in the last chapter, the software described in this chapter was designed and implemented. During development some speed optimizations were implemented. These are described here as well.

5.1 Overview

The design of the software consists of four major algorithm parts. For an accurate description, some of these are divided into smaller parts, which are described later. Within the software, *work orders* are used as instruction messages to drive the system. These work orders are created by the test generator and sent to a work thread. After processing, the result is sent to the result catcher, and the test generator is notified that the work thread is free for more processing. The four major parts are the ones that can be seen in Figure 5.1.

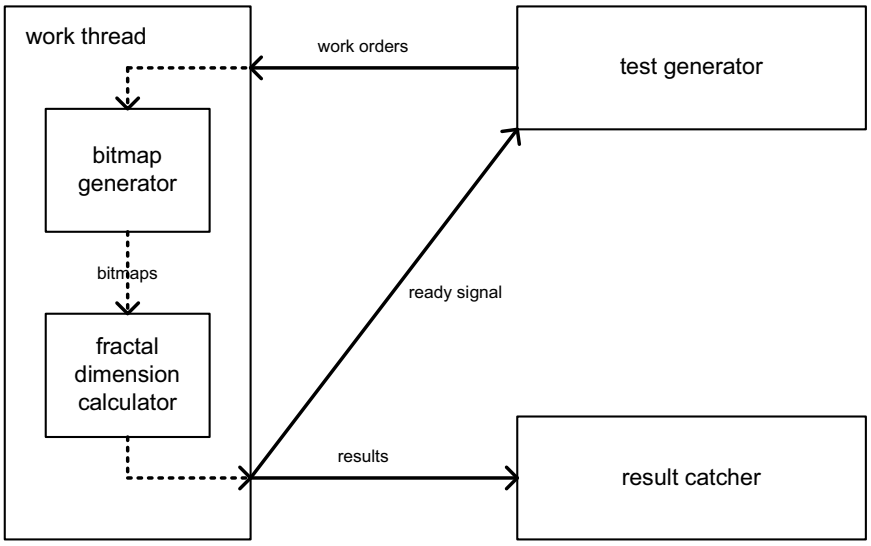


Figure 5.1: Overview of the software design

- **Test generator** - This part of the software generates the test set, as described in Section 4.3. The test set is made up of an amount of work orders, that are sent one by one to the work thread. When the work thread finishes a work order, the result is sent to the result catcher,

and the test generator receives a ready notification. The test generator then sends the work thread a new work order.

- **Bitmap generator** - This part receives a work order as its input and generates a city boundary bitmap as described in the last chapter. This part entails the Perlin noise algorithm, the transformation function and the threshold method. When a bitmap is created it is sent on to the part that does the calculation of the fractal dimension.
- **Fractal dimension calculator** - This block represents the algorithm that calculates fractal dimension from a city boundary bitmap. To do so this part first converts the hull found in the bitmap to a vector representation, calculates a suitable r_0 as described in Section 4.1, and then runs the structured walk test for a number of scales. The fractal dimension for the different scales is calculated and added to the result structure, which is sent on to the result catcher.
- **Result catcher** - This part of the software has the task to calculate some statistics per result, such as average fractal dimension and standard deviation. It also does the linear regression analysis and calculates the standard deviation to this line. Since all the results are gathered here it can also calculate averages and a fail ratio per iteration. When all results have been gathered, they are saved to a CSV[1] file to enable further analysis with help of for instance Microsoft Excel. The result catcher is not elaborated any further in this chapter, since it exactly follows the statistics guidelines as described in Section 4.4.

[1] Comma Separated Values

5.2 Detail

5.2.1 Data types

In the last section two data types were implicitly defined, a work order and a result type. These types are discussed here shortly.
The work order is a simple struct-like class, containing information for the work thread, as is displayed in Figure 5.2. This information for instance contains the random seed that is used by the Perlin noise implementation, but also the input parameters as defined in Section 3.5.
The work result is also a simple struct-like class, and is used as a container for the results of the work thread. After it has been filled by the work thread, it contains information such as whether the thresholding was successful, the fill ratio that was obtained, and a list of fractal dimensions for different scales, as described in Section 4.1.

5.2.2 Test generator

In the software implementation the test generator has two main functions, creating the work orders and controlling the execution of these work orders. A work order is created by iterating through all the ranges of the input parameters. Prepared work orders are sent to a work thread as soon as there is one free. The test generator then waits for a signal from the work thread, and sends the next work order, until all possibilities have been processed.

5.2.3 Bitmap generator

The bitmap generator consist of four defined parts as can be seen in Figure 5.2.

Random generator

As a random generator, the publicly available C implementation of the Mersenne Twister [16] is used. The implementation had to be slightly adapted to ensure correct functioning in a possibly multi-threaded environment. Beforehand, the internal state of the random generator was saved in static variables, allowing only one instance of the generator to be used.
To enable the use of multiple instances of the Mersenne Twister algorithm, a class was defined containing the already available code. The static state variables were transformed into private class members and the static functions into methods. Compilation of the methods did not give any trouble, because member names were not changed from their corresponding static variables.

2D Perlin noise generator

The Perlin noise implementation consists of two parts. First, the noise class is *prepared*, as described in method 2 of Section 2.4.2. This means that the prepare function is called with a start layer and an end layer, after which a bitmap is created for every layer. For these layers, numbered l, these bitmaps have sizes of 2^l x 2^l. Consequently these bitmaps are filled with random floating point values between 0 and 255. This range was chosen to be able to easily convert the bitmaps to an image format that can be viewed with a simple image viewer.
Once the source data is prepared, Perlin noise can be output. The output function also accepts a start and an end layer number, which are checked to lie in the range of the prepared data, as well as an output image width. The output image is sized to the given width and the same height and initialized with zero values. After this a normalization factor is calculated, which is represented by the denominator in Equation 2.24, by summing all separate layer factors. For these layer factors Equation 2.23 is used.
To fill the output image, every pixel is evaluated separately. Since the corresponding image to every prepared layer has a different size, the coordinates of the pixels in the output image are converted to normalized coordinates between 0 and 1. With these coordinates the layer images are sampled. To sample an image the four pixels around the given coordinate are looked up and interpolated. This

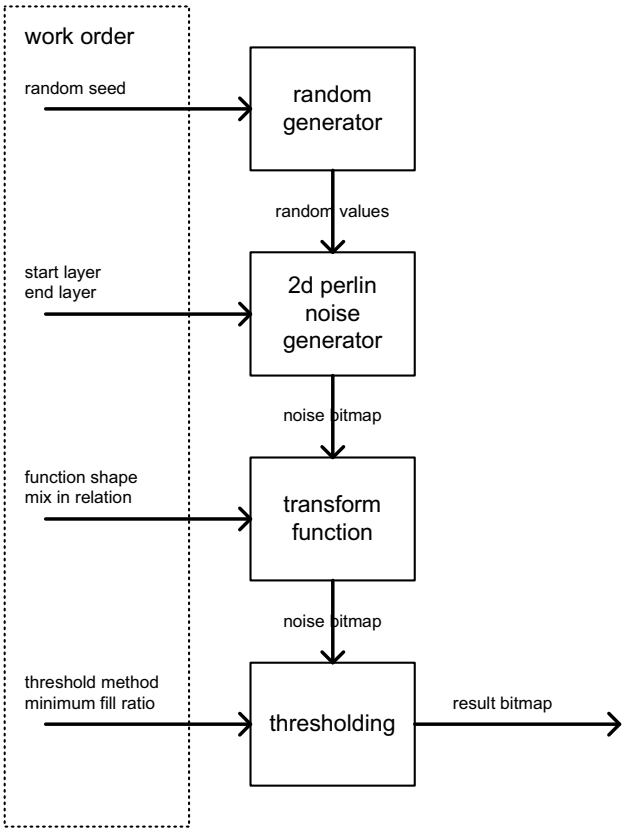


Figure 5.2: Bitmap generator design overview

process can be seen in Figure 5.3; Two layer images, of layers 1 and 2, are shown overlayed over each other. The thicker lines and dots mark the pixel borders and middles of layer 1, the thinner ones show layer 2. The cross marks a coordinate to be sampled. The thinner arrows show the distance to the four layer 2 pixels around this coordinate, the thicker arrows to the corresponding layer 1 pixels. These distances are used for the interpolation function.
As mentioned before in Section 2.4 linear interpolation is used. Since all values in the layer images are in the range from 0 to 255, the interpolated values are also in this range. As in Equation 2.22 every interpolated value is multiplied with its own layer factor. All sampled values are then summed together and divided by the summed layer factors, creating again a value between 0 and 255. This value is written to the output image.

Transform function

The implementation of the transform function is very straight-forward; First it determines the minimum and maximum pixel value in the bitmap. The difference between these two is defined as the image range. Then, every pixel in the bitmap is normalized using the minimum pixel value and the image range. After normalization, Equation 3.1 is used to transform the image. For D_{xy} the distance of every pixel to the center of the bitmap is calculated. The distance from a pixel on the rim of the image, lying orthogonal to the center, is defined as 1. If a distance is greater than 1, for instance in the corners of the image, the value is capped to 1.

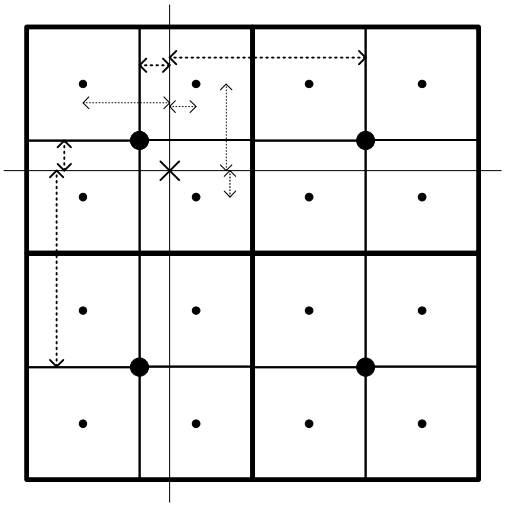


Figure 5.3: Layer image pixel sampling

Thresholding

To apply a threshold to the image, method 4 as described in Section 3.3 is used. As a first step for this method, the absolute minimum threshold is defined by the highest value on the rim. This can be obtained very easily by evaluating all pixels on the rim and taking the highest value.
Another threshold is defined by the minimum fill ratio demanded from the work order. This is converted to a minimum amount of filled pixels by multiplying it by the total amount of pixels in the bitmap. To find the threshold this minimum amount of filled pixels should be filled by the pixels with the highest values in the bitmap. To accomplish this efficiently, all pixel values are first indexed in an array by counting them. This process is depicted in Figure 5.4.

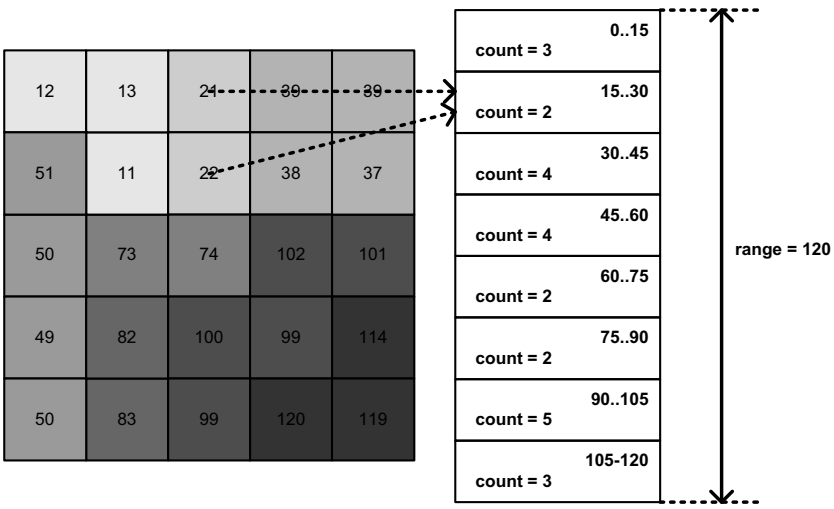


Figure 5.4: Pixel value indexing

The figure shows an input bitmap of 5 x 5 pixels on the left side and an indexed array on the right. The array represents the total pixel value range as calculated in the transformation function step,

in this example a range of 120 is shown. This total range is translated to a separate subrange for every item in the array. Every array element counts the amount of pixels that have a value in its own subrange. In the example it can be seen that the second element represents the subrange of 15..30 and that there are two pixels counted in this range.
In the real implementation the numbers used are 1024 index array elements and of course input images of 512 x 512 pixels. The subrange of a single element is therefore defined by *range*/1024. Filling the index array is as simple as for every pixel dividing its value by the subrange and rounding that value down. This value is the index in the array of the element that should be incremented by one.
After the array has been filled it can be used to find the threshold that is as high as possible but still makes sure that at least the needed minimum fill ratio is fulfilled. The minimum amount of pixels to fill was already derived from the minimum fill ratio. A loop is started at the last element in the array, which represents the highest pixel values in the bitmap. The value of this element is subtracted from the amount of pixels to fill. This is repeated with every lower value in the array until the amount of pixels left to fill is zero or below zero. The index of the last element used is then multiplied by the subrange per element again to find the threshold needed to fulfill the minimum fill ratio requirement. This process is shown in Figure 5.5.

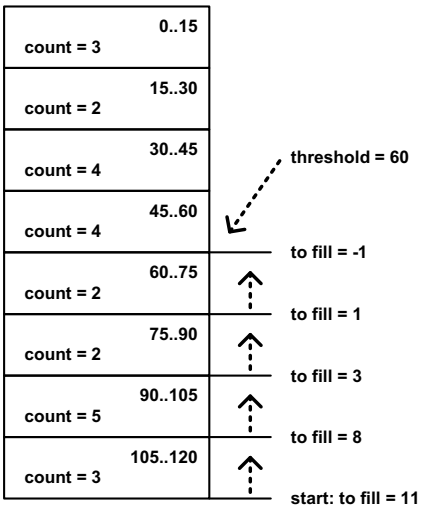


Figure 5.5: Threshold selection

During this step in the process, some of the members of the work result structure are set. With regards to the thresholding the following values are saved: The absolute value of the amount of pixels left to fill after the threshold has been found, the same value but expressed as a percentage of the total amount of pixels in the image, and the threshold found in the process. This threshold is then finally applied to the bitmap. All pixels in the bitmap are evaluated; When its value is higher than the threshold it remains unchanged, if the value is lower it is set to zero.

5.2.4 Fractal dimension calculator

After the bitmap has been generated it is analyzed by the test program. In order to do so, the hull is extracted from the bitmap into a vector format, and the fractal dimension of this hull is calculated at multiple scale levels. This is illustrated in the overview in Figure 5.6. The extraction of the hull into a vector representation has been implemented exactly as described before in Section 3.3 and will not be elaborated any further here. To make sure the algorithm picks up a form in the bitmap, it

is started at 16 equally distributed points on the left side of the image. To find the biggest enclosed shape, the resulting array with the highest amount of coordinates is selected.

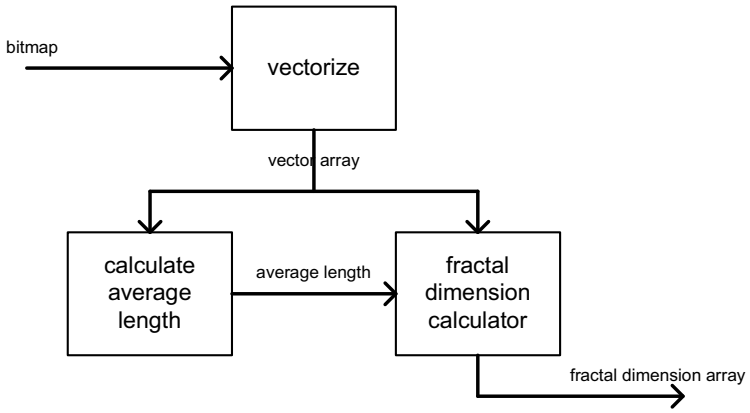


Figure 5.6: Fractal dimension calculator design overview

Calculate average length

To calculate the fractal dimension, a few inputs are needed for the structured walk algorithm. One of these inputs is defined in Section 4.1 as l_0, the input chord length. To stay as close as possible to the implementation used in [1], the average length of the line segments in the extracted hull is used. In the implementation this is simply done by taking the length of every line segment and calculating the average value.

Fractal dimension calculator

The fractal dimension calculator uses an implementation of the structured walk algorithm as described in Sections 2.1.4 and 4.1. Since the algorithm has already been described extensively this is not elaborated any further here.

5.3 Optimizations

Since the original tests showed processing times of about 15 seconds per work order and the total amount of tests would have taken more than 33 days to complete, ways were found to speed up the process. The most effective optimizations will be discussed here.

5.3.1 Multi-core

The original implementation was built on a single-core AMD Athlon XP 2600+ processor. Since more powerful computers are available these days, it seemed logical to make use of a 2.13 GHz Intel Core 2 Duo to speed up the execution of the algorithm. Alas, since the original implementation was single-threaded, only one of two available CPU cores was used[2] and the algorithm executed a mere 1.5 times faster, despite the computer being about 2.5 years newer.
To spread an application over multiple processing units it is necessary to make it multi-threaded, and with at least as many threads as cores are available. To do so, the most demanding parts of the algorithm, the bitmap generator and the fractal dimension calculator were placed in a thread. To create as little thread interdependencies[3] as possible it was chosen to feed all input at once at algorithm start. During execution results are then gathered internally in the thread and only sent back to the main thread when the algorithm is finished.
The easiest way of ensuring that a piece of memory is not read and written, or just written simultaneously by multiple threads is by giving each thread its own instance of all classes needed. This for instance goes for the Perlin noise generator, but also for the random generator. In case of the random generator there is even an extra reason for this.
Even if the implementation of the random generator would be made thread-safe, there would be no way of predicting which thread would get the next random number. This is not a problem per se, but output of the algorithm would not be fully deterministic anymore, leading to the fact that a later run with the same random seed could give different results. This situation is not preferable from a benchmarking point of view. Firstly, it would mean that it is not enough to store only the input set of a test, because the results could vary. Secondly, it is in this case also not possible to compare old results with new results after optimizations or other alterations have been made. These problems are solved by assigning each work thread its own instance of the random generator.

Results

On Microsoft Windows, as on other operating systems, an application can retrieve the amount of separate processing units the host machine has available. To make use of all cores at least the same amount of threads must be started. On the other hand, to prevent unnecessary overhead of thread switching the amount of threads should be kept a low as possible. It is thus chosen in the application to start the same amount of threads as the system reports cores being available.
One work order still takes the same amount of time to complete, but two work orders can be carried out at the same time. As a result, the total test set could be processed in about half the time, the Core 2 Duo now being three times faster than the Athlon XP. Reasons for the good performance scaling after the enabling of the second thread can be the fact that the threads can execute almost without communication to other threads, thus not interfering with each other, and the availability of enough cache memory (2 MB) to run both threads in.

5.3.2 Structured walk algorithm

As described in Section 2.1.4, according to Equations 4.6 and 4.8 it is necessary to start the structured walk once for every starting point $x \in 1..V$. The structured walk test is, due to its nature, of magnitude

[2] This is easily visible in a system monitor application such as the Windows Task Manager: On a dual core processor the Task Manager reports two cores with separate CPU usage percentages. When running one single-threaded application these CPU usages will report 100% only one at a time.

[3] Also with the main thread in which the test generator as well as the user interface code runs.

$O[V]$. Doing this for every starting point means a magnitude of $O[V^2]$. Another problem is that this whole procedure has to be repeated for every value of l_n, where n is chosen out of a set of $1..W$. This brings the total magnitude of the fractal dimension calculations of magnitude $O[V^2W]$. For reference: In these tests W is usually chosen as $W = 12$.
In practice this problem can easily be seen. Depending on the amount of input vectors in the output of the vectorization step, the calculation of the fractal dimension takes up about 50% to 95% of the processing time needed, because of the exponential nature of the algorithm. If the structured walk algorithm were to execute only twice as fast, total execution time could go down by 25 to 47.5%.

Solution

It is proposed to, instead of starting the structured walk test once for every starting point $x \in 1..V$, only execute it once for every starting point where $x \equiv 1 \pmod{s}$, in which s represents the value added to x in every iteration of the loop. In the work order type shown in Section 5.2.1 this is incorporated with the `m_Structured_Skip` member. A value of $s = 1$ naturally gives the same results as the original algorithm.
When choosing higher values for s the algorithm will execute faster, it now has a magnitude of $O[V^2W/s]$. Although since $s < V$ this improvement does not solve the quadratic V^2 part, it should still result in a total execution speed-up of s times. On the other hand, since not all possible input is tested accuracy will be lost. The following method was used to test this inaccuracy:

- Define a representative input set of work orders.
- Create a reference set of fractal dimensions by running this set of work orders with $s = 1$.
- Run the same set of work orders multiple times with different values of $s > 1$.
- For every tested value of $s > 1$, calculate:
  - Maximum absolute difference[4] $Diff_{max}$
  - Average absolute difference $Diff_{avg} \leq Diff_{max}$

Results

The following tolerances were defined as acceptable:

- $Diff_{max} < 0.0025$. The values measured in datasets of existing cities were given with a significance of at maximum 0.01 in [1]. To stay on the safe side in determining minimum accuracy, it was defined that the maximum absolute difference should not be greater than a quarter of this. This prevents the inaccuracy from being so big that false assumptions can be made.
- $Diff_{avg} < 0.0005$. When maximum absolute difference is kept under a certain value this ensures that, although $Diff_{max}$ is not that high, not every value has this relatively high difference. For instance when $Diff_{max} = 0.0020$, which is well within limits, it would be possible that $Diff_{avg} = 0.0020$ as well, meaning that every fractal dimension calculated has the same relatively high inaccuracy.

The total test containing 5368 different work orders took about two hours to complete. The range of s was defined as $1 \leq s \leq 20$. The results are shown in Table 5.1. As discussed earlier more time is saved with higher values of s. Therefore, the highest value of s should be chosen from the table for which the corresponding $Diff_{max} < 0.0025$ and $Diff_{avg} < 0.0005$, meaning $s = 15$.
For the rest of the testing therefore member `m_Structured_Skip` has become this value. Measured speed-up by this optimization was about 10 times on average for the execution of a full work order, which means from bitmap generation to fractal dimension calculation.

[4] Absolute difference is defined as $|D_{s=1} - D_{s=x}|$.

s	$Diff_{max}$	$Diff_{avg}$
1	0.00000	0.00000
2	0.00057	0.00003
3	0.00061	0.00005
4	0.00079	0.00006
5	0.00136	0.00008
6	0.00147	0.00009
7	0.00129	0.00010
8	0.00164	0.00011
9	0.00211	0.00013
10	0.00201	0.00013
11	0.00192	0.00014
12	0.00226	0.00015
13	0.00230	0.00016
14	0.00216	0.00017
15	0.00230	0.00018
16	0.00264	0.00018
17	0.00380	0.00019
18	0.00251	0.00020
19	0.00340	0.00021
20	0.00317	0.00021

Table 5.1: Overview of $Diff_{max}$ and $Diff_{avg}$ for different skip values

Chapter 6

Results

6.1 Results of the *what works* test

Running the *what works* test took in total about an hour. Within this time-span 10240 city hulls were created and evaluated. This means that on average around 0.35 seconds were needed per result. In total 735 of the generated hulls passed validation, which comes down to 7.1% of the entire output. Unexpectedly, the only Perlin sets in the results that passed validation were the ones with and end layer of 8 or 9. These were distributed as follows:

Start layer	End layer	Passed count	Percentage
3	8	176	$\approx$ 24%
2	9	488	$\approx$ 66%
6	9	71	$\approx$ 10%

As was expected, when the minimum fill ratio increases, the amount of valid results decreases. This is displayed in Figure 6.1.

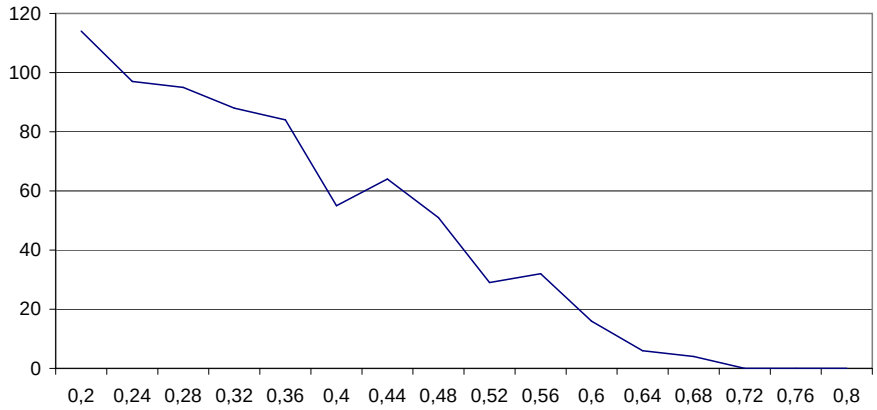


Figure 6.1: The amount of valid results (vertical axis) related to the minimum fill ratio (horizontal axis), of the *what works* test.

6.2 Results of the full test

Running the full test took in total about 16 hours. Within this time-span 175680 city hulls were created and evaluated. This means that on average around 0.33 seconds were needed per result. In total 29016 of the generated hulls passed validation, which amounts to 16.5%.

6.2.1 Summaries of valid results

As in the *what works* test, the only Perlin sets in the results that passed validation were the ones with an end layer of 8 or 9. These were distributed as follows:

Start layer	End layer	Percentage passed of possible
2	8	6.12%
3	8	8.47%
4	8	9.03%
5	8	4.41%
2	9	25.14%
3	9	29.64%
4	9	33.98%
5	9	27.55%
6	9	4.31%

Per Perlin set there is a total amount of $B \cdot H \cdot M = 19520$ results, as explained in Section 4.3.2. The percentages shown in this table represent the amount of those 19520 that were found to be valid. As defined in the same section, all sets that have a validity percentage smaller than 5% were deleted from the database, leaving 27314 valid results.

Transformation slope form

As expected, the amount of passed results per slope form are almost equal. For $n = 1$ the amount is 13259 and for $n = 2$ it is 14055. Since it is hardly possible to do a correlation test on a variable that can only take two different values, this will not be done.

Transform mix-in relation

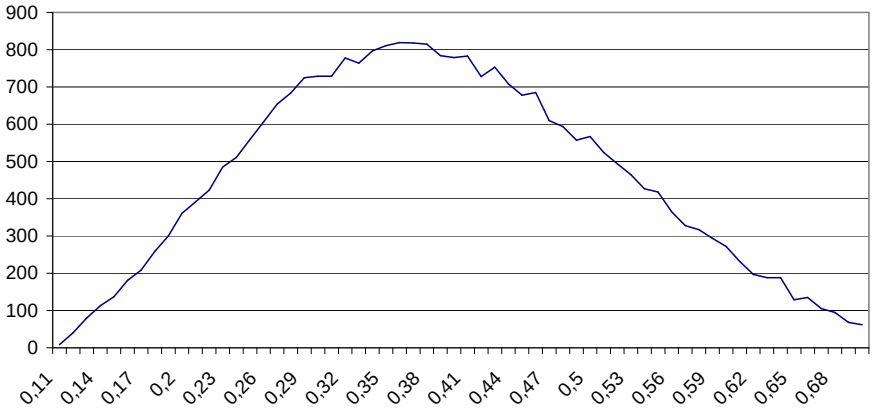


Figure 6.2: The complete result set: The amount of valid results (vertical axis) in relation to the Perlin mix-in relation c_2 (horizontal axis).

When looking at Figure 6.2, it is clearly visible that the expectation of low on the sides and high in the middle, as described in Section 4.4.3, was fully correct. On the top of the graph there are only 818 valid, out of a maximum possible 2240, results. This is a little more than 35%.

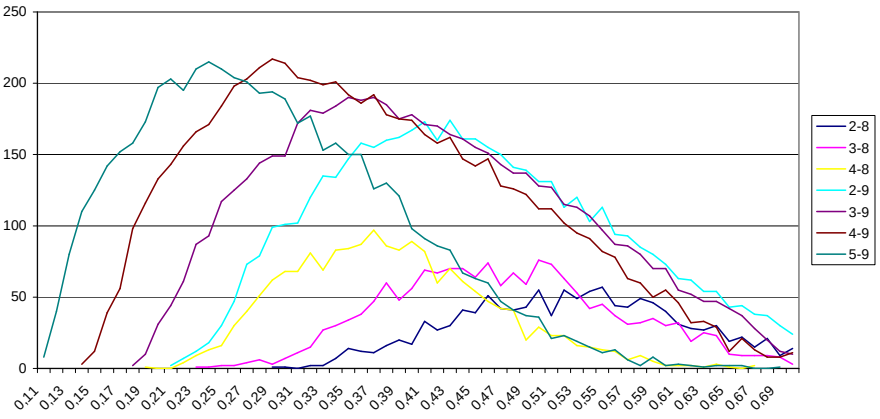


Figure 6.3: The complete result set: The amount of valid results (vertical axis) in relation to the Perlin mix-in relation c_2 (horizontal axis). Each line in the figure represents a Perlin set, as defined in the legend.

In Figure 6.3, a more detailed image is displayed. It shows that there is a separate top for every Perlin set, each one slightly shifted horizontally. While moving to right the tops also become flatter. The highest tops here have validity percentages of up to 67%. The total amount of results possible for every combination is 320 in this case.

Minimum fill ratio

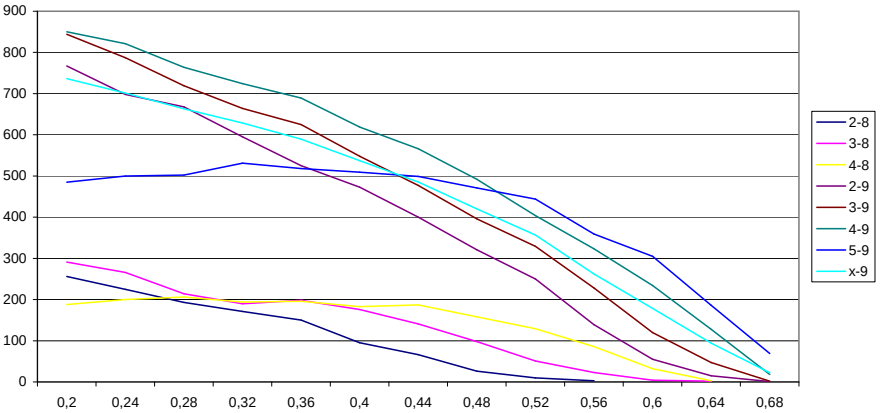


Figure 6.4: The complete result set: The amount of valid results in relation to the minimum fill ratio. Each line in the figure represents a Perlin set, as defined in the legend. The item *x-9* in the legend represent the average of the four lines of the Perlin sets that end on layer 9.

As can be seen in Figure 6.4, a lower minimum fill ratio ensures more valid results. At a minimum fill ratio of 0.2 on the 4-9 line there are even 850 valid results, which is almost 70% of the total amount of 1220 possible results. It is also visible that at a minimum fill ratio of 0.4 there is on average still

about 45% valid in the Perlin sets that end in layer 9. As the minimum fill ratio grows larger than 0.4, the percentage of valid results rapidly drops to zero.

6.2.2 Correlation results

Perlin noise layers

In Figure 6.5 the distribution of the fractal dimension outcome over the different Perlin sets is displayed. It shows that there is a difference between the sets that include layer 9 and the sets that go up to layer 8, in that they range up to a higher dimension. It also shows that, at least in the part that is visible here, there are no noticeable differences between the sets with different start layers. Of course, only the part within the range $1.18 < D_R < 1.28$ is visible here, so it is possible that outside of this range a difference would be visible.

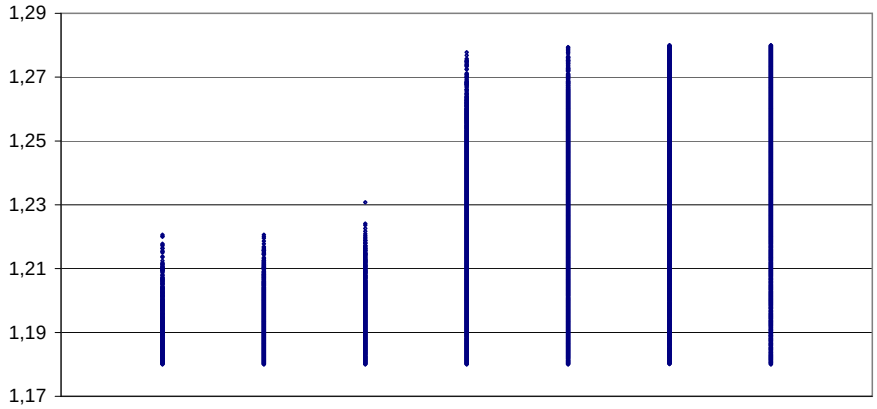


Figure 6.5: The complete result set: The distribution of fractal dimension D_R (vertical axis) over the different Perlin sets (horizontal axis). Displayed from left to right: Perlin sets 2-8, 3-8, 4-8, 2-9, 3-9, 4-9 and 5-9.

Even though the figure does not show much correlation, a correlation value of almost 0.60 could be attained by using a combined value for start and end layer. The highest coefficient was found using the formula $endLayer \cdot 5 + startLayer$ to combine these two, the end layer thus being most significant. Apparently the correlation formula picks up on the difference between the sets with the end layers 8 and 9 already mentioned above.

Transform mix-in relation

It can be seen from Figure 6.6 that there is hardly any relation between the Perlin mix-in factor and fractal dimension when regarding the whole result set at once. Therefore the same correlation test was also tried with the results of just one Perlin set. This showed results similar to this graph, although a little more relation was visible.
To try to find a direct correlation between the Perlin mix-in factor and fractal dimension, correlation graphs were also made of results with only a variable mix-in factor, all other parameters remaining constant. One of these cases[1] is shown in Figure 6.7. A positive correlation is visible here, which can also be found in the correlation coefficient of 0.72.

[1] For the displayed case, the values for the other parameters are: Perlin start layer 2, Perlin end layer 9, minimum fill ratio 0.4, slope form $n = 2$

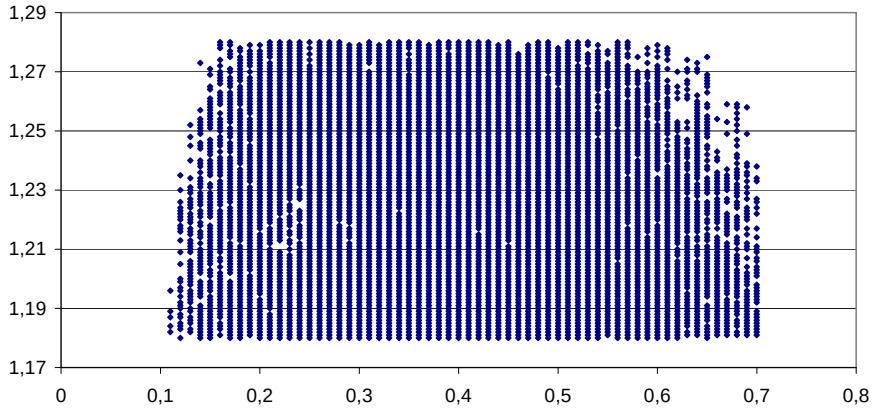


Figure 6.6: The relation between the Perlin mix-in factor c_2 (horizontal axis) and fractal dimension D_R (vertical axis).

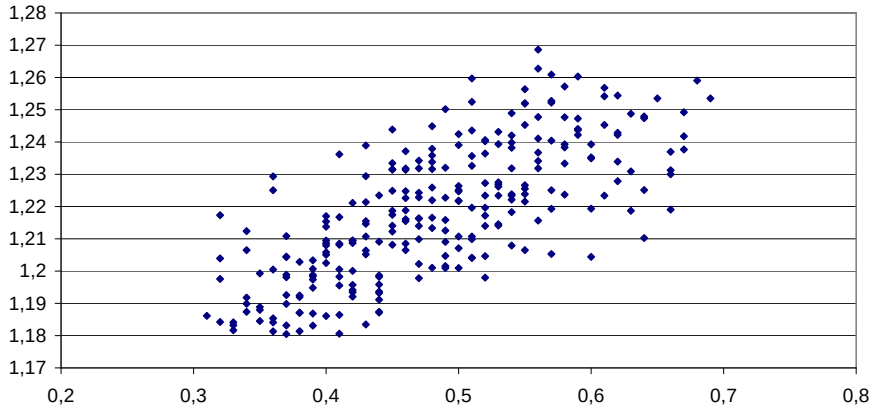


Figure 6.7: The relation between the Perlin mix-in factor c_2 (horizontal axis) and fractal dimension (vertical axis), with the mix-in factor as the only variable.

Minimum fill ratio

Figure 6.8 shows that when looking at the whole result set, there is no relation between the minimum fill ratio and fractal dimension.
As was done with the Perlin mix-in factor it was also tried to find a correlation when the fill ratio is the only variable parameter. An example[2] graph of that can be seen in Figure 6.9. This is one of the test cases where it seemed as though some correlation was found, but even this case has an, absolute, correlation coefficient lower than 0.5.

[2]For the displayed case, the values for the other parameters are: Perlin start layer 2, Perlin end layer 9, perlin factor $c_2 = 0.4$, slope form $n = 2$

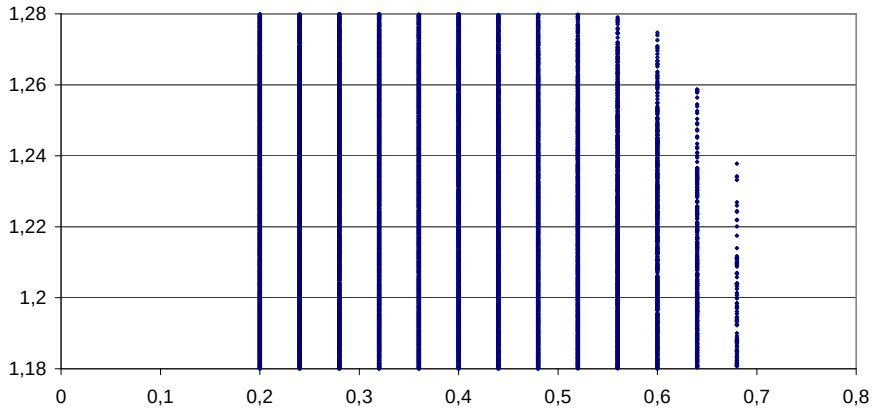


Figure 6.8: The relation between the minimum fill ratio (horizontal axis) and fractal dimension D_R (vertical axis).

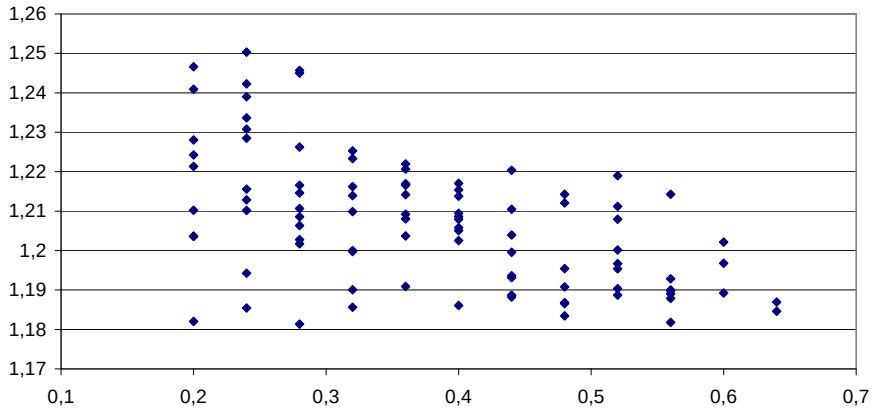


Figure 6.9: The relation between the minimum fill ratio (horizontal axis) and fractal dimension (vertical axis), with the fill ratio as the only variable.

6.3 Reflection

Perlin noise layers

It was expected that the choice of layers included in the input noise image determines to a large part the fractal dimension of the created hull, with higher fractal dimensions only being possible with more layers of noise. The results also show that there is some positive correlation between the number of the end layer and the range in which the fractal dimensions lie. Unfortunately, within the valid range of city dimensions, the different Perlin layer ranges do not show much coherency in the output. It is very well possible that the differences show on a wider scale of the fractal dimension range, but that does not apply to the topic in this research.
With respect to the amount of valid results it is clearly visible that for Perlin sets 2-9, 3-9, 4-9, 5-9 these numbers are higher than for any other sets. They all have validity percentages of at least 25%. This number can still be increased by choosing the other parameters in a way that the extremities are cut off.

Transform mix-in relation

It was shown in the last section that there is a slight positive correlation between the Perlin mix-in relation parameter and the fractal dimension of the result. This can be explained by the fact that if the amount that the Perlin noise is mixed into the final bitmap is lower, the effect of the Perlin noise will be less, and the effect of the smooth transform function will be higher. Since the Perlin noise is the source of the fractality in the image, it is logical that when lowering the Perlin mix-in factor, the fractal dimension will also be lower. On the other hand, this effect is not very reliable. The mix-in factor can only be used to steer the result a little, but not in a way that it helps predict the fractal dimension within a certain range.
To ensure a high validity percentage, this parameter should be kept within the following ranges. These ranges have been determined from the data that was used for Figure 6.3. These are the tops of the graphs, everything above a value of 150:

- Start layer = 2, end layer = 9: Perlin mix-in factor between 0.35 and 0.47
- Start layer = 3, end layer = 9: Perlin mix-in factor between 0.30 and 0.46
- Start layer = 4, end layer = 9: Perlin mix-in factor between 0.22 and 0.44
- Start layer = 5, end layer = 9: Perlin mix-in factor between 0.17 and 0.36

Minimum fill ratio

From Figure 6.8 it is visible that, in order to reach the full 1.18 to 1.28 range in fractal dimension the minimum fill ratio should not be chosen higher than 0.4. Also Figure 6.4 shows that the chance on a valid result drops when the fill ratio is chosen higher than 0.4. Furthermore it was shown in the last section that this parameter has no such effect on the result dimension that it would allow this parameter to be used as a guiding effect.

6.3.1 Summary

When keeping to the ranges for the input parameters as recommended in this section, there are 6393 valid results in the result set. The amounts of items in these ranges are:

- Items in the ranges of the Perlin mix-in factors: 73
  - 0.35 to 0.47: 13
  - 0.30 to 0.46: 17
  - 0.22 to 0.44: 23

  - 0.17 to 0.36: 20

- Items in the range of the minimum fill ratio: 6
- Amount of iterations: 10
- Amount of different slope forms: 2

Which yields a total amount of results in these ranges of $73 \cdot 6 \cdot 10 \cdot 2 = 8760$. This means that it is possible to reach a total validity of 73.0%.

Research questions

The research questions that should be answered by this chapter are

- What parameters affect the result of the algorithm?
- How do these parameters affect the result of the algorithm?
- Is the output stable for the same input?

As has been shown it is hard to say what parameters affect the fractal dimension of the result in what way. Recommendations have been done about the ranges of all parameters in such a way that a validity percentage of 73.0% is obtained, and which guarantee that the full 1.18 to 1.28 range is covered.
The stability of D_R in the output has also been measured between different iterations of the same input parameters, and the deviations range between 0.03 and 0.07. Since the entire valid range for D_R is only 0.10, these values represent deviations of 30 to 70%. This can not be qualified as stable.

Part III

Conclusion

Chapter 7

Conclusions

In this chapter, first the main research questions are repeated and reviewed, after which a more general conclusion will be given. This chapter will end with some recommendations for further research.

7.1 Overview of the main research questions

How does a hull creation algorithm fit into the rest of city generation?

An overview has been given of current city generators. The one that can be taken most seriously, is the City Engine described in Section 2.3.5. This system is very comparable in function to the design re-lion is looking for, and can use separately generated city hulls without any problems. Since the boundary creation algorithm has been designed with the city generator of re-lion in mind, the proposed algorithm is also usable with that system.

What defines a realistic shape of a city?

It has been discussed in Section 2.2 that a city hull should have fractal properties, with a fractal dimension between 1.18 and 1.28. Furthermore, an estimation has been made for the minimum so called *enclosed area ratio*, which is the relation between the enclosed area and the maximum enclosed area for equal maximum diameter. This estimation results in a value of $r_{min\_encl} \approx 0.0635 = 6.35\%$.

Is Perlin noise a form of fractal?

It has been discussed in Section 2.4.3, using the qualitative properties of fractality, that Perlin noise can be seen as a fractal. The quantitative properties have been shown to be fulfilled in the experiment part of this project.

Do the generated shapes fulfill the earlier "what defines the shape of a city?" question?

When using an optimal set of input parameters, of which an overview has been given in Section 6.3, 73.0% of all generated shapes fulfill the given definition of a realistic city boundary. This is a positive result, since it means that on average three out of every four generated hulls can be used as the base for a virtual city.

What parameters affect the result of the algorithm?

As was shown in Chapter 6, this question can not be fully answered by this research. As a result of this, the question *How do these parameters affect the result of the algorithm?* can also not be answered. However, as discussed before, it is possible to define a set of input parameters that will have a success rate of 73.0%.

7.2 Conclusion

In order to answer the main question of this project, it is first necessary to describe a system that could be built using the algorithm of this project. This system would ask a user for a desired fill ratio between 20% and 40%, and for the other input parameters use the most optimal input settings, as described before. The generated city hull will then be tested for validity, since only 73.0% of all generated cases will be valid.
If this hull is found to be invalid, the process is repeated until a valid one is found. After three iterations, the probability that no valid case has been found is 0.27^3, which is already less than 2%. Even on the older hardware used in the beginning of this project, an AMD Athlon XP 2600+, the evaluation of one test case takes less than a second, so in praxis this process should not be problematic.

> *How can two-dimensional Perlin noise be used for automatic generation of realistic city boundaries within easy-to-use city content creation systems?*

The conclusion of this project is that it is very well possible to use two-dimensional Perlin noise within easy-to-use city content creation systems, such as the planned system of re-lion. It is possible to create the boundaries for the generated city using the algorithm proposed in this thesis, and a test is available to validate their realism. This algorithm does not interfere with the ease-of-use of the rest of the system, since there is only one possible user option.

7.3 Recommendations for future research

There are some questions that this project has left unanswered, and there are some that have arisen after the project had been carried out. These recommendations for further research are given in this section.

How can fractals be used to build up a city shape directly?

The algorithm used in this project uses a two-dimensional shape, generated by Perlin noise, to create a city boundary. One of the factors of realism in these boundaries is their fractality. In the literature on these subjects, ways to use fractal definitions to create city shapes were also found, for instance in [1]. It would be interesting to find out whether it is possible to implement these methods, and if the results thereof pass the same validity tests as used in this project.

How can the algorithm be changed to make it possible to better steer the result?

One of the conclusions of this project is that it is at the moment not possible to predict the exact outcome of the algorithm, in terms of fractal dimension and validity. It would be interesting to find out if modifications can be made to the proposed algorithm, in such a way that its outcome becomes more predictable. If it is possible to narrow this down further, results do not necessarily have to be validated after generation anymore. Furthermore, this might lead to more insight in the mathematical functioning of the algorithm.

How can the relation between suburbs and the main city be implemented by the algorithm? Is it possible to use the fact that sometimes multiple *islands* can be found within the generated shape?

It is mentioned at the end of Section 3.4 that, currently, only the biggest contiguous shape within the bitmap is used as a city hull. However, sometimes multiple, smaller, shapes can also be found within the same bitmap. Future research could analyze how these extra shapes can be used to represent suburbs of the city. Also, it might be possible to find other ways to have the algorithm generate a city including suburbs.

Which other quantifiable properties of cities can be defined to be used in more advanced city generators?

In this project, two quantifiable properties of cities have been defined, the fractality of its boundaries, and the enclosed area ratio. In order to create more advanced city generators, it is necessary to find and describe more of these properties.

How can three-dimensional Perlin noise be used to generate virtual cities? Is it possible to let it define building heights?

This project was focused on the use of two-dimensional Perlin noise. However, Perlin noise can be created in any wanted dimension. An interesting follow-up project would be to use three-dimensional Perlin noise, for instance in order to define building height at a certain position.

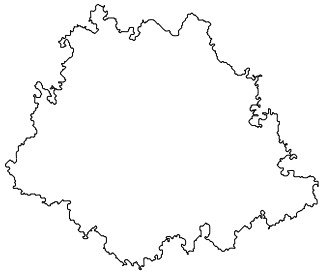
Appendix A

Examples of generated city boundaries

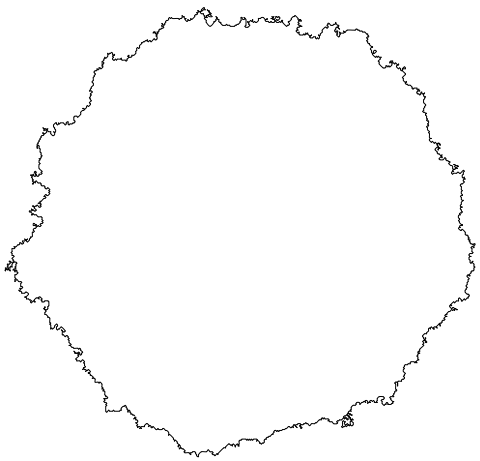
This appendix displays some examples of city boundaries that were created during the full test. The values of their input parameters, and the variables calculated during the validity test are noted with every image. These variables are the following:

- **Random seed** - The random number that was used to seed the Mersenne Twister random number generator, to fill the Perlin noise layers.
- **Perlin layers** - The layers that were used in the Perlin noise.
- **Slope form** n - The form of the slope that was used in the transformation function, as displayed in Equation 3.1.
- **Perlin factor** c_2 - The factor that the Perlin noise was multiplied with, as shown in Equation 3.1.
- **Minimum fill ratio** - The minimum fill ratio that was used, as described in Section 3.3.
- **Regression line slope** a - The calculated slope of the regression line, as displayed in Equation 4.9.
- **Fractal dimension** D_R - The comparable fractal dimension that was calculated, as described in the last paragraph of Section 4.1.
- **Standard deviation** - The standard deviation of the dimensions on different scales from the regression line, according to Equation 4.13.

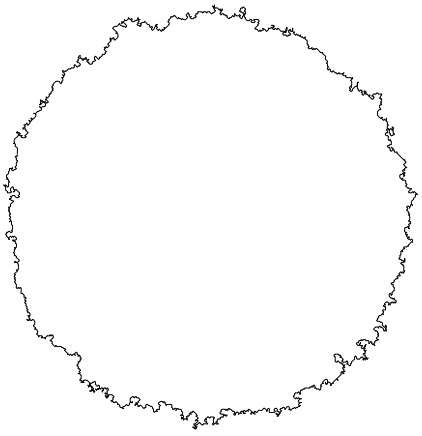
A.1 Valid results



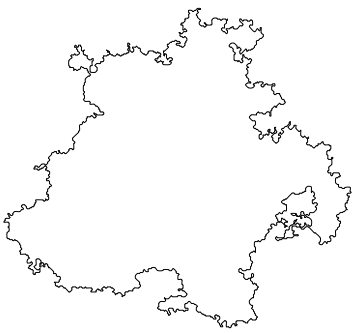
(a) Random seed 2770721032, Perlin layers 3 to 8, slope form $n = 1$, Perlin factor $c_2 = 0.38$, minimum fill ratio 0.20, regression line slope $a = 0.004$, fractal dimension $D_R = 1.180$, standard deviation 0.0146



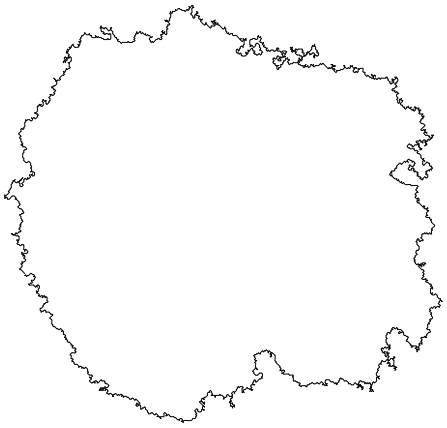
(b) Random seed 0624619307, Perlin layers 3 to 9, slope form $n = 1$, Perlin factor $c_2 = 0.18$, minimum fill ratio 0.60, regression line slope $a = -0.021$, fractal dimension $D_R = 1.180$, standard deviation 0.0124



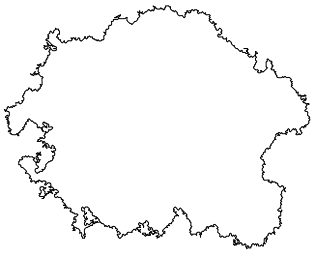
(c) Random seed 0397699697, Perlin layers 5 to 9, slope form $n = 2$, Perlin factor $c_2 = 0.22$, minimum fill ratio 0.52, regression line slope $a = -0.020$, fractal dimension $D_R = 1.200$, standard deviation 0.0167



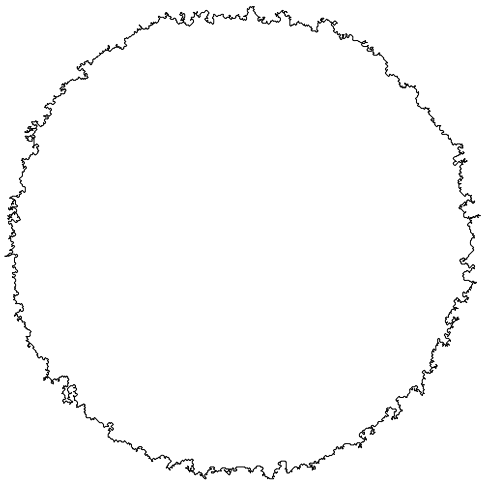
(d) Random seed 0345220084, Perlin layers 2 to 8, slope form $n = 2$, Perlin factor $c_2 = 0.42$, minimum fill ratio 0.24, regression line slope $a = 0.027$, fractal dimension $D_R = 1.200$, standard deviation 0.0101



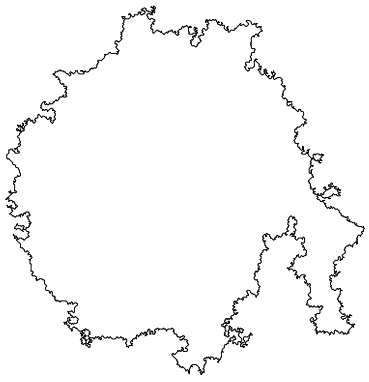
(e) Random seed 3283582743, Perlin layers 2 to 9, slope form $n = 2$, Perlin factor $c_2 = 0.42$, minimum fill ratio 0.52, regression line slope $a = -0.010$, fractal dimension $D_R = 1.220$, standard deviation 0.0128



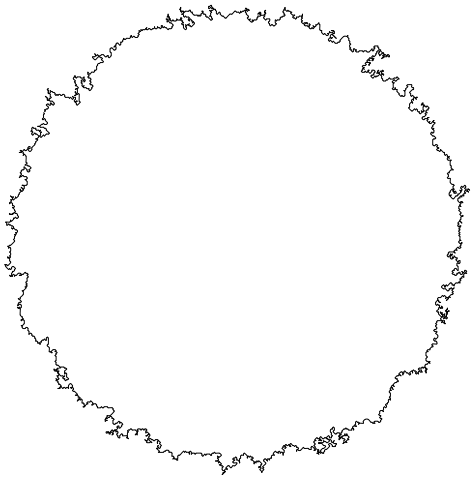
(f) Random seed 0246066605, Perlin layers 3 to 9, slope form $n = 1$, Perlin factor $c_2 = 0.34$, minimum fill ratio 0.20, regression line slope $a = -0.001$, fractal dimension $D_R = 1.220$, standard deviation 0.0127



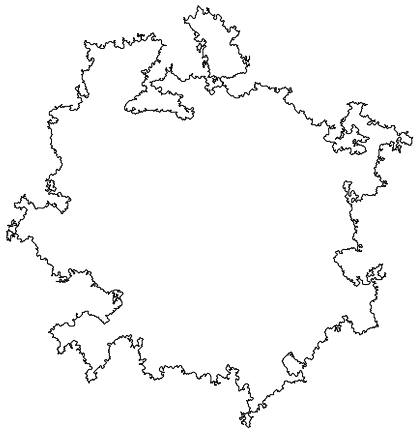
(g) Random seed 2956227368, Perlin layers 6 to 9, slope form $n = 2$, Perlin factor $c_2 = 0.22$, minimum fill ratio 0.68, regression line slope $a = -0.019$, fractal dimension $D_R = 1.240$, standard deviation 0.0197



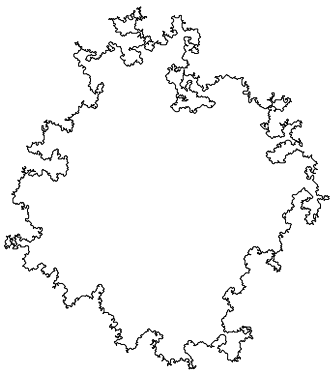
(h) Random seed 1589012818, Perlin layers 3 to 9, slope form $n = 1$, Perlin factor $c_2 = 0.42$, minimum fill ratio 0.32, regression line slope $a = 0.000$, fractal dimension $D_R = 1.240$, standard deviation 0.0095



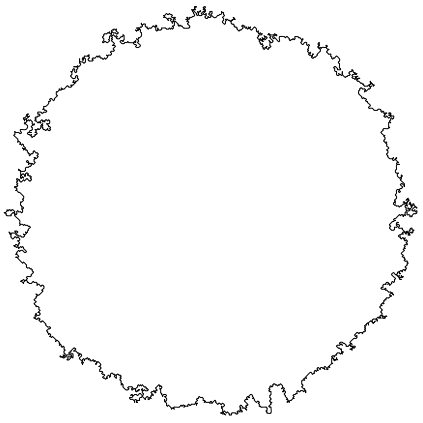
(i) Random seed 1419021824, Perlin layers 6 to 9, slope form $n = 2$, Perlin factor $c_2 = 0.30$, minimum fill ratio 0.64, regression line slope $a = -0.010$, fractal dimension $D_R = 1.260$, standard deviation 0.0182



(j) Random seed 3468463229, Perlin layers 4 to 9, slope form $n = 1$, Perlin factor $c_2 = 0.46$, minimum fill ratio 0.36, regression line slope $a = 0.020$, fractal dimension $D_R = 1.260$, standard deviation 0.0111

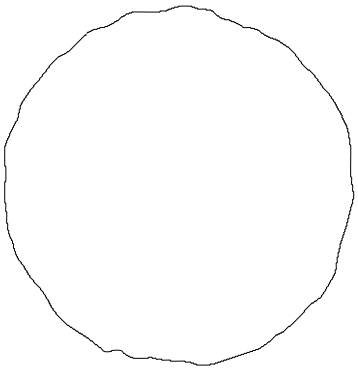


(k) Random seed 2390627975, Perlin layers 5 to 9, slope form $n = 2$, Perlin factor $c_2 = 0.54$, minimum fill ratio 0.24, regression line slope $a = 0.029$, fractal dimension $D_R = 1.280$, standard deviation 0.0095

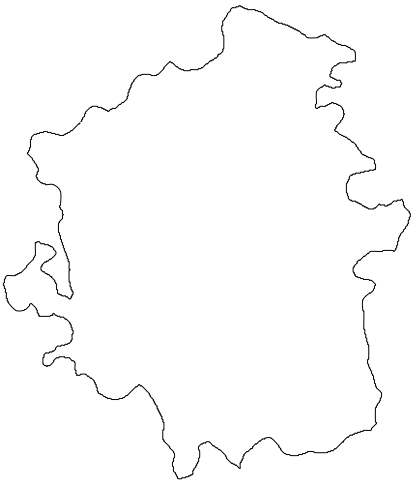


(l) Random seed 0888641242, Perlin layers 6 to 9, slope form $n = 2$, Perlin factor $c_2 = 0.34$, minimum fill ratio 0.48, regression line slope $a = -0.018$, fractal dimension $D_R = 1.280$, standard deviation 0.0183

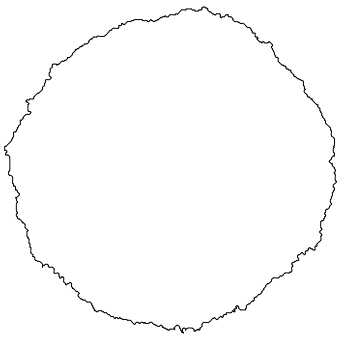
A.2 Invalid results



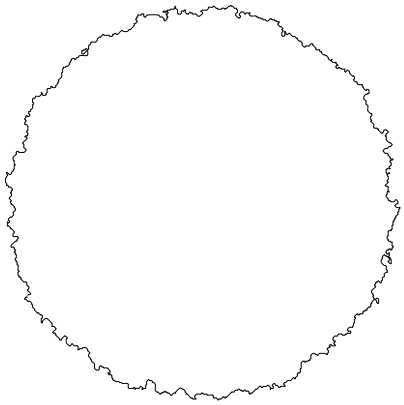
(a) Random seed 1713280110, Perlin layers 2 to 5, slope form $n = 1$, Perlin factor $c_2 = 0.10$, minimum fill ratio 0.40, regression line slope $a = -0.006$, fractal dimension $D_R = 1.000$, standard deviation 0.0013



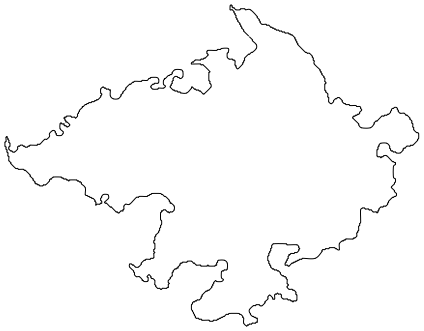
(b) Random seed 0942495785, Perlin layers 2 to 5, slope form $n = 1$, Perlin factor $c_2 = 0.42$, minimum fill ratio 0.48, regression line slope $a = 0.028$, fractal dimension $D_R = 1.050$, standard deviation 0.0090



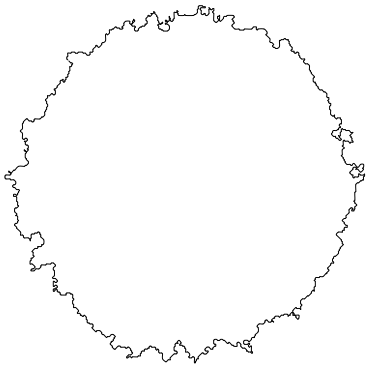
(c) Random seed 2043237144, Perlin layers 3 to 8, slope form $n = 2$, Perlin factor $c_2 = 0.14$, minimum fill ratio 0.32, regression line slope $a = -0.017$, fractal dimension $D_R = 1.050$, standard deviation 0.0019



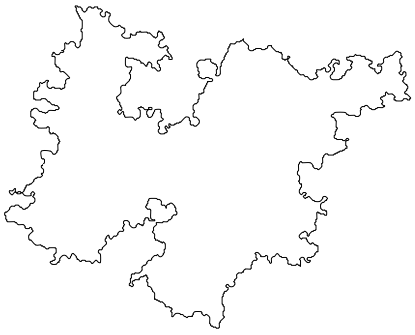
(d) Random seed 1880265999, Perlin layers 5 to 8, slope form $n = 2$, Perlin factor $c_2 = 0.14$, minimum fill ratio 0.48, regression line slope $a = -0.012$, fractal dimension $D_R = 1.100$, standard deviation 0.0124



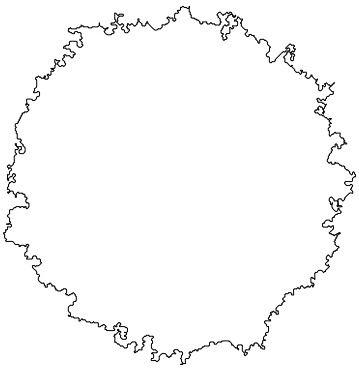
(e) Random seed 0520309446, Perlin layers 2 to 6, slope form $n = 1$, Perlin factor $c_2 = 0.46$, minimum fill ratio 0.24, regression line slope $a = 0.037$, fractal dimension $D_R = 1.100$, standard deviation 0.0098



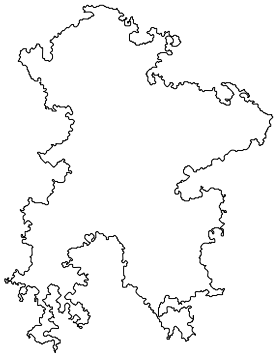
(f) Random seed 2022283867, Perlin layers 5 to 8, slope form $n = 2$, Perlin factor $c_2 = 0.18$, minimum fill ratio 0.36, regression line slope $a = -0.002$, fractal dimension $D_R = 1.150$, standard deviation 0.0212



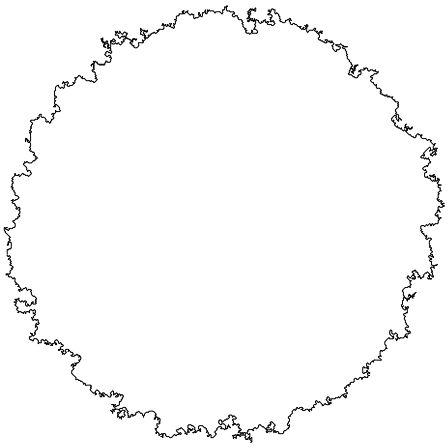
(g) Random seed 0244350575, Perlin layers 3 to 7, slope form $n = 2$, Perlin factor $c_2 = 0.62$, minimum fill ratio 0.24, regression line slope $a = 0.031$, fractal dimension $D_R = 1.150$, standard deviation 0.0129



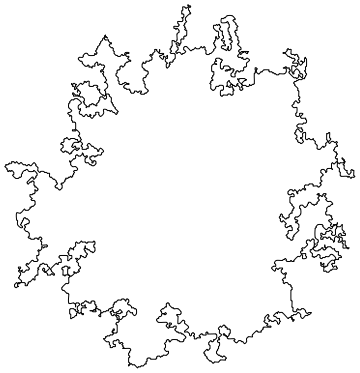
(h) Random seed 3757382766, Perlin layers 5 to 8, slope form $n = 1$, Perlin factor $c_2 = 0.22$, minimum fill ratio 0.36, regression line slope $a = 0.012$, fractal dimension $D_R = 1.200$, standard deviation 0.0244



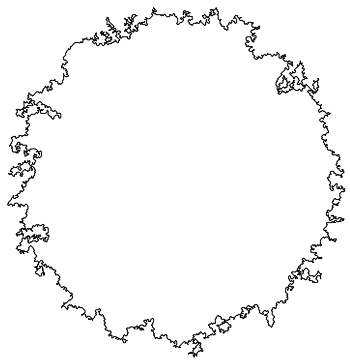
(i) Random seed 2275459955, Perlin layers 3 to 8, slope form $n = 2$, Perlin factor $c_2 = 0.66$, minimum fill ratio 0.24, regression line slope $a = 0.033$, fractal dimension $D_R = 1.200$, standard deviation 0.0119



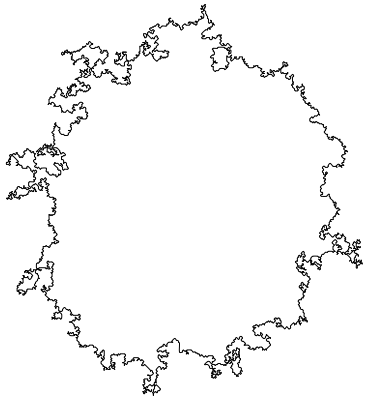
(j) Random seed 2271958465, Perlin layers 5 to 9, slope form $n = 1$, Perlin factor $c_2 = 0.18$, minimum fill ratio 0.56, regression line slope $a = -0.004$, fractal dimension $D_R = 1.250$, standard deviation 0.0240



(k) Random seed 2288246422, Perlin layers 5 to 8, slope form $n = 2$, Perlin factor $c_2 = 0.50$, minimum fill ratio 0.28, regression line slope $a = 0.060$, fractal dimension $D_R = 1.251$, standard deviation 0.0067



(l) Random seed 1152870436, Perlin layers 6 to 9, slope form $n = 2$, Perlin factor $c_2 = 0.34$, minimum fill ratio 0.32, regression line slope $a = 0.023$, fractal dimension $D_R = 1.301$, standard deviation 0.0262



(m) Random seed 1690639016, Perlin layers 5 to 9, slope form $n = 1$, Perlin factor $c_2 = 0.42$, minimum fill ratio 0.32, regression line slope $a = 0.032$, fractal dimension $D_R = 1.301$, standard deviation 0.0102

Appendix B

Urban ecology

From the beginning of the 20th century, models of urban social structure have been developed. These models are focussed on the socioeconomic patterns in cities, for instance ethnic communities. [6] Although most of these models have been developed a long time ago and have received serious criticism, they are still used today as a basis for explaining urban processes [2].

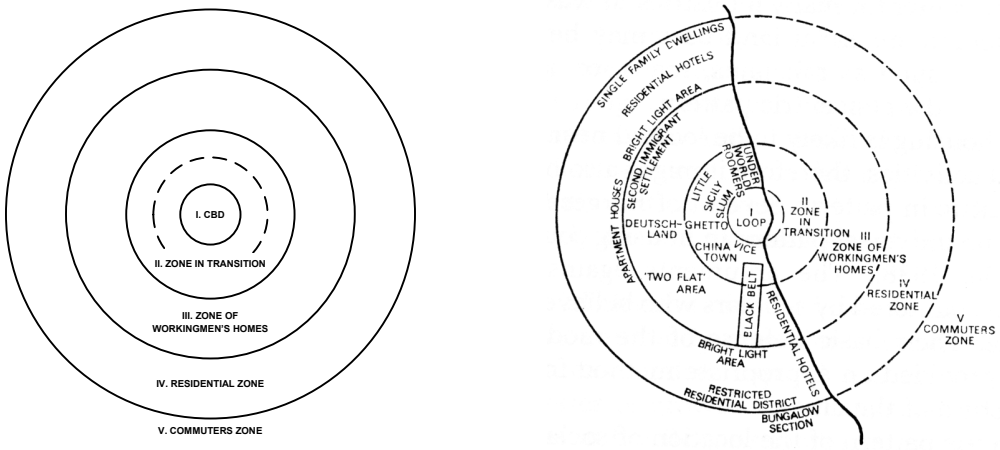
B.1 The Concentric Zone model

In the beginning of the 1920s Robert E. Park and Ernest W. Burgess developed an urban research program within the sociology department of the university of Chicago. During this period their students created numerous social maps of Chicago (the Social Science Research Base Map), which resulted in the first publication of the Concentric Zone model in 1924. These initial zones were [6]:

1. Central business district (CBD)
2. Zone in transition
3. Zone of independent workingmen's homes
4. Zone of better residences
5. Commuter zone

The complete model was first published in 1925 in *The City* [19] and describes six concentric circles originating from the center of the city, as shown in Figure B.1(a). These rings represent zones with different land use and population groups: [21]

- **Zone I** - The Central Business District (CBD), tertiary employment, convergence of urban transport infrastructure and therefore most accessible.
- **Zone II** - A dual zone, containing:
  - Industrial activities taking advantage of nearby labor and markets. Further used for transport terminals adjacent to the CBD.
  - Transition zone, gradually reconverting to other uses by expanding commercial and industrial activities. Poorest segment of the urban population, lowest housing conditions.
- **Zone III** - Residential zone dominated by the working class. Located near the major zones of employment (1 and 2). Low commuting cost location for the working class.
- **Zone IV** - Higher quality housing, longer commuting, higher traveling costs.



(a) The Burgess Concentric Zones Model

(b) Burgess' map of Chicago

Figure B.1: The maps of Burgess

- **Zone V** - High class and expensive housing in a rural, suburbanized, setting with highest commuting costs.

The model is often named *urban ecology*, because Burgess describes processes in cities that follow ecosystem-like rules, the most important one being competition between social groups for scarce urban resources like land [2]. Low-SES [1] groups claim space in the center of the city, pushing other population groups from the center toward the edge of city, a process referred to as *succession*. A zone where the population is changing is called a transition zone.

Criticism

The model of Burgess has been heavily criticized for a number of different reasons. First of all it was easily proven that the CBD is not the only place where commercial activities develop. Industries and commercial land use also concentrates along linear features such as rail lines, roads or water. These activities also attract Low-SES housing for its workers, which then follows a derivative of said linear features. Both of these land use types can thus break the concentric pattern. [6]
Secondly, the model is said to be outdated. It was developed in a time when American cities had fast growing populations and when motorized transportation, such as the automobile, was hardly used as opposed to public transport. Since public transport had good connections to the inner city, that is where the opportunities for growth were, leading to the succession process. From halfway the 20th century highways have enabled expansion to take place in suburbs of a city, avoiding the expensive reconversion process. [21]
Other points of criticism are the fact that the model only described the - relatively young - cities of America, while cities that had started development in pre-industrial times did not follow the concentric circles model at all, and that it assumes spatial separation of place of work and place of residence, only occurring later in the twentieth century.
However, all sources agree that even today the Burgess model can still function as a useful concept explaining the concentric layout of some cities, as the foundation of succession theory and as a way to explain urban growth in American cities in the early-mid 20th century.

[1] SES: Social Economic Status. Burgess measured this mostly by income.

B.2 Hoyts Sector Model [17]

Homer Hoyt proposed his sector model in 1939, which was originally designed as an analytical tool for research on real estate markets in the USA. It was based on a study of 142 American cities, but still received similar criticism as the model of Burgess.
The most important point in Hoyts theory, is the fact that high-SES groups dominate the land use patterns. The starting point is that the highest rental values are in one or more sectors at the edge of the city. The location of these sectors is dynamic over time. High rent areas expand along established lines of travel and amenity areas such as high ground, main transport axes, or shorelines. Next to high rent areas are lower and lower rent areas, in such a way that very low-SES groups do not live next to very high-SES groups.
The process that the sectors are dynamic over time, they move through the city over the years, is called *Filtering*. The richer groups move outward, and their old housing is converted to be used by a group with lower SES. An example of this is large single-family houses being converted to more-apartment buildings, which is possible due to housing deterioration. This way, the lesser rich follow the rich, in the ongoing process of Filtering.

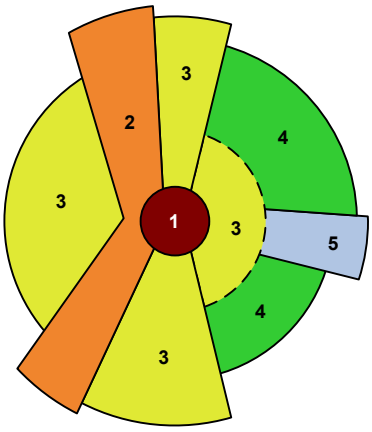


Figure B.2: The sector model of Hoyt

In the theory of Hoyt, linear patterns and main transport routes are very important, as well as the outskirts of the city. Therefore, it arranges the zones in sectors radiating from the CBD. As an example, the following zones are displayed in Figure B.2:

1. CBD
2. Wholesale and light manufacturing
3. Low-class residential
4. Middle-class residential
5. High-class residential

Criticism

Hoyt was criticized in his own time by Walter Firey (1947), who said that urban patterns follow from cultural and emotional factors rather than economic ones. Sentimental attachment to place, community ties. Research in the Boston city. Later both Hoyt and Burgess' models came under attack in the age of the automobile.

B.3 The Multi-Nuclei Theory

Some time after the publication of Hoyts sector model, it became clear that the emerging of larger cities, made possible by better means of transportation, required a different model description. Faster transportation meant that it was not necessary anymore to concentrate all activities in one central place [15]. The multi-nuclei theory, introduced by Chauncy Harris and Edward Ullman in 1945, suggests that a city may have multiple concentrations of activity apart from the CBD [21].

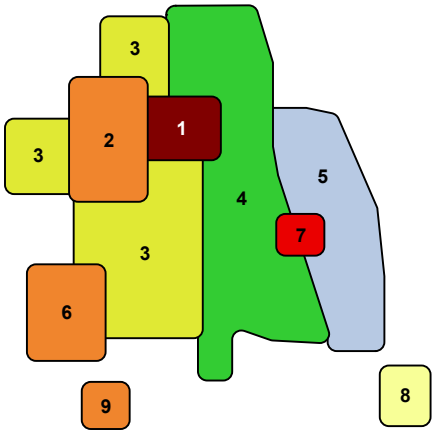


Figure B.3: The multi nuclei model of Ullman and Harris

Figure B.3 is an example of a city layout described by the model of Ullman and Harris. The numbers represent the following areas:

1. CBD
2. Wholesale and light manufacturing
3. Low-class residential
4. Middle-class residential
5. High-class residential
6. Heavy manufacturing
7. Sub business district
8. Residential suburb
9. Industrial suburb

The theory suggests that, over time, many towns and almost all larger cities do not grow around one CBD, but around multiple central places in the urban pattern. While better transportation makes it possible to get lesser needed goods or services from farther away, but while it is still useful to have more needed services located nearby, it is possible for these central places to specialize and thus differentiate from the others. Decisive in these specializations is not only the matter of pure distance, but a more sophisticated set of push and pull factors: [21]

- **Differential accessibility** - Highly specialized facilities are needed by some activities to be successful. Heavy industry for instance requires large amounts of space, and cargo connections such as ports or railway stations. These features can all simply not be found in the middle of a city.

- **Land use compatibility** – Similar activities make different groups pull together. Service activities such as banks, insurances companies and governmental institutions naturally have a large amount of interaction with each other, and will therefore attract each other. Migrants of the same cultural background will attract each other because of a shared view on living, but also groups of similar SES. This way, land use compatibility forms an important pull factor.
- **Land use incompatibility** – It is obvious that certain activities do not go well together, such as heavy industry and residential areas of a high SES. Commercial activities such as banks and insurance companies do not tend to build their offices within areas of lower SES. Land use incompatibility forms an important push factor.
- **Location suitability** – It is not always possible for a group to afford the optimal location within a city. These groups will have to locate to cheaper areas. Although these places are not optimal, they are at least suitable for the activity.

The polynuclear model of Harris and Ullman was the first to describe a fragmentation of urban areas with a high amount of specialization. This was also the first model to take urban sprawl into account, the possibility for a city to grow large suburbs because of better means of transportation. The concept of distance to the CBD is therefore less important in this model.

B.4 Similarities

From the different models of Burgess until the multi-nuclei model, the following similarities can be recognized.

- **Classification** - All theories describe mainly commercial activities, industrial activities, and residential areas. The residential areas are ranked from a low to high SES.
- **CBD** - There is a central role in all theories for one or more business districts, which are the most accessible parts of a city, the most concentrated, and therefore also the most expensive parts. These centers, being a concentration of commercial activity, attract residents to nearby areas because of their shopping and working needs. Burgess still saw the city center as an area of lower SES, but this can be explained by the fact that in the early industrialized cities, especially Chicago, cities were built around the industries. The workers had to live close to the industries, and therefore the CBD was of lower SES [15]. This was changing around the time of Burgess, industries later moved outward because the means of travel for the workers improved.
- **Ecology** - All models indicate movement of different social groups through the cities over time. Burgess already pointed out that the lower SES groups moving to the center were pushing the other groups more and more outward. Hoyt refined this concept and called it filtering, the repeating process where groups of lower SES are attracted by areas of higher SES, after which the groups of higher SES move away. Ullman and Harris define push and pull factors more explicitly, by saying that certain activities attract and other activities repel each other.
- **City features** - Already in the theory of Burgess it was recognized that groups of higher SES generally move more to the outside of a city. This was later confirmed by Hoyt. In the theory of Harris and Ullman industries that only need a lot of space for little money these also move to the edge. Another feature of cities mentioned from Hoyt on, are the main transport axes through the city. In bid-rent theories of later times (eg. [14] transport axes are named as an important pull factor, which thus increase land rent around it.

Bibliography

[1] Michael Batty and Paul Longley, *Fractal cities*, Academic Press, London and San Diego, August 1994.

[2] Nina Brown, *Robert park and ernest burgess: Urban ecology studies, 1925*, http://www.csiss.org/classics/content/26, 2006.

[3] Eric Bruneton, *Modeling and rendering Rama*, (2005).

[4] Jon Callas, *Using and creating cryptographic-quality random numbers*, http://www.merrymeet.com/jon/usingrandom.html, June 1996.

[5] Donald E. Eastlake 3rd, Jeffrey I. Schiller, and Steve Crocker, *RFC 4086*, http://rfc.net/rfc4086.html, June 2005.

[6] Christopher H. Exline, Gary L. Peters, and Robert P. Larkin, *The city, patterns and processes in the urban ecosystem*, Westview Press, Inc., 1982.

[7] Agner Fog, *Website: Pseudo random number generators*, http://www.agner.org/random/, 2006.

[8] David G. Green, *Fractals and scale*, http://parallel.hpc.unsw.edu.au/complex/tutorials/tutorial3.html, 1995.

[9] Stefan Greuter, *Real-time procedural generation of pseudo infinite cities*, (2003).

[10] ———, *Towards a real time procedural world generation framework*, (2003).

[11] Mads Haahr, *Introduction to randomness and random numbers*, http://www.random.org/essay.html, June 1999.

[12] Eduardo Hidalgo, *City Generator website*, http://aig.cs.man.ac.uk/gallery/oldgallery/projects/citygen/index.html, 2000.

[13] John Hoggard, *Fractal geometry*, http://www.math.vt.edu/people/hoggard/FracGeomReport/, May 1997.

[14] Edgar M. Hoover and Frank Giarratani, *An introduction to regional economics*, Regional Research Institute, WVU, http://www.rri.wvu.edu/WebBook/Giarratani/main.htm, 1999.

[15] Homer Hoyt, *Forces of urban centralization and decentralization*, The American Journal of Sociology **46** (1941), no. 6, 843–852.

[16] Makoto Matsumoto and Takuji Nishimura, *Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator*, ACM Transactions on Modeling and Computer Simulations **8** (1998), no. 1, 3–30.

[17] Raymond Edward Pahl, Robert Flynn, and Nick H. Buck, *Structures and processes of urban life*, Longman, London and New York, 1983.

[18] Yoav I. H. Parish and Pascal Müller, *Procedural modeling of cities*, Proceedings of ACM SIGGRAPH 2001 (2001), 301–308.

[19] Robert Park, Ernest W. Burgess, and Roderick D. McKenzie, *The city*, University of Chicago Press, 1925.

[20] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical recipes in c*, Cambridge University Press, 1988.

[21] J-P Rodrigue et al., *The geography of transport systems*, Hofstra University, Department of Economics & Geography, `http://people.hofstra.edu/geotrans`, 2006.

[22] *Virtual Cityscape website*, `http://aig.cs.man.ac.uk/gallery/oldgallery/cityscape.html`, 2000.

List of Figures