Institutionen för systemteknik
Department of Electrical Engineering

Examensarbete

Procedural generation of road networks using L-systems

Examensarbete utfört i Informationskodning
vid Tekniska högskolan vid Linköpings universitet
av

Martin Jormedal

LiTH-ISY-EX--13/4706--SE

Linköping 2013

Department of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings tekniska högskola
Linköpings universitet
581 83 Linköping

Procedural generation of road networks using L-systems

Examensarbete utfört i Informationskodning
vid Tekniska högskolan i Linköping
av

Martin Jormedal

LiTH-ISY-EX--13/4706--SE

Handledare: **Nobuyuki Okamura**
From Software

Examinator: **Ingemar Ragnemalm**
ISY, Linköpings universitet

Linköping, 10 August, 2013

Avdelning, Institution
Division, Department

Division of Information Coding
Department of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping, Sweden

Datum
Date

2013-08-10

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English
☐ _____

Rapporttyp
Report category

☐ Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ _____

ISBN
—

ISRN
LiTH-ISY-EX--13/4706--SE

Serietitel och serienummer **ISSN**
Title of series, numbering —

URL för elektronisk version
http://www.icg.isy.liu.se
http://www.ep.liu.se

Titel
Title

Procedurell generering av vägnätverk med hjälp av L-system

Procedural generation of road networks using L-systems

Författare
Author

Martin Jormedal

Sammanfattning
Abstract

This thesis details the results and conclusions of a project conducted at the game studio From Software in Tokyo, Japan during the autumn of 2008. The aim of the project was the design and implementation of a system able to generate 3D graphical representations of road networks.

Nyckelord
Keywords computer graphics, procedural generation, procedural modeling, l-system

Abstract

This thesis details the results and conclusions of a project conducted at the game studio From Software in Tokyo, Japan during the autumn of 2008. The aim of the project was the design and implementation of a system able to generate 3D graphical representations of road networks.

Sammanfattning

Den här rapporten behandlar resultat och slutsatser från ett projekt utfört på spelstudion From Software i Tokyo, Japan under hösten 2008. Syftet med projektet var design och implementation av ett system för att generera 3D-representationer av vägnätverk.

Acknowledgments

I would like to thank my parents for their support and patience during my studies. Thanks are also due From Software for the opportunity to do this project while getting a chance to experience living in the wonderful city of Tokyo for half a year. A special thanks is directed towards the Sweden Japan Foundation who awarded a stipend towards the completion of this project. Finally I would like to express my sincere gratitude to my examinating professor Ingemar Ragnemalm for not only bearing with me despite dragging my feet with completing this report but also encouraging me to finalise it....

Contents

Chapter 1

Introduction

1.1 A word on procedural generation

This section is intended to give a quick introduction to the concept of procedural generation, if you are already familiar with the topic please feel free to skip forward to section 1.2.

Procedural generation rather obviously implies generating something by means of a procedural algorithm, as opposed to doing the same work by hand. The result can be pretty much anything from music to images to animation data to 3D models.

As the reader is might be aware of at this point, the project described in this paper was conducted at the gaming studio From Software in Tokyo, Japan, and while methods for procedural generation have uses in a range of fields, our focus will remain on it's role in content creation in the field of game design.

There are a few different motivational factors behind choosing the procedural approach but any combination of the following tend to be the main ones:

- Saving on storage space.
- Saving on manual labour.
- The possibility of generating a better result than can generally be achieved by hand.

Saving on space was of great concern during the early days of computer games and methods of procedural generation found some of their earliest practical uses here to generate, for example, levels on the fly from a small set of resources. Saving on manual labour has constantly been the main factor behind the use of procedural generation and continues to be so. The project described in this paper also finds itself squarely in this segment. The possibility of achieving a better result than by doing something by hand has arisen fairly recently as computing power has reached new heights. Concentrating on the subject of procedural generation

of 3D models (a.k.a. procedural modelling) we can conclude this is well exemplified in the highly successful work that has been done in the field of plant and tree generation, much due to the similarities between actual naturally occurring shapes of plants and fractal patterns. Fractal based algorithms are often employed in procedural modelling and such is also the case with the work described in this paper.

Practical use of procedural generation as well as research in the field has experienced quite a renaissance since the mid 2000's, mainly due to the increased complexity of environments and ever increasing demands of realism in computer games and computer animated films. If we limit ourselves to the world of computer games, areas where procedural methods are used successfully, or are making inroads, include texture generation, animation and 3D model generation. When it comes to procedurally generated animation data the line between procedural generation and simulation can get a bit blurred at times as much of the computation is often done in real time.

Relative to the amount of work that has been done on procedural modelling of natural phenomena like plants and terrain, comparatively little work has been put into generating man made structures which tend to have far less stochastic properties making it decidedly more difficult to employ fractal algorithms and still achieve believable results.

1.2 Introducing the problem

As was briefly pointed out in the previous section (1.1), procedurally modelling of man made structures presents a different set of difficulties as opposed to modelling of naturally occurring shapes and patterns. In this paper we will deal with a type of man made structure that combines both properties of a purposefully designed system and some distinct stochastic properties that have been shown to lend themselves well to procedural generation, namely road networks. The purpose of this study was to investigate one possible practical use of procedural generation in a game development setting with the goal of implementing a system that could be used for game map prototyping and/or background scenery. The reasons for choosing the prototyping route instead of aiming for production quality results were, in part the importance of game map layout to the game mechanics in the type of games where the system is applicable, and in part due to the high demands put on graphics quality in modern games versus the limited amount of man hours available for the project. Thus the focus of the project was on developing a sound algorithm that produces believable road networks, accompanied by a flexible system to control the algorithm.

While focusing the project on procedural generation was decided upon from early on, the exact subject matter was decided upon after a series of interviews with

programmers, producers and designers at From Software. While other ideas were also floated and considered there was a degree of consensus that rapid prototyping of urban landscapes for use as, for example, support for level design and backdrops, could prove quite useful in a multitude of scenarios.

1.3 Specifying the task

The vision for the project was initially highly, I dare say overly, ambitious and imagined the system being able to generate entire cities, buildings and all, as well as the ability to nicely render these in order to preview the result. The reality of the limited time allotted and the fact that it was a one-man-project after all, quickly brought some restraint to the scope of the project. A set of more reasonable basic requirements on the project were decided upon:

- Create a system to input easily editable environment data.
- Construct and implement a sound algorithm for generating a road network from the environmental data.
- Output the result in an intermediary format that can be refined using other 3D software.

Some of the earlier ideas that were kept on as nice-to-haves included some sort of building generation and a renderer to quickly view the result of a generation cycle. Due to time constraints these only resulted in some rudimentary work on building generation.

The above stated requirements were however fulfilled to a point that was deemed acceptable.

This project was never awarded an official name while the research was performed but to facilitate referencing to it and avoid confusion I've decided to name it ToshiGen, from the Japanese word for city "toshi" (都市) and "gen" which is a slight play on words, referring both to "GENeration" and "gen" which is a common pronunciation for the Japanese kanji pictogram for origin or source (元).

1.4 Previous work

The generation of road networks by means of procedural generation is by no means a novel idea, there has been a fair bit of research into the problem and an interesting variety of approaches.

The shape of road networks are intrinsically influenced by two strong forces that sometimes act in synergy and sometimes at odds with one and other. One is the surrounding environment, the other is the will of man. These two competing forces are also apparent in the strategies employed when attempting to synthesise

these structures. On one side are the techniques that attempt to purely imitate the geometric shape of interconnected roads in an environment. On the other side are the agent-based methods that seek to emulate human planning behaviour. Hybrids have certainly been attempted and such a method was indeed considered for ToshiGen as well. The possibility of overly increasing level of complexity of the system in relation to the possible actual visual improvement gained ultimately led to such plans being abandoned.

Research into the different approaches also showed that while the agent-based systems such as Lechner et al CityBuilder[1] resulted in interesting simulations they did not necessarily produce more believable visual results.

Other methods have relied heavily on the idea of a cell as a basic component of road networks. Grid based methods [2] and those based on Voronoi diagrams [3] fall under this class. While these systems succeed in capturing some of the regularity found in road networks well, they do tend to produce results that are both overly uniform and rather simplistic in their design when applied to levels in the road hierarchy above a set of city blocks. Another problem in using what is essentially a loop as a basic building block is the lack of dead ends and the singular roads that usually connect areas of higher population density.

Finally we have the fractal pattern method, primarily represented by L-systems, a method originally designed to model the branch and root structures of plants. L-systems have proved to be able to modify in a manner that produces road networks with very believable visual results. As the visual representation rather than correct simulation of an urban environment was the goal of the project the L-system based method was selected for the project which is the subject matter of this paper. L-systems are explained in further detail in chapter 2 on the theory behind this project.
The work that has been the greatest influence on this project is that of Pascal Müller [4] who has made a great effort of analysing the L-system approach and implementing practical solutions based on it.

Chapter 2

Theory

The generation of road networks as described herein is driven by an L-system. L-systems are a family of string rewriting systems and essentially constitute a subset of a field of study called *formal languages*. An introduction to these concepts and how they work is in order to facilitate in describing the road generation system.

2.1 Formal grammars and languages

Formal language theory, the field concerned with the study of formal languages and grammar is related to applied mathematics trough its relation to formal logic. A *formal grammar* is described as a set of *production rules* for rewriting a string and a starting string, called an *axiom*, from which the rewriting begins. As an example let's consider this simple grammar:

Example 2.1: Formal grammar

Production rules:
1. S $\rightarrow$ aSb
2. S $\rightarrow$ ba

Starting string (axiom): S

Step 0:	S	we apply rule 1
Step 1:	aSb	we apply rule 1
Step 2:	aaSbb	we apply rule 1
Step 3:	aaaSbbb	we apply rule 2
Step 4:	aaababbb	

We then apply the production rules to the string one at a time. Applying rule '1' to the starting string renders "aSb". Applying the same rule twice more gives us "aaaSbbb". Finally applying rule '2' renders us the string "aaababbb". From this we can easily deduct that applying the production rules in different order and different number of times we can generate an infinite set of strings from this

grammar. The total set of strings that can be generated from a formal grammar is the *formal language* of that grammar.

2.2 Deterministic and stochastic L-systems

L-systems are generated in the same way a formal language is generated from a formal grammar. The crucial difference is that whereas in a formal grammar only one production rule is chosen and applied per evolution step, in an L-system as many rules as possible are applied in parallel for each evolution step. To put it in different words, there is no conscious selection of one production to be applied over the entire string but all applicable productions are considered for each and every variable in the string and if there are more than one a mechanism needs to be in place to prioritise between them.

This approach has resulted in a plethora of variations of L-systems. The general trend being towards finding ways to apply the most suitable production rule at the right place in the string. As we will see later on in this chapter this has resulted in L-systems moving away from the concept of a "string" to rewrite and towards data structures that are programmatically easier to handle and thus more suited to solving more complex problems.

For the time being we will content ourselves with the idea of an L-system as a string rewriting system as demonstrated in section 2.1 on formal grammar and languages above. L-systems are generally described as a *tuple*:

G = V, w, P

Where 'G' denotes the system, 'V' the vocabulary, or set of *variables* available to the system. 'w' represents the *axiom* which describes the state of the string before the L-system is applied. The axiom is usually a quite simple string but theoretically could be the output from the L-system, or any L-system with the same vocabulary or a subset thereof, as applied to a previous, simpler axiom. Indeed this is what happens at every iteration of an L-system. Finally 'P' is the set of production rules that determine how the system evolves, for those variables in the vocabulary 'V' that do not have any applicable production rules the identity production "S $\rightarrow$ S" is assumed. Such variables are sometimes referred to as *constants* or *terminals*.

If we only have one single production rule to match any one variable in the string we have a *deterministic* L-system on our hands. This follows logically as a system like this will produce the same result every time it is executed.

L-systems, first detailed in 1968 [5] are the brainchild of theoretical biologist Aristid Lindenmayer (hence "L"-system) and was initially used to simulate the growth of simple multicellular organisms, specifically algae and fungi which were

the main focus of Prof. Lindenmayers research at the time.

Deterministic L-systems though, soon proved to be quite restrictive. Plants are rarely, if ever, deterministic in their appearance (two trees rarely look exactly alike) and only having a single production rule for each variable severely limits the patterns one is able to generate. Thus the first natural extension to L-systems was the introduction of support for multiple production rules for any variable and, by necessity, a measure of randomness in order to choose between the patterns.

In order to illustrate this let us take a second look at the example 2.1, this time in the terms of an L-system. The vocabulary V consists of a, b, S where 'a' and 'b' are constants. The production rules in 'P' would be S $\rightarrow$ aSb, S $\rightarrow$ ba where the variable on the left of the arrow is called the *predecessor* and the string or variable to the right is, logically, called the *successor*. The element of randomness that warrants the name stochastic L-systems is constituted by attaching a probability factor to each production rule, like so: S $\rightarrow$ aSb : 0.6, S $\rightarrow$ ba : 0.4. The numbers after the colons mean that the first production rule is applied with a probability of 60% and the second production rule with a probability of 40%. This addition at the same time allows us to have several production rules that are applicable to the same variable in the string, and to adjust the influence of said production rules in relation to one another. Given that one is using a seeded pseudo-random number generator to choose between the production rules, which is normally the case with implementations of L-systems, one has the dual possibilities of generating infinite variations of the system using different seeds and recreating any system given that one has the proper seed to initialise the random number generator.

2.3 The turtle system

Now all of this talk of formal grammars and L-systems is very interesting I'm sure, but just how does it pertain to the generation of computer graphics? The beginning of the answer, and an important link to understanding how L-systems are used and why they have developed the way they have we find in the *turtle system* method of visualising these systems.

The Turtle system gets its name from the idea of a turtle crawling along a path while pulling a pen. Wherever the turtle goes it leaves a line behind it. Now imagine having a very well trained turtle which obeys our every instruction. The connection to visualising L-systems becomes apparent if we let these instructions be the sting of symbols generated by an L-system and assign a motion to each symbol. Thus 'F' could be used to represent moving forward one step '+' could symbolise turning say 45°to the right and '-' turning that same amount to the left.

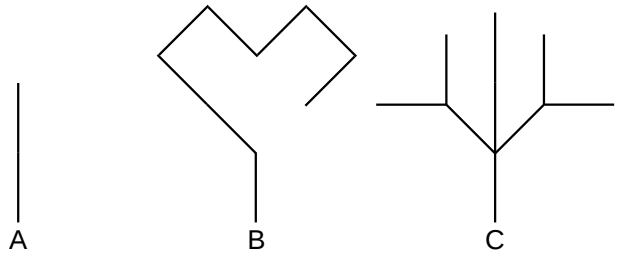
With this set of symbols we could specify L-systems that generate instructions for the turtle system to generate any number of squiggly lines. However, to generate the branching structures we are looking for, both for plants and road networks,

we need to introduce symbols that allow the turtle system to save its position before drawing one branch and then return to the saved position in order to draw the next branch, and so on. Usually this function is implemented as a stack of positions where the '[' symbol in a string denotes a push onto said stack and the ']' symbol a pop and return to the stored position.

Example 2.2: Turtle system

In a turtle system where "F" denotes a movement forward, "+" a positive rotation of 45 degrees and "-" a negative rotation of 45 degrees the figures below correspond to the following strings respectively.

A: FF
B: F+FF–F–F++F–F–F
C: F[+F[+F][-F]][FF][-F[-F][+F]]



2.4 Contextual L-systems

Contextual L-systems provide another mechanism for determining the most suitable rule to apply at any one point in the string. By allowing feedback in the form of the data already stored in the string itself, specifically which symbols are present to the direct left and right of the symbol we are considering, i.e. in which "context" the symbol is located. When typing rules this is symbolised by adding hooked brackets, '<' and '>' to the left and/or right of the target symbol with the context(s) to match against on the pointy end of said brackets.

Example 2.3: Contextual replacement rule

A contexutal replacement rule to replace 'S' with 'bSa' only when 'S' has an 'a' on it's left and a 'b' on it's right would look like so:
a<S>b → bSa

With a randomly selected axiom of: baSba
The first iteration of the system would give us: babSaba
After this point the system cannot be evolved further without adding more rules due to the context constraints specified by the single replacement rule.

2.5 Parametric L-systems

The model for L-systems presented so far is fine for creating nice fractal patterns but as we strive to achieve higher levels of realism a problem soon becomes very apparent; lack of information. All instructions to the turtle system are discrete, '+' always rotates the turtle a pre-set amount of degrees and 'F' always moves it a pre-set unit-length forward. Problems with this emerge as soon as we want to create branches or streets with different length and with non-uniform angles between them. Using our current paradigm we could solve this by increasing the vocabulary, thus creating symbols indicating translations and transformations of different length and angles. This however is not only a cumbersome approach but it still limits us to a predetermined set of lengths and angles.

To alleviate this problem Hanan [6] introduces the concept of parametric L-systems. In parametric L-systems the symbols are replaced with *modules* that besides the symbol also carry a list of parameters that can specify among other things the extent of the action. When fed to a turtle system these parameters also open up the possibility to associate other data with each movement of the turtle, such as the colour of the line being drawn. A module is constructed from a symbol, for example 'F' that we've used to denote a forward motion of a turtle. This symbol is then combined with a parameter list. The meaning of each parameter are up to the implementation of the L-system and turtle system and thus requires there to be an understanding between the two with regards to this. In this example we might have specified the first parameter to denote the length of the movement and the following three the RGB components of the colour of the line to be drawn.

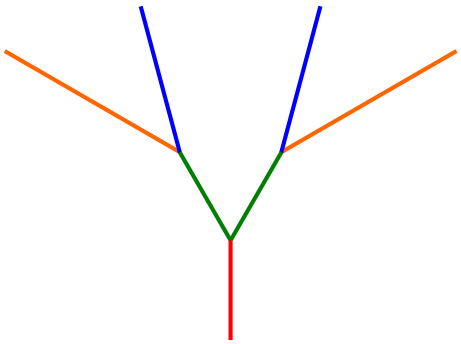
Example 2.4: Module

A module representing forward movement for a turtle system could be constructed like so:
Symbol: F
Parameter list definition: (length, colour)
Resulting module: F(length, colour)

A module representing a change in rotation of the turtle could be constructed like so:
Symbol: R
Parameter list definition: (angle)
Resulting module: R(angle)

Example string constructed with these modules:
F(1.0, "red") [R(30)F(1.0, "green") [R(30)F(2.0, "orange")] [R(-15)F(1.5, "blue")]]
[R(-30)F(1.0, "green") [R(-30)F(2.0, "orange")] [R(15)F(1.5, "blue")]]]

The result of which, if fed to the turtle system would look like the figure below.



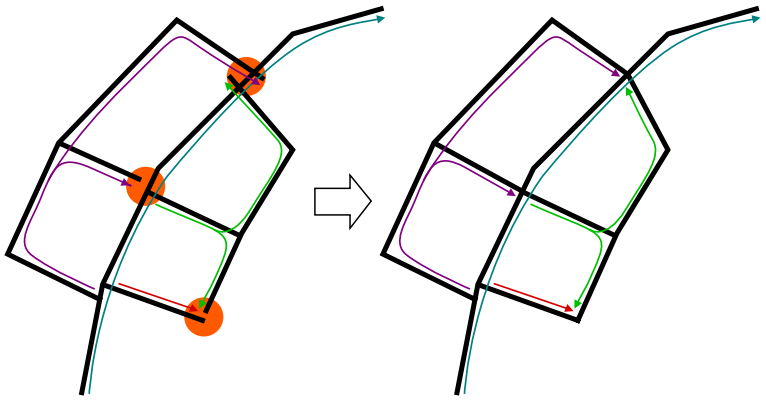
2.6 Allowing loops, an L-system as a graph

Now that we have a basic theoretical understanding of L-systems it's appropriate that we take a closer look at it's applicability for solving the postulated problem and more specifically the theoretical modifications that were necessary to reach an acceptable result.

The big, and rather obvious, problem when adapting an algorithm for generating directed, acyclic structures such as trees, to generate decidedly cyclic structures, such as the road networks we are looking to do are loops. The model of L-systems as described so far with a string as the information carrier from step to step in the evolution process presents quite an obstacle here, since allowing for loops would at best require some convoluted mechanism to reference a literal from multiple other literals in the string. Detecting when loops should be created by branches crossing one another would also require us to create some intermediary logical representation of the structure described by the string and the commands to a hypothetical turtle-system contained therein. We would also need to recreate this intermediary structure between each evolutionary step of the L-system as new data is added. Surely there must be a better way.

Example 2.5: Problems with intersecting paths

The figure below shows a L-system that has started to fold in on itself with some particular problem areas marked out. The coloured lines illustrate the origin and direction of the different paths. With a graph-based data structure we have a method of detecting when paths intersect in place we are able to avoid formations that would seem improbable in an actual road network. It is also a much simpler task to adjust the properties of nodes and edges to remedy the problem areas.



Now, one could argue that detecting these crossing segments is not necessary, that simply allowing the path of the turtle to intersect would suffice in order to generate the visual patterns we're striving for. Alas, this would allow the network of line-segments to fold in on itself removing any sense of control and likely resulting in a far more chaotic appearance than intended. Also, crossings play a central and special part in a road network which further increases the importance of knowing when and where they occur.
The conclusion then, would be that we sorely need some other data structure than a string of literals on which we can apply the concepts of an L-system. Preferably this data structure should incorporate the properties of all the L-system variants described above.
Lucky for us there is a rather simple and elegant solution to our woes. We can achieve what we want by using a graph. At the same time we also need to let the production rules work with the basic building blocks present in this new context. So instead of symbols in a string we now deal with nodes and edges in a graph. Thus a replacement rule may, instead of replacing a symbol 'S' with a sting '[-FS][FS][+FS]', replace an 'S' type node with three edges with an 'S' node at the end of each.

Example 2.6: Identical string based and graph based systems

Let us construct a simple L-system with the intention of feeding the output to a turtle-system. For sake of simplicity let the symbols "+" and "-" imply a positive of negative rotation of 45 degrees and the symbol F imply a forward movement of length 1.0 in the context of the turtle-system. We will let this system iterate 5 times over the string before taking a look at the result.

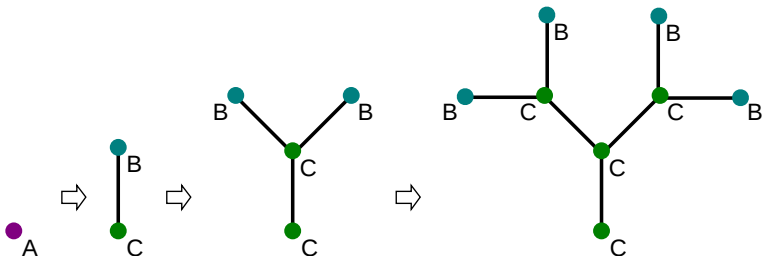
Rule 1: A $\rightarrow$ FB
Rule 2: B $\rightarrow$ [+A][-A]
Axiom: A

Step 0	A
Step 1	FB
Step 2	F[+A][-A]
Step 3	F[+FB][-FB]
Step 4	F[+F[+A][-A]][-F[+A][-A]]
Step 5	F[+F[+FB][-FB]][-F[+FB][-FB]]

"B" means nothing to the turtle system so for all intents and purposes it sees the above string as equivalent to:
F[+F[+F][-F]][-F[+F][-F]]

We will attempt to create the same visual result using a graph based L-system using the following production rules:

Rule 1:	Match node type:	A	Rule 2:	Match node type:	B
	Change to type:	C		Change to type:	C
	Spawn type:	B		Spawn type:	B
	Spawn count:	1		Spawn count:	2
	Edge length:	1.0		Edge length:	1.0
				Edge angle:	45



As we can see the two production rules from the two systems closely resemble each other and they both produce the same graphical result.

This has the added benefit that we are essentially incorporating the instructions meant for the turtle system into the data structure itself in the form of edges. We are thus evolving a two (or three if we so wish) dimensional graphical representation of the system directly instead of a set of a one dimensional string of instructions on how to draw it. Thereby our problem of creating loops is solved, as it is easy enough to add or detract edges and nodes from one another. The problem with detecting branch overlaps, or near overlaps, in order to create loops also presents a much smaller challenge due to the current state of the system (i.e locations of nodes and edges) being instantly available to us at each evolution step. While there are distinct advantages to this approach it also requires us to expand a bit on the concept of parametric L-systems when it comes to production rules as they now create the edges between nodes and thus need to carry information about how to go about doing that. We may also need additional node types/variables in

order to mark nodes as already expanded and avoid it getting matched multiple times. As can be seen in example 2.6 this results in a slightly different approach than with string replacement.

Chapter 3

Implementation

The main focus of the project was to produce believable output, thus the most effort was put towards implementing a good generation algorithm while usability and polished UI was considered to be "nice to have" rather than a "must have". Thus the system in it's current form is implemented as a black box solution with input in the form of an XML-file with instructions for the generation algorithm and a set of image maps that can be used to input information such as a height field and population density. Output is rendered in the form of an output file that contains a simple mesh representing the generated road network, and if so indicated, a terrain-mesh generated from a height-field. The output file is in an XML-based format called Collada (more on Collada in section 3.4).

3.1 System overview

As seen in figure 3.1 the system in itself is made up of three rather elementary units, input, generation and output. Each of these units has been awarded it's own chapter that further describe the methods and technologies used. The input data set and Collada format output is described in conjunction with the input and output units respectively. Focus is quite naturally on the generation step as that is where most of the magic happens.

3.2 Input and XML module

Input to the generation system is provided in two forms, an XML-file and an arbitrary number of image maps. These are combined into an initial environmental model that acts as an resource for the L-system to gather information from and modify as needed.

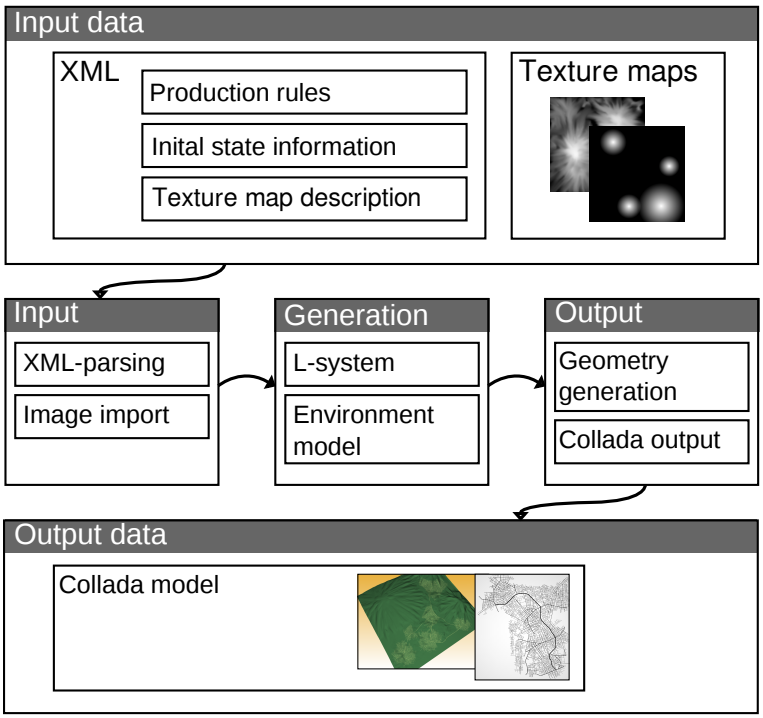


Figure 3.1. Overview of the system

3.2.1 XML input

XML was chosen as the input format mainly due to two properties, it's widely used and quite verbose. These properties combine to make for a relatively easily human readable text-based input that despite the lack of a polished GUI allows for transparent control of the system. It is also easily imported and exported which allows it to act as an intermediate format between the generation system and a GUI were one be added in the future.
The XML file contains some general information about the target environment and also describe the production rules (production rules are explained in section 2.1) that are to be used by the L-system.
The XML reader module was designed around an Open Source library called TinyXML.

3.2.2 Image map input

Image maps gives us a simple and effective way of providing densities and boundary information related to the environment onto which the road network is to be generated.
The role of the image maps are to provide extra data to be used by the production

rules such as topology and population density data. These can optionally be used by the production rules to, for example, avoid steep inclines or prefer to branch off side roads within areas with higher population density.
Access to the image maps in the system is uniform and technically an arbitrary number of image maps can be defined for inclusion. The inclusion of an image map is specified in the XML configuration file, use of these maps however needs to be implemented programmatically which means that in reality is quite nontrivial to increase the amount of image maps actually used by the system. The rules implemented at the end of the the project described herein make use of at most a height map for topology and a map for population density.

3.2.3 The initial environment model

The environmental model is an encapsulation of the graph representing the road network, the available node types, the production rules for the L-system and additional data that support the operation of the L-system at large. Such additional data includes the seed for the random number generator, the total size of the domain that the network is allowed to grow within and the number of evolution steps, i.e. iterations, to execute. An example of how this information is encoded in the XML input file can be seen in example 3.1.

Example 3.1: Environment XML tags

The initial environment information is provided with the XML input file as shown below.

```
<RandomSeed value="69584251" />
<DomainSizeX value="100"/>
<DomainSizeZ value="100"/>
<EvolveStepNum value="100"/>

<Map2DDef>
    <Semantics value="Terrain"/>
    <Path value="../../Resource/heightmap-51x51.tga"/>
    <Scale value="0.02"/>
</Map2DDef>

<Map2DDef>
    <Semantics value="Population"/>
    <Path value="../../Resource/popmap-512x512.tga"/>
    <Scale value="100"/>
</Map2DDef>

<AxiomNode>
    <Type value="MainRoadLeaf"/>
    <Pos valueX="30.1" valueY="20.1"/>
    <Width value="0.05"/>
</AxiomNode>
```

3.3 Generation, the graph-based L-system

3.3.1 Initial challenges

The first attempts at implementing an L-system to drive the generation process resulted in a system highly based around the idea of the classic representation of a string of instructions to a turtle system. The string was implemented as a double-linked list of literals that were replaced according to the production rules of the system. As indicated in section 2.6 this representation presents some great difficulties when it comes to intersecting segments and the crossings they create. This was a hard learned lesson that cost a fair deal of time and rewriting of code.

3.3.2 The graph based L-system

Some solution was needed to solve the problems described in the previous section (3.3.1). As described in the theory section (2.6) the solution was to be found in representing the state of the L-system after each evolution step as a graph. The literals were replaced with nodes of different types and just as with literals the production rules were constructed to replace a node for another or a set of others. The need for a turtle system to generate a visual representation of the resulting structure was eliminated by also storing the edges between connected nodes as they were created.

The nodes are also a very practical place to store the hierarchy of the network. Thus the type label of each node is used to designate it's position in said hierarchy (main road, street, alley, etc.). These type labels are defined dynamically in the XML environment specification as illustrated in example 3.2, technically allowing for an arbitrary hierarchy depth. This also provided the possibility to fulfil another requested feature. This feature was the generation of "rail" structures. The initial source of this request was that much of Tokyos urban planning and transportation, to a degree not commonly seen in other cities, is centred around the train systems and there was a wish to be able to represent structures intended for different modes of transport than automobiles. The dynamic node type labelling feature allows for the classification of nodes as rail with specific rules to grow a rail network, the possibility to create dedicated footpaths or bicycle routes that all can be marked as such and treated separately post generation is provided analogy.

In the nomenclature used in example 3.2 "leaf" is used to denote nodes at the fringe of the network. These are thus freshly generated and are only connected to one edge. As they are matched against a production rule that targets their specific type label they will be changed into a "knot" if the production rule is successful in generating one or more nodes and their connecting edges. A "knot" thus signifies

any node in the network with two or more edges attached.

Example 3.2: Example of node types declared in XML

```
<RoadNodeType name="MainRoadLeaf" id="100" />
<RoadNodeType name="MainRoadKnot0" id="101" />
<RoadNodeType name="MainRoadKnot1" id="102" />
<RoadNodeType name="MainRoadKnot_Branched" id="103"/>
<RoadNodeType name="StreetLeaf" id="200"/>
<RoadNodeType name="StreetLeaf_JustGrow" id="201"/>
<RoadNodeType name="StreetKnot" id="202"/>
<RoadNodeType name="StreetKnot_Branched" id="203"/>
<RoadNodeType name="AlleyLeaf" id="300"/>
<RoadNodeType name="AlleyKnot" id="301"/>
<RoadNodeType name="AlleyKnot_Branched" id="302"/>
```

3.3.3 Growing the network, the production rule model

Each production rule in ToshiGen is made up from two distinct parts, a piece of XML data that specifies the node to target and parameters to use, perhaps more importantly though, through the "name" attribute of the "Production" XML-element it also points to the particular C++ class that contains the logic that uses the other parameters to apply a certain operation on applicable nodes of the network.
Example 3.3 shows an example of a production rule as defined in the XML input file for the system. It points to the production class "BranchAtAngle" and contains the parameters to be used when the logic contained in that class is applied to nodes of type "MainRoadLeaf", such as which type of new nodes to spawn, the minimum and maximum length of the edge connecting the spawned node to the parent node, parameters concerning the angle of the spawned edges and the width to be used when generating geometry for the spawned edges.

Production rules go through a few steps when applied to the graph, as first touched upon in section 2.6:

1. Find a node of matching type.
2. Check if the conditions to apply the production rule for the matched node are fulfilled.
3. If the check above is positive, apply the rule and spawn edges and child nodes according to the parameters provided.

4. Check that the criteria for the spawned nodes are fulfilled.

5. If the check above is positive, add the spawned nodes and edges to the network.

6. Change the type of the originally matched node according to the parameters provided.

Example 3.3: Production rule in XML

This example details the structure of a production rule and it's most common parameters as defined in the XML input file.

```
<Production name="BranchAtAngle">
    <MatchRoadType value="MainRoadLeaf"/>
    <ChangeToType value="MainRoadKnot_Branched"/>
    <SpawnType value="MainRoadLeaf"/>
    <MaxLength value="1"/>
    <MinLength value="0.5"/>
    <Width value="0.05"/>
    <IntersectRadius value="0.1"/>
    <MaxAngle value="50"/>
    <MinAngle value="10"/>
    <MinInbetweenAngle value="45"/>
</Production>
```

Step 4 above deserves some further attention. The criteria mentioned here include checks angles against other spawned nodes and edges, as well as against the orientation of the parent node. In case the tests fail here there is generally an attempt to modify the angle of one or both of the edges and their corresponding nodes to fall within the parameters.
There is also a check if the spawned node is out of the bounds specified for the whole network. If a spawned node were to fall outside of the bounds it is generally rejected and the node and edge will not be added to the network.
Finally there is a check for intersections with other edges and proximity to other nodes in the network. This check generally goes through the steps listed below.

1. Check for intersections with other edges.

2. Check for other nodes within the "IntersectRadius" specified in the production rule parameters.

3. Check for other edges within the "IntersectRadius" specified in the production rule parameters.

Should any of these tests return a positive result there are corresponding procedures to rearrange the spawned nodes and edges to maintain the integrity of the network as well as to strive for a more believable result by creating crossings where applicable.

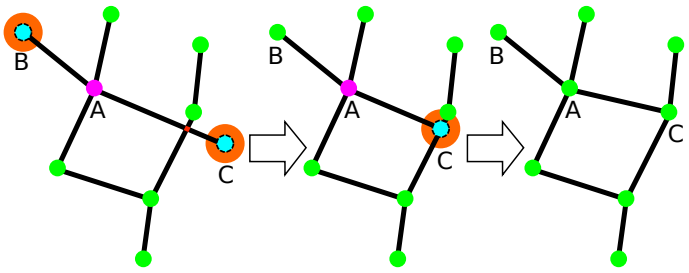


Figure 3.2. Check for edge intersections

Figure 3.2 illustrates the check for intersections between a newly created edge and any pre-existing edges. This check takes precedent and should one or more such intersections be found the newly spawned node will be moved to the point of intersection closest to the parent node and the spawned edge shortened correspondingly. Thereafter the check for nearby nodes detailed below and illustrated in both 3.2 and 3.3 is performed. Should it yield a result the spawned node will be discarded and the spawned edge will instead be connected to the closest found node. Should no nearby nodes be found the same procedure as for nearby edges, illustrated in 3.4, will be executed and the existing edge with which an intersection was found will be split into two edges and the spawned node will be inserted between them and connected to both as well as the spawned edge.

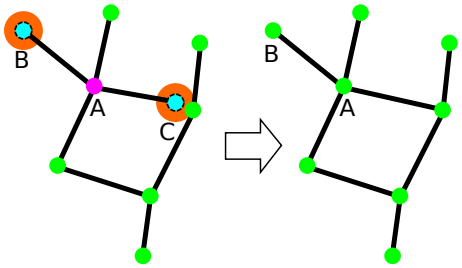


Figure 3.3. Check for nearby nodes

Every production rule contains a parameter called "IntersectRadius" which specifys a circular area around any newly spawned node in which to check for

existing nodes or edges. As detailed above in the paragraph regarding edge intersections, should any nodes be found the newly spawned node will be discarded and the spawned edge will instead connect the parent node with the existing node that the check found.

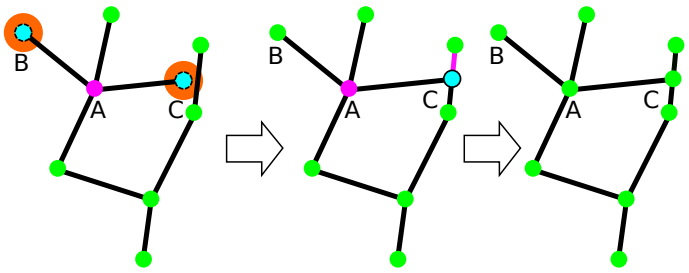


Figure 3.4. Check for nearby edges

Should no nodes but an existing edge be found within the intersect radius described above, the spawned edge will be extended or shortened so that the spawned node is on the closest found existing edge which then be split into to edges and the node will be inserted between them and connected to both as well as the spawned edge. This procedure is illustrated in 3.4.

3.4 Output and Collada module

The system in in it's current form exports the road network as a set of line segments each represented by a simple rectangular polygons. These are created by passing the network created by the L-system trough a geometry generation step. Optionally one may also pass along the height map used and geometry representing it will also be created.
The output data is thereafter exported as a Collada file. Collada is an XML-based file format originally created by Sony Computer Entertainment to facilitate importing and exporting data between different software that handle 3D models. Support for it is fairly widespread. Both these properties marked it as a good choice as output format considering the projects purpose as a prototyping tool.
Collada is currently under the maintenance of the Khronos Group who also handle maintenance and standardisation of OpenGL. There exist a few libraries for aiding in interaction with Collada files but for this project the official Collada DOM (DOcument Object Model) that is also maintained by the Khronos Group was chosen, partly due to its liberal licensing and partly because of high likelihood of continued development and compliance with future versions of Collada.

The output form ToshiGen was never intended to be production quality level models but rather a sketch to be refined. The system is however designed to allow for relatively easy extension. The representation of the line segments can be changed to any model of arbitrary complexity by modifying or exchanging the geometry generation step. To allow automatic blending for any complex segment models would however require a fair bit of additional work. Some road generation software such as the work of Yamauchi [7] go to great lengths to achieve automatic blending of road segments and generation of crossings and the like, including pavements and road markings. It was not considered a priority for this project due to the refinement work that was expected to occur on any generated result that went on to production among other things.

Chapter 4

Results

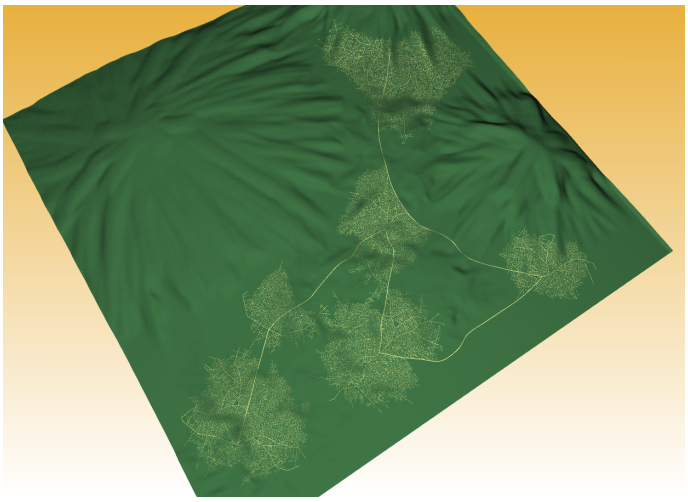


Figure 4.1. Example of a bigger network with separate population centers

As touched upon in the introduction of chapter 3 the main focus of the project was to produce output that subjectively would be viewed as a believable road network. In that respect I would suggest the project was a success, and in this chapter I will present some examples in an effort to persuade the reader to agree. In the very beginning of chapter 1.1 there were three reasons listed as motivating factors to leverage procedural methods to solve a problem:

- Saving on storage space.
- Saving on manual labour.
- The possibility of generating a better result than can generally be achieved by hand.

Saving on storage space was never a driving factor behind the project but the other were indeed relevant. In both those cases I would like to proclaim at least moderate success. The system has a very demanding user interface which diminishes the savings in manual labour, more on that as well as thoughts on what can, or at least, could have been done about that in chapter 5. As for creating better results than can generally be achieved by hand, it's a very subjective matter and also depends on whose hand we're referring to. For the general person the system will will in all probability generate a superior result, and even for the artisticly inclined it will do so in less time (provided you are comfortable with manual XML editing, again see chapter 5) and with greater consistency.

Besides the points above, performance is an important factor when considering the usefulness of the system. Indeed one of the major motivations behind the project was to quickly iterate prototypes of urban landscapes from which the most promising ones could be selected and refined.
The general nature of the environment we are trying to generate, and in particular the growth strategy inherent in the L-system gives us a structure that increases in complexity exponentially as leaf nodes are expanded into several new leaf nodes. This results in every pass taking longer time than the last until some resource runs out. The resource in general tends to be space, either of the entire designated environment or of the populated area if using a population map.
The result is that the cost in time and thus performance increases quickly with the sought after size of the network. Performance is quite good for smaller networks but generating massive networks can take hours depending on your hardware. Some thoughts on improvements in relation to performance can be found in chapter 5.

4.1 Examples

Figure 4.2. The growing of a network visualised

The networks displayed in 4.1 show the same network in three different stages,

the generation times were measured on a Core i7@1.9GHz:

- After 20 iterations, generation time: 11s
- After 40 iterations, generation time: 35s
- After 60 iterations, generation time: 46s

The figure 4.3 shows the population density map used. We can see that the increase in generation time is largest between the first two versions of the network, thereafter the lack of populated area reduces the amount of growth possible.

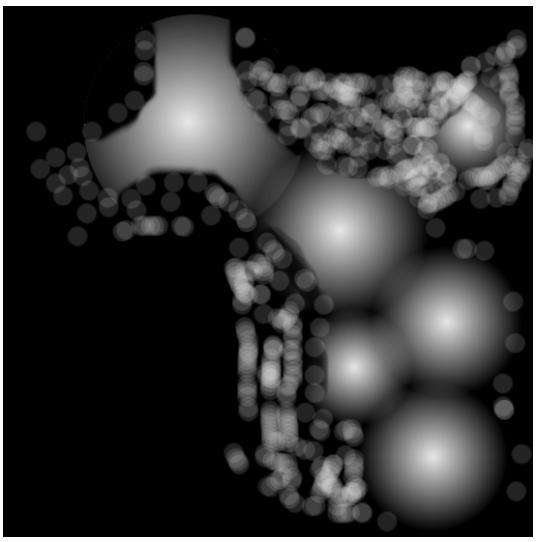


Figure 4.3. Population map

Figure 4.4. Small road network generated by the system. This network was generated in approximately 60 seconds on a machine equipped with an Intel Pentium 4 CPU running at 3GHz

Chapter 5

Discussion

Any piece of software is inherently a product of decisions made during it's development, both functional and non-functional, good and bad. In this section we will take a closer look at some of these decisions and the effects they had on the end result of the project. In the light of this we will also discuss what could have been done better and how the system might be improved if it were be continued.

5.1 Design choices and limitations

A project like this is doomed to have it's scope limited by the available resources, and certain problems were simply too complex to overcome within the scope of the project. As always when designing software both good and less good choices were also made in the process. I would like to dedicate this section to reflection upon some of these problems and choices with a focus on the limitations they impose and what might have been done better.

5.1.1 Usability

One big, or rather *the* big challenge with procedural generation systems is achieving balance between complexity and control. As the systems one tries to generate increase in complexity it inevitably becomes more difficult to leverage control over them in an intuitive way. By way of extension this causes problems with usability for any tool based around these methods.
This became quite apparent during the course of this project and it is also where I feel that ToshiGen falters the most. While the algorithm is sound and the structure as a whole is flexible and fairly easily extendible, to leverage control of this flexible system is quite a challenge. It can be a time consuming task of interleaving XML-file edits and test generations to create good results. This regrettably has quite a negative impact on the usefulness of the software as a whole. I've chosen to place some thoughts on how this may be alleviated under section 5.3 "Future work" later in this chapter.

5.1.2 Intermediate output of network topology without geometry generation

The generation pipeline as described in 3.1 is fixed, That is to say there is no intermediate output of data, besides some basic logging, between reading of the input data and output of the road network geometry. This all well and good in the general case but as has been discussed elsewhere in this text (3.4) one of the goals of the design was to have the geometry generation module easily exchangeable. Having an intermediary step with the ability to output the layout of the road network would allow for inputting the same layout into different geometry generators to find the one that produced the best result. It would also allow for the possibility to give the generated network a preliminary check before passing it through more time consuming geometry generation than the current, quite rudimentary, one.

5.1.3 Separation of network types

The system does not provide for a means to disallow different network types to interconnect. The most notable effect of this is that the rail network and road network will inevitably grow together become difficult to discern. In some cases this might be the desired effect in order to create a crossing between the to two. Mode often than not though, especially in highly populated areas, the two traffic types are kept separate and it would make sense to for example let rail block roads, which will then have to run alongside the rail until some designated point where a crossing or underpass can be generated. In order to implement this the edges will need to carry more information than they currently do. These modifications are detailed in section 5.2.2 as suggested future work.

5.1.4 Improved handling of terrain obstacles

As suggested in section 5.1.3 above, letting some features like parts of the network ferrying different modes of transport have an impact on how the road network is grown would yield more believable results. In the same manner this applies to certain terrain features as well. Most notably bodies of water and very steep terrain. While the system has some ability to avoid overly abrupt inclines there is currently no representation for rivers or lakes which makes avoiding them rather difficult.
It would however be a quite simple task to leverage the mechanism for image maps already present, as well as some gentle modification of the production rules in order to to achieve the desired effect. One could create a map with areas corresponding to the obstacles we wish to avoid when growing the network and let the production rules check against this. It would likewise be a simple task to avoid deep valleys and mountains by checking against the terrain topology map.

Unfortunately the improvements suggested above can't be described solely in XML so some modification of the actual C++ code would be needed. This incidentally brings us on to the subject matter of the next section (5.1.5).

5.1.5 Improved XML format

The the handling of the XML file describing the initial environment and the parameters for the production rules as well as the structure used in the file itself would benefit greatly from some improvement. The structure of the file, as it is, is very flat and has a lot of focus on the description of the production rule parameters.
As an example the hypothetical solutions to both the suggested improvements described in section 5.1.3 and section 5.1.4 would benefit significantly from more, and better presented, information about the road network edge segments and image maps respectively.

Road edge segments would be easy to discern from each other by adding types and descriptions for the them in a similar manner done for the nodes. This would also simplify things for any geometry generator that created more complex structures as it would allow it to know the exact type and purpose of any road edge segment.

A more verbose definition of image maps would make their use more flexible and allow them do be defined dynamically in XML alone. As things stand a lot of the code pertaining to the access and function of image maps is hard coded in the production rules. Expanding on the effects the image maps may have on the system and encoding these parameters along with the map definition in the environment XML file would eliminate the need of a great deal of that hard coded behaviour. The added parameters could detail for example whether to repel growth in from some areas and/or encouraging it in others, define which node types are affected and which production rules should take a specific map into account.

5.2 Future work

Many of the limitations described in section 5.1 would be good candidates for any extension of the system, but I've selected three specific tasks to feature as especially fitting for a (hypothetical) continuation of the project. The thinking behind the choices is that the tasks should represent well defined projects of a larger scope. All of these features were also discussed in some way or form during the original project.

5.2.1 GUI

As discussed in section 5.1.1 one of the foremost obstacles to using the system is in the general usability. Manually editing the XML parameters for the production rules requires both some script language knowledge and a rather good understanding of how the system is implemented and how L-systems work in general.
To alleviate this a GUI would be most welcome. It could be used both to con-

trol the parameters, preview the generated content, as well as to make manual adjustments after the generation step.

5.2.2 Detection and subdivision of sectors

This task is linked to and a direct prerequisite to the building generation described in section 5.2.3 below, it would however be quite possible to take on as a separate task and to achieve good results might be a challenge in and of itself.
Quite naturally when a network is generated we also wind up with open spaces in between the roads, in order to generate buildings, parks, town squares and other features of an urban landscape these spaces need to be detected and their dimensions registered. To accommodate buildings they also need to be subdivided into plots of suitable geometry.

5.2.3 Building generation

As mentioned in section 5.2.2 above, in order to mesh the generation of buildings with the networks generated by ToshiGen one would first have to tackle the problem of detection and subdividing the open areas between the roads in the network. However, the generation of buildings is in and of itself a task of substantial complexity. Depending on the level of accuracy the effort could easily surpass the scope of the project described herein. It is an interesting topic and much work has already been done, alas there is still much room for improvement. It would also be a natural progression in the case of ToshiGen towards extending it to generate urban environments. Indeed there is some small pockets of code dedicated towards this goal floating around in the system that unfortunately had to be abandoned due to the scale of taking on the task with good results.

5.2.4 Parallelisation of the generation algorithm

As the network grows the amount of nodes to match with production rules and check for expansion increase exponentially which of course has a very adverse effect on performance. The L-system as it stands is also completely single-threaded. This is of course rather silly for a process that might lend itself quite well to parallelisation to be handled this way in this age of multi-core CPUs and GPGPU. In order to achieve a substantial speed-up it would be interesting to investigate how to handle this parallelisation. The obvious way to start would be to divide the area into subsections, this however may cause some difficulties when edges grow into another section and may need to interact with nodes and edges in said section at the same time as another thread is processing and modifying the topology there. Finding some nice way to pass these cases between threads might be a solution but one could also imagine some kind of dispatcher that ensures that all threads are working as far apart as possible at all times in order to avoid problems with simultaneous attempts to modify a node and/or edge.

5.3 Conclusion

While a fair few shortcomings of the system have been detailed in this particular chapter the result of the project was none the less a very flexible system that fulfilled the basic demands placed on it, and in some respects did even more. The modular nature of the pipeline allows for easy extension and the general approach taken in the L-system implementation would easily allow the system to be modified to generate other structures than road networks that share similar properties such as rivers, and given the systems ability to modify texture maps rudimentary land topology, such as valleys and gorges should not be outside the scope of possibility either.

The project was intended to be an investigative one and by all accounts the results were satisfactory to the client in question. It's failures served as lessons and it's successes as guidance to the people involved, myself in particular.

Bibliography

[1] Thomas Lechner, Ben Watson, Uri Wilensky, Martin Felsen, *Procedural City Modeling*. Northwestern University, Illinois Institute of Technology 2003.

[2] Stefan Greuter, Jeremy Parker, Nigel Stewart, Geoff Leach, *Undiscovered Worlds – Towards a Framework for Real-Time Procedural World Generation.*. Royal Melbourne Institute of Technology 2003.

[3] Jing Sun, George Baciu, Xiaobo Yu, Mark Green, *Template-Based Generation of Road Networks for Virtual City Modeling.*. Hong Kong Polytechnic University 2002.

[4] Yoav I H Parish, Pascal Müller, *Procedural Modeling of Cities*. SIGGRAPH 2001.

[5] Aristid Lindenmayer, *Mathematical models for cellular interaction in development.*. Journal of Theoretical Biology 1968.

[6] James Hanan, *Parametric L-Systems and their application to the modelling and visualization of plants*. University of Regina 1992.

[7] Daisuke Yamauchi (山内大介), *Procedural Generation of Road Networks in a Dynamically Changing Virtual City (*時間変化する仮想都市における 道路網の自動生成*)*. Tsukuba University 2004.

Appendix A

Appendix A: List of external libraries used

SOIL
TinyXML
ColladaDOM

Upphovsrätt

Copyright