

# **An Interactive System for Procedural City Generation**

**George Kelly  
M.Sc. in Computing**

**Institute of Technology  
Blanchardstown**

**2008**

**Supervisor: Hugh McCabe**

# **Declaration**

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Masters in Computer Science in the Institute of Technology Blanchardstown, is entirely my own work except where otherwise stated, and has not been submitted for assessment for an academic purpose at this or any other academic institution other than in partial fulfilment of the requirements of that stated above.

**Signed:** \_\_\_\_\_

**Dated:** \_\_\_\_/\_\_\_\_/\_\_\_\_

# **Abstract**

The rapidly growing computer game industry requires a highly skilled workforce and this combined with the complexity of modern games, means that production costs are extremely high. One of the most time-consuming aspects is the creation of game geometry, the virtual world which the players inhabit. We have developed techniques to automatically generate such geometry, thus removing the need for developers to construct it manually.

In this thesis a city generation system is presented that employs procedural techniques to rapidly create the urban geometry typical of a modern city. The approach taken is unique in that users are provided with an interactive interface to control the generation process. The system enables the generation of the underlying road networks which form the structure of cities and urban neighbourhoods. These road networks are automatically mapped to any terrain model, and adapt themselves to the specific geometry of the underlying terrain. The regions enclosed by roads are automatically extracted from the resulting road graph and building lots are determined using a subdivision process. The buildings are placed within the boundary of selected lots and basic geometric shapes are generated with advanced materials containing shaders to simulate additional geometry. Tactile control is provided by allowing the user to directly manipulate high level elements such as road intersection nodes and to control the many other aspects of city generation via intuitive property inspectors. As users alter the model the results are updated in real time, thus facilitating an interactive design process. The system can be used to pre-generate geometry in advance or to enable dynamic game environments where world geometry can be generated on demand.

# Contents

1 Introduction.....	1
1.1 Background.....	1
1.2 Procedural Generation.....	3
1.3 City Generation.....	4
1.4 Aims and Objectives.....	5
1.5 Achievements.....	6
1.6 Thesis Outline.....	6
2 Procedural Techniques Research.....	8
2.1 Key Properties.....	8
2.2 Fractals.....	10
2.3 L-Systems.....	11
2.4 Perlin Noise.....	13
2.5 Tiling.....	17
3 City Generation Research.....	19
3.1 The Structure of a City .....	19
3.1.1 Primary Transit Network.....	20
3.1.2 Neighbourhoods / Districts.....	21
3.2 The Evaluation of City Generation Systems.....	23
3.2.1 Critical Framework.....	23
3.3 Grid Layout & Geometric Primitives.....	24
3.3.1 Buildings: Geometric Primitives.....	24
3.3.2 Real-time Optimisations.....	25
3.3.3 Discussion.....	26
3.4 L-systems.....	27
3.4.1 Road Network: L-systems.....	27
3.4.2 Buildings: L-systems.....	29
3.4.3 Discussion.....	32
3.5 Agent Based Simulation.....	33

3.5.1 Road Network: Agent Based Simulation.....	33
3.5.2 Buildings: Agent Based Simulation.....	35
3.5.3 Discussion.....	36
3.6 Template Based Generation.....	37
3.6.1 Road Network: Template Based Generation.....	37
3.6.2 Discussion.....	39
3.7 Split Grammars.....	40
3.7.1 Buildings: Split Grammars.....	40
3.7.2 Discussion.....	42
3.8 Conclusions.....	43
<b>4 Interactive City Generation Design.....</b>	<b>44</b>
4.1 Overview.....	44
4.2 Primary Road Network.....	46
4.2.1 Adaptive Roads.....	47
4.2.2 Sampling.....	48
4.2.3 Sample Selection Strategies.....	49
4.3 Secondary Road Generation.....	50
4.3.1 City Cells.....	51
4.3.2 Secondary Road Growth.....	52
4.3.3 Snap Algorithm.....	55
4.3.4 Summary of Secondary Road Generation .....	61
4.4 Building Generation.....	61
4.4.1 Blocks.....	62
4.4.2 Lot Subdivision.....	63
4.4.3 Building Construction .....	70
4.5 Summary.....	74
<b>5 Citygen Implementation .....</b>	<b>75</b>
5.1 An Introduction to the Citygen UI.....	75
5.2 View Edit.....	78
5.3 Node Edit.....	79
5.3.1 Add Node - Chain Tool.....	80
5.3.2 Validity Checking.....	80

5.4 Road Edit.....	82
5.4.1 Adaptive Road Control Properties.....	83
5.5 Cell Edit.....	84
5.5.1 Cell Generation Properties.....	84
5.6 Building Tiles.....	87
5.7 Integrated Game Engine: OGRE.....	88
5.8 Graph Structures: BGL (Boost Graph Library).....	89
5.9 GUI Platform: wxWidgets.....	90
5.10 Accessible Export: COLLADA.....	91
5.11 Summary.....	91
<b>6 City Generation Results.....</b>	<b>92</b>
6.1 Primary Road Network.....	92
6.2 Adaptive Primary Roads.....	95
6.3 Secondary Road Growth.....	97
6.4 Building Generation.....	99
6.5 Performance.....	102
6.6 Development Integration.....	103
6.7 Analysis.....	104
6.8 Final Output.....	106
<b>7 Conclusions.....</b>	<b>108</b>
7.1 Summary.....	108
7.2 Achievements.....	109
7.3 Project Assessment.....	111
7.4 Future Work.....	112
7.5 Conclusions.....	114

# Related Publications

KELLY G. AND McCABE H. 2007. Citygen: An Interactive System for Procedural City Generation. In *GDTW '07: Fifth International Conference on Game Design, Liverpool*, ACM. Pages 8–16.

KELLY G. AND McCABE H. 2007. Interactive City Generation Methods. Poster presentation at *SIGGRAPH '07, San Diego*, ACM.

KELLY G. AND McCABE H. 2006. Interactive Generation of Cities for Real-time Applications. Poster presentation at *SIGGRAPH '06, Boston*, ACM.

# Chapter 1

## Introduction

The focus of our research is on the generation of city models via the application of procedural techniques. By using a sequence of computer instructions we aspire to automatically generate all of the geometry, materials and textures that constitute a 3D city model. The motivation of our research is to achieve the efficient construction of realistic, detailed and large scale urban environments and help solve the content creation problems facing the graphics industry. In addition we aim to enable new features like dynamic game environments, compact distribution models and accelerated rendering for large city scenes.

### 1.1 Background

Advances in the field of computer graphics are consistent and we are now in an era where real-time photo-realistic rendering is common place and graphics processors exceed the complexity of CPUs in terms of transistor count by a ratio of up to 4:1 [Nvidia 2007][AMD 2007]. The evolution of computer graphics has dramatically increased the computing power available to developers and it has enabled the display of more realistic, detailed and large scale 3D worlds than ever before. However, displaying these worlds on screen is only part of the challenge. To provide this level of graphic detail in a final product, the detailed content—including the geometry, materials and textures that make up the 3D worlds—must be created by a fleet of artists. Content is traditionally defined as static assets and requires manual construction. The authoring of such detailed and large scale content is both time consuming and expensive. Even with the latest advancements in graphics hardware, the industry finds itself in a position where the new level of visual fidelity can only be achieved with massive financial resources to fund a new level of content creation.

The prohibitive cost of content creation results in the graphics industry, including games, films, advertising and television, struggling to meet the consumers' expectations, as set by the largest and most expensive titles. For those studios who can afford to, increasing the number of artists working on a project is a simple method that can be used to create more content. However, the effectiveness of this method is limited by the artistic pipeline not scaling, additional artist numbers do not necessarily generate a proportional yield of content. These inefficiencies add to the already high development costs of computer graphics. The result is an increased barrier of entry for new development firms, thus stifling innovation in the industry.

One potential solution to the content creation problem is the application of procedural techniques [IDV Inc. 2006][Wright 2005]. Traditional approaches to content creation rely on the use of static assets to define the world. These static assets are largely inflexible, not easily modified and their reuse is limited. Procedural techniques define assets using a set of computer instructions. The geometry, textures or behaviours of the asset are then generated automatically using these instructions. Furthermore by parametrizing the generation functions a wide range of output can be created. This property enables procedural assets to be far more flexible than static assets, offering much greater re-usability and range of output.

Additional benefits can be provided by the application of procedural techniques, with some particularly novel applications in gaming. By encoding the behaviour of the entities in a game world, it is possible to create several different instances of each entity, enabling the creation a unique environment with each play of the game. Using these dynamic game environments new game-play aspects can be introduced, the longevity of a title can be improved and entirely new gaming concepts can be applied [Wright 2005]. Procedural techniques can also provide practical benefits to the distribution of games. The concise nature of procedural assets stand in direct contrast to traditional static assets where several CDs, DVDs, and even Blu-ray discs are required to distribute a single application. Procedural techniques employ algorithms to generate content on-the-fly, which allows applications to make significant space savings. This is most evident in the *Demo Scene* where graphically detailed applications are distributed in a number of Kilobytes not Gigabytes [Scene Awards 2004] [Farbrausch 2007]. In practice a hybrid approach is used, where some static assets are necessary but procedural techniques are applied for selected assets to enable enhanced detail and more efficient distribution.

## 1.2 Procedural Generation

Procedural generation can be defined as the application of computer instructions to automatically generate geometry and textures. The construction of complex geometric objects is a new phenomenon in computer graphics, even though procedural techniques have been used for over 20 years to one degree or another. Recently these techniques have been extended to successfully model complex natural objects such as trees, waterfalls and clouds [IDV Inc. 2006][Ebert et al. 2003].

Early procedural algorithms were primarily concerned with the construction of textures. This can be seen in Perlin Noise, which was devised to add a natural appearance to textures by creating a coherent form of noise by layering noise textures to add a more natural appearance [Perlin 1985]. This technique has also been applied to generate solid textures of natural materials such as marble [Perlin 1999]. Other techniques emulate natural cellular materials by using Voronoi Diagrams to create textures of skin, bark and cobblestone [Ebert et al. 2003].

The construction of complex geometry in games has only recently been carried out using procedural techniques. One area to which these techniques have been applied successfully is to the generation of trees and plants. In 1990 A. Lindenmayer and P. Prusinkiewicz released a book titled *Algorithmic Beauty of Plants* that introduced a system of graphically modelling plants using a rewriting system [Prusinkiewicz and Lindenmayer 1990]. This book illustrated how complex plants could be generated from a concise definition and documented a formal grammar to describe the structure of these plants. Although no immediate effects were noticeable in game development it did provide inspiration for further research. A commercial real-time tree generation system titled *SpeedTree RT* by IDV was first licensed in 2002 and provides a solution for generating trees in games and other computer graphics applications. *SpeedTree RT* has been used by numerous game studios including Rockstar, Microsoft, Epic and Sony [IDV Inc. 2006]. The system has also received several middle-ware awards and garnered noticeable attention for its application of procedural generation [Gamasutra 2002].

Entire worlds can be constructed from procedural techniques, where assets including realistic natural features such as terrain, lakes, trees and shrubs are procedurally generated [Pandromeda 2006]. The widespread application of procedural techniques has been largely confined to technical demonstrations and show-cases.



*Figure 1: SpeedTree RT [IDV Inc. 2006]*

In the computer graphics industry, and in particular the games industry, procedural generation is seen as a complementary technology that can be used to supplement traditional artist-authored content. The application of procedural assets has been limited to the construction of natural phenomena such as trees, shrubs and terrain, but has equal relevancy for the construction of man-made phenomena. With games such as Spore continuing to expand the boundaries of procedural generation, these techniques and their range of applications will continue to grow beyond simple flora [Wright 2005].

### 1.3 City Generation

Cityscapes are difficult to model. They are both visually and functionally complex and are a result of an elaborate evolutionary process that takes place over hundreds of years under the influence of countless factors. Some of the major influential factors affecting cities include population, transport, environment, elevation, vegetation, geology and cultural influences. It is a formidable challenge to create a realistic model of such a large and complex system.

To design a procedural generation system that can construct realistic cityscapes, it is important to identify and carefully select a reduced set of factors to model. A number of urban design and architecture authors have discussed many aspects of cityscapes including the patterns present and the constituent components. Kevin Lynch writes about the image of the city and human perception. He itemises constituent elements of cities such as paths,

edges, landmarks, nodes and regions [Lynch 1960]. Alexander et al. documents a number of patterns found within cities such as neighbourhoods, public areas and special buildings [Alexander et. al. 1977]. Using this research suitable candidates are identified and only the most predominant patterns and features of cities are selected for modelling.

Specifically our research focuses on the patterns of road networks, the divisions of neighbourhoods and building construction. The primary, or main roads act as traffic flow arteries, whose function it is to transport people and goods around the city from one district to another. In addition, the primary roads often convey essential characteristics of the city and form tangible boundaries that divide the city into regions or neighbourhoods. Within each of these neighbourhoods we find the secondary roads that service the local area by providing access to and from the primary road network. Buildings are only situated or placed within access of the primary or secondary road network.

## 1.4 Aims and Objectives

Our general aim is to develop a city generation system that produces the required geometry, materials and textures to model a cityscape. In particular the system should be capable of:

- producing a city model that is realistic, detailed, large scale and fits into the surrounding environment.
- generating road networks that reproduce or emulate a number of distinct styles found in real city road networks.
- constructing primitive buildings that are suitable for use in real-time rendering but detailed enough to maintain the realism of the city.

If successful, the system should do the above in such a way that:

- interactive and tactile control of city generation is possible.
- it is easy to use and accessible to novice users.
- output can be easily used by those working in the graphics industry.

In addition to this, some practical objectives are to:

- implement a portable multi-platform workspace for city generation.
- implement the system such that it can be easily developed and extended.

## 1.5 Achievements

In this thesis we document our research in the area of procedural city generation. The concepts and ideas proposed to solve the city generation problem are outlined, the generation system is described and the results are evaluated. Notable achievements of the work can be summarised as follows:

- Study of procedural techniques: their background, principles and the key properties which distinguish successful algorithms. Analysis of related city generation research, including an outline of the algorithms, and an evaluation of the output generated.
- Design of an adaptive road system that automatically plots the path each road takes by sampling terrain and fitting the road to the environment according to a pre-specified strategy.
- Design of a real-time road growth algorithm and an efficient system to calculate intersection and proximity status for each road segment added.
- Implementation of an interactive city generation system. A cross-platform graphical application provides an integrated workspace to view, edit and interactively control the complete city generation process.
- Testing and evaluation of the city generation system where the operation, performance and output of the system is evaluated.

## 1.6 Thesis Outline

In this introduction some background information has been provided, procedural techniques have been introduced, the motivation for research has been outlined and the goals and major achievements have been listed. On this basis an understanding of the research has been established and the rest of this thesis will cover in more detail the related literature, theory, design, implementation and results.

In the next chapter, Chapter 2, we provide an overview of the subject of procedural techniques and present a number of key techniques and algorithms that have been applied successfully in the field of computer graphics. In Chapter 3 previous research into the procedural generation of cities is reviewed and an analysis of the existing solutions is provided. Chapter 4 outlines the design for our interactive city generation system called

Citygen, and explains the operation of the main components. Chapter 5 documents the implementation of Citygen including the tools, libraries and algorithms used. In Chapter 6 the results of the city generation system are presented and areas for possible future research are considered. Finally some conclusions are provided.

# Chapter 2

## Procedural Techniques Research

In this chapter an overview is provided into the field of procedural techniques and their application in the computer graphics world. A description of general procedural techniques is included and several key properties of effective algorithms are identified. In order to gain an insight into the application of procedural techniques an outline into the operation and results of several of the most influential techniques is provided. This study includes techniques such as Fractals, Perlin Noise, L-Systems and Cellular Basis algorithms.

### 2.1 Key Properties

A procedural technique describes an entity, geometry, texture or effect, in terms of a sequence of generation instructions rather than as a static block of data. These instructions can then be executed to create instances of the asset and parameters can be used to vary characteristics. Procedural techniques can thus be employed to produce a wide range of assets, from generating simple noise for use in texturing and natural formations [Perlin 1985], to more complex recursive algorithms such as fractals or L-systems that can recreate organic structures such as snow flakes and trees [Prusinkiewicz and Lindenmayer 1990].

Key properties of successful procedural techniques include [Ebert et al. 2003]:

- **Abstraction:** Data is not specified in the conventional sense as geometry, textures, etc. but instead the data and behaviour of the entity is abstracted into an algorithm or a set of procedures. Minimal knowledge is required by the operator and model data can be manipulated easily without requiring details of the implementation.

- **Parametric Control:** Parameters directly correspond to a specific behaviour in the procedural generation. The developer can define as many useful controls as required for the artists to operate effectively. Example of parameters include the height of the mountains in a terrain algorithm or the number of segments in a procedural sphere.
- **Flexibility:** It is possible to capture the essence of an entity without explicitly bounding it within real-world limits. Parameters can then be varied to produce different results as desired and even results outside the normal range of the original model can be generated.

Procedural techniques have been applied successfully in the generation of numerous complex phenomena in computer graphics and have proved beneficial for a number of reasons.

Textures, geometry or effects abstracted into procedural algorithms are not fixed at a set resolution or number of polygons. Procedural techniques are therefore inherently multi-resolution in nature and the complexity of their output can be varied. This capability is of particular interest to computer graphics practitioners. For example *level of detail* (LOD) is important in any 3D rendering system and essential to real-time rendering applications [Akenine-Möller and Haines 2002]. The concept behind LOD is to use more simple versions of an entity if it contributes less to the final rendered image. So for an object that occupies only 4 pixels in the final image, 10,000 polygons are not required and a basic representation using 10 polygons would be sufficient. The multi-resolution nature of procedural techniques allows models to be automatically generated at several levels of detail [Ebert et al. 2003].

Concise descriptions for generated objects are possible and can often be expressed in terms of a few simple parameters. These small descriptions can be used to create large amounts of detailed textures and geometry. This effect is known as data amplification [Ebert et al. 2003] and provides developers with the means to create an entire world that is easily distributable over low-bandwidth network connections. The conciseness of procedural techniques are exploited by *Demo Scene* creators who create and distribute scenes that are complex and rich in detail in the form of tiny executable files as small as 2KB [Scene Awards 2004].

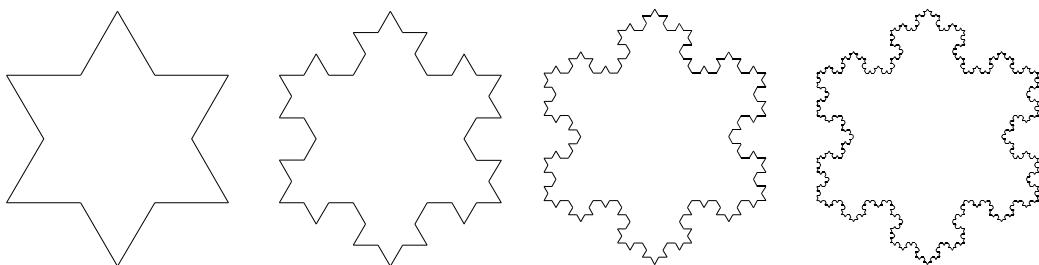
The flexibility and control provided by procedural techniques give the designer a platform for artistic freedom and experimentation. New visual effects and original objects can be created by experimenting with parameter values that exceed normal boundaries [SideEffects 2005]. Typically procedural algorithms are implemented in advance on software, however with recent advances in graphics hardware it is possible to execute techniques in real-time on the

GPU. For example, complex procedural techniques like volumetric textures that were previously impossible to run in real-time can now be implemented in this manner [Hart 2002] [Spitzer et al. 2003].

A number of fundamental procedural techniques and algorithms are now described that have been successfully employed within the domain of computer graphics.

## 2.2 Fractals

Natural shapes are not easily described by conventional geometric methods. Clouds are not spheres and mountains are not cones. Natural shapes tend to be irregular and fragmented and exhibit a complexity incomparable to regular geometry [Mandelbrot 1982]. However these shapes can be described using a branch of mathematics called *fractal* mathematics. Benoît Mandelbrot, regarded as the 'father of fractals', coined the term fractal in 1975 from the Latin *fractus* meaning broken. The basic concept of fractals is that they contain a large degree of *self similarity*. This means that they usually contain little copies of themselves buried deep within the original, like the stars embedded in the Koch Snowflake [Ebert et al. 2003], as shown in Figure 2. Also, fractals possess infinite detail, so for any given fractal, the closer we look at it, the more detail it can reveal [Linden and Schachinger 2002].



*Figure 2: The first four iterations of the Koch snowflake*

The Koch snowflake in Figure 2 shows four recursions. Self-similarity is achieved by generating the same shapes or patterns at smaller and smaller scales as the recursion progresses, a property referred to as *scale invariance*. There is no theoretical limit to the amount of recursion that can be done and hence infinite levels of detail can exist within the shape. Visualizing fractals manually is difficult, and therefore computer based implementations of fractal algorithms have been present from the start. Mandelbrot used computers to visualise complex fractals including the Mandelbrot Set shown in Figure 3 [Mandelbrot 1982]. In addition, a wide range of natural structures from simple plants like

ferns as shown in Figure 4, to detailed terrain, contain fractal properties and can be generated using simple recursive algorithms [Barnsley 1988]. Fractal algorithms are particularly suited to procedural generation because of the effective *abstraction* they provide from the structural complexity of the natural objects they represent. Also, fractal algorithms yield a high level of *data amplification*. Complex models can be generated from a few simple equations. Finally fractal algorithms can utilize recursion to provide varying levels of detail.

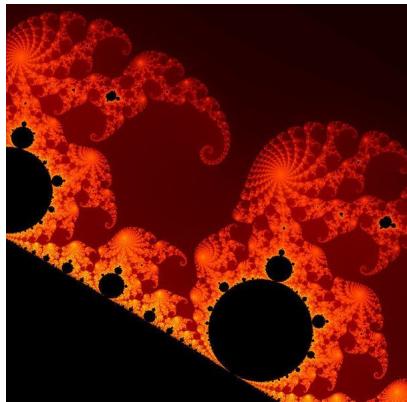


Figure 3: Mandelbrot Set



Figure 4: IFS Fractal Ferns [Barnsley 1988]

Fractals are limited however to self similar structures and the objects we are seeking to model may not necessarily contain this self-similarity. They are superseded in many contexts by other more flexible algorithms like formal grammars such as L-systems.

### 2.3 L-Systems

L-systems are a formal grammar devised by biologist A. Lindenmayer as a mathematical theory for biological development. L-systems were originally developed to study bacteria replication and the growth patterns of simple organisms [Lindenmayer 1968]. Since then the system has been extended to define more complex objects such as plants and branching structures. In the book, *Algorithmic Beauty of Plants*, the developmental process of plants is captured using the formalism of L-systems and visualised with computer graphics [Prusinkiewicz and Lindenmayer 1990].

The central concept of L-systems is that of rewriting, a technique for defining complex objects by successively replacing parts of a simple initial object using rewriting rules. An example of a simple L-system is shown in Figure 5. An initial state or axiom,  $\omega$ , is a string of symbols and constants that define the initial state of the system. A series of rewriting rules or

productions,  $\mathbf{P}$ , are then defined. Each of these consist of two strings: the predecessor and the successor, that specify the way variables can be replaced. These rules are applied successively, allowing large complex objects to be quickly generated from a simple axiom.

$$\begin{aligned}
 V : & \{a, b\} & n=0 : & a \\
 \omega : & a & n=1 : & ab \\
 P_1 : & a \rightarrow ab & n=2 : & aba \\
 P_2 : & b \rightarrow a & n=3 : & abaab
 \end{aligned}$$

Figure 5: Algae Growth: three iterations

L-systems can be used to visualise structures by embedding graphical symbols in the vocabulary of the axiom or productions. Turtle commands are used to describe and visualize a range of L-systems including plants and branching structures. The idea behind *turtle graphics* is that the '*turtle*' can be given instructions relative to its current position and as it moves it leaves a pen line mark behind it. The bracket extension was proposed by Lindenmayer to support the branching structures that are common in nature [Lindenmayer 1968]. Figure 6 displays an example of such a structure defined as an L-system using the bracket extension.

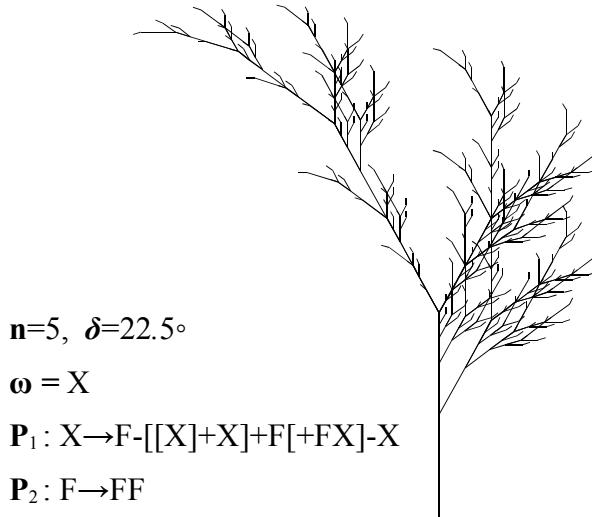


Figure 6: L-system branch generated in turtle graphics[Prusinkiewicz and Lindenmayer 1990]



*Figure 7: Procedurally generated tree used in a modern 3D game [IDV Inc. 2006].*

Research into L-systems has continued, and significant advances have been made with commercial packages now available that can apply similar techniques to generate rich landscapes with detailed foliage including shrubs, plants and trees. Figure 7 shows a demonstration of the SpeedTree plug-in from IDV Inc. which enables graphics developers to easily populate scenes with a realistic and diverse range of plants and trees [IDV Inc. 2006].

L-systems work well as a procedural technique for a number of reasons. They allow complex models and organic structures to be defined, modelled and visualised using a concise set of productions. A varying level of complexity can be supported by parameters such as the recursion level of the L-system [Lluch et al. 2003]. The algorithms can be defined in a compact and intuitive manner and can effectively abstract the recursive structure of many natural phenomena. L-system generation can be adjusted easily via external parameters and is extensible by design, in a similar way to other formal grammars.

## 2.4 Perlin Noise

Perlin Noise is an algorithm that can be used to create more natural looking textures. The technique was originally developed by Ken Perlin and was first applied in the feature film Tron released in 1982 [Perlin 1985]. The technique has a range of applications in computer graphics including the creation of effects like fire and clouds and the generation of fractal geometry like terrain.

The concept behind Perlin Noise is to combine a number of noise layers together to produce a single texture of coherent noise with fractal like detail. A *Coherent* noise function can be defined as one in which the values change smoothly from one point to the other.

Two major components are used to accomplish this: a *noise generation* function and an *interpolation* function.

*Noise generation* is achieved by employing a simple random function to construct an initial noise data set. It is important that the data output is controlled and reproducible. For this reason a seeded random generator is used which can produce consistent results for a given input seed and maintain a random output pattern.

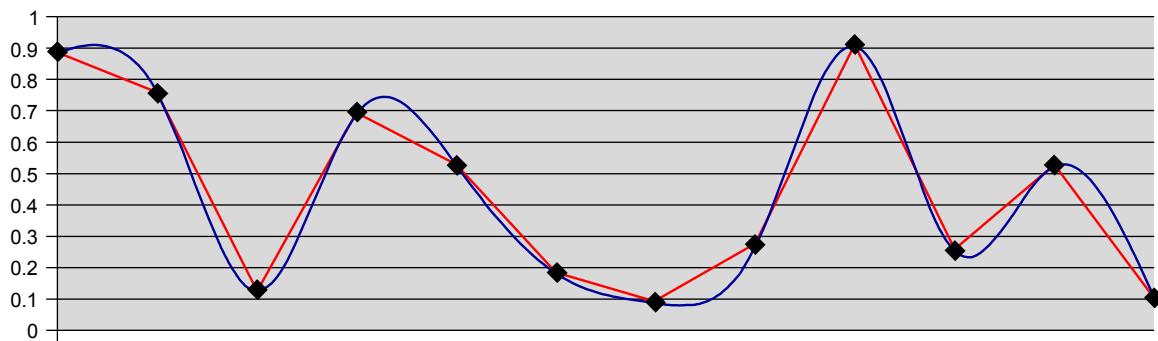
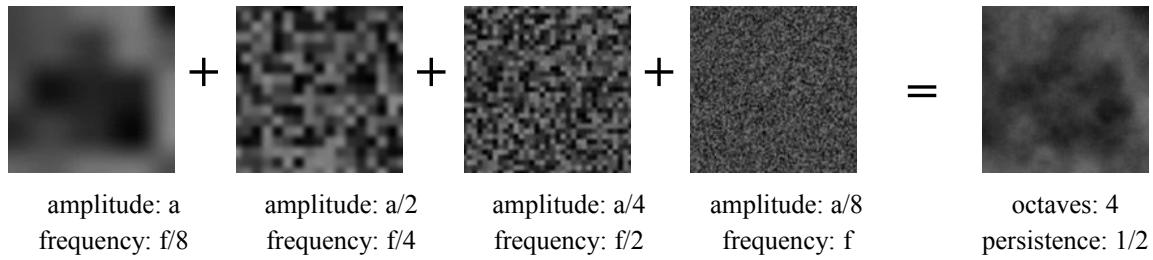


Figure 8: Interpolation Linear and Cubic

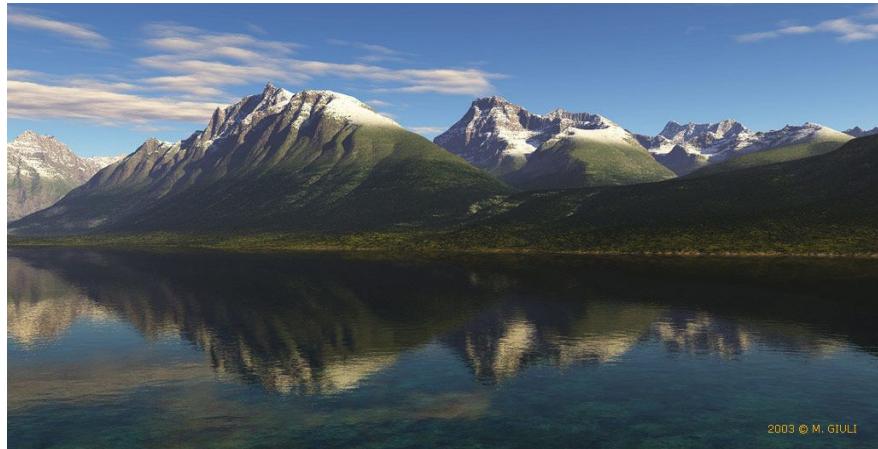
*Interpolation* is a process of curve fitting in which a function is constructed that can match a given data set. Using this function, new data points can be calculated from a initial set of values, in this case the data generated by the noise function. No specific interpolation algorithm is required for Perlin Noise and each algorithm can vary in computational complexity, smoothness of function curve, accuracy and number of data points required. Linear interpolation is a basic method, fast and of low quality. Cubic interpolation in contrast is more complex, significantly slower, outputs a high quality curve and requires four points to obtain a single value.

The interpolation process allows any random noise data to be expressed as a continuous function. From these functions a number of noise texture layers can be created using any specified frequencies and amplitudes.



*Figure 9: Combination of several layers of noise.*

*Turbulence* is finally applied by combining several noise texture layers of differing scales together. This creates a form of coherent noise. Each layer is referred to as an *Octave* and the ratio between amplitude and frequency of the layers can be expressed as a constant known as *persistence*. The resulting output is a natural looking procedural texture that can be defined in terms of a few simple parameters.



*Figure 10: Photo realistic scenery and rendered using Terragen with procedural geometry generation and procedural texturing. ©2003 M. GIULI.*

In nature, there are several scales of detail present. For example, in terrain, large features like mountains are most predominant, but also smaller features like hills and crests, and even fine detail such as scree are also present. These layers of detail make procedural techniques like Perlin Noise especially suited to generating natural phenomena. Terragen uses the Perlin algorithm to generate photo realistic terrain, clouds and seas [Planetside 2004]. Figure 10 showcases the detail and scale of output that is possible using procedural techniques. The *persistence* parameter can be used to control aspects of terrain generation. A low ratio of persistence can produce smooth terrain with very fine detail, and high persistence may result in more jagged terrain with less fine detail. For real-time rendering applications, the Perlin Noise algorithm can generate any specific region of the terrain on demand and vary the level of detail present without needing to store the massive data-set of the terrain geometry.

Solid textures, also known as three dimensional or volumetric textures, can be generated using Perlin Noise. Solid textures differ from conventional two dimensional textures in that they allow objects to be virtually carved from the texture as they would be carved out of a solid block of material [Perlin 1999]. An example is shown in Figure 11 of a vase that is carved out of a volumetric marble texture created using the Perlin noise algorithm. The texture replicates the veins of darker material running through the marble and achieves a higher level of realism than is possible using basic 2D texturing techniques. Solid textures are computationally expensive to render with high memory and storage requirements. Compression such as S3TC can partly alleviate this problem but still results in high requirements. Perlin Noise can be used to solve this problem as it requires minimal storage due to its procedural nature, and can even be used to render volumetric textures in real-time using the pixel-shader hardware on the GPU, effectively removing any memory constraints [Hart 2002].



*Figure 11: Marble vase textured with a solid texture [Perlin 1999].*

Perlin Noise provides a comprehensive set of benefits. Effective parametrization gives the developer control of the output using high level parameters. Geometry and textures created using the algorithm have minimal storage requirements and can be generated from a concise definition consisting of a few simple parameters. Textures of any size and detail can be produced and the innate behaviour of the algorithm can be exploited to support varying levels of detail. The output generated is tileable and can join seamlessly, thus enabling techniques like repeating and layering for multi-texturing. Additionally, the technique can be used as a method of enabling real-time volumetric textures on modern graphics hardware. Perlin Noise is one of the most useful and frequently used procedural techniques and is beneficial in a wide range of computer graphics applications.

## 2.5 Tiling

Tiling is one of the most basic procedural techniques and has a long tradition in game development. Many of the classic platform titles such as *Mario* employed tiling to repeat sections of 2D graphics creating a virtual world. Games such as the *Shoot Em Up Construction Kit*, released in 1987 by *Sensible Software*, allowed the user to construct and edit game maps by providing a simple interface to select and position tiles from a tile library [Sensible 1987]. Figure 12, shown below, demonstrates an example *tileset* and a corresponding screenshot for the Super Mario Bros game by Nintendo.



Figure 12: Super Mario Bros. 2 (*Lost Levels*), © Nintendo Japan Ltd.

More recently multi-texturing techniques have evolved and use repeatable tiles layered together to create highly detailed and varied textures. New materials can be created by combining a set of detailed textures, colour maps and blending maps. Using this technique terrain can be procedurally textured by applying several layers of detailed tile-able textures. Examples of texture layers could include rock, grass, sand and snow. Also texture layers can be combined with varying degrees of influence on the final texture. Textures are applied to the terrain according to a variety of specified parameters, they can be selected according to height, slope, or specified explicitly using an image map [Planetside 2004]. This solution allows vast areas to be textured in detail, something that is not possible using a single high resolution texture.



*Figure 13: From left to right: a) Regular and transition patterns and the probability map.  
b) Virtual texture. c) Virtual texture mapped on terrain. [Lefebvre and Neyret 2003]*

Extended algorithms have also been developed that use stochastic information such as probability distribution maps to procedurally texture landscape [Lefebvre and Neyret 2003]. An image map for the terrain area is supplied that stores the probability of using various tiles. Constraints can be specified to state which tiles can be joined under what conditions and whether they may be joined directly or require transitional tiles. Using a random function thousands of different permutations of worlds are possible from a single probability map.

Procedural tiling systems provide several advantages for graphics applications. Vast and detailed landscape or terrain for virtual worlds can be created from stochastic information and small sets of texture tiles. These maps and game worlds can then be easily distributed for on-line gaming which is particularly useful for massively multi-player on-line role-playing games (MMORG) and other on-line applications where game resources are shared. Storage and memory requirements are minimised so it is possible to optimally store and render worlds of vast dimensions in real-time on commodity hardware. Tiling is a good example of how a simple procedural technique can be applied and extended successfully in computer graphics.

# Chapter 3

## City Generation Research

In the previous chapter we introduced procedural techniques and discussed the benefits they can bring to computer graphics. In this chapter we shall look at the structure of a modern city, and discuss the work of urban design and architecture authors who identify some key patterns and constituent components. Before reviewing the existing city generation research, a critical framework is established. Then an overview of each city generation system is presented, and an insight is provided into the operation of each systems' algorithms. A discussion accompanies each overview, using the critical framework to evaluate the strengths and weaknesses of each approach.

### 3.1 The Structure of a City

Cities are both visually and functionally complex. To obtain a better understanding of these complex systems and their structures, we look at the research which identifies the patterns and the constituent elements. Urban design and architecture authors have discussed a large number of wide-ranging topics relating to cityscapes. From the research studied we have selected work that relates closely to city structure and presents the most direct correlation to city generation. Two publications in particular fulfil these criteria: Kevin Lynch writes about the image of the city and human perception, he itemises the constituent elements of cities as paths, edges, districts, nodes, and landmarks [Lynch 1960]. Alexander et al. document the patterns found within cities such as neighbourhoods, public areas and special buildings [Alexander et. al. 1977]. The following sections discusses this research in two sections: Primary Transit Network and Neighbourhoods / Districts.

### 3.1.1 Primary Transit Network

The primary transit network consists of the main roads or highways of a city, the major rail lines and waterways. This network is an important element in determining the sub-structure of a city and is instrumental in our interpretation of

From a visual perspective the primary transit network is perceived by both occupants and outside observers as the most predominant and recognisable pattern present in city structure. This theory can be reinforced by the research of Kevin Lynch. In *The Image of the City* he documents a number of reasons for the importance of the primary transit network and its significance on our perception of the city. Firstly the concept of *Paths* is introduced: “Paths are the channels along which the observer customarily, occasionally, or potentially moves” [Lynch 1960]. In cities most transport is accomplished via road and the primary road network forms the main arteries and transport channels of the city. These *Paths* specify the location and route from which the city is observed and thus form a lasting impression on our image of the city. Lynch does not underestimate the importance of Paths on our perception of cities, he states that they are the “predominant elements” in our image of the city [1960]. Secondly the more abstract concept of *Nodes* is described. Nodes are termed as “points, the strategic spots in a city into which the observer can enter, and which are the intensive foci to and from which he is travelling” [1960]. Although they are not specific to the transit network the relationship between Nodes and the network is explained: “They may be primarily junctions, places of a break in transportation, a crossing or convergence of paths”[1960]. So in relation to the primary transit network we can easily draw a parallel between the junctions as *Nodes* and roads or tracks as *Paths*. Both have a significant impact on our perception and thus play an important role in determining the recognisable character of a cityscape.

Functionally, the primary transit network is most significant to us as users of the city. The locations and structure of this network must be understood to navigate and move around the city. For this reason when arriving at a new city one of the first things many of us will do is purchase a map. Inside a city map the primary road network is emphasised to appear most evident and the rail network map will often occupy a separate page. Later we will frequently reference these networks on the map to find our bearings. Our comprehension of the transit network is essential to our understanding of a city and it is a key factor in our recognition of city characteristics.



Milan – Concentric Rings

Paris – Radial Spokes

Figure 14: Predominant Pattern in Existing Cities (Map)

Shown above in Figure 14 are two predominant patterns from two modern cities, Milan and Paris. The images illustrate how visible the patterns of the primary road network are and how they form an intrinsic part of city character.

### 3.1.2 Neighbourhoods / Districts

Neighbourhoods, or districts, are areas of the city which have an identifiable character and a tangible boundary. Each city is composed of component districts, for example: the city centre, the financial district, residential areas, industrial areas, suburbs and so on.

Urban research helps define these areas and explain the relationships with the city occupants. Lynch defines districts as “medium-to-large sections of the city, which the observer mentally enters ‘inside of’ and which are recognizable as having some common, identifying character”[Lynch 1960]. Alexander et al. discuss the importance of neighbourhoods and their relationship to the occupants of the city. In the pattern titled *Identifiable Neighbourhood* he describes these regions and the relationship with the occupants: “People need an identifiable spatial unit to belong to. They want to be able to identify the part of the city where they live as distinct from all others”[Alexander et. al. 1977].

The boundaries of each neighbourhood are important to maintaining the integrity and character of the unit. Alexander et al. liken neighbourhoods to cells, and state the importance of boundaries. “The strength of the boundary is essential to a neighbourhood. If the boundary is too weak the neighbourhood will not be able to maintain its own identifiable character”

[1977]. The specific components of the cityscape that constitute the boundaries for these regions are described in the patterns *Subculture* and *Neighbourhood Boundary* [1977].

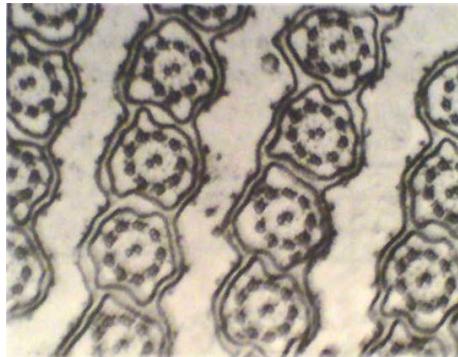


Figure 15: Cells [Alexander et. al. 1977]

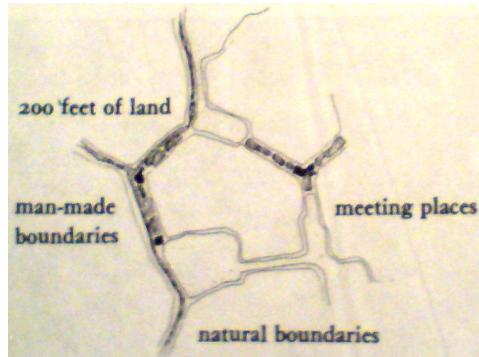


Figure 16: Neighbourhood / District Boundaries [Alexander et. al. 1977]

These cityscape components include both natural boundaries – such as rivers and lakes and also man-made boundaries – such as major roads and rail roads [1977]. Lynch terms the boundaries between districts as *Edges*, which he describes as linear breaks not considered as paths [1960]. These *Edges* may consist of elements such as shores, rivers, railroad-cuts and major roads. Although rail tracks and roads are also paths, they may be considered as *Edges* when they act as a barrier to the access of a district. For example consider a major road running through a residential area, in this case the road can restrict the movement of pedestrians in the local area and now acts more as an *Edge* than as a *Path*. Alexander et. al. reference the “Appleyard-Lintell” study and define a metric stating that a road with more than 200 cars per hour deteriorates the quality of the neighbourhood and forms a barrier to free pedestrian movement [1977].

Identifying components that constitute districts and their associated boundaries provide us with an insight into the structure of cities. It is clear from the urban research that the city-wide primary transit network alongside natural features are key to determining the boundaries of more local structures such as districts or neighbourhoods. Based on this research we can conclude that the primary transit network is an important aspect of the cityscape and its affect on the sub-structure is wide ranging. Therefore, an obvious recommendation can be made, to create a successful city generation system an accessible method of control for this influential element should be provided.

## 3.2 The Evaluation of City Generation Systems

City generation is achieved in a series of stages, with each applying one or more algorithms to generate a constituent component of a city. There is no predefined scope for any stage and each system has a unique approach making direct comparisons difficult. The generation process can however be divided into two main stages: Road Generation and Building Generation. For these stages an overview is presented for each city generation system and an insight is provided into the operation of the procedural techniques applied.

### 3.2.1 Critical Framework

In order to evaluate the output created by the generation systems and the effectiveness of the applied procedural techniques we identify a common set of criteria:

1. **Realism** – Does the output of the city generation system look like a real city? How much detail is present and how true is the generated model to a real city model?
2. **Scale** – Is the urban landscape generated up to the scale of a city? How many roads, neighbourhoods and buildings are generated?
3. **Variation** – Can the city generation system recreate the variation of road networks and buildings found in real cities or is the output homogeneous?
4. **Input** – What is the minimal input data required to generate basic output and what input data is required for the best output?
5. **Efficiency** – How long does it take to create the examples shown and on what hardware are they generated? How computational efficient is the algorithm?
6. **Control** – Can the user control the city generation process? To what degree can the user influence the generation results? Are the control methods applied intuitive or restricted?
7. **Real-time** – Are city geometry and textures generated in real-time? Are any rendering optimisations applied and can the city be rendered or explored in real-time?

The results of each approach are evaluated, using the criteria outlined above, to determine the effectiveness of the city generation systems and their associated procedural techniques.

### 3.3 Grid Layout & Geometric Primitives

Stefan Greuter et al. describe a system to procedurally generate a city in real-time [Greuter et al. 2003][Greuter et al. 2004]. The techniques applied to generate the city are discussed in a number of papers and demonstrated in a virtual city application titled *Undiscovered City*. The application creates a road network using a simple grid layout upon which it can place buildings generated using a combination of simple geometric primitives. The research is specifically targeted at real-time applications and the *Undiscovered City* application is a proof of concept for this idea. The system runs in real-time and renders at interactive frame rates.

#### 3.3.1 Buildings: Geometric Primitives

The building generation system uses the location of buildings in the form of grid coordinates as a seed for building generation. The appearance of each building is determined by this seed including properties such as height, width and number of floors. Generating buildings using a similar set of numbers such as neighbouring grid coordinates can result in similar looking buildings, so to overcome this a hashing function (shown in Figure 17) is implemented in order to provide more random distribution.

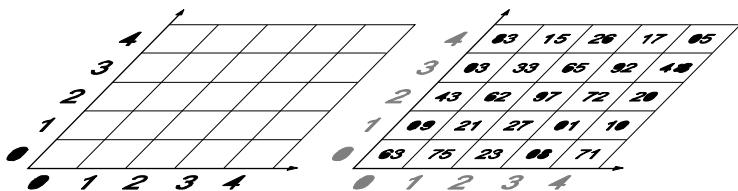


Figure 17: Grid Layout Coordinates & Hashing [Greuter et al. 2003]

Building geometry is generated by combining geometric primitives to form building sections. Each building section is constructed using a different floor plan. The top most section of the buildings are created by extruding a three dimensional volume from the most basic of floor plans, composed from only a few primitive shapes. In subsequent sections below, another primitive shape is added to the previous floor plan and a three dimensional volume is extruded in the same fashion. Figure 18 illustrates how the creation of consecutive sections are combined to form the complete geometric model of a building. Figure 20 shows the generated buildings with their textured faces which are not procedurally generated but are selected from a set of 10 building window textures.

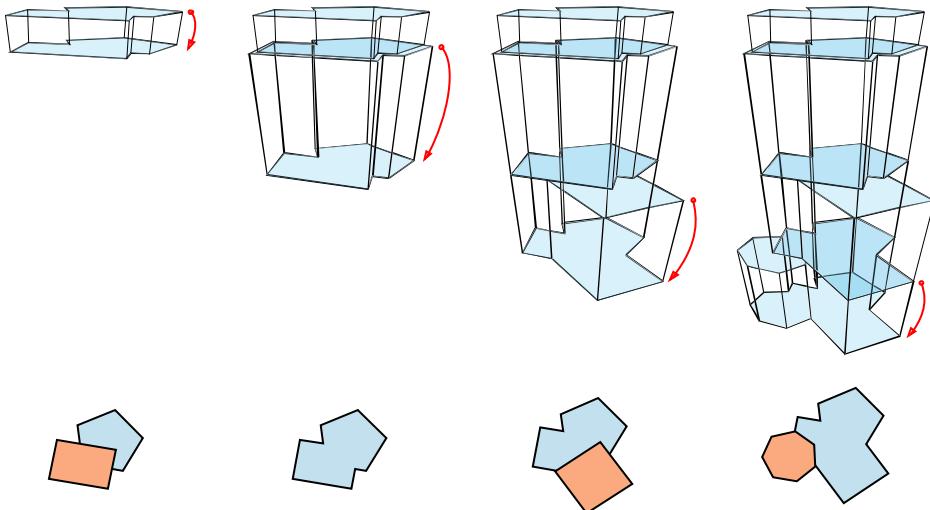


Figure 18: *Floor Plan Generation* [Greuter et al. 2003]

### 3.3.2 Real-time Optimisations

The *Undiscovered City* is designed with real-time applications in mind and implements optimisations such as a geometry caching and view frustum culling. The culling technique, referred to as *View Frustum Filling* (shown in Figure 19), renders only the buildings visible within the view frustum. By loading and rendering a reduced set of buildings the amount of memory required to store the scene and the graphical processing power required to render the scene are minimised, enabling the real-time rendering of a large data set like a city. The grid road network allows easy detection of building visibility within the view frustum and hence provides a computationally efficient method to cull superfluous buildings from view.

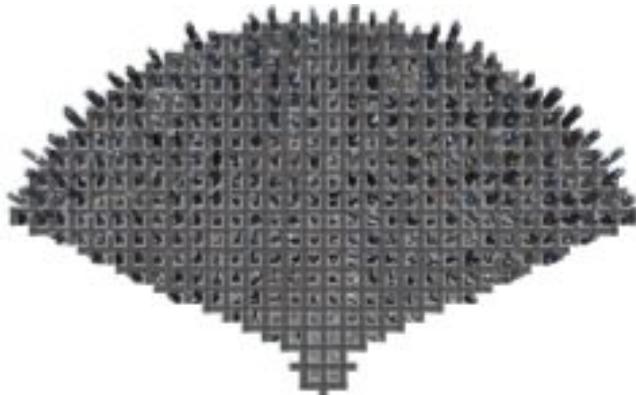


Figure 19: *View Frustum Filling* [Greuter et al. 2003]

In addition to culling building geometry, a building cache is also implemented. Buildings are generated in advance and defined as OpenGL display lists that can be stored in the building cache. The cache employs a least recently used algorithm: recently accessed buildings are

kept in the cache while older less recently accessed items are replaced. As a result of using the building cache, memory use is optimized and buildings can be recalled from cache for display an order of magnitude faster than they can be generated from scratch.



Figure 20: Screen shot at street level in the *Undiscovered City* demo

### 3.3.3 Discussion

**Realism:** The single grid pattern does not reflect real cities where a number of patterns are present, as a result the road network appears artificial and homogeneous. All of the buildings appear angular and modern and are somewhat realistic but unconvincing. Simple windowed faces are used and the buildings are not geometrically detailed.

**Scale:** The grid layout system can create road networks on a very large scale and is limited only by the size of the integer based coordinates. At  $2^{32}$  cells wide, the size is not a practical restriction for city generation.

**Variation:** The grid system is required for real-time optimization in the system. However, the resulting road network has little variation with the only control parameter being the grid spacing. Only a single building type is constructed and although the geometry for each building is different the cityscape still appears homogeneous.

**Input:** No input maps or geo-statistical data are required.

**Efficiency:** Road network and building generation take place in real-time, and figures are provided for the generation and rendering of the *Undiscovered City*.

**Control:** Grid spacing can be adjusted using short-cut keys in the application and the changes can be viewed in real-time. The building generation process is not interactive and all buildings are generated using a random seed created from the grid hashing technique.

**Real-time:** The system is designed for real-time applications and can render views of large scale cities in real-time on commodity hardware from 2003 at interactive frame rates. [Performance for numbers of buildings being displayed on screen: 200 buildings @60fps, 500 buildings @20fps, 1000 buildings @5fps].

## 3.4 L-systems

Parish and Müller present one of the most complete city generation solutions, the *CityEngine*, in a paper titled *Procedural Modeling of Cities* [Parish and Müller 2001]. The *CityEngine* consists of a suite of components including road generation, building construction and building face creation that unite to form a pipeline for city generation. L-systems [Lindenmayer 1968] are selected as the key technique for procedural generation in the *CityEngine*. Lindenmayer-systems have traditionally been used to model natural phenomena but are also suitable for the generation of cities due to their concise nature, computational efficiency and data amplification properties.

### 3.4.1 Road Network: L-systems

L-systems, as previously discussed, have been used to model the development of plants and branching structures. These contain some similarities in structure to road networks. The *CityEngine* uses an extended form of L-systems titled *Self-sensitive L-systems* to construct road networks in a manner which takes existing growth into account.

Input is taken in the form of 2D image maps. Geographical information on elevation, vegetation and water boundaries is required, and additional socio statistical image maps can also be included specifying information such as population density, land usage, street patterns and maximum building heights. A road network generation application, shown in Figure 22, is used to manage the generation of roads, and allows the operating user to specify extra parameters such as the smoothing angle of road network edges, road width, etc. Although only a geographical input map is required, the examples included in the paper, such as Virtual Manhattan in Figure 26, utilize a number of different input maps.

Road generation is accomplished through the use of two rule sets: the *Global Goals* and the *Local Constraints*. Road segments are initially plotted according to the Global Goals which are similar to the goals that a city designer may have. These tentative plans are then refined by the Local Constraints which reflect the practical constraints of the real world and the state of the existing road network.

### **Global Goals**

- There are two different types of roads: highways or major roads connect population density centres which can be identified from a grey-scale population density map supplied at input, small roads connect to the nearest highway.
- Streets follow some super imposed geometric pattern.
- Streets follow the path of least elevation.

### **Local Constraints**

- Road segments are pruned to fit inside a legal area: line segments extending into water are pruned.
- Roads are rotated to fit inside a legal area: a road to the coast bends around the coastline like a coastal road.
- Highways are allowed to cross an illegal area of a certain distance: a highway approaching a limited span of water will cross over it like a bridge.
- Roads segments are checked to see if they intersect with existing roads or if they come within a certain distance of an existing road junction: Figure 21 shows how proposed road segments are modified to satisfy the self-sensitive rules.

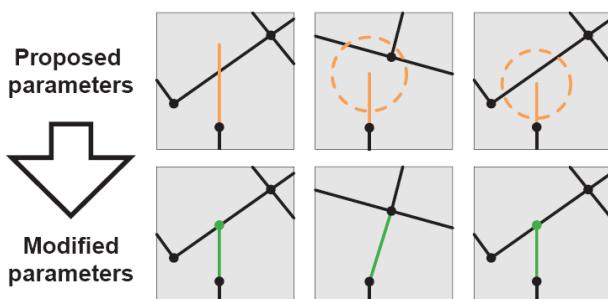


Figure 21: Self-sensitive road L-system [Parish and Müller 2001]

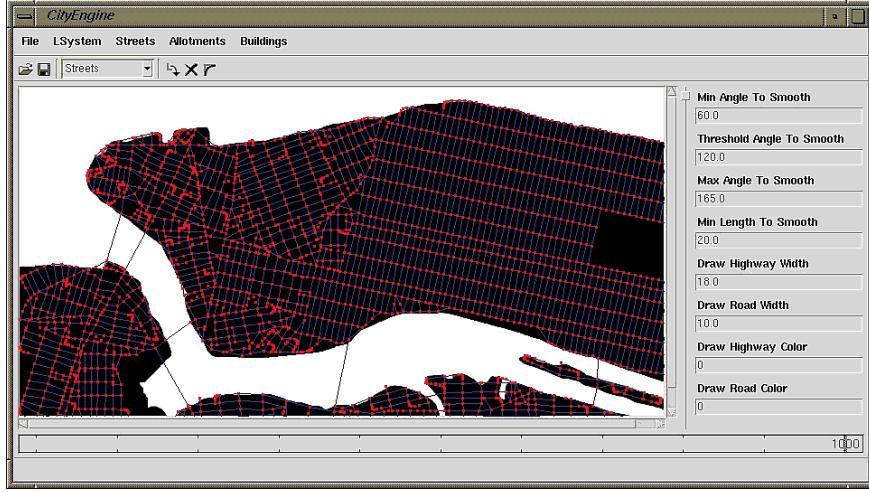


Figure 22: CityEngine GUI displaying Virtual Manhattan after 100 steps [Müller 2006].

### 3.4.2 Buildings: L-systems

The CityEngine constructs buildings on the road network in a series of distinct stages: define building allotments, create building geometry and generate textured faces. To define building allotments the CityEngine utilizes data from the previous road network generation stage. Figure 23 outlines the stages of allotment generation. Allotments or lots are calculated by first extracting blocks from the road network using the roads of the network as the dividing borders. Each basic extracted block is then divided into a series of potential lots via randomized subdivision. Lots that are too small or have no immediate street access are culled and removed from the system. The final lots generated by the CityEngine are shown in the right-most image of Figure 23 and appear both varied and practical.

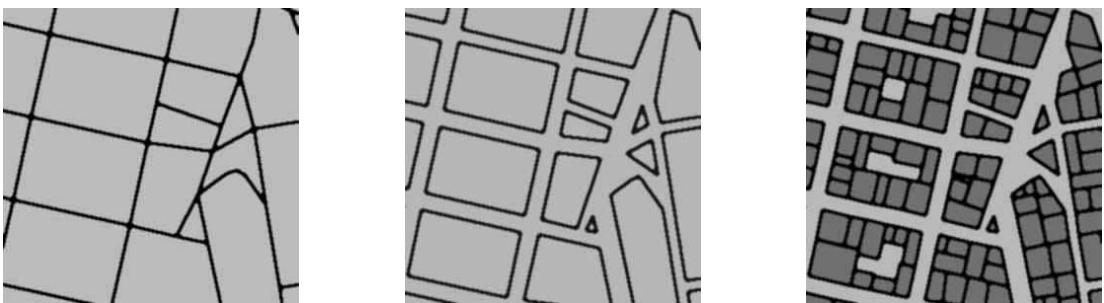
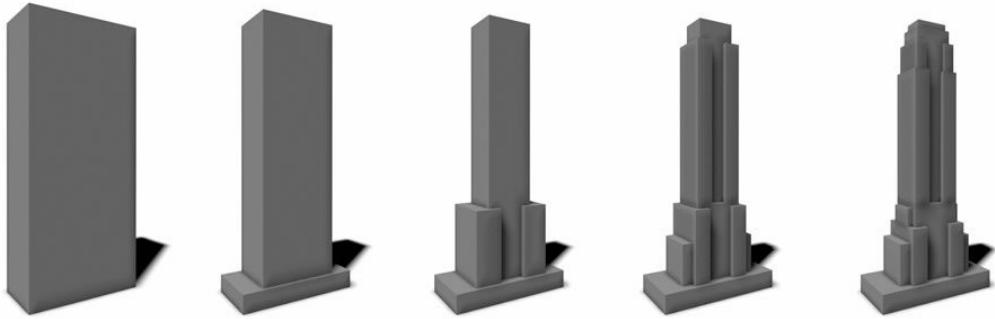


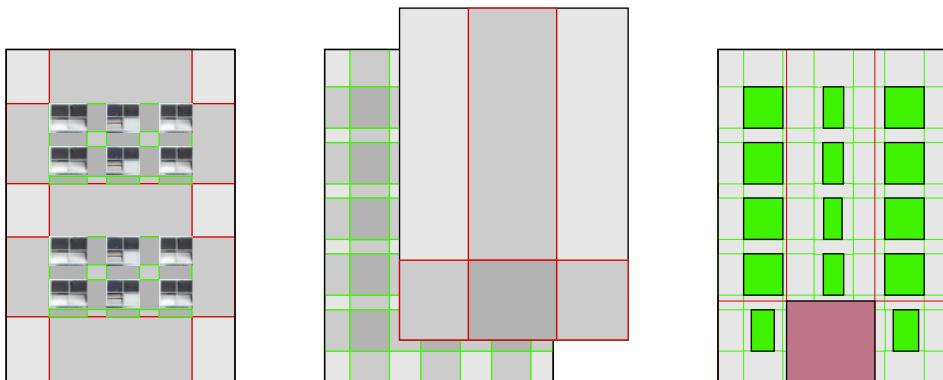
Figure 23: Lot Division Stages [Parish and Müller 2001]

Building geometry is generated through the use of a parametric L-system. Several different building styles are implemented including skyscrapers, commercial and residential, with each type using a different set of L-system productions. The building type is determined from a zone map which can be passed in as an image map input.



*Figure 24: L-System building refinement from bounding box to the Empire State Building [Müller 2006].*

The initial state, or axiom, of the building L-system is a bounding box generated from the lot footprint and a building height image map, if available. L-system operations consist of transformations (scale and move), extrusions, branching and termination, and the use of geometric templates for roofs, antennae, etc. L-systems allow for the addition of more productions and provide an extensible solution. A basic level of detail implementation is possible since each iteration of the building L-system is a refinement of a basic building bounding box as shown above in Figure 24.



*Figure 25: Building face construction [Parish and Müller 2001]*

Building faces are created procedurally by generating textures using an over-laid series of grid-like structures. Several layers of grid-like structures are used with functions that define how the layers are combined. The functions dictate which cells from what layer are selected to create the final face and can use conditional and statistical information to select cells. Cells typically contain doors or windows but can contain any building face feature. The construction of a face is shown in Figure 25. The red layer influences the selection of cells from the green layer. The resulting face is a conditional combination of multiple layers.

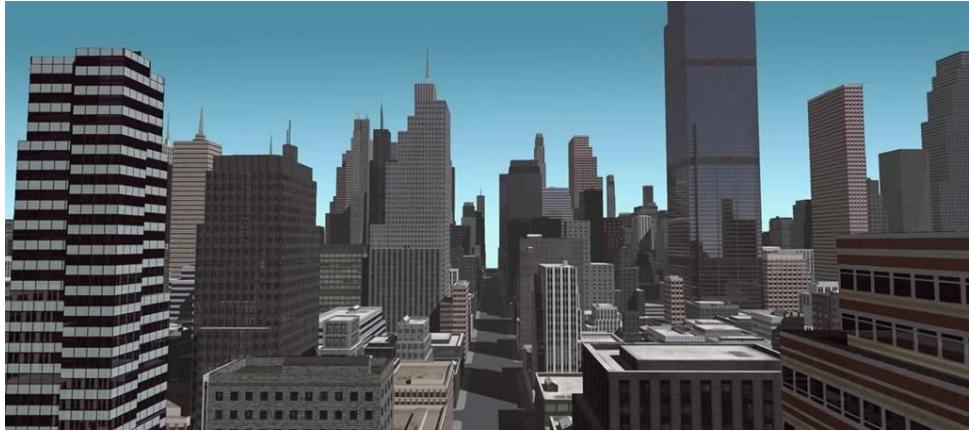


Figure 26: *CityEngine - Virtual Manhattan – Maya render [Parish and Müller 2001]*

The *CityEngine* produces data that can be imported into Maya, a commercial 3D package, for final rendering. The sample shown in Figure 26 illustrates such a rendering from Maya, in this case a showcase of Virtual Manhattan.

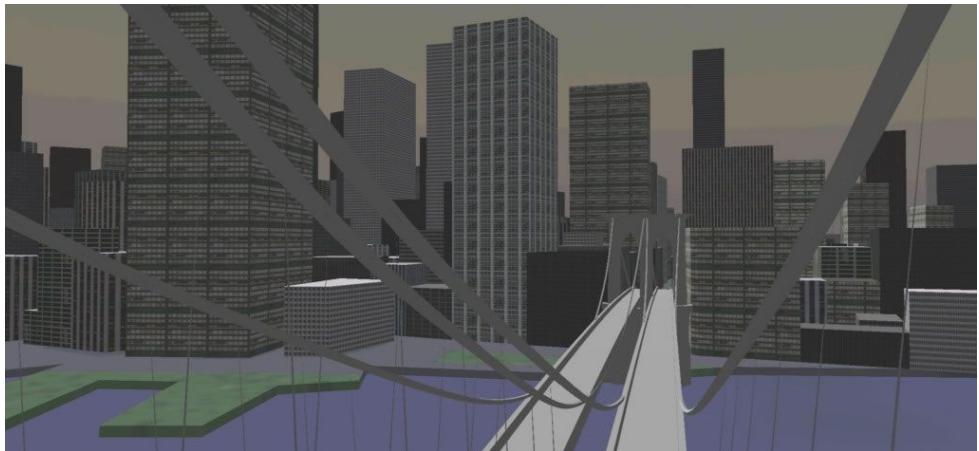


Figure 27: *CityEngine - Virtual Manhattan – DV/reality [Parish and Müller 2001]*

A real-time implementation is available utilizing *DV/reality* software from *Dimension*. *DV/reality* is a large scale visualisation tool designed to run on super computers and distributed rendering applications. There are no real-time rendering features such as level of detail or geometry culling discussed and from the screen-shot of *DV/reality* in action in Figure 27 it is clearly evident that a reduced complexity model is being displayed. (Notice how the buildings appear more similar to the left most image of Figure 24 in contrast to the right).

### 3.4.3 Discussion

**Realism:** The *CityEngine* can create a complex and detailed road network but utilizes real statistical data making the generative capability of the system difficult to assess. The blocks from the road network are divided into realistic and practical lots upon which buildings can be constructed. L-system building generation provides an effective method of generating a realistic cityscape although the resulting buildings are basic. Several different types of buildings including skyscrapers, commercial and residential buildings can be created and green areas are also displayed. Overall a good visual balance is achieved with practical positioning.

**Scale:** Scale does not appear to be an limiting factor for the system and is possibly restricted only by the size of the input data maps.

**Variation:** A range of road networks can be created and examples of different cities are shown including Paris – Circular, New York – Grid and San Francisco – Terrain wrapping. A range of building function types are catered for, but only a limited range of styles are demonstrated. In Virtual Manhattan a convincing clone of New York is shown but it may be more difficult to generate other cities where different architectural styles are required.

**Input:** The minimum input required is a geography map, however all of the samples shown utilize numerous input maps and include statistical data from real world cities. This dependence on real-world data requires the acquisition of geo-statistical data to use the system, which is not desirable. Also, from an evaluation point of view it is difficult to determine which patterns are created by the L-systems and which are created as a result of the input data. All of the samples shown utilize numerous image maps to create realistic output like that illustrated in Figure 26.

**Efficiency:** Road network generation is relatively efficient. The large road network of the Manhattan sample shown in Figure 26 is created in under 10 seconds. The next stage of generation, the building stage, takes longer to complete. Virtual Manhattan requires approximately 10 minutes to sub-divide the road network into lots, construct buildings and create textured faces. It is important to note that although the generation time is documented, the time required for Maya to render Virtual Manhattan is not disclosed, and would likely take substantially longer than both of the previous stages combined.

**Control:** It is unclear how much user interaction is allowed and no interactive features are documented. It appears that the system is controlled by writing specific rules for each region and thus it would require advanced knowledge and expertise to use the system. Control is also limited with the use of image maps and is not suitable for editing by a novice user.

**Real-time:** A real-time demonstration is available using the *DV/Reality* software shown in Figure 27 that displays a simplified version of Virtual Manhattan. *DV/Reality*[DVR] is a visualisation tool designed to provide real-time rendering through the use of high powered graphics workstations and distributed rendering. No documentation on any real-time features of the *CityEngine* is provided.

## 3.5 Agent Based Simulation

Lechner et al. [Lechner et al. 2003] apply an agent based technique to generate cities in their solution titled *CityBuilder*. The system is built on the NetLogo™ platform which is a multi-agent programmable modelling environment based on the Logo programming language and is designed to provide users with a platform to explore emergent phenomena. The city generation is implemented by simulating cities using a set of agents that can model specific city entities such as developers, planning authorities and road builders. The *CityBuilder* system models not only the road network and buildings, but also simulates the growth and development of the city over time.

### 3.5.1 Road Network: Agent Based Simulation

Roads are created from road segments that are assembled according to a grid pattern. Deviation from the pattern is allowed and can be specified via a parameter. A deviation value of zero will result in a strictly uniform grid-like road network, a deviation value near one will result in an organic like network. The interconnectivity of the network can also be altered via constants that dictate the road density and the distance between road intersections.

Input in the form of a terrain height map is required along with a specified water level to determine the legal area in which roads and buildings can be placed. Extra parameters such as road density, grid spacing, and deviation from grid can be adjusted using sliders in the interface shown in Figure 28 to alter the behaviour of the agents. Additionally users can specify certain parameter values for specific areas by painting on the map using a brush similar to that in a simple paint application.

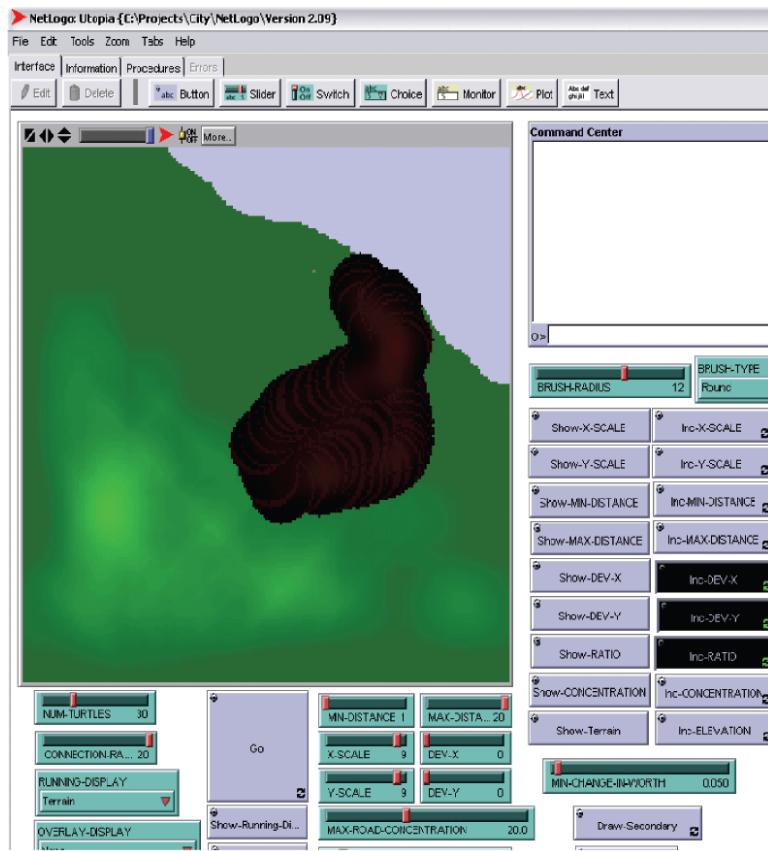


Figure 28: NetLogo™ City Builder Interface [Lechner et al. 2003]

The road segments are created by two types of agents – *extenders* and *connectors*:

- *Extenders* roam around terrain near to existing developments to search for land that is not serviced by the road network. Once that area of land has been discovered, it is assessed according to road density, proximity to existing junctions, and deviation from the start point. Roads follow parcel boundaries and try not to make large changes in elevation.
- *Connectors* roam over the existing road network sampling the distance taken to travel to a point within a given radius using a breadth first search of the road network. If this distance is too long the connector will propose a road segment between the two points, the proposed segment is subject to the same checks as extenders.

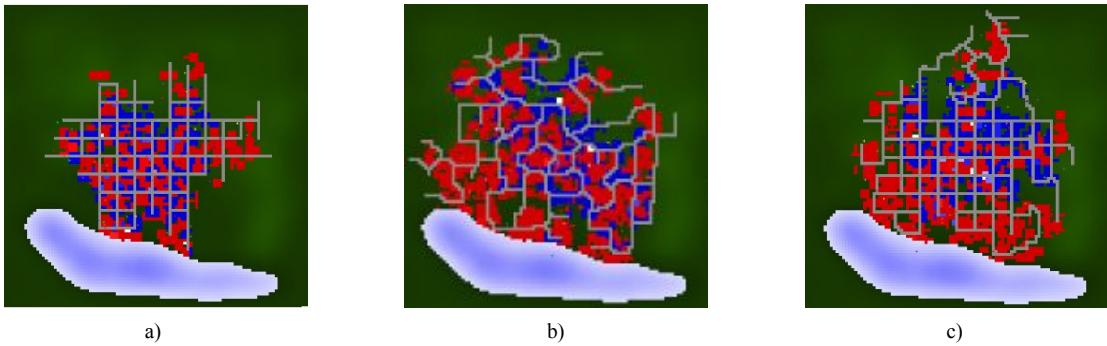


Figure 29: Example output of differing city structures: a) Gridded, b) Organic & c) Mixed Gridded and Organic [Lechner et al. 2003]

Road networks can be viewed evolving in real-time, and the examples shown were created in 15 to 30 minutes. Figure 29 c) shows one of the main strengths of the agent based system by effectively blending between raster and suburban road styles.

### 3.5.2 Buildings: Agent Based Simulation

The generation of land usage for buildings is completed via the interaction of a number of agents but is primarily due to the work of *Developer agents*. *Developer agents* perform the role of urban developers and have similar goals: buy land, request planning permission, build and sell. A rectangular grid of patches represent the world and each patch may be occupied by a building or road. Patches are grouped into parcels under the ownership of the *building agent*. The building agent determines the zoning information of each parcel and tracks attributes of the buildings.

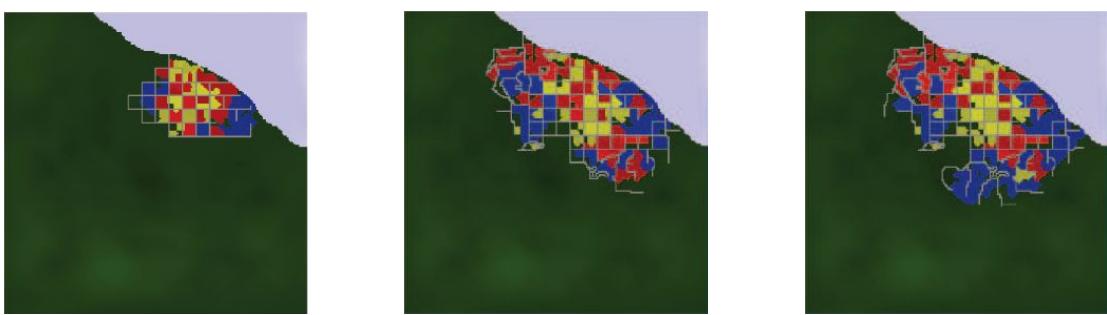


Figure 30: Development Sequence. Yellow is residential, red is commercial, blue is industrial. Roads are grey. [Lechner et al. 2003]

Three distinct developer types are defined: residential, commercial and industrial. All developers seek to increase the value of their land and each developer type evaluates the value of land differently and uses a different set of rules to complete its goals. For example:

residential developers seek land near the less busy areas of the road network in contrast to commercial developers who look for the busiest sections of the road network. Property is reviewed and a site is chosen. A proposal is then prepared that satisfies the clients needs and meets the city's restrictions. The proposal must then be reviewed by the city. A developers' proposal is only successful if it passes the city regulations and makes a net positive impact on the community by providing a service or increasing the value of the land. After this process is complete the developer agent starts again looking for more property. Figure 30 illustrates the evolution of a small city with snapshot images from left to right.

The *CityBuilder* system creates a road network and defines land use that is then used to determine building types but does not generate actual building geometry and textures. The visualization of the city buildings is not a feature of the system but takes place externally in the proprietary SimCity game engine.

### 3.5.3 Discussion

**Realism:** The road network appears realistic and has the ability to effectively transition between road patterns, particularly between central urban areas and less dense suburban areas. No buildings are generated, but the land usage map appears realistic resembling real statistical data similar to that showcased in the *chil.us* project [UrbanLab 2006].

**Scale:** The output created from the system (as shown in Figure 30) is limited in scale and is of a comparable scale to that of a village or small town rather than a city.

**Variation:** A realistic range of road network patterns is displayed although they appear decidedly random. Different zones are supported with commercial zones using rigid block like road structures and residential areas using sprawling roads. Three different land usage and building types are defined: commercial, residential and industrial.

**Input:** A terrain height map and a water level input are required to determine the legal areas in which buildings can be placed. Other input can be specified by the user through the interactive application.

**Efficiency:** *CityBuilder* models not only the structure of a city but also its evolution and as a result of the added complexity the algorithm is computationally intensive and time consuming. A city of only limited scale similar to a village can be generated over a period of

approximately 15 minutes (no hardware specification stated), not including the generation of any building geometry or textures.

**Control:** An innovative feature is available in the form of a paint tool that can be used to paint parameter values on the map. Numerical parameters such as road concentration, deviation and scale can be specified via an interactive application using the various sliders and widgets of the GUI.

**Real-time:** There are no real-time considerations or even a 3D model of the city.

The system could be easily expanded but with an algorithm of high computational complexity it is not suited for real-time procedural generation and at the moment is more suitable for simulation applications.

### 3.6 Template Based Generation

Sun, Baciu et al propose an alternative approach to creating cities in their 2002 paper *Template-Based Generation of Road Networks for City Modeling*. They use a collection of simple templates and a population adaptive template [Sun et al. 2002]. The basic concept of the system is that a road network template is applied to a geographic map as a plan and then the roads are deformed subject to local constraints.

#### 3.6.1 Road Network: Template Based Generation

Several inputs are required in the form of 2D image maps. A colour image map which contains geographical information on land/water/vegetation is required. A grey-scale height map image to specify elevation is required. A population density map is required for the population-based template and is used to determine the varying road network density.

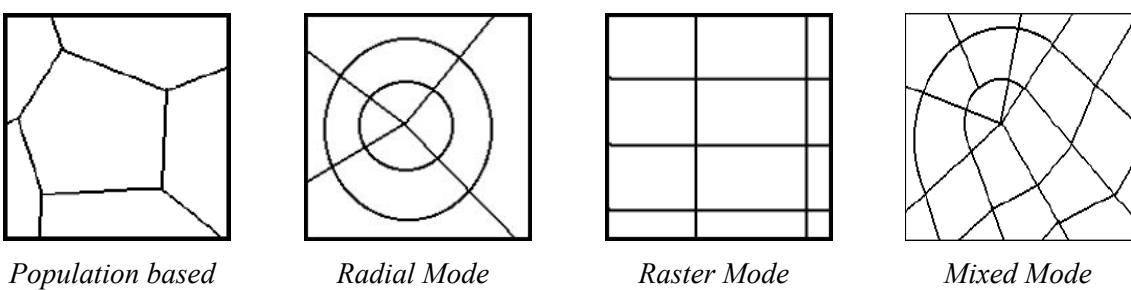
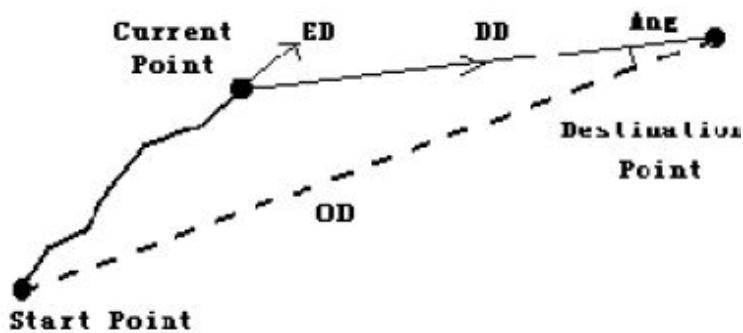


Figure 31: Road Patterns [Sun et al. 2002]

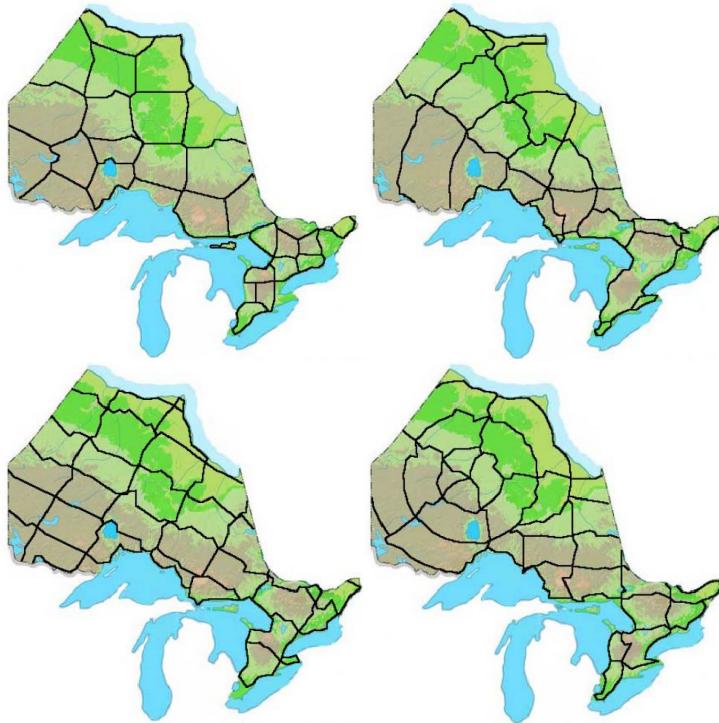
The population-based template is implemented using a Voronoi diagram [Sun et al. 2002]. A road system is created that is representative of the population distribution. Road networks are

suitably dense in highly populated areas and sparse in less populated areas. This is made possible by extracting density points from the population density input map and using the points as input sites for the Voronoi diagram. The edges or cell boundaries from the resulting diagram are used to create the interconnected road network. The other templates use procedural patterns to create the road network. The Raster Mode, Radial Mode and Mixed Mode templates serve as simplistic growing patterns, with roads starting from a defined centre point and growing in an iterative process toward the edges of a bounded area. The Mixed Mode is simply a compound of one or more of the other basic templates.

Templates define only the desired road pattern, and just as road planners must respond to practical constraints, so must the pattern. Roads deviate from the supplied pattern changing direction rapidly to avoid obstacles such as water and curve gradually to avoid large changes in elevation. Roads are created in short steps. Figure 32 shows a diagram of the process in action. At each step the system emits several fixed length radials and selects the radial with the least variation in elevation that is in a legal zone. In the case of a tie between two radials the path of least deviation from the original path is chosen. The angles at which the radials are drawn is restricted by a freedom factor,  $F$ , which limits the maximum angle of deviation for each radial. The final shape of the road is a result of terrain deviation and the selected pattern is followed only as strictly as the freedom factor dictates it to be followed.



*Figure 32: Adaptive Roads [Sun et al. 2002]*



*Figure 33: Results clockwise: Population-Based Template, Radial Mode, Raster Mode and Mixed Mode. [Sun et al. 2002]*

### 3.6.2 Discussion

**Realism:** The applied template technique includes patterns found in real cities and the mixed mode pattern helps recreate the combination of patterns also present in city road networks. However, the results do not achieve the complexity and scale of real city road networks.

**Scale:** The examples shown in Figure 33 demonstrate limited complexity and are insufficient in scale to be classed as city scale road networks.

**Variation:** A choice of four templates is demonstrated and each can be deformed by the random terrain providing limited though varied output.

**Input:** Few inputs are required but include several image maps such as a terrain height maps, a standard geographic map and a population density map for the population based template.

**Efficiency:** No information is provided on the performance of the generation process.

**Control:** A reliance on statistical data and no indication of any user interaction to control the road network generation would imply that this solution is rigid and inflexible.

**Real-time:** No 3D implementation is shown and no performance figures are provided.

## 3.7 Split Grammars

The *Instant Architecture* solution presented by Wonka et al. focuses only on the generation of realistic buildings through the use of a new type of formal grammar called split grammars. These grammars are based on the concept of shape [Wonka et al. 2003].

### 3.7.1 Buildings: Split Grammars

Split grammars are based upon the previous research and principles of shape grammars pioneered by Stiny [Stiny 1980]. A shape grammar is a formal grammar not unlike L-systems but it is based on the fundamental primitive of shapes rather than letters or symbols. Rules or productions map a shape, or a number of shapes to be replaced by another shape, or number of shapes. An initial set of shapes is supplied to start with, and the rules are applied in an iterative manner.

The basic building blocks of the system, and the objects that the grammar manipulates, are simple attributed, parametrized, labelled shapes called *basic shapes*. A large number of rules or productions are required to transform the shapes. For the examples shown in the paper a database of around 200 rules and 40 attributes was assembled. Figure 34 shows an initial state and simple set of sample rules.

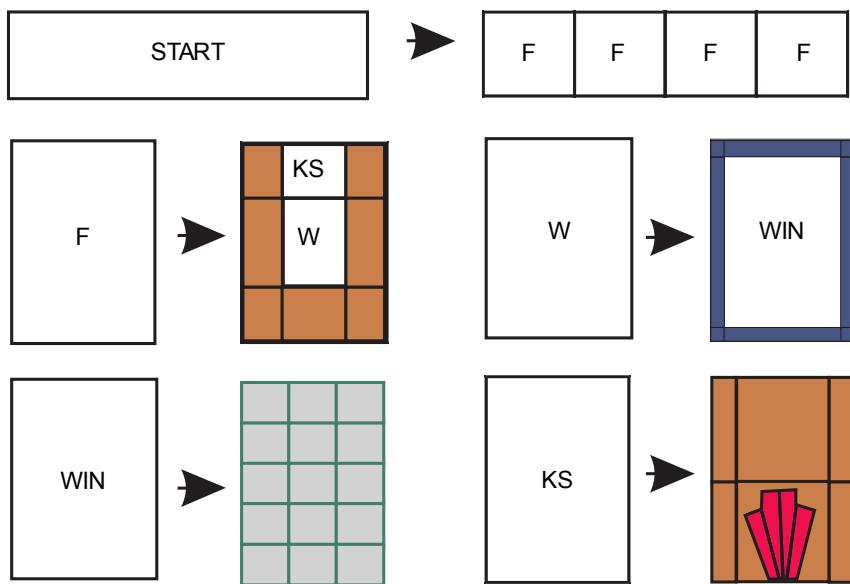
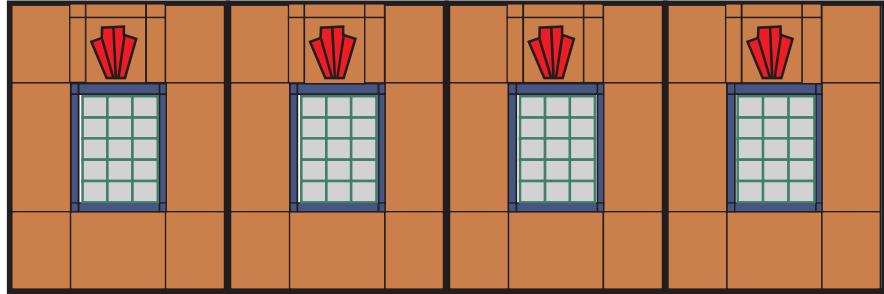


Figure 34: An example of split grammar [Wonka et al. 2003].

An initial starting state is provided and then transformed by means of an iterative process using rules from the database. The rules split buildings into faces, faces into structural

sections and structural sections into components such as windows and so on. This is shown in Figure 34 with the end result shown of the completed derivation in Figure 35.



*Figure 35: Completed derivation of the grammar in Figure 34 [Wonka et al. 2003].*

Attributes assigned to shapes are propagated from the initial state down through the system. The attributes store information about the building like its symmetry, age, use and visual properties. These are later used to render the building but are also used to help match transformation rules and find relevant replacements. In addition, a control grammar is applied that can change the attributes of basic shapes in order to apply spacial design concepts, such as setting the first floor of a building to be a shop or applying a vertical detail to a column of shapes. The resulting building models produced by the instant architecture system contain detailed local features such as window sills but also distinctive building features such as vertical details on the edges of buildings.



*Figure 36: Screen shot of Instant Architecture [Wonka et al. 2003].*

### 3.7.2 Discussion

**Realism:** The split grammar technique produces very realistic buildings even going as far as to effectively recreate different styles of architecture.

**Scale:** The examples shown in Instant Architecture are limited in scale, but serve to demonstrate the strengths of the system by creating a small group of buildings in a town square or centre. A high level of variation is shown in the examples but the number of buildings is limited and is not of city scale.

**Variation:** Building style varies greatly helping to produce very realistic output, however it is not clear how many different buildings types can be produced.

**Input:** The system requires substantial initial input with samples like those shown in Figure 34 requiring a database containing approx. 200 rules and 40 attributes. The authors report that this took around two weeks to assemble. From this database a variety of buildings of different styles can be created and the data can be distributed with the system, without requiring the user to assemble their own dataset.

**Efficiency:** The algorithm, although complex, is quite efficient. It can create buildings of up to 10,000 polygons in around 3 seconds on an Intel Pentium 4 at 2Ghz.

**Control:** No interactive editor or GUI is described, but the split grammar rules can be edited in the database manually. This process is described as non trivial and requires a level of expertise and experience using the split grammars. It could well be a barrier to extending the system. There may also be constraints on the size of the system and the number of rules that it can manage with a reservation expressed that some derived designs may not even make sense if more rules are added.

**Real-time:** The detailed buildings that the system produces can be explored in real-time however the number of buildings on display at any one time is limited. It is clearly a limit of the system with such a high polygon count. Level of detail support would be essential to use the system for real-time applications.

In conclusion, the Instant Architecture solution produces realistic and detailed buildings but may require a level of expertise to operate that restricts it to an academic audience.

### 3.8 Conclusions

In this Chapter we have reviewed previous research into the procedural generation of cities. It is important for us to recognise the areas that can be improved and to identify additional goals. In evaluating the existing research we used a common set of criteria: Realism, Scale, Variation, Inputs, Efficiency, Control and Real-time provisions. The systems adopt unique approaches to the city generation problem and have individual strengths and weaknesses. The highlights include: L-systems road generation, Subdivision lot creation and the interactive user interface of the agent based approach. After completing this analysis, it can be concluded that in general the previous research efforts can achieve a high level of variation, realism and scale but with the exception of the agent based approach do not provide an effective method of control.

The goals for our city generation system to improve upon this existing work are:

- Accessibility – input data such as geo-statistical data or complex architectural rules should not be required to use the system.
- Interactivity – the system should be capable of fully autonomous generation but also facilitate interactive control.
- Real-Time generation – for effective control over generation and to expand the range of applications, the system should generate a city model in near real-time speed.

The goal of our research is to create a city generation system suitable for real-time applications that is capable of creating realistic, varied and large scale cities in an efficient manner. Additionally it should be accessible to non-expert users and provide an effective method of interactive control.

# Chapter 4

# Interactive City Generation Design

In the previous chapter we outlined and evaluated the previous research into city generation systems. In this chapter we present the design of our city generation system. We describe the algorithms and techniques involved in generating primary roads, secondary roads, blocks, lots and buildings.

## 4.1 Overview

A key design goal of the system was that it would allow the user close control over the generation process by means of direct manipulation of the algorithm parameters via an accessible and intuitive visual interface. Furthermore it was regarded as crucial that the results of these manipulations would be computed and rendered in real time so that the user gets immediate feedback on their actions . This is what we mean when we state that *Citygen* is an interactive procedural city generation system. The motivation for this is so the user can engage with the system by moving nodes or changing parameters, see the effect immediately, tweak again, and so on. This iterative design process can be continued as long as necessary until the desired results are obtained.

Similar to previous research, the generation of a city model is achieved via the execution of a number of steps [Parish and Müller 2001]. Each step may employ a different algorithm to: facilitate user interaction, generate a feature or obtain a relevant data structure. To explain the city generation process these steps are grouped into three main stages. Each stage corresponds to one of the constituent city structure components, previously identified by urban design authors [Lynch 1960][Alexander et. al. 1977] and discussed in Chapter 3.

Before describing the individual steps and algorithms applied in the city generation system, we list the main stages and provide a general outline of our design.

1. **Primary Road Network:** serves as the traffic flow arteries of the city, whose function it is to provide transport around the city and from one region to another.
2. **Secondary/Neighbourhood Roads:** are the roads inside the regions enclosed by primary road network. Neighbourhood roads work by providing access for the area to and from the primary road network. Each region may display a distinct style.
3. **Building Construction:** Buildings are situated and constructed on lots which are identified from the enclosed areas within the secondary or local road network.

The essential character of a city and the boundaries of the neighbourhoods are dictated by the pattern of the primary road network (grid-like, radial etc..) and therefore this acts as the starting point of the generation process. The user can create, and manipulate, a graphical representation of a primary road network. Vertices of the primary road graph are called *nodes* and act as control points. They can be added, moved, deleted and so on. The roads which connect these nodes are procedurally generated by the system, mapped to the underlying terrain model, and rendered in real time.

Once a region is enclosed the secondary road generation process is initiated automatically. The system contains a variety of road network patterns and the user can select which road network pattern to apply based on the results desired (for example, grid-like or meandering suburban roads) and assign different patterns to different parts of the city. Several pattern presets are defined for convenience and each can be easily modified to alter the efficiency, connectivity, scale and flow of the resulting road network. Once again, the results are computed and rendered in real time, allowing direct interactive manipulation of the process.

The final stage of generation is the construction of buildings. In order to accomplish this it is necessary to compute building footprints onto which the buildings should be placed. This is done by calculating all of the enclosed areas between the secondary roads and then subdividing them into *lots*. The buildings are then placed within the lots and the shaders and materials are applied to the generated geometry. During the generation process the user can make changes to any stage and see the results of the changes propagate through in real-time.

Taking each of these generation stages in turn, we explain the operation of the system, paying particular attention to the problems that arise, and describing the solutions and algorithms employed to solve these problems.

## 4.2 Primary Road Network

Road networks are represented as *undirected planar graphs* and are implemented as *adjacency lists*. An adjacency list contains an entry for each node and each of these entries comprises of a list of nodes that this node is directly connected to (see Figure 37). This data structure provides an efficient way to store, edit and perform operations on graph representations of road networks.

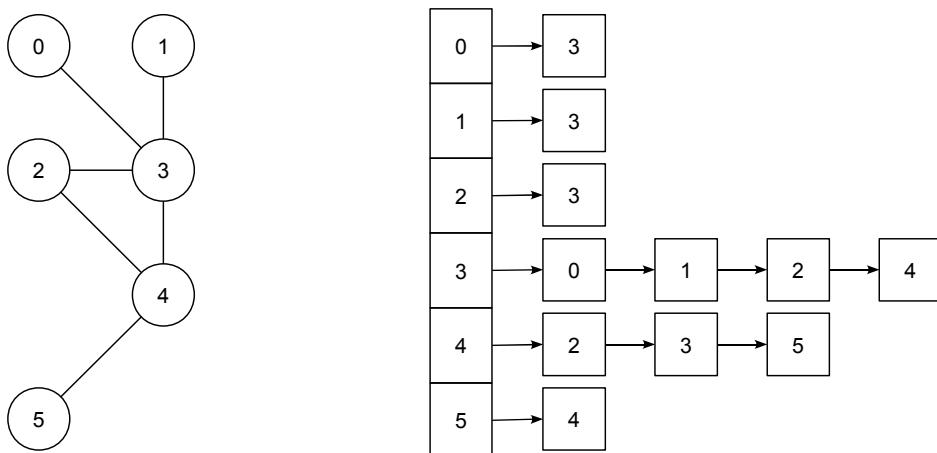
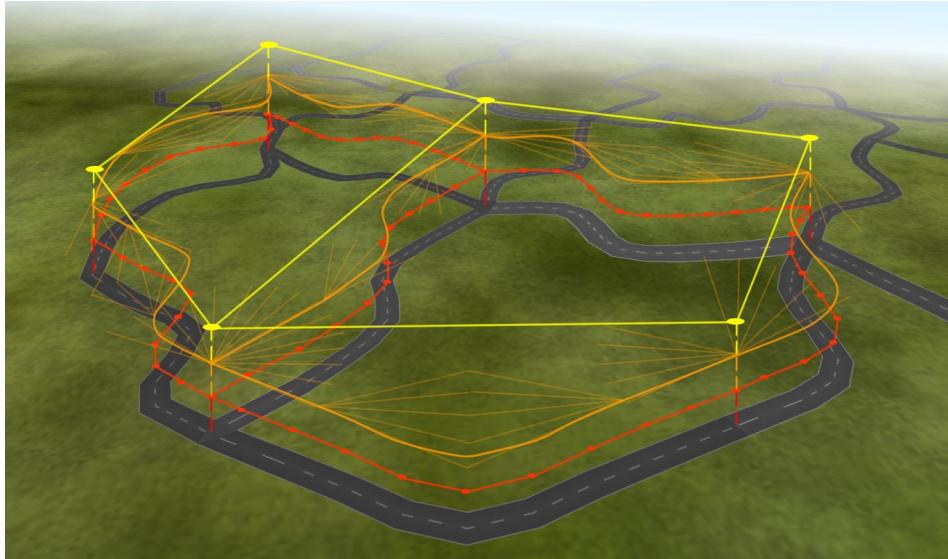


Figure 37 Adjacency List Data Structure

These structures form the basis for all of our road network graphs, including both primary and secondary roads. We use two different graphs to store data solely for the primary road network, shown in Figure 38 and each neighbourhood region also uses an additional graph to store data for its' own secondary road network.

The two primary road network graphs are simply termed, the *high level graph*, and the *low level graph*. The nodes of the high level graph correspond directly to primary road intersections and an edge between two nodes indicates that these nodes are connected together with a primary road. So, in other words it stores the *topological structure* of the primary road network. The low level graph defines the actual path each road takes across the terrain. It will also have nodes corresponding to primary nodes but also many more nodes between these indicating points on the terrain through which the road passes. By keeping the

high level topological road graph separate from the low level graph, we minimise the data set for processing and provide a means for the efficient extraction of connectivity information.



*Figure 38 Primary road network graphs, Yellow: High level graph, Red: Low-level graph, Orange: Plot samples and interpolation spline.*

Nodes of the high level graph function as *control nodes* and can be interactively manipulated within the application in order to adjust the topography of the primary road network. The nodes and edges of the low level graph are then computed by the system using the sampling, plotting and interpolation processes to construct the actual road routes through the terrain. We call these *adaptive roads*. After each manipulation the low level road graph contains the data required to render the roads. The manipulations of control nodes in the topological graph take place in a graphical interface with a real-time display of the final adaptive roads.

#### 4.2.1 Adaptive Roads

The concept behind adaptive roads is to fit road segments into the surrounding environment and ensure that the roads reflect the world in which they occupy. This is accomplished by plotting the road automatically using a sampling technique and various plotting strategies to adapt the roads to the terrain. In practice, to use adaptive roads, the user simply positions the source node and destination node of the road. These nodes correspond to the control nodes of the high level graph. The system then plots the path in real-time, providing immediate feedback and tactile control for the user to fine tune each segment. In addition constraints are employed to maintain the integrity of the road graph. Each proposed segment is automatically snapped to existing infrastructure whenever possible. Aside from aiding the user to rapidly

create a road graph, these constraints ensure that the user cannot create an invalid road segment or leave the road graph in an unusable state. By fitting each road to the environment a sense of cohesion is achieved in the resulting road network, along with increased realism and character.

#### 4.2.2 Sampling

Roads are plotted by starting from a source point and sampling a set of points at regular intervals to define a set of possible paths to the destination. The road graph is stored as an undirected graph and the plotting operation is designed to be commutative (i.e.  $\text{plot}(a \rightarrow b) == \text{plot}(b \rightarrow a)$ ). Therefore the algorithm was designed to operate bidirectionally by sampling simultaneously from both the source node and the destination node, and then finally terminating by meeting in the middle.

Parameters are used to control the size of the samples, the number of samples taken and the maximum deviation allowed from the target direction.

$d_{SAMPLE}$  : sample size

$n_{SAMPLE}$  : number of samples

$\theta_{DEV}$  : angle of deviation

Each control point travels a distance  $d_{SAMPLE}$  and deviates from the direction of the destination point less than an angle  $\theta_{DEV}$ . A set of possible control points is obtained from a fan of  $n_{SAMPLE}$  evenly spaced samples which are evenly distributed over an arc of degree  $2\theta_{DEV}$ .

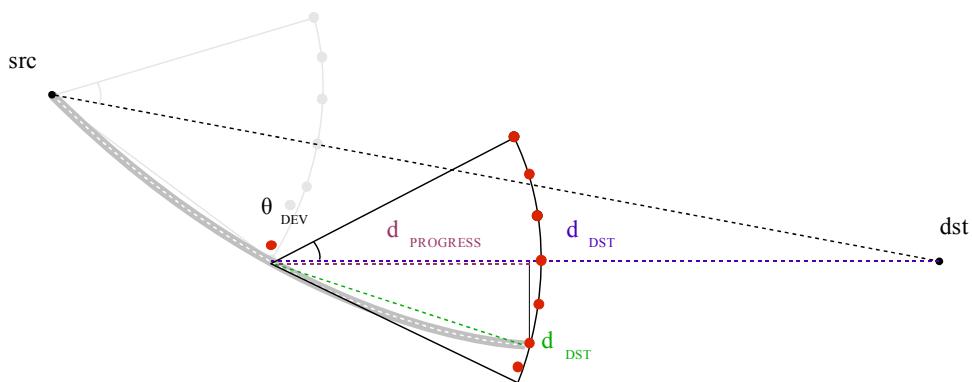


Figure 39 Road interval sampling

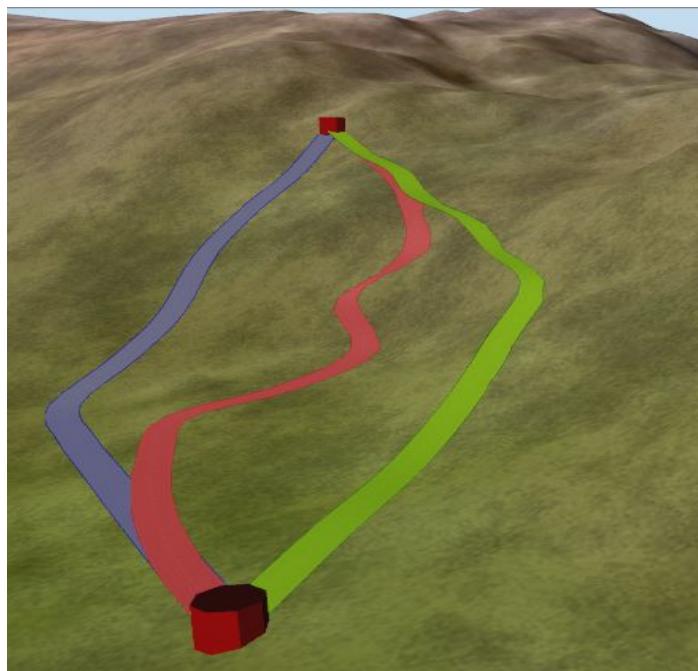
The road plotting process is complete when a sample is within a constant  $d_{SNAP}$  of the destination point, this is guaranteed by ensuring  $\theta_{DEV} < 45^\circ$  and  $d_{SNAP} > d_{STEP} * \cos(\theta_{DEV})$ . By

limiting the deviance angle of the road samples the resulting roads are free to meander when necessary but not without purpose as they are bounded to travel towards their goal.

When the sampling is complete and the path has been plotted, the selected samples are inserted into a spline where fine grained segments can be interpolated and extracted for insertion into the low level road graph for final rendering. A Cubic Hermite spline is used with the Catmull-Rom approach applied to tangent generation [Paeth et al. 1995]. This spline technique was selected because it is easy to compute and produces an exact fit to the sample values. Approximated splines can result in the road terrain intersections.

#### 4.2.3 Sample Selection Strategies

Different selection strategies are employed to choose the samples acquired in the sampling process. Samples are primarily selected from the elevation difference between the sample and the previous plot point. In some strategies additional measurements are taken into account. A number of these different sample selection strategies are demonstrated in Figure 40 and described below.



*Figure 40 Adaptive roads in Citygen. Blue - Minimum Elevation, Red - Least Elevation Difference, Green - Even Elevation Difference.*

##### 4.2.3.1 Minimum Elevation Strategy

This is the most basic strategy in which the sample with the lowest elevation is selected resulting in a road path similar to the route a river or a stream would take.

#### 4.2.3.2 Minimum Elevation Difference

A more competent strategy than the first, this strategy avoids elevation drops or rises, and seeks to maintain an even elevation for the complete road segment. However a problem can occur when constructing roads between a source and destination node with a large elevation difference. In this case the *Minimum Elevation Difference* strategy will avoid the required ascent and descent until the last step when it has to join in the middle. On certain terrains this can result in a road with two smooth road sections and a steep section joining the two.

#### 4.2.3.3 Even Elevation Difference

To improve on the *Minimum Elevation Difference* strategy a technique with some more foresight was required. This is the impetus for the *Even Elevation Difference* strategy which aims to plot an even and smooth path for the road by looking ahead and re-evaluating the elevation goal as it progresses. This strategy operates by calculating the elevation difference between the current position and goal position. Based on the progress being made towards the goal, the algorithm seeks to ascend or descend an even portion of total elevation for each plot point. This strategy operates by selecting the sample with the minimum difference between distance covered and elevation ratio, and the goal distance and elevation ratio.

$$\text{MinAbs}(\text{elevation}_{\text{STEP}} / d_{\text{PROGRESS}} - \text{elevation}_{\text{DEST}} / d_{\text{DST}})$$

Each sample aims to cover an even portion of the total road elevation. The resulting roads are smooth curves that meander when necessary weaving through hilly terrain and searching for even paths to ascend or descend large elevation differences.

### 4.3 Secondary Road Generation

Secondary roads service the local area within districts by providing access to and from the primary road network. In our system, districts are the regions of the terrain enclosed by primary roads. We call these *city cells* and they form the basic units upon which the secondary road generation process operates. The generation of the secondary road network within cells is accomplished using a growth based algorithm similar to the L-systems technique applied extensively to the generation of natural phenomena. There are important aspects to this process that will be described:

**City Cell Extraction:** How do we extract the cells from the primary road network graph?

**Secondary Road Growth:** How do we generate a range of road patterns in the secondary road network within these cells?

**Snap Algorithm:** How do we efficiently obtain information on the intersection status and proximity to the existing roads in the network?

#### 4.3.1 City Cells

City cells are formed from the enclosed regions of the primary road network. These regions can be determined by extracting the closed loops from the high level primary road graph. To extract the cells we execute a *Minimum Cycle Basis (MCB)* algorithm [Eberly 2005] on the primary road network graph and store the cell data in self contained units. Our design enables the efficient parallel execution of road generation within cells by ensuring that all cells are self contained and that the shared data is minimal.

The Minimum Cycle Basis is defined as the unique set of minimum cycles in a graph that all other cycles can be constructed from. There are numerous algorithms available to compute the MCB but few take the position of vertices into account, instead operating solely on the structure of the graph.

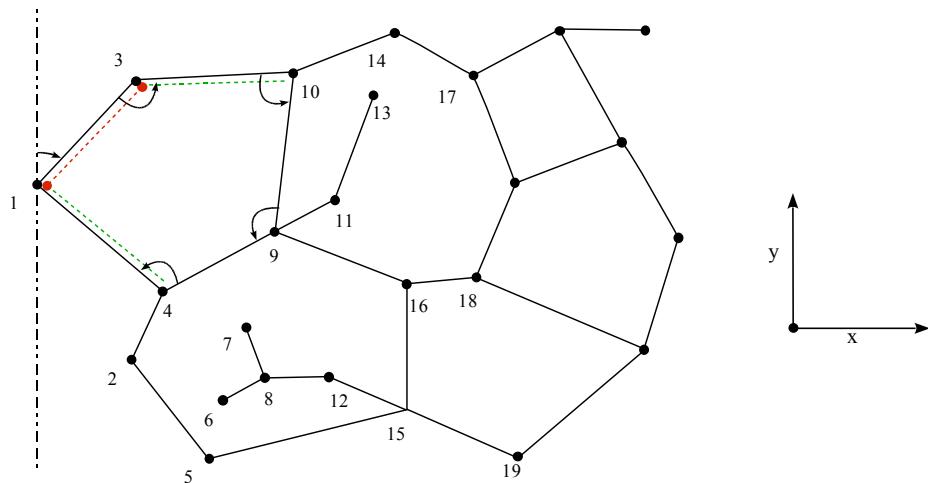


Figure 41: MCB algorithm illustration

The MCB algorithm used in *Citygen* is that described by David Eberly in [Eberly 2005]. Our implementation of the algorithm works by first sorting the nodes by the x location and then extracting cell cycles in a left to right order. Cycles are extracted by using the clockwise orientation of edges to prioritise exploration paths. As cycles are found they are marked and

removed from the graph so as not to influence further searches. Filaments are ordered sequences of vertices where the end vertices are either end points or branch points and those in the middle have exactly two adjacent vertices. In Figure 41 the cycle  $\{1,3,10,9,4\}$  is first extracted and the edges are marked cycle edges, then edge  $\{1,3\}$  is removed and filaments connected to vertices 1 and 3 are removed and not stored as they are not part of any further cycles. This brief outline is included to provide an insight into the operation of the algorithm, for a detailed explanation of the algorithms operation see [Eberly 2005].

After the cycles and filaments have been extracted, their containing cycles are determined and the data is grouped and then stored in a cell data structure. Each cell is self-contained and consists of a private road graph with the boundary cycle, filament roads and a small set of parameters to control road generation. As a result of this self contained design, it is possible for secondary road generation to be executed efficiently in parallel. Parallel generation of cells is currently implemented for multi-core systems running *Citygen*, but has a clear path to be extended further to use GPGPU programming.

### 4.3.2 Secondary Road Growth

Once a city cell is created, the generation of secondary roads is initiated within it using a growth based algorithm. The choice of using a growth based algorithm was based on success of the prior L-system work of Prusinkiewicz on plants [Prusinkiewicz and Lindenmayer 1990] and Parish and Müller on city generation [Parish and Müller 2001]. Although our application does not use L-systems it shares the concept of parallel growth. Our generation algorithm is distinct in that it is computationally efficient and contains a number of optimisations to enable it to run in real-time. The generation is flexible, producing a wide range of output and functions by adding road segments in a parallel fashion similar to organic growth.

Road construction begins from the bordering primary boundary roads and grows inwards in a parallel fashion. The starting point for the initial road segment is obtained using a deviated midpoint from a selection of the longest cell boundary sides. Once the initial segment's position and direction are calculated they are placed on a queue for processing. The road generation process is sensitive to existing infrastructure and new segments can connect to, and extend, existing roads. Information about the status of a proposed segment in relation to the road graph is obtained via an extensive snapping algorithm. This algorithm (described in

detail in section 4.3.2.1) provides information on intersections and on the proximity to neighbouring segments and nodes. Using this information, the road growth algorithm can make an informed decision to modify each proposed segment and join it to the existing network, or discard it if it does not meet the criteria set by the cell control parameters. Shown below in Figure 42 is a pseudo-code description of the algorithm.

```

// calculate initial road segments
for each boundaryRoad in longest(boundaryRoads)
    midPoint = calculate deviated road midpoint
    sourceNode = insertNode(boundaryRoad, midPoint)
    roadDirection = calculate deviated boundaryRoad perpendicular
    if placeSegment(sourceNode, roadDirection, ref newNode)
        nodeQueue.push(newNode, roadDirection)

// process road growth
while(nodeQueue is not empty)
    sourceNode = nodeQueue.front().node;
    roadDirection = nodeQueue.front().direction;
    nodeQueue.pop();

    for ParamDegree iterations
        newDirection = rotate roadDirection by i*(360° / ParamDegree)
        deviate newDirection
        if placeSegment(sourceNode, newDirection, ref newNode)
            nodeQueue.push(newNode, newDirection)

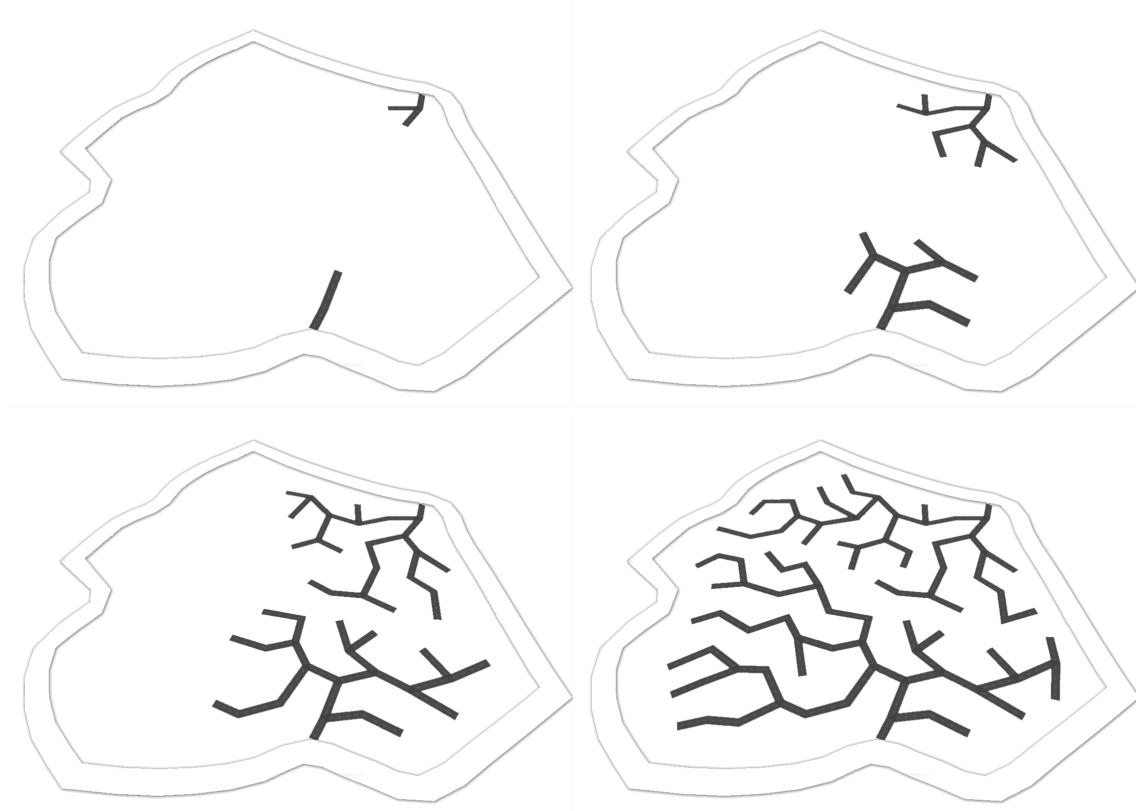
// function to place road segment, returns true on success
placeSegment(sourceNode, roadDirection, ref newNode)

    roadLength = calculate deviated ParamSegmentLength
    targetPos = sourceNode.pos + (roadDirection * roadLength)

    switch(snapInformation(ParamSnapSize, sourceNode, targetPos))
        case: no snap event
            targetNode = createNode(targetPos, roadLength)
            createRoad(sourceNode, targetNode)
            return true
        case: road snap event
            if random value is less than ParamConnectivity
                snapNode = insertNode(snapRoad, snapPoint)
                newNode = createRoad(sourceNode, snapNode)
                return true
        case: node snap event
            if random value is less than ParamConnectivity
                newNode = createRoad(sourceNode, snapNode)
                return true
    return false

```

Figure 42: Road Growth Pseudo-Code(Control Parameters shown underlined and orange)



*Figure 43: Road Growth 10, 100, 300 & 1000 steps.*

Each cell specifies a *control parameter* set which is used by the growth algorithm to control the generation process. Example parameter sets are shown in section 5.5.1. Control parameters used for road generation include *segment size*, *degree*, *snap size* and *connectivity*:

- *Segment size* controls the size of each proposed segment and hence granularity of the neighbourhood road network. Small segment sizes result in tightly packed streets whereas larger ones will give a more sparse road network.
- *Degree* controls the number of times a road branches at any given node.
- *Snap size* alters the distance threshold used to connect to existing infrastructure and hence influences the efficiency of the road network.
- *Connectivity* changes the probability that segments will connect together thus affecting road network flow.

A *deviance parameter* partners each control parameter and enables relevant noise to be introduced by altering the parameters at each step of road generation. A seeded random generator is used to ensure that all generation is 100% reproducible.

Using this relatively simple growth algorithm along with a concise parameter set, we can generate a range of different road patterns which can be specified on a cell by cell basis. These patterns range from regular raster patterns typical in modern city centres, to irregular industrial patterns and even random meandering patterns of organic development.

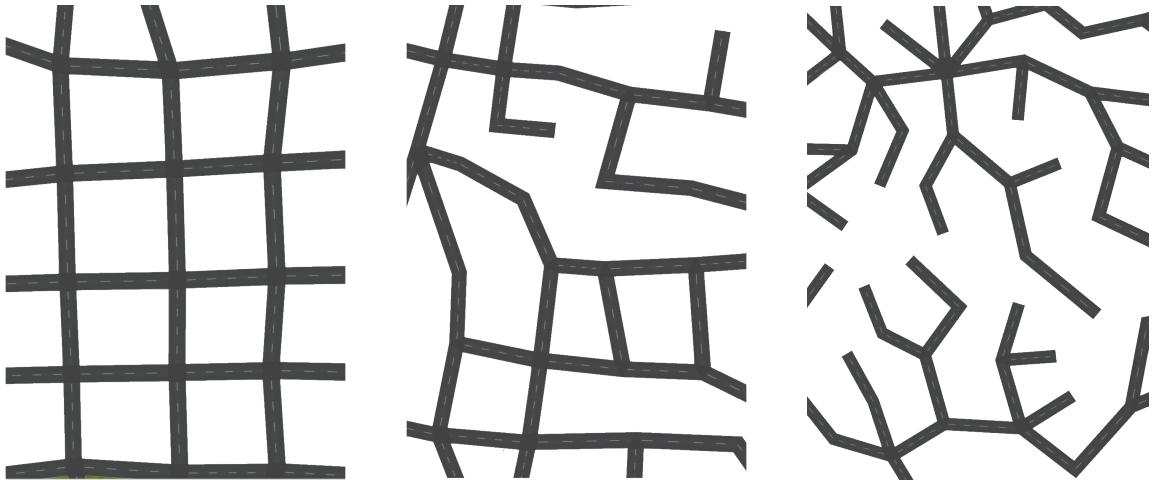


Figure 44: Early road growth patterns: Raster, Industrial and Organic

#### 4.3.3 Snap Algorithm

The snap algorithm provides information on the proximity and intersection status of a proposed segment relative to the existing local road network graph. Proposed segments are then refined by the road growth algorithm using this data. The snap algorithm is called frequently by the growth algorithm and has a strong influence on road generation performance in the system.

The snap algorithm receives the parameters:  $a$  – the root node,  $b$  – the target position and also a specified *Snap Size*.

Figure 47 illustrates the resulting snap area (shaded in grey) for the proposed segment  $ab$ , with the *Snap Size* specifying the width and radius of the snap area. A *Snap Event* is created for any segment that causes an intersection or is within the defined snap area. Several events may occur on the proposed segment  $ab$ , but each of these events are prioritised by their distance to the root  $a$ , with the closest events being the most important.

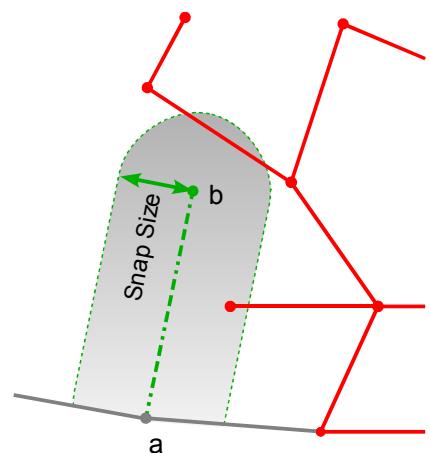


Figure 45: Snap area shaded in grey.

In order to calculate snap information for the desired snap area illustrated in Figure 47 a number of proximity and intersection tests are required. These series of tests are computationally intensive, but a number of optimisations have been devised that improve the performance significantly. We will now describe these optimisations and outline the operation of the required tests.

#### 4.3.3.1 Extended Bounding Box Exclusion

The first step to optimising the speed of the snap algorithm is to minimise the number of segments that require testing. This is achieved by using a technique of minimal computational expense, in the form of a two dimensional axis-aligned bounding box test. The snap algorithm must not only report on the intersection status of each proposed segment, but also on the proximity to existing segments. For this reason an extended bounding box is used (see Figure 46). The extended bounding box is calculated by simply extending the standard bounding box for only the proposed segment by the distance specified in the *Snap Size* parameter. Using this method we can guarantee that segments which may not cause an intersection, but may infringe on the snap area, are not excluded prematurely.

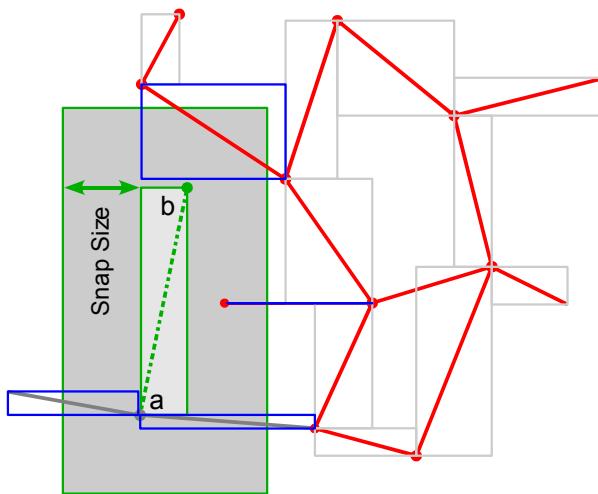


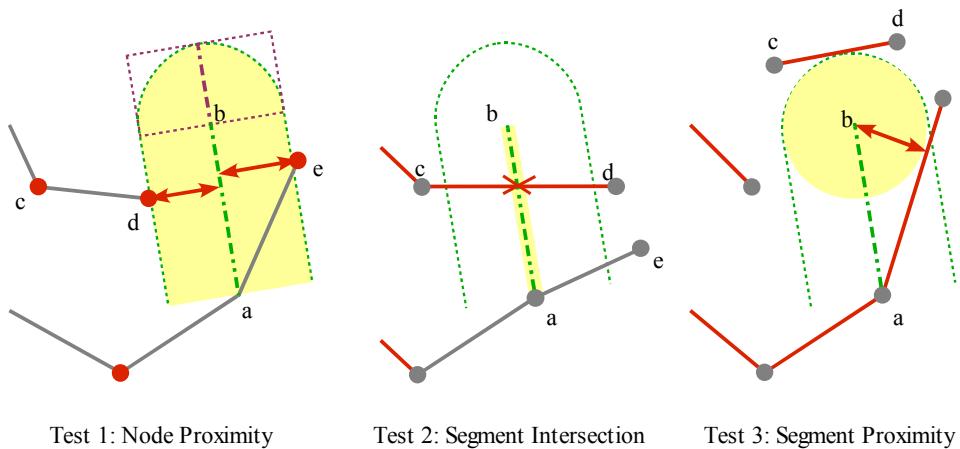
Figure 46: Extended Bounding Box Test:

The extents and position for each segment bounding box can be obtained by calculating a segment half vector for the extents value, and then using this also to determine the midpoint for the position value. The execution of the extended bounding box test excludes up to 90% of road segments from further testing in a typical road network thus providing a considerable performance improvement with little expense.

### 4.3.3.2 Testing Procedure

As previously discussed a number of different proximity and intersection tests are required to evaluate the snap area (shown in Figure 45). The snap algorithm testing procedure is composed of three ordered tests:

1. Proposed segment to existing node proximity test.
2. Proposed segment to existing segment intersection test.
3. Proposed node to existing segment proximity test.



*Figure 47: Snap Algorithm Tests: the active elements in each test are coloured in red.*

These three tests define the procedure, that segments not excluded by the extended bounding box test are subject to. Due to the frequency of which the Snap Algorithm is executed, further optimisation is still important to maintain a high level of performance. A basic optimisation is implemented by ordering the tests according to the priority of the snap events they can provide. As previously discussed in section 4.3.3, the priority of a snap event is dictated by its position relative to the root  $a$  of a proposed segment  $ab$ , with the events closer to the root assigned the highest priority. Since only a single snap event is returned, new snap event of lower priority can be ignored. Tests 1 & 2, the node proximity and segment intersection tests are executed first as they can provide snap events along the full length of segment  $ab$ . In comparison the third test, segment proximity, can only provide snap events at the target position  $b$ , which is assigned the lowest priority. So, if a snap event occurs in either of first two tests, then the entire series of third tests are not executed as a snap event closer to the root,  $a$ , cannot be provided.

### 4.3.3.3 Test 1: Node Proximity

The node proximity test checks the distance between the existing nodes in the road graph and the proposed segment  $ab$ . To calculate the distance we use an algorithm which calculates the distance from a point to a line [Rourke 1998]. A pseudo-code description is listed below:

```
// distance from point p to line ab
distanceToLine(a, b, p)
    ab = b - a
    ap = p - a
    r = (ap).dotProduct(ab) / ab.squaredLength()
    s = (ap.perpendicular()).dotProduct(ab) / ab.squaredLength()
    return abs(s) * ab.length()
```

Figure 48: Distance from point to line pseudo-code

In addition to calculating the distance between nodes and the proposed segment, this algorithm can also provide information on the location of each node. The values of  $r$  and  $s$  indicate the position of each node relative to the proposed segment  $ab$ .

- $r$ , indicates the position of a node relative along the length of  $ab$ . For example: if a node is located at  $a$ , the  $r$  value will be 0 and if it is located at  $b$ , the  $r$  value will be 1. Any node where  $0 \leq r \leq 1$  is located within the grey band illustrated in Figure 49.
- $s$ , indicates the position of a node relative to the perpendicular of  $ab$ . For example: if a node is located on the left of  $ab$ , the  $s$  value will be less than 0 and if it is located on the right of  $ab$ , the  $s$  value will be greater than 0.

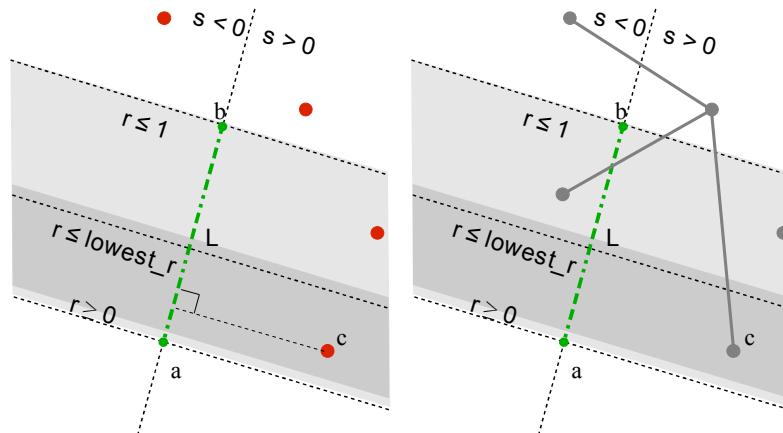


Figure 49: Node distance function scalars  $r$  &  $s$

The values of  $r$  and  $s$  are used by the node proximity test to enable a number of optimisations:

The  $s$ -value is proportional to the distance that a point is located from the line segment. Once the *Snap Size* is expressed in terms of  $s$ , the full distance does not need to be calculated and the computation for node proximity tests can be reduced.

The  $r$ -value indicates the location of a node along the line  $ab$  and is important in treating how possible snap events are processed. Normal node proximity events require that the condition  $0 \leq r \leq 1$  hold true. This ensures that the snap event occurs within the line segment  $ab$  and the node is located within the lines passing through  $a$  and  $b$  perpendicular to  $ab$ .

If the  $r$ -value exceeds 1 an additional test may be triggered. If  $1 > r \leq 1 + (\text{Snap Size} / ab.length)$ , then the node is located in the purple area, shown in Figure 50. In this case an additional point to point distance test is triggered which checks whether a snap event occurs in a circular area around  $b$ . Executing this additional distance test only when a snap event is probable reduces the overall computation required.

Finally, because snap events are prioritised closest to the root, and the  $r$ -value indicates the location of a node on segment  $ab$ , this value can also be used as a priority indicator. By using the  $r$ -value as a priority indicator we remove the need for any additional computation to determine snap event priority. If a snap event occurs, the  $r$ -value of the offending event is stored. Since only higher priority events are significant, this new  $r$ -value sets the benchmark for all future tests. A test is only executed if  $r$  is less than the lowest previous  $r$  and greater than zero. For example: if a snap event occurs on the segment body  $ab$ , the  $r$ -value obtained will lie between 0 and 1, so any snap events beyond 1 will now be irrelevant, and so the additional distance test will not be executed.

If no snap events occur in the node proximity test the values of  $r$  and  $s$  for each node are still stored. These values can be used by future tests to implement optimisations.

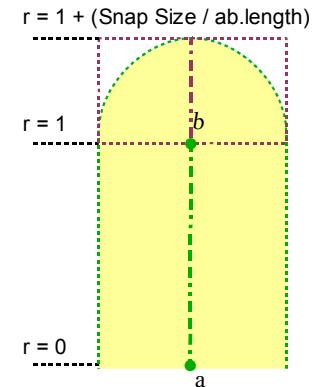


Figure 50: Test 1: Node Proximity:  $r$ -values

#### 4.3.3.4 Test 2: Segment Proximity

The segment intersection test checks for intersections between existing segments in the road graph and the proposed segment  $ab$ . Efficient execution is achieved by executing the full segment intersection test only when an intersection has been determined probable.

The probability of an intersection is calculated by using the data obtained in Test 1. As explained in the previous section, the node distance data defines the position of nodes relative to the proposed segment. Therefore, if the nodes of the segment being tested contain s-values of a different sign, then they must lie on different sides of the proposed segment and an intersection on the line  $ab$  occurs. Also, if both node r-values place the nodes' location within the proposed segment  $ab$  or on opposing extensions of  $ab$ , then an intersection is probable within the bounds of the line segment  $ab$ . The application of this optimisation results in a 98% reduction of all segments intersection tests in a typical city neighbourhood cell.

Once an intersection has been determined as probable the following segment intersection test is executed. A pseudo-code description of the algorithm is listed below [Rourke 1998].

```
// check for intersection between line segment ab & cd
lineIntersection(a, b, c, d, r, s)

ab = b - a
cd = d - c
denom = (ab.x * cd.y) - (ab.y * cd.x);
if(denom == 0) return false;           // lines are parallel
ca = a - c
r = ((ca.y * cd.x) - (ca.x * cd.y)) / denom; // r = pos on cd
s = ((ca.y * ab.x) - (ca.x * ab.y)) / denom; // s = pos on ab
if(r == 0 && s == 0) return false; // lines are coincident
return true;
```

Figure 51: Distance from point to line pseudo-code

In a similar style to the operation of test 1, the distance between a snap event and the root node is not calculated explicitly and a scalar r-value is again used as the priority indicator. The same benefits apply in this stage. No additional computation is required to determine snap event priority and the r-value provides another means of discarding unnecessary tests that cannot yield an event of significant priority and lower r-value.

#### 4.3.3.5 Test 3: Segment Proximity

The segment proximity test checks the distance between the existing segments in the road graph and the target position defined by  $b$ . To calculate the distance from a segment to a point the same algorithm as described in Test 1 and shown in Figure 39 is used [Rourke 1998].

This is the last test executed by the Snap Algorithm procedure and is only called if no snap event occurred in the previous tests. This test is scheduled last because of the fact that it can only detect snap events located at the end of the proposed  $ab$  which is assigned the lowest priority. No special optimisations are implemented in this test and the check simply calculates the distance between node  $b$  and every other line segment.

#### 4.3.4 Summary of Secondary Road Generation

We have discussed City Cells, Road Growth and the Snap Algorithm. City Cells are extracted from the primary road network and each cell maintains a local road network graph thus reducing the relevant data set for secondary road generation within the cell. Road Growth is achieved by using a simple growth algorithm with a minimal *control parameter set*. Even though the secondary road network is divided among the City Cells the network data set is still of a significant size. The Snap Algorithm employed by the growth algorithm is called frequently by the growth algorithm and its efficient operation is critical for road generation performance. The snap algorithm applies an extended bounding box test to filter the road segments for testing. Then, three tests are implemented with a number of optimisations to enable the snap algorithm to provide proximity and intersection information efficiently. Once the secondary road network is complete the next stage of generation is the identification of lots and the construction of building footprints and geometry.

### 4.4 Building Generation

The building construction stage of Citygen is accomplished in three stages. Firstly the enclosed regions are extracted from the secondary road graph by applying the Minimum Cycle Basis algorithm as described previously in Section 4.2. Secondly the lots are identified by splitting the regions into minimal tracts or parcels of land suitable for development. Thirdly and finally the building footprints are inset from the lot boundaries and the building geometry is constructed and textured.

### 4.4.1 Blocks

Blocks represent the enclosed regions of the secondary road network. The role of the block is to add any additional geometry such as footpaths, signposts, traffic lights or post boxes onto the region. Currently only the footpaths are added. These footpaths are constructed by first insetting the cell boundary by the value specified in the *footpath width* parameter. The original cell boundary and the inset boundary are then combined to create an internal boundary strip. This strip is then extruded upwards by the specified *footpath height* parameter. The boundary cycle is finally toured and the length of each edge is measured. An accumulated length value is stored in each vertex and transformed to form the UV texture coordinates. These texture coordinates ensure that the footpath textures are tiled correctly.

#### 4.4.1.1 Straight Skeleton Inset

The boundary consists of small segments interpolated from the primary road network and a diverse mix of patterns generated in the secondary road network. For this reason a basic inset operation can frequently fail. This failure results in undesirable artefacts in the inset boundary. Hence, an algorithm was used that can take the potential problems into account. The technique used is that of the Straight Skeleton, a concept first proposed by Aichholzer et al. in the paper titled “A Novel Type of Skeleton for Polygons” [Aichholzer 1995].

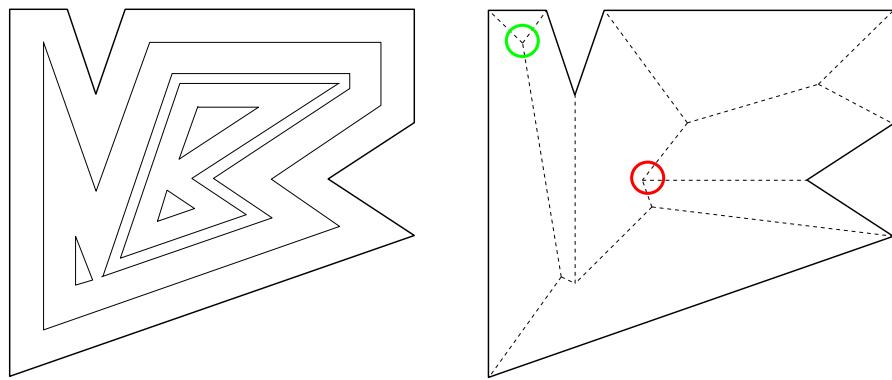


Figure 52: Straight Skeleton [Aichholzer 1995]

A straight skeleton is defined as the union of pieces of the angular bisectors traced out by the polygon vertices during the shrinking process [Aichholzer 1995]. During the shrinking process two types of events can occur:

- An *Edge Event*, occurs when an edge shrinks to zero and the neighbouring edges become adjacent. An example of this event is circled in green in Figure 52.

- A *Split Event*, occurs when an edge is split by a reflex vertex, thus dividing the whole polygon into two. An example of this event is circled in red in Figure 52.

Following the description of the concepts by Aichholzer et al., an implementation of the algorithm was later provided by Felkel and Obdrzalek when they published the “Straight Skeleton Implementation” in 1998 [Felkel and Obdrzalek 1998]. This is the implementation that our algorithm is based upon. The algorithm obtains the Straight Skeleton of any polygon in  $O(nm + n \log(n))$  time, where  $n$  is the total number of vertices, and  $m$  is the number of reflex vertices. This algorithm works by storing the polygon in a data structure called a *Set of circular Lists of Active Vertices*, or *SLAV* for short. This data structure is used to store the polygon vertices and the Straight Skeleton structure. Events are then processed in the order they are encountered.

There are two major differences in our implementation of the inset algorithm and the algorithm described by Felkel et al. Firstly the inset algorithm does not generate a complete Straight Skeleton. Instead, the algorithm generates a partial skeleton by tracking the inset progress and detecting just enough events to inset the polygon by the required amount. Secondly the algorithm operates in  $O(n \log(n))$  time. This is achieved, not by improving the complete algorithm, but by customising the algorithm to our application and ignoring *Split Events*. During testing it was found that the *Split Event* rarely occurs and so can be removed without causing significant problems. This approach has the added benefits of simplicity and performance but has a dangerous side effect of creating potentially invalid regions. However, efficient error checking is employed in the later stages to help correct any errors that may occur during the inset process. The result is an inset algorithm that is balanced by creating few errors yet remaining computationally efficient.

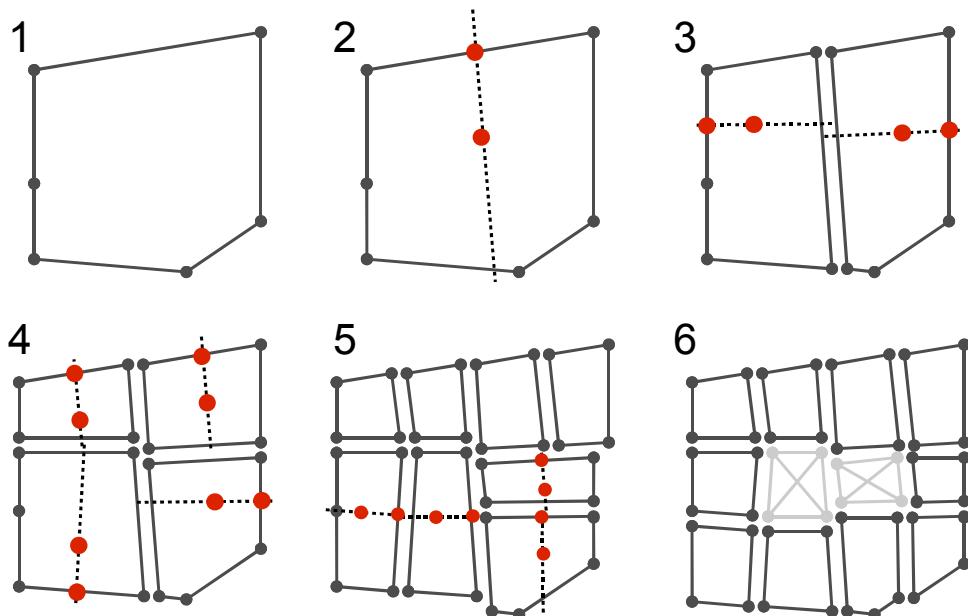
#### 4.4.2 Lot Subdivision

The lot subdivision process operates on the region boundary defined in each city block. A subdivision algorithm is applied which is based on that described by Parish et al. [2001]. Also, a number of extra features and optimisations have been implemented to extend this technique. These features include:

- a more even and accurate method of dividing lots
- an optimisation to orientate lots perpendicular to their access roads
- the ability to process both concave and convex regions

- the addition of individual lot width and lot depth parameters

The lot subdivision process operates by measuring the regions and dividing them recursively until all of the regions are of a size approaching the values specified in the cell control parameters. Two cell control parameters, lot depth and lot width, along with deviation values are used to control the subdivision process. The lot subdivision algorithm recursively splits each region into two or more sub-regions using a split line to define each split operation. These split lines are calculated by obtaining a perpendicular from a point near the middle of the longest side of each region. Deviation is used to introduce noise into the subdivision process. This application of noise results in a more natural lot distribution. The deviation amount can be specified in the cell control parameters and a range of variation between regular and randomised lot sizes is possible. Also, throughout the subdivision process road access information is maintained in the region data structures. This information is used to determine if regions have direct access to the road network. Those regions without direct access to the road network are not considered suitable for building development, and are discarded, or used for green space.



*Figure 53: Illustration of the Lot Subdivision Algorithm*

Shown above in Figure 53 is an example of the lot subdivision algorithm dividing a regular convex polygon typical of an inner city block. The red points indicate the split points used to define the split line shown as dashed lines. The light grey regions shown in the last stage

represent the lots that do not have direct road access and are excluded. For more detail on the lot subdivision algorithm, a pseudo-code description is shown below in Figure 54.

```

subdivideLots(blockRegion)
    regionQueue.push(blockRegion)

    while(regionQueue is not empty)
        region = regionQueue.front()

        // calc the longest road edge and split size
        longestEdge = region.getLongestRoad()
        if(longestSide.length < lotWidth)
            // calc the longest non-road edge and split size
            longestEdge = region.getLongestNonRoad()
            if(longestSide.length < lotDepth)
                // if lot is small enough, add completed region
                outputRegions.add(region)
                regionQueue.pop()
                continue
            else
                splitSize = lotDepth
        else
            splitSize = lotWidth

        // calculate the split points
        sp1 = calcSplitPoint(longestEdge, splitSize, lotDeviance)
        sp2 = sp1 + longestEdge.perpendicular()

        // split and process the new regions
        newRegions = SplitRegion(region, sp1, sp2)
        regionQueue.pop();
        foreach(Region r in newRegions)
            if(r.hasRoadAccess())
                regionQueue.push(r)      // add to processing queue
            else
                delete r              // discard region

    return outputRegions

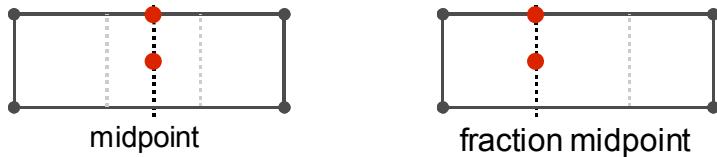
```

*Figure 54: Lot Subdivision Algorithm pseudo-code*

#### 4.4.2.1 Even and Accurate Lot Subdivision

Using a binary split operation limits the accuracy to the nearest power of two. By taking the target size into account we can offset the division point for more even and accurate divisions for odd sized lots. The function called to calculate the deviated mid point for the longest edge is of key importance to the generation of evenly and accurately sized lots. Although a lot deviance parameter can be specified in the cell control parameters, the ability of the original

lot subdivision algorithm to deliver evenly sized lots is restricted. When testing the initial version of the lot subdivision we quickly found that by using a deviated midpoint the accuracy of the lot subdivision was limited to the nearest power of two.



*Figure 55: Split Point Calculation*

In the example shown above the region is 60 units wide and 20 units deep. The desired lot width and lot depth is 20 units and the deviance is 0. The desired result is obvious, three lots should be obtained. However, if the midpoint is used as the first split point in the region, four lots, not three lots will be returned. To solve this problem we take the measurement of the longest side and determine the number of splits that may be required. Using this number we can then select the middle-most fraction for an accurate split. Although it is not evident in the example above, it is very important that the middle-most split fraction is used, otherwise thin slices are cut from large regions and lot distribution is compromised. This feature is necessary for cells which specify low lot deviance values and enables accurate and evenly sized lots to be obtained. For cells with high deviance values the application of noise is not negatively affected by the implementation of this feature. Shown below is a short snippet of pseudo-code that demonstrates how this feature can be implemented.

```

calcSplitPoint(longestEdge, splitSize, lotDeviance)
    factor = Round(longestEdge.length / splitSize)
    fraction = 1/factor
    midPosition = Round(factor/2) * fraction

    // calculate longest edge vector src → dst
    longestEdgeVec = longestEdge.dstPos() - longestEdge.srcPos()

    return longestEdge.srcPos() + longestEdgeVec *
        (midPosition + (lotDeviance * (random() - 0.5) * fraction))

```

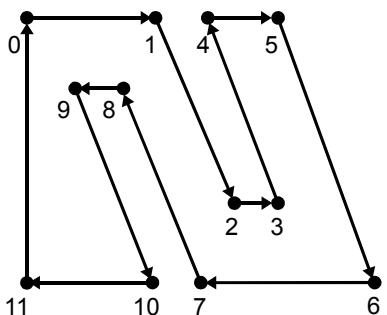
*Figure 56: Lot Subdivision Algorithm pseudo-code*

#### 4.4.2.2 Perpendicular Lot Orientation

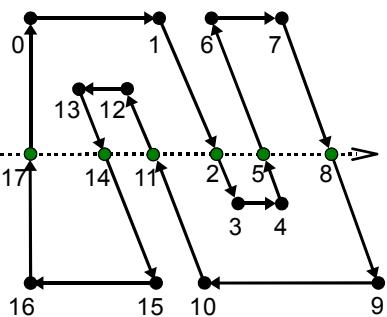
Another modification was required to improve the orientation of lots obtained from irregular or complex regions. The original lot division algorithm worked fine for rectangular regions, however, when the algorithm was tested on suburban road networks the resulting lots were angular and irregular. To combat this and provide more realistic lots for suburban networks a modification was made to the lot division process in which division was prioritised along sides with road access. This modification can be seen in Figure 54 where the function `getLongestRoad` is invoked before the function `getLongestNonRoad`. As a result of this simple modification almost all lots are now oriented perpendicular to their access roads. This more closely reflects the conditions found in the real world and improves the quality and realism of lots in suburban regions without affecting other regions negatively.

#### 4.4.2.3 Concave and Convex Lot Subdivision

The last and most important improvement to the lot subdivision algorithm is the removal of the restriction for regions to be convex. By requiring that all regions be convex the lot subdivision process is limited to operating on regular blocks, like those found in Manhattan. However, most regions are not convex. Suburban, industrial or any district with filament roads can not be processed using the basic subdivision algorithm. The solution was to develop a more complex split algorithm capable of splitting both concave and convex regions into two or more subregions.

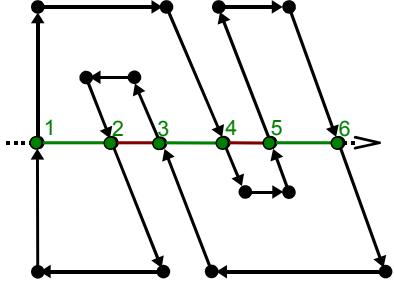


Regions or lots are represented as graphs of directed edges. These directed edge graphs are implemented in a circular linked list. Using this data structure each of the vertex neighbours are easily accessible and additional data such as road access information and process variables can be attached to each edge.

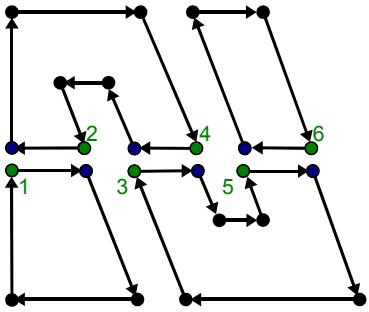


The intersections between the boundary edges and the split line are calculated in two stages: The first stage determines the position of each vertex relative to the perpendicular of the split line using the same technique as described in section 4.2.3.3. In the second stage, edges with a vertex on each side of the line, are deemed to have

an intersection present. The position of the intersections and their relative locations along the split line are then calculated and inserted into the regions circular list and also into a list for intersection vertices.



The list of intersection vertices are then sorted by their relative position on the split line. Each alternate edge between the ordered intersection vertices is considered part of the region as per Jordans Curve Theorem (these edges are marked in green in the illustration shown on the left). For each of these pairs a bridge function is invoked to modify the structure of the region into several constituent regions.



The bridge function creates two additional vertices and two new edges for each intersection pair. It is important that the bridge function modifies the graph so that each of the original intersection edges is assigned to a different sub region. Finally the sub regions are extracted by cycling through the connected edges for each intersection edge. Since a sub-region may contain more than one intersection edge, edges are marked as visited to avoid any duplication.

*Figure 57: Region Split Algorithm*

The original version of algorithm was developed using vectors and indices instead of the circular list. However, there were performance implications, even though the size of each vector was determined in advance and the memory was allocated in advance, the resulting copy time for each region was still noticeable. After further research a useful description of an algorithm was found in Graphic Gems V [Paeth et al. 1995]. This new revision of the algorithm, as shown in Figure 58, is based on the clipping algorithm described in chapter II.3 of Graphic Gems V. While the algorithm is not identical, important concepts including the bridge function and the circular list were instrumental in improving the performance of lot subdivision. On the next page a pseudo-code listing of the split region code is provided for those who may require more detail on the algorithms' operation.

```

splitRegion(region, a, b)
ab = b - a
Lsq = ab.squaredLength()
foreach(edge in region)
    edge.s = (-ac.z * ab.x + ac.x * ab.z) / Lsq

foreach(edge in region)
    if((edge.s > 0 && edge.next().s <= 0)
       || (edge.s <= 0 && edge.next().s > 0))

        cd = edge.dstPos() - edge.srcPos()
        denom = (ab.x * cd.z) - (ab.z * cd.x)
        ca = a - edge.srcPos()
        r = ((ca.z * cd.x) - (ca.x * cd.z)) / denom // loc on ab
        s = ((ca.z * ab.x) - (ca.x * ab.z)) / denom // loc on cd

        if(edge.s == 0)           // if split on src
            intersectingEdge = curr
        else if(edge.next().s == 0) // if split on dst
            intersectingEdge = curr.next()
        else
            // intersection point calc using cd, splitline ab is flat
            intersectingEdge = region.insert(edge, edge.srcPos()+(s*cd))

        intersectingEdge.s = r
        createdEdges.add(d)

    // sort the created list by location on ab
    sort(createdEdges, sortDirectedEdgeByS)

    // mark edges as unvisited
    foreach(edge in region) edge.s = 0

    // bridge intersection pairs
    for(i=0; i<createdEdges.size(); i+=2)
        DirectedEdge::bridge(created[i], created[i+1])

    // finally extract the new regions
    foreach(createdEdge in createdEdge)
        edge = createdEdge
        skipDuplicate = false
        do
            if (edge.s > 0)
                skipDuplicate = true
                break
            edge.s = 1           // mark edge as visited
            edge = edge.next()   // advance to next edge
            while(edge != createdEdge)
                if(!skipDuplicate) outputRegions.add(new Region(createdEdge))

        return outputRegions
    
```

Figure 58: Lot Subdivision Algorithm pseudo-code

### 4.4.3 Building Construction

Buildings are generated on the lots created by the lot subdivision process. Hints attached to each neighbourhood as part of the *control parameter set* advise the building generation algorithm on what class of building to be generated and how it should be positioned on the lot. The actual building placement is achieved using a combination of polygon insetting and shape fitting. Finally the building geometry is constructed by performing an extrude operation on the footprint and materials are applied to simulate additional geometric detail.

#### 4.4.3.1 Building Footprints

Building generation begins with the positioning of the building footprint within each lot. Different strategies are applied to obtain the footprint based on the type of development that is indicated by the *cell control parameters*. The footprints are calculated using the inset algorithm described in section 4.4.1.1. Road access information is required by some development types to inset the road access sides by a different amount to the others. To implement this feature the Straight Skeleton Inset function has been extended to support a weighted inset routine, sometimes referred to as a *Weighted Straight Skeleton* or *Camp Skeleton*. This extension is implemented by calculating the angle bisector taking edge weights into account. In this case the optimisation applied by Felkel et al.[Felkel and Obdrzalek 1998] to limit the execution of the *Split Event* (see section 4.4.1.1 for a definition) test is no longer applicable. In a non-linear inset, used by the Suburban example below, a different set of vertices can cause a *Split Event* and not just the reflex vertices. The illustration below shows the inset operations for a number of different building types.

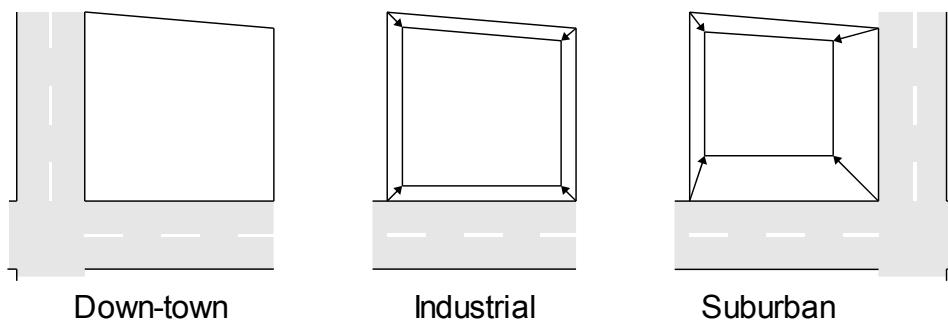


Figure 59: Building Type Lot Insets

As shown in Figure 59, the down-town type buildings attempt to make maximum use of lot space and apply zero inset while the suburban buildings retreat in from the road access sides and also from each other in order to make room for a garden area. The industrial buildings

retreat in from their boundaries to emulate the green space found in industrial estates. After the inset has been performed the industrial building footprints and down-town building footprints are finalized. However, the suburban building does not use an irregular shaped footprint and instead attempts to fit a regular shape such as a rectangle inside the inset region.

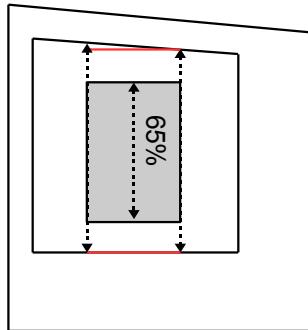


Figure 60: Building Type Lot Insets

This simple shape fitting algorithm operates by first selecting a primary road access side and obtaining a centre point. Two lines are then calculated parallel to the primary road edge and offset from the centre point. Using these lines intersection tests are executed and the minimum distances between the intersections and the centre point are obtained. Using these minimum distance values a proportional rectangle is constructed as the final footprint.

#### 4.4.3.2 Building Geometry

Once the footprint has been calculated, building geometry is then generated by simply extruding the footprint upwards to produce a solid object. The height of each building is determined from the *building height* parameter, an accompanying deviance parameter is used to add controlled noise to the building height values.

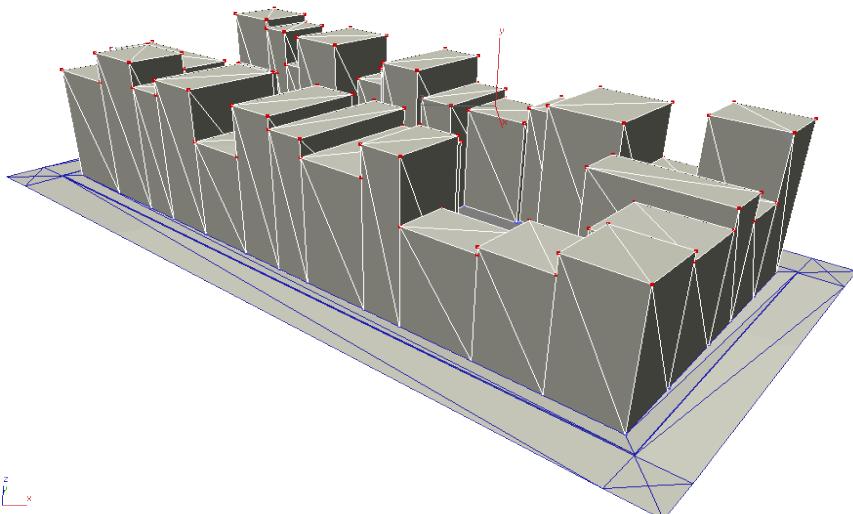


Figure 61: Primitive Building Geometry

The basic building shapes that are generated by extruding the footprints are primitive. However, these shapes can then be used as a starting point for a shape grammar transformation or any other procedural building technique. In our implementation, we try to improve the level of detail and visual quality of the cityscape by applying advanced materials with shaders to simulate additional geometry.

#### 4.4.3.3 Building Materials and Shaders

In order to improve the appearance of basic building meshes, materials are applied that employ shaders to add more detail. One of the main advantages of this approach is that the shaders can utilize the graphics hardware and not the main processor to simulate the additional geometry. Thus, the appearance of more complex meshes is provided without incurring the processing time of generating and rendering the actual geometry.

Several techniques are available to simulate additional geometry, three examples are shown in Figure 62. The first technique is *normal mapping*, which is a standard technique frequently applied in computer graphics. It succeeds *bump mapping* by using a colour texture to define the normals instead of using a grey-scale texture to perturb the normals. The three colour components in the texture are used to define the x, y and z components of the surface normals. These normals define the direction each surface is facing, and are then used by the lighting system to compute shading. The second technique shown in Figure 62 is *parallax mapping* [Kaneko 2001]. Parallax refers to the effect where the apparent position of an object changes with the observation point. This technique overcomes the flat appearance of normal mapping by displacing the texture coordinates to give an appearance of depth in the final rendered object.

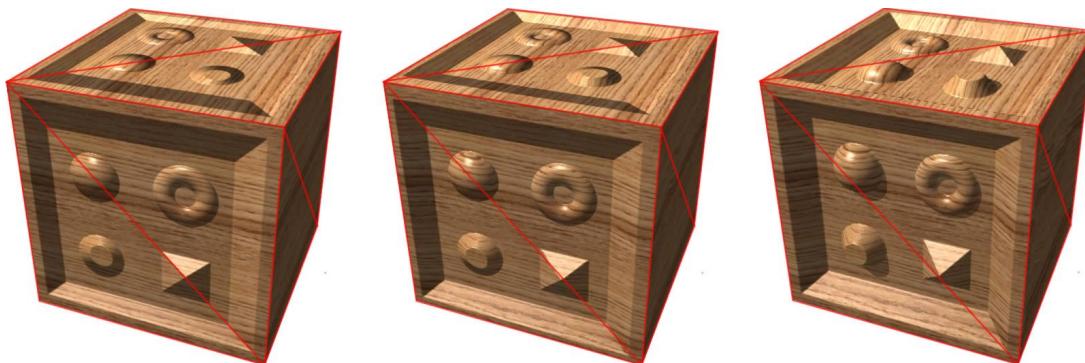


Figure 62: Three techniques to simulate additional geometry

The third technique shown in Figure 62 is called *relief mapping* [Policarpo 2005], and this is the technique applied to our buildings. Relief mapping overcomes the weaknesses of the other techniques by providing realistic shading, parallax effects, occlusion and self-shadowing. Occlusion is the effect where an object is masked, hidden or occluded from view by an object that is closer to the observation point. An example of occlusion can be seen in Figure 62 where the far side of the pyramid is occluded on the top side of the right most cube. Here the far side of the pyramid is not visible as it is hidden or occluded by the near sides. The effects of self-shadowing can also be seen on the front side of the same cube. Here the detailed objects on the face cast shadows across the face just if they were implemented with geometry.

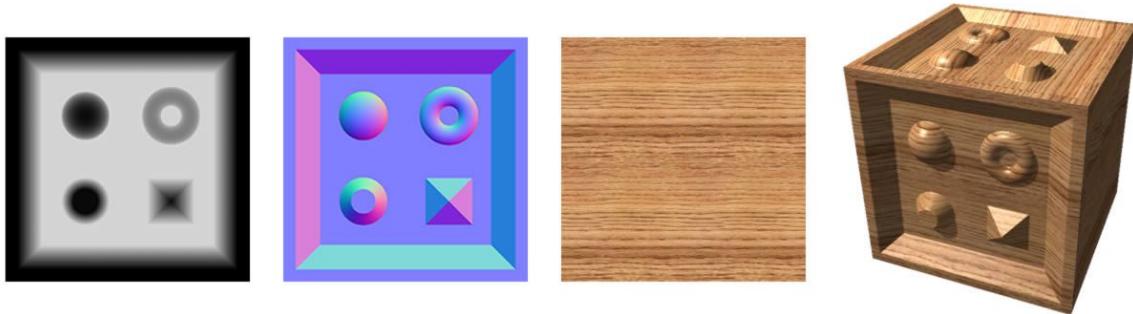


Figure 63: Relief Mapping: Height-map, Normal-map, Diffuse-map and Output

The textures required to implement relief mapping are shown in Figure 63 above. Relief mapping uses a grey-scale texture, or the alpha value of a colour texture to define a height-field in addition to using a colour texture for the normal map. The relief mapping technique defines a shader to compute an efficient ray to height-field intersection algorithm which executes on the GPU [Policarpo 2005]. More details on this algorithm can be found in “Real-Time Relief Mapping on Arbitrary Polygonal Surfaces”.

Each relief mapping texture set is used to define a *building tile*. These building tiles contain information in their material description to specify the input textures and the desired dimensions for their application. The tiles are then applied to the building geometry by generating texture coordinates with a best fit algorithm. This algorithm aims to match the tiles dimensions without splitting or dividing a tile unnecessarily. By using this system a range of different sized geometric building units can be defined and applied to any building while scale and proportion are maintained.

The process used to create the texture sets for the building tiles is discussed in the next chapter, in section 5.6. A screen-shot of city buildings with a range of building geometry tiles can be viewed in the results chapter. To summarize, this approach provides a method to define building geometry in tile form and the application of this technique results in significantly improved detail in the final rendered cityscapes.

## 4.5 Summary

In this chapter we have discussed the concepts and algorithms that have been applied in our design. The three main stages of city generation: primary road networks, secondary road networks and building generation have been outlined. The primary road network stage described the adjacency list data structure, the adaptive roads and the sampling strategies available. Secondary road network generation encompassed the extraction of cells via the minimum cycle basis algorithm, a description of the growth algorithm and a detailed look at the snap algorithm which is critical to performance. The final stage, building generation, explained the operation of the lot subdivision algorithm and outlined the improvements made. Finally the algorithm used to obtain building footprints was explained and the geometry tile system that is powered by the shaders of the building materials was described. In the next chapter we show how these concepts and algorithms were developed into an interactive procedural city generation system, titled Citygen.

# Chapter 5

## Citygen Implementation

In this chapter we describe our implementation for the procedural city generation project. A solution has been developed in the form of a standalone application titled *Citygen*. We introduce the user interface and provide an outline of all the major interface components describing their functionality and role in the system. Furthermore an explanation is included of how the application can be used, the tools that it provides and how these tools can be applied to quickly and easily construct a cityscape.

The application has been built from scratch using C++ and takes advantage of a number of key technologies to fulfil the design goals. Several libraries and standards help to create an accessible, interactive application that can achieve real-time city generation, and fit into the development pipeline of the graphics industry. In addition to fulfilling the design goals, the careful choice of libraries have provided added benefits and these are also discussed. An outline into each of these libraries and standards is included alongside the reasoning for their selection.

### 5.1 An Introduction to the Citygen UI

The Citygen application is designed for accessibility and ease of use. It provides a unified workspace for procedural city generation and the user interface controls are integrated together in a single application incorporating a built-in game engine view. Individual interface components such as the toolbars, property lists and game engine render view can be moved, docked or separated from the main window in a similar style to commercial graphics application or IDEs. The main components of the workspace are illustrated in Figure 64.

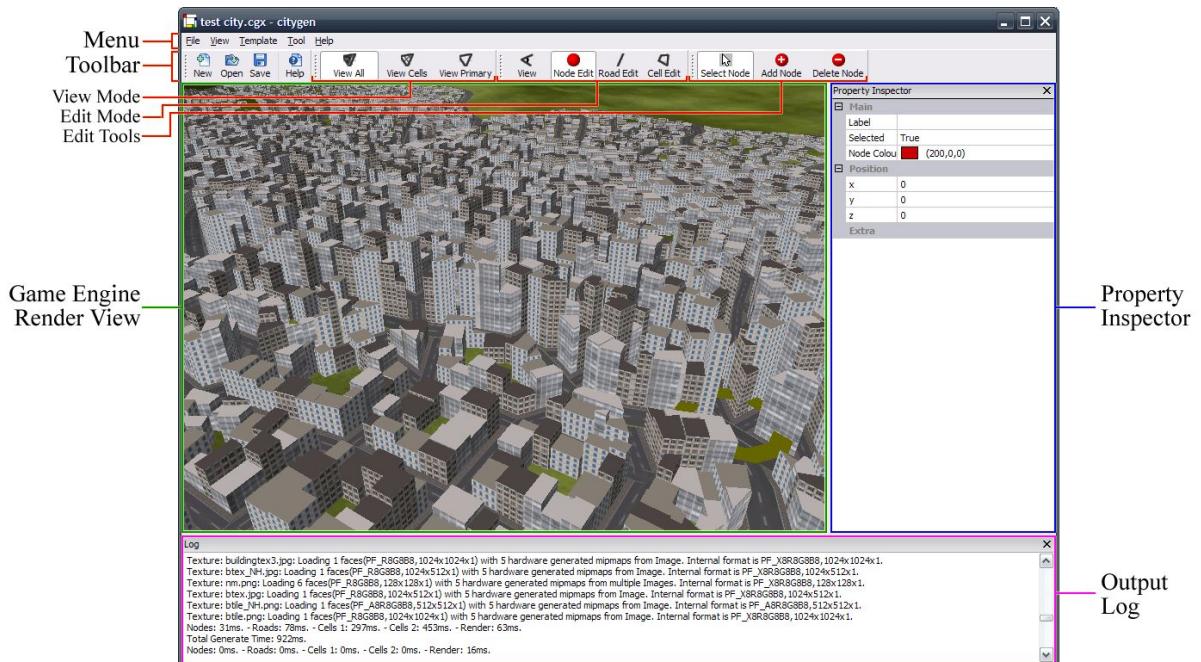


Figure 64: Citygen User Interface

## Menu and Toolbar

Citygen contains a *Standard Toolbar*, a *View Mode Toolbar*, an *Edit Mode Toolbar* and an *Edit Tools Toolbar*. The menu has a copy of the toolbar functions with the addition of infrequently used tools such as *Options* and *Export*. The *Standard Toolbar* contains the generic functions commonly found in most desktop applications: *New File*, *Open File*, *Save* and *Help*. The remaining toolbars contain application specific functions and their operation and behaviour is detailed in the next sections.

## View Mode Toolbar

The *View Mode Toolbar* can be used to select what elements of the city are displayed in the *Game Engine Render View*. A number of views are available:

- *View Primary* – displays only primary roads.
- *View Roads* – displays primary roads and secondary roads.
- *View All* – displays primary roads, secondary road and buildings.

The view mode can be changed at any point in the generation process and is independent of the edit mode. This allows the user to see how their changes affect each layer of city

generation. Figure 65, displayed below, illustrates the effect of different view modes in action on the *Game Engine Render View*.



Figure 65: View Modes from left to right: View Primary, View Roads and View All.

### Edit Mode Toolbar

The Edit Mode Toolbar is used to change the current edit mode of the application. Four distinct edit modes are available: View, Node, Road and Cell. By selecting a different edit mode the user changes the operational mode of the application. For example, selecting Node Edit displays the Node Edit Tools and the Node Property Inspector. The user can then select and modify nodes in the *Game Engine Render View*. *Road Edit Mode* and *Cell Edit Mode* provide similar functionality for roads and cells. A more detailed description of the operation of individual edit modes is provided in sections 5.2, 5.3 and 5.4.

### Property Inspector

The property inspector is used to view and modify the properties of city scene objects. Properties that can be edited include basic coordinates, display preferences and most importantly the control parameters for generation.

### Output Log

The output log provides extra information targeted towards more technical users. This extra technical information contains specific details on the city model currently being generated. Some examples of this information are the format of the textures loaded, the generation times, and the polygon and vertex counts of the different city components.

## 5.2 View Edit

This mode is used to change the view of the city presented to the user in the *Game Engine Render View*. The view system in Citygen follows the same design goals as the rest of the application and is designed to be accessible and easy to use. The camera model used replicates the functionality found in mainstream 3D authoring tools and should be familiar to any user with experience in computer graphics.

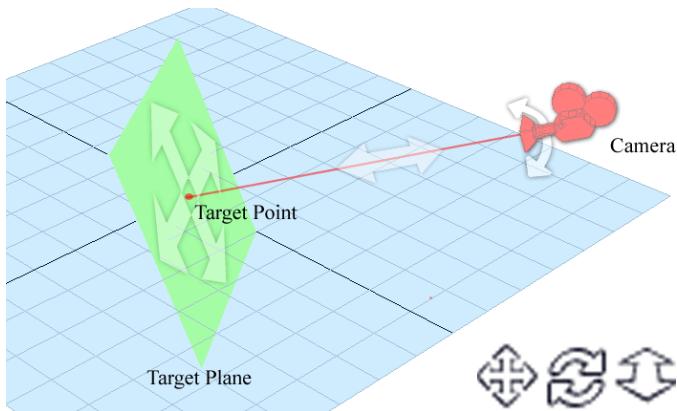


Figure 66: Camera Model and Cursor Bitmaps

- *Translation* operations are performed relative to the target plane so that the user can effectively grab and move the screen. Translation magnitude is accurately calculated to keep the object in focus aligned with the mouse cursor as the user drags the screen. The correct translation magnitude is calculated via a mouse ray plane intersection.
- *Rotation* operations are made around the target point. The user can rotate the camera horizontally and vertically at the same time, using a mouse drag action and relative screen coordinates to control the rotation.
- *Zoom* operations are activated using either a mouse drag action or the scroll wheel. The zoom magnitude is determined relative to the distance between the camera and target point, in order to have rapid zooming at distance, and fine-grained control nearer.

Cursor bitmaps, shown in Figure 66, were created using vector graphics in the Fireworks application and are similar in style to those found in mainstream authoring tools. The cursors are used in Citygen to indicate the current camera manipulation being performed.

### 5.3 Node Edit

The *Citygen* application provides a user interface to rapidly construct the primary road network using a simple and intuitive process. A point and click system is employed that enables users to quickly create the series of nodes and adaptive roads that constitute the primary road network.

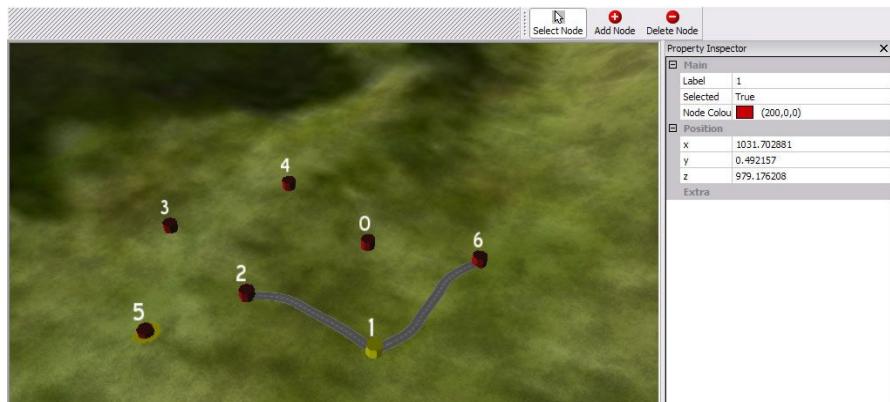


Figure 67: Node Edit Workspace

Nodes are represented in the application by red control points which can be selected, moved, modified and deleted using the point-and-click interface. A label is assigned to each node with the instance count used as a default value, custom text can also be assigned by the user via the Property Inspector. Nodes are displayed according to their status: normal nodes are drawn in red, highlighted nodes are surrounded with a yellow disc and selected nodes are drawn in yellow.

Three node specific tools are provided in the toolbar: *Select Node*, *Add Node* and *Delete Node*. To add a node to the system the user simply selects the *Add Node* tool and points and clicks on an area of the terrain and a node is then constructed and displayed at that point. If the user would like to modify the position of an existing node the user can activate the *Select Node* tool and then click on any node and simply drag it to the desired location to reposition it. In the event that specific and exact world coordinates are required for a node the application can facilitate this process also. Using the node property inspector, displayed on the left in Figure 67, the user can specify the exact world coordinates by typing the desired position into the x and z properties of the property inspector. The y property is the height coordinate and is determined automatically by plotting the 2D coordinate to a point on the surface of the terrain. To delete a node, a similar point-and-click system is employed. The

user simply selects the *Delete Node* tool and points and clicks near an existing node on the terrain.

### 5.3.1 Add Node - Chain Tool

The add node tool contains an additional mode of operation that is designed to facilitate the construction of roads. This mode provides a method to add roads quickly using the least number of clicks possible.

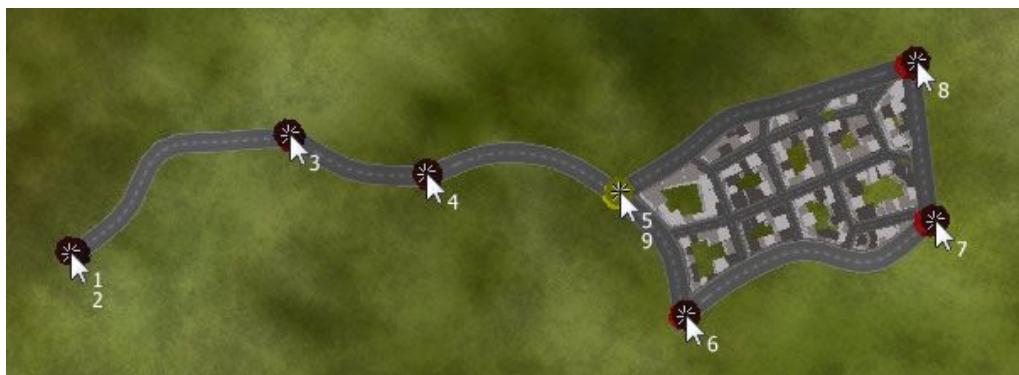


Figure 68: Chain Tool – Plough of roads in 9 clicks

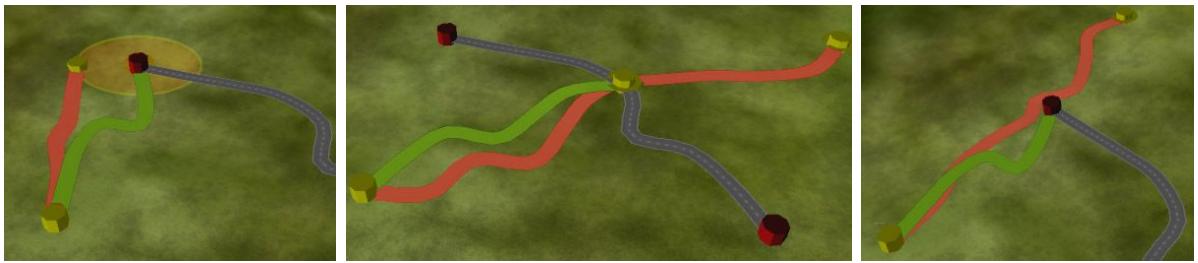
The road construction mode of the Add Node tool is activated by clicking on an existing node with the Add Node tool selected, then a road is automatically generated and displayed between the selected node and a proposed node, positioned underneath the mouse cursor. As the user moves the cursor around the screen they can preview this new road and choose to add it by simply pointing and clicking on the terrain. A new node is then added, along with a connecting road identical to the preview road that was displayed. To add further roads, this process does not have to be initiated again, as the selected node is advanced to the newly added node, allowing the user to create a long string of nodes all connected by roads. In addition, this tool is not limited to creating singular strings of nodes, but can also be used to create loops and connect to existing infrastructure. When connecting a road to existing nodes, the selected node is still advanced, but in this case it is advanced to the target node, and the user can continue to add a road per click on the terrain.

### 5.3.2 Validity Checking

Validity checking is an important part of the Citygen system and is applied to maintain the validity of the road network graph and to assist the user in the rapid construction of the

primary road network. Just like the secondary road generation, where the growth algorithms' actions are refined to create a valid road network, here the users actions are refined.

Road network graphs are planar so their edges cannot overlap and all road to road intersections require a node to model each and every junction. Citygen provides real-time validity checking, so as the user moves the cursor, or proposes a change to the network, their actions are refined in real-time by the validity checking mechanisms. This enables the user to see a live view of how their actions will effect the road graph and what modifications shall be made by the system to ensure the integrity of the road graph. For example, in the event that a user wishes to create a road that overlaps with another road, the system will automatically detect the intersection event, display the corrected road graph, and insert a node in the existing road at the offending intersection point upon commit.



*Figure 69: Validity Checking: Node Snap, Road Intersection and Road Snap. The red road illustrates the users proposed action and green road shows the corrected version.*

In addition to maintaining the integrity of the road network, the validity checking algorithms assist the user in the rapid creation of nodes and roads by snapping automatically to existing infrastructure. As the user selects nodes, and connects nodes with roads, the cursor will automatically snap to the closest existing node within a defined snap distance. With the aid of this technique users can select nodes and add roads without having to worry about the exact position of the cursor or without requiring precise selections. Also users can, with a single action, propose invalid or intersecting roads, view the corrected modifications in real-time and click to accept these as additions to the road network.

## 5.4 Road Edit

The Road Tools provide a means to add, modify and delete roads in the primary road network. Primary roads in Citygen are adaptive and change to fit the environment in which they are placed. The road Property Inspector provides access to the parameters that control

the adaptive road generation process and these can be modified with their effect viewable instantaneously.



Figure 70: Road Edit Workspace

Roads are rendered with textured geometry in Citygen and appear identically in the exported model that the application creates. Junction plates are constructed separately as part of the node object but are displayed in the road edit mode. A point-and-click interface is used again to select, modify, delete and add roads. Selected roads are indicated by being displayed in a shade of yellow.

Three road specific tools are displayed in the toolbar: *Select Road*, *Add Road* and *Delete Road*. In order to create a new road the user selects the *Add Road* tool and clicks on a pair of nodes  $a$  and  $b$  sequentially to create a new road from node  $a$  to node  $b$ . In the event that a road cannot be constructed, for instance if the road intersects with a number of other existing roads, the user is presented with a warning message that explains the reason why their desired road was not constructed. The *Select Road* tool functions by allowing the user to select any road with a single click and then the properties for that road are loaded and displayed in the road *Property Inspector* dialogue. The *Delete Road* tool provides a means for the user to delete individual roads from the road network graph. The operation of the tool is again very simple, to delete a road the user simply selects the *Delete Road* tool and clicks on the road to be deleted.

#### 5.4.1 Adaptive Road Control Properties

The road *Property Inspector* displays the adaptive road generation properties for the currently selected road and also the default generation properties for new roads when a road is not selected. The road *Property Inspector* consist of 7 properties in two categories, *Adaptive*

*Road Parameters and Display Options.* Figure 71 shows the road property inspector in action.

The *Adaptive Road Parameters* category lists the control parameters involved in the generation of adaptive roads. The *Algorithm* property provides a drop down list to select the active road generation algorithm, a description of which is included in the previous chapter in section 4.2. The remaining properties in this category control the behaviour of the sampling for road generation and specify the width of the road. Figure 72, which is displayed below, shows the effect of these control parameters on adaptive road generation.

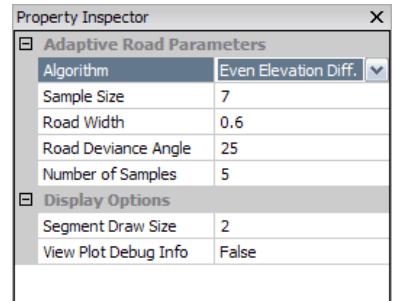


Figure 71: Road Property Inspector

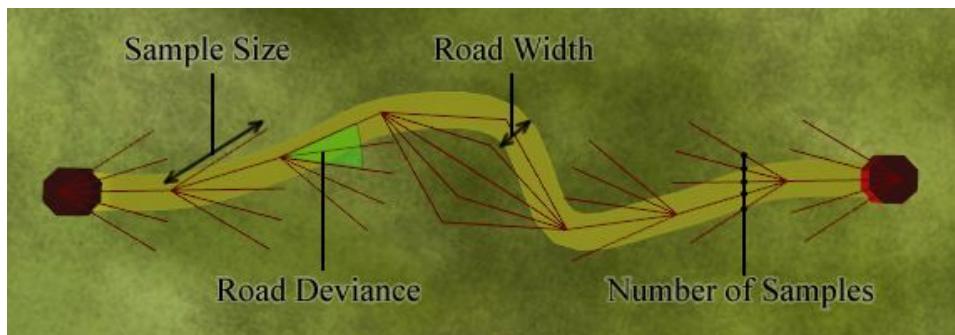


Figure 72: Adaptive Road Control Parameters

The Display Options category allows the user to change the display of each road and how it is rendered. The *Segment Draw Size* property directly effects the smoothness of the road geometry by defining the size of road segments to be interpolated from the road path curve. The last parameter, *View Plot Debug Info*, can be used to enable a debug overlay for each road. The debug overlay is illustrated in Figure 72 and displays the samples used by the adaptive road generation algorithm.

## 5.5 Cell Edit

In *Citygen*, *City Cells* represent the neighbourhoods or districts of the city and are formed from the enclosed regions of the primary road network. Inside each *City Cell* the secondary road network is generated using a growth based algorithm. The Cell Edit mode provides an accessible interface in which the user can simply select cells and view their generation

control parameters. Control parameters for both the secondary road generation and building construction are displayed in the *Property Inspector*.

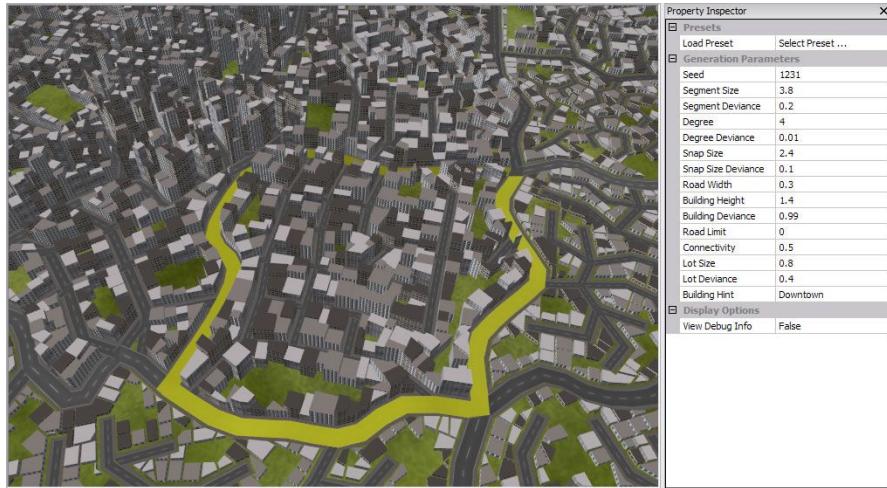


Figure 73: Cell Edit Workspace

Cell Edit mode, in contrast to the other edit modes, does not provide a bar of specialized tools. Instead a single tool, the *Cell Select* tool, is all that is required. No “add cell” or “delete cell” tools are needed as cells are created via the construction of the primary road network whenever a region is enclosed. The boundaries of cells are automatically extracted from the primary road network using the Minimum Cycle Basis (MCB) algorithm described in section 4.2.1. *Cell Select* operates in a similar fashion to the other tools with a point-and-click interface, if a user clicks within the boundary of a cell it will be selected. Selected cells are indicated by a yellow outline displayed around the cell boundary as shown in Figure 73.

### 5.5.1 Cell Generation Properties

The cell *Property Inspector* provides an editable view of the generation control parameters. If no cells are currently selected, the *Property Inspector* displays the default property set, which can be modified to change the initial property values used for new cells. Upon selection of a cell, its parameters are read and loaded into the *Property Inspector*, where they can be viewed and modified. Three categories of properties are listed. *Presets* contains a property to load a predefined set of generation properties that encapsulate a distinctive style or type of neighbourhood. *Generation Parameters* controls the operation of the secondary road growth algorithm and building construction. Lastly *Display Options* provides a facility to display additional debug information.

*Generation Parameters* is the core category of the *City Cell Property Inspector* and contains all of the control parameters required for generation. The *Seed* property is passed to the random generator in the generation process and ensures that the results are consistently reproducible. Five properties and their accompanying deviance values are used to control the secondary road growth algorithm. They are *Segment Size*, *Degree*, *Snap Size*, *Road Width* and *Connectivity*. Two properties are used to control Lot Subdivision: *Lot Size* and *Lot Deviance*. Three properties govern the construction of buildings: *Building Height*, *Building Deviance* and *Building Hint*. Any modification made to the control parameters updates the *Game Engine Render View* in real-time giving the user an element of close control over the generation process. A detailed outline of the function and effect of these parameters is documented in the previous chapter.

The *Display Options* category of the cell property inspector contains a single property to enable a debug overlay. This debug overlay displays additional generation information such as block boundaries, lot boundaries and lot subdivision lines. The display of these lines can be useful for gaining an insight into the operation of the lot subdivision algorithm and can help the user obtain their desired effect from the generation process.

The *Presets* category contains the *Load Preset* property that allows the user to load a pre-defined set of values into the generation parameters with one click. These presets are an added convenience for the user and can act as a starting point or template for the generation of a number of styles of secondary road networks and different types of neighbourhoods. Once the parameters have been loaded they can be modified and tailored to fit the users specific requirements. It is also important to note that each *Preset* does not output a single map, but is seeded and has  $2^{32}$  combinations like every other parameter set. Figure 76 displays three distinct presets, each with their parameter sets and an accompanying screenshot.

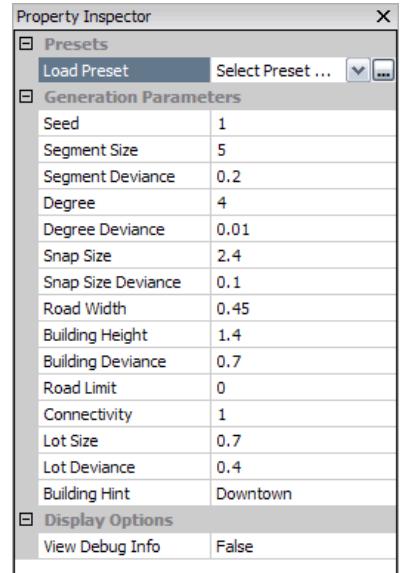


Figure 74: Cell Property Inspector



Figure 75: Cell Debug View

<b>Manhattan</b> <i>Segment Size:</i> $5 \pm 20\%$ <i>Degree:</i> $4 \pm 1\%$ <i>Snap Size:</i> $2.4 \pm 10\%$ <i>Road Width:</i> 0.45 <i>Connectivity:</i> 1 <i>Building Height:</i> $1.7 \pm 70\%$ <i>Lot Size:</i> $0.7 \pm 40\%$	
<b>Suburbia</b> <i>Segment Size:</i> $2.4 \pm 30\%$ <i>Degree:</i> $18 \pm 60\%$ <i>Snap Size:</i> $2 \pm 10\%$ <i>Road Width:</i> 0.2 <i>Connectivity:</i> 0.05 <i>Building Height:</i> $0.3 \pm 10\%$ <i>Lot Size:</i> $0.5 \pm 50\%$	
<b>Industrial</b> <i>Segment Size:</i> $3 \pm 10\%$ <i>Degree:</i> $4 \pm 1\%$ <i>Snap Size:</i> $2 \pm 10\%$ <i>Road Width:</i> 0.3 <i>Connectivity:</i> 0.05 <i>Building Height:</i> $0.6 \pm 30\%$ <i>Lot Size:</i> $1.2 \pm 60\%$	

Figure 76: Preset Examples

## 5.6 Building Tiles

The building tiles define the material which is used to simulate additional geometry on building meshes. In this section we will show how the building tiles are implemented and provide a short outline of the steps required to build a tile. Each tile consists of a height map, a normal map, a diffuse colour map and a material description.

The first step is to obtain a photo of the building that we wish to model. In order to create good textures for the building it is important that the photo is taken as straight-on as possible. The next step is to model the building geometry. One approach is to create a height map by shading the areas of the photo in shades of grey that correspond to the displacement of each area. However, I found that this approach is error prone and it is more reliable to apply the image as a projective texture and then model the building using a 3D authoring tool.

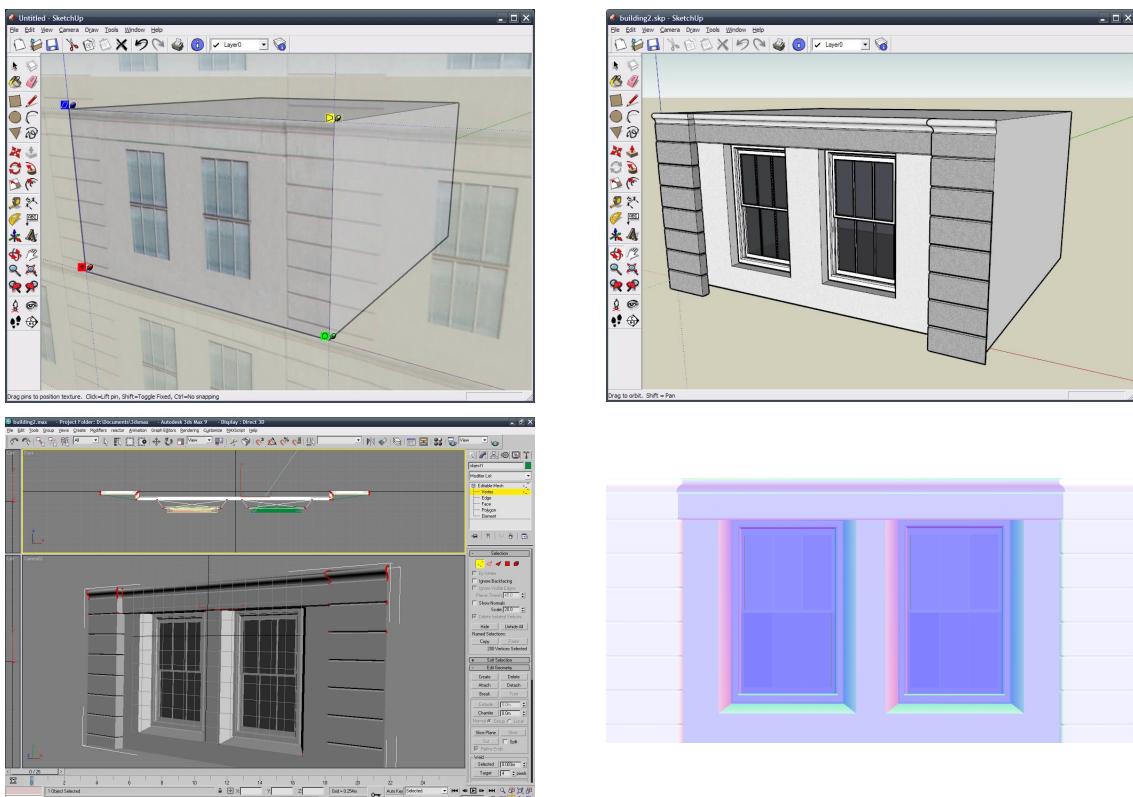


Figure 77: Building Tile Authoring

After modelling we have a complete geometric mesh of the building. The vertices can then be tweaked so that the façade angles are limited to 80 degrees, this avoid artefacts occurring. Finally the building side can be exported from 3DMax using the plug-in provided by Policarpo et al. [Policarpo 2005]. The texture that is produced contains both the height and normal map. This exported texture is included with the diffuse texture in a material script

which specifies the desired dimensions of the tile and completes our definition of the building tile. These tiles are then applied to the buildings to achieve significantly improved detail.

## 5.7 Integrated Game Engine: OGRE

Citygen includes an integrated game engine view within the application so that all cities can be generated in a WYSIWYG environment. OGRE [OGRE 2007] was selected for use in Citygen because of its clean design and comprehensive graphics feature-set which is comparable to even the most recent commercial game engines. It is frequently used in the education and research sectors as the engine of choice for rapidly developing 3D applications. OGRE features of particular relevance to Citygen include:

- Efficient and versatile rendering engine with OpenGL support.
- Powerful material system that defines materials outside of code and supports multi-texturing, multi-pass blending, vertex and fragment programs.
- Numerous mesh data formats accepted, vertex buffers, index buffers, vertex declarations and buffer mappings.
- Flexible scene management with the option of custom scene managers for complete control over scene organisation
- Clean object-oriented design, well-documented API and an active community.
- Cross-platform: Windows, Linux & OS X.

OGRE is officially defined as a rendering engine (**O**bject-oriented **G**raphic **R**endering **E**ngine) and not a game engine. However, game engine components not found in the standard OGRE package can be easily integrated into the system using a series of add-ons or plug-ins. Examples of available add-on components include audio, control, scripting and physics to name but a few. OGRE is built on a well tested and mature code-base and has been applied in a number of commercial game titles.



Figure 78: OGRE game screenshots: Ankh, Building World & Pacific Storm

## 5.8 Graph Structures: BGL (Boost Graph Library)

A number of graphs are employed within the Citygen application to store road network data and perform graph algorithms to extract enclosed regions, adjacency information and graph traits. The **Boost Graph Library**[Boost 2007] is used to serve as a standardized generic interface for traversing the road network graphs and is independent of the graph data structures used. Hence the algorithms that have been developed as part of Citygen can be applied to any graph that implements the standard BGL interface.

In addition to providing a standardised interface for graph traversal the BGL library is also used in Citygen because it implements several types of graph data structures in efficient and well-tested code. The main graph data structure used by the Citygen application is the adjacency list model. This template can be specialised further by specifying the STL container components used for both edges and vertices. In the case of Citygen road networks:

- Vertices are stored in the `<list>` container to ensure efficient iterator access and quick insertion/deletion at the cost of some additional storage space.
- Edges in the road network graph are undirected and are stored in the `<set>` container so that parallel or duplicate edges can be excluded with little overhead and the consistency of the road network graph can be maintained.

The `RoadGraph` class models road networks in the Citygen application and defines a simple API to modify and perform algorithms on the network. Several `RoadGraph` instances are used in the application: the primary high-level graph, the primary low-level graph and additionally a graph is used for each cell. Each visual entity, which is an item displayed in the world, such as nodes, roads, cells, buildings etc., is associated with one or more graphs. The associations are structured so that each entity can use its private member data to access neighbours and related objects in the city scene. For example: it is the role of nodes to create junction plates and for this to be achieved each node must access its parent graph and extract the connected roads and the position of adjacent nodes. After the construction of junctions, roads must access their source and target nodes to obtain the correct coordinates to join to the junction plate. Numerous relationships are modelled in the Citygen application and the `RoadGraph` is used to provide relationship information for all entities.

## 5.9 GUI Platform: wxWidgets

Citygen is designed from the ground up to be a portable application and support users running different platforms and operating systems such as Windows, Linux and OS X. Several user interface libraries exist to enable cross platform development, the wxWidgets [wxWidgets 2007] library was chosen because of a number of distinct advantages it provides:

- **Native look and feel:** wxWidgets creates user interfaces that perfectly match the look and feel of the host operating system by utilizing the native APIs.
- **Comprehensive GUI widgets:** wxWidgets contains basic widgets such as menus, toolbars, buttons and sliders but also provides advanced controls like Property Grids.
- **Advanced features:** customisable workspaces are an example of an advanced feature that can be supported easily with little effort using the wxWidgets library.

Cross-platform user interface libraries generally operate by emulating the display of interface components, resulting in applications that appear out-of-place and inconsistent with the look of the host operating system. wxWidgets is different in that its applications are always consistent with the look and feel of their host operating system. This is accomplished by not emulating the user interface components within the library but instead translating the API calls to use the native user interface libraries. wxWidgets provides a single API that acts as a thin layer of abstraction over the native APIs. In practice this means that a wxWidgets application running on Windows uses the Win32 API, on Linux uses the GTK+ API and on OS X uses the Cocoa API. By using wxWidgets cross-platform applications can be developed without expert knowledge of any specific platform and a single unified code base can be used for all platforms.



Figure 79: wxWidgets Native Look and Feel: Windows, Linux & OS X

## 5.10 Accessible Export: COLLADA

The main goal of the procedural city generation project was to create a tool that could help developers and artists working on computer graphics projects create large urban environments. In order to accomplish this goal it was important that the Citygen application could fit into the development pipelines used in the computer graphics industry. A wide range of digital content creation tools are used within the industry and it is not feasible or practical to provide a plug-in, or export file formats, for each tool. For this reason a unified exchange format was selected that could be read by all tools. The COLLADA(**COLL**Aborative **D**esign **A**ctivity) [COLLADA 2007] specification was chosen as the format to achieve this.

*COLLADA defines an XML database schema that enables 3-D authoring applications to freely exchange digital assets without loss of information, enabling multiple software packages to be combined into extremely powerful tool chains. [COLLADA 2007]*

COLLADA is an XML format originally devised by Sony Computer Entertainment as a digital asset exchange format. Since then a number of major graphics developers such as Alias, Discreet and SoftImage have joined the consortium to extend and refine the specifications adding support for advanced materials and physics. The COLLADA schema, now at revision 1.4.1, is well supported by most major graphics tools and game engines including 3DS Max, Maya, Blender, SoftImage XSI, Deep Exploration and the Unreal Engine 3.0. Citygen provides complete COLLADA export for generated city models and supports the export of all geometry, materials and textures.

## 5.11 Summary

In this chapter we have described our implementation for interactive procedural city generation. The user interface was introduced and an outline of the major interface components was provided. We have shown how the tools in Citygen can be used to create the constituent components of a city-scape. Also, information on the technologies used in the application were documented, along with the reasons for their selection. In the next chapter, we display the results of the system. This encompasses: studying the cityscapes generated, analysing the performance, and determining the accessibility of the generated models.

# Chapter 6

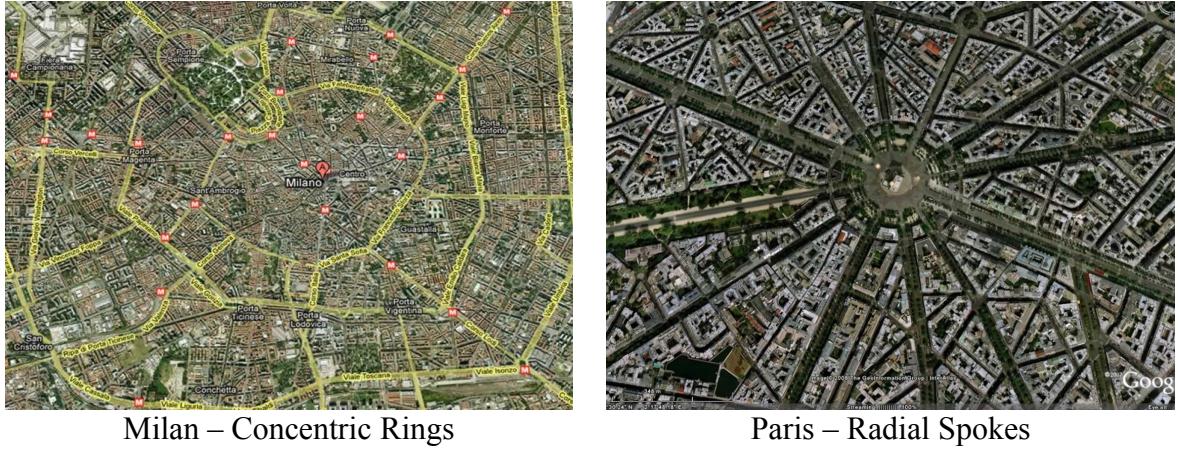
## City Generation Results

In this chapter we analyse the results of our procedural city generation system. The primary road editing facility is demonstrated by recreating predominant city patterns from real-world cities. The adaptive road system is then illustrated showing the construction of several common road types, each of which are mapped to the terrain. Next, the distinctive patterns of the secondary road networks are illustrated with samples, results and the associated generation parameters. Then, the lot subdivision and building tile results are displayed. The overall performance of the system is then analysed as we break down the computation times into stage times, comparing the performance with and without optimisations. Following that, we gather some statistics on the 3D authoring tools used in the graphics industry and estimate the percentage of users that can fit Citygen into their work-flow. Finally, we show a selection of screen-shots of generated cities. We begin with the primary road editing facility.

### 6.1 Primary Road Network

In *Citygen*, the construction of the primary road network is facilitated using the interactive graph editor, thus enabling users to incorporate any number of high level patterns into their cities. The primary road network is critical to determining city structure and plays an important role in influencing our impression of city character. Previously, we discussed in Chapter 3 how roads correspond to *Paths* and *Edges*, and how these roads form the defining boundaries for each neighbourhood or district. We also discussed how the primary road network affects our perception of the city, and in particular how these road networks define the predominant patterns that we use to identify distinctive city areas. The examples we used to illustrate this point were maps of the concentric ring roads that loop around the city centre

of Milan and the radial spokes surrounding the Arc de Triomphe in Paris. We can now revisit these examples and see how the user can specify the key characteristics of a cityscape by recreating these patterns in *Citygen*.



*Figure 80: Predominant Pattern in Existing Cities (Satellite)*

*Citygen* is designed to provide the user with interactive control over road generation so input data is not required. However, satellite images or road maps can be useful as a form of reference to observe and select desirable patterns, which can then be incorporated into our generated cities. The following images, shown in Figure 81 and Figure 82, illustrate how these patterns can be accurately reproduced in procedural cities.



*Figure 81: Milan Primary Network in Citygen*

Figure 81 illustrates the concentric ring road pattern of Milan applied to an area of approximately 25Km<sup>2</sup>. Even though the secondary road network is significantly different from the original, the cityscape can still be easily recognised as that of Milan, due to the predominance of the same high level patterns in the primary road network.



*Figure 82: Paris Primary Network in Citygen*

Figure 82 shows the radial spoke pattern of Paris applied to a much smaller area of around 1.5Km<sup>2</sup>. Again the pattern is clearly identifiable, and shows how the patterns of the primary road network have a strong influence on the perceived character of a city.

By recreating familiar patterns from established cities we have demonstrated that the interactive editor is capable of creating accurate and distinct patterns. The primary road network has been identified as the most predominant and recognisable feature of a cityscape. Using our system, any number of high level patterns can be mixed and matched, applied to the primary road graph and incorporated into the final cityscape. In the next section we look at adaptive roads, a technique which helps the user rapidly build the primary road network.

## 6.2 Adaptive Primary Roads

Adaptive roads form an important part of our procedural generation system. The path that roads follow is rarely an exact straight line. Roads in the real world are subject to environmental constraints and these constraints affect the route each road takes. For an author to create realistic roads: the constraints of the environment must be taken into consideration, the effect of each constraint must be evaluated and a suitable path for the road must be plotted. With regard to city generation applications this would mean that each road would need to be manually planned and plotted, therefore making the construction of roads a time consuming task. For this reason the adaptive roads system was developed to enable the rapid construction of the primary road network.

Adaptive roads react to same stimulus as real roads and adapt to fit the surrounding environment. Roads are plotted to the contours of the terrain and are controlled using a concise set of parameters and a number of strategies. The adaptive roads are generated instantly and the route can be previewed in real-time as the user moves the cursor. Effective parametrization is achieved which results in road control parameters that directly relate to the road properties. Two key parameters are deviation and sample size. The deviation parameter affects the roads range of movement and the sample size parameter affects the roads granularity of movement. By simply varying these two parameters we can capture the behaviour of a wide range of roads and use the obtained values to recreate similar roads in our generated cities.



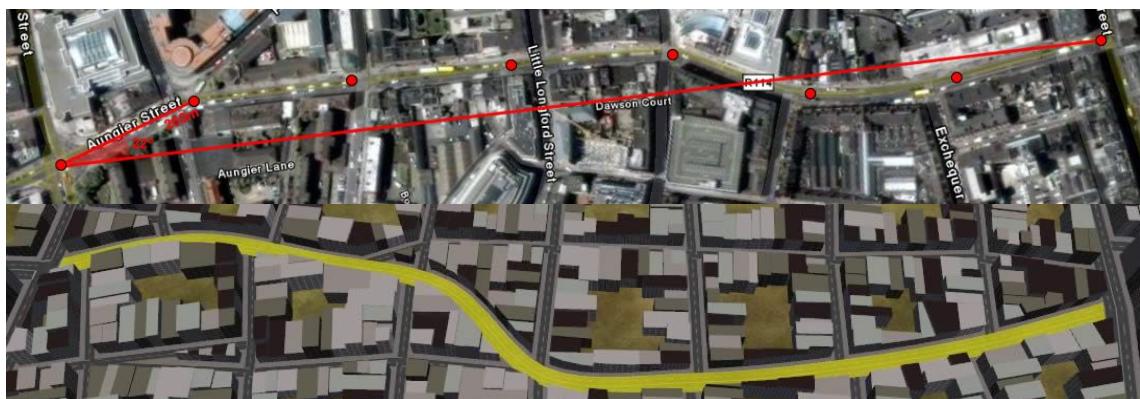
Figure 83: Major Road : Sample Size 640m, Sample Deviance 15°

Major roads such as motorways or highways are designed for high speed transit and as a result of this have no sharp corners or tight turns. Motorways can be recreated in *Citygen* by using a high sample size value with low deviance, the interpolation system employed in the application results in a road with smooth and gentle curves.



*Figure 84: Primary Road : Sample Size 260m, Sample Deviance 25°*

Primary roads that encounter uneven terrain must navigate a smooth path through while maintaining minimal elevation changes. Figure 84 demonstrates how a road of this type follows the terrain with substantially more curvature than the motorway using a moderate sample size and lenient or high deviance value.



*Figure 85: Inner City Road : Sample Size 90m, Sample Deviance 22°*

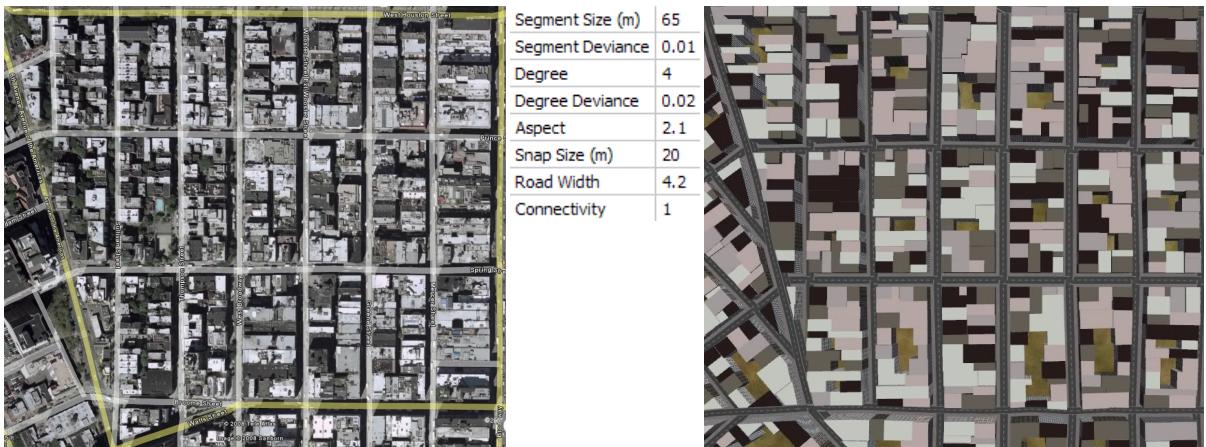
Inner city roads do not have to cope with high speed traffic and as a result can meander moderately to weave through the city. The direction of the road can change quite quickly due to the overlapping transport paths and frequent intersections. To abstract the behaviour of

inner city roads a low sample size value is used and a moderate to low deviation value. For many inner city routes, especially in planned cities, there is little or no deviation present.

In this section we have illustrated how a range of road types can be automatically generated using the adaptive road system. The road sampling and plotting algorithms ensure that roads react to the surrounding terrain and fit into the environment. Road networks can be created quickly and easily. A flexible generation algorithm means a range of road types can be created. Effective parametrisation enables the user to accurately match the road requirements. In the next section we will look at generation of the secondary road network.

### 6.3 Secondary Road Growth

In Citygen, secondary road generation operates on the regions enclosed by the primary road network. The secondary roads perform the job of servicing the enclosed land area by providing access to and from the primary road network. Each enclosed region represents a neighbourhood or district of the city, and each can display a distinct style and character. A growth algorithm is employed to construct the road segments inside each cell. This forms the secondary road network. A parameter set is defined for each cell to control the generation process and thus the style of output generated. We will now take a selection of secondary road patterns and illustrate how they can be generated using the Citygen application.



*Figure 86: Manhattan (Grid) Street Pattern*

The Manhattan pattern is formed from a grid of streets and is the most simple pattern to recreate. To create a grid pattern a degree value of four is specified with minimal degree deviation and segment deviation values. The road segment size and aspect ratio can be customised to modify the block size and block aspect for each instance. The aspect parameter allows directional bias to be implemented without affecting the generation of other patterns.



Figure 87: Industrial Street Pattern

The industrial zones are not as highly connected as the grid networks and function to provide efficient road access to the industrial buildings. Through traffic is not a goal of these areas, but is enabled since for industrial areas it is important to have easy access to the primary road network. For these reasons a much lower connectivity value is used in the generation parameters. Higher deviance values for segment length and junction degree are applied to add some noise into the resulting patterns. A large segment size is used to facilitate the large block area required by industrial buildings.



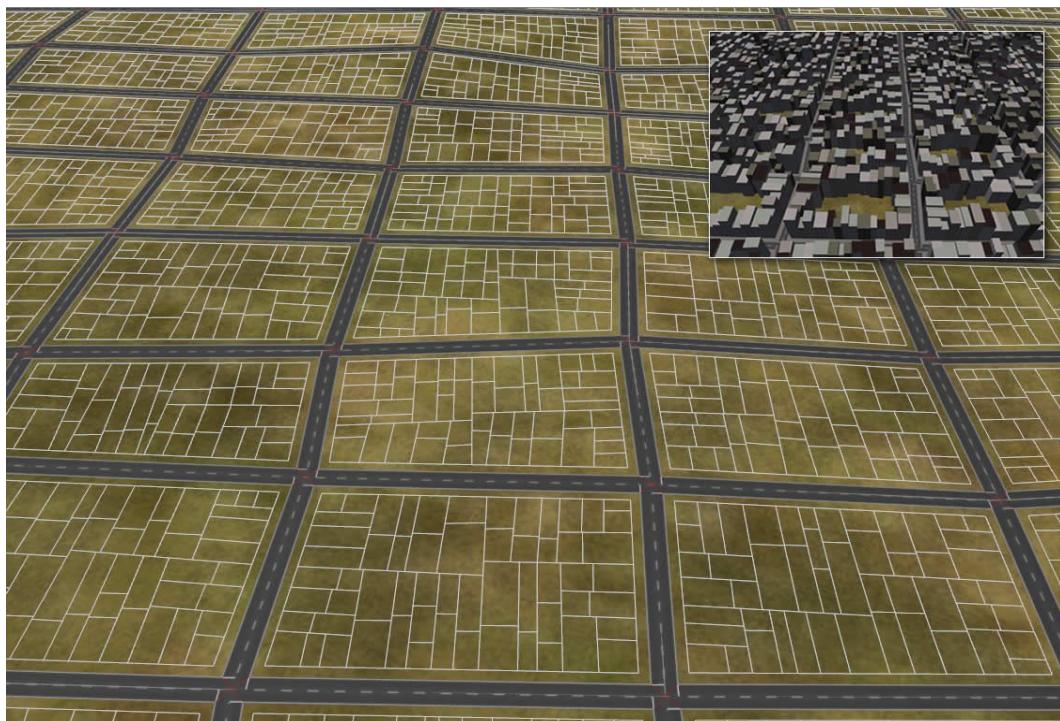
Figure 88: Suburban Street Pattern

Suburban areas use tree like road networks with almost all roads terminating at dead ends. Although the planned layout of suburban areas is difficult for the growth algorithm to emulate the general structure can be recreated accurately. The most significant parameters to set when creating suburban districts are degree deviance, snap size and connectivity. The connectivity of a suburban area is low by design, traffic through the neighbourhood is discouraged since these areas are often residential. Degree deviance is set to a relatively high value to enable the road growth to meander throughout the neighbourhood. A high snap size value is also required to ensure even road distribution and to contain untidy growth.

In this section we have illustrated how a range of neighbourhood styles can be generated using the road growth algorithm. The influential parameters that define each style type were outlined, and the results of the secondary road generation were shown side by side with real world examples. The results show that the road growth algorithm enables the user to generate a wide range of road networks. Also, real-world road networks can be matched by similar procedural networks using only a few specific parameters.

## 6.4 Building Generation

Building generation operates on the enclosed land areas of the secondary road network. The lot subdivision process divides these areas into lots that are suitable for building construction. Once the lots have been identified the construction of buildings can begin. Basic buildings are constructed by extruding the building footprints upwards. Building tiles, which simulate detailed geometry by using relief mapping, are then applied to the final mesh. We first look at the results of the improved lot subdivision algorithm and assess the effect of our extensions.



*Figure 89: Manhattan Lot Subdivision – Lots Displayed in Debug View*

The original lot subdivision technique described by Parish and Müller [2001] worked well on regular grid-like road networks similar to those found in Manhattan. However, the implementation of the algorithm in Citygen differs from the original in that it contains a number of extensions. These extensions include: more even and accurate method of lot

division, the prioritisation of division along road access sides, the ability to process both concave and convex regions and the addition of individual lot width and depth parameters. Even though these changes alter the behaviour of the process, we found that the generation of lots in regular grid networks like Manhattan was not adversely affected. In fact, the lot width and depth parameters were beneficial in creating lots which match those found in Manhattan.

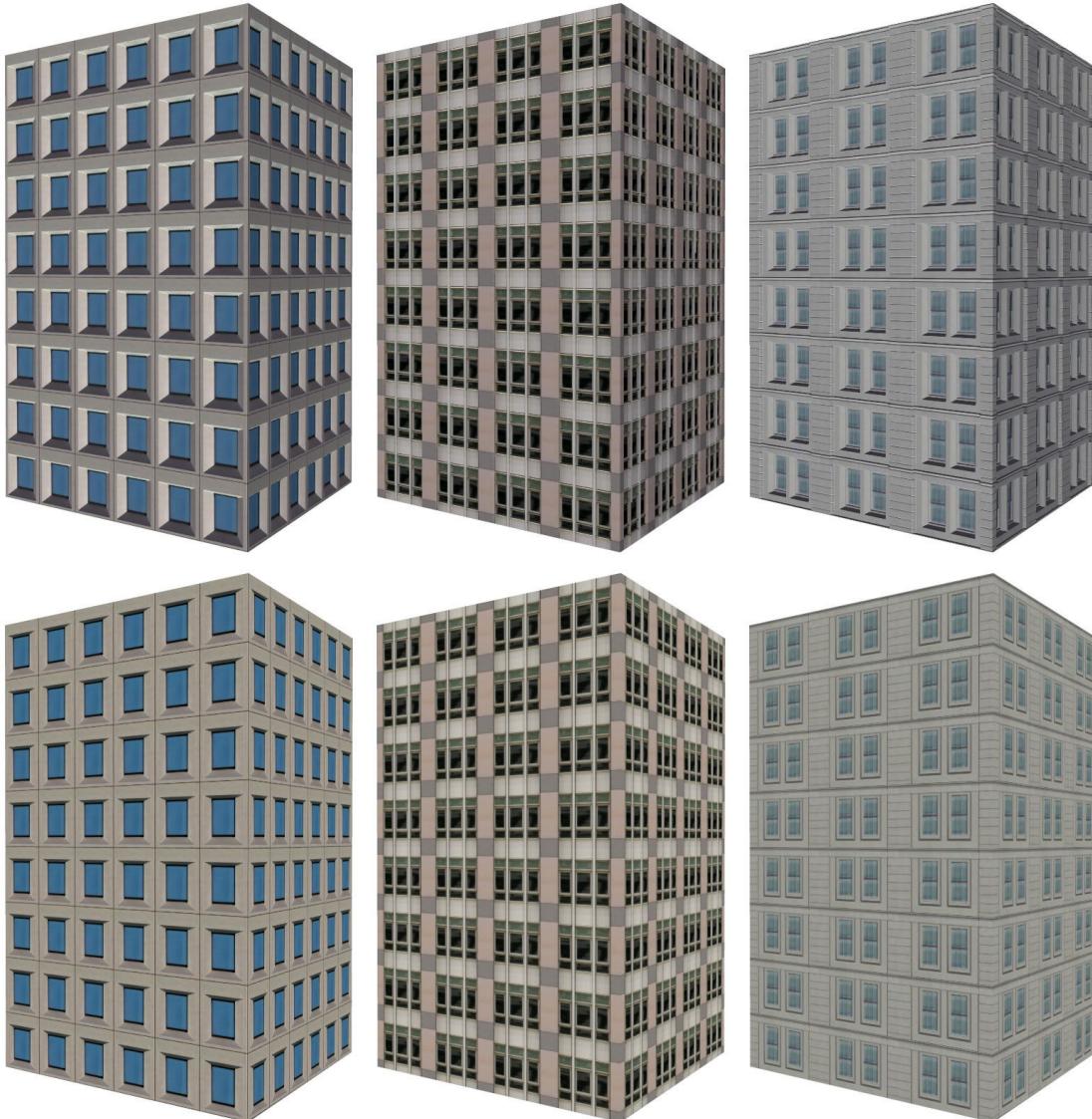
The motivation for the extensions was to extend the application of the lot division algorithm to a wider range of road networks. Networks like the suburban type, shown below in Figure 90, are significantly different from the regular grid-like networks. Here, the suburban generation process can frequently cause irregular, angular and concave regions. The lot subdivision algorithm needs to accommodate these concave regions and generate regular rectangular lots from irregular input. Prioritising the division along road access sides and supporting concave regions are essential to the success of the algorithm. The extensions to lot subdivision result in a substantial reduction of irregular lots and provide a feasible method of obtaining lots from suburban and other irregular road networks.



*Figure 90: Suburban Lot Subdivision – Lots Displayed in Debug View*

We now look at the results of the building tiles. After the lot subdivision process is complete the building footprints are calculated and basic geometric primitives are extruded. The resulting geometry is then textured with building tiles which simulate additional geometry to provide more realism and detail. Shown below in Figure 91, is a selection of building tiles

applied to basic cuboid geometry. The top row shows the building with relief mapping enabled and the bottom row shows the buildings without relief mapping enabled.



*Figure 91: A selection of Building Tiles*

Looking at the images above, the individual benefits that relief mapping provides can be observed. Namely, the added depth, parallax, occlusion and shadowing. Even though the flat textures contain some emphasised shading the relief mapping technique is distinctly superior and provides a higher level of detail and realism. In this section we have discussed the effectiveness of the lot subdivision extensions and shown the benefits of relief mapping on building tiles. In the next section we outline the general performance of our system and discuss the effect of the algorithms and optimisations on performance.

## 6.5 Performance

Performance is an important factor in our city generation system. Interactive control is one of our main objectives and in order to provide this feature we aimed to generate and render the city at interactive frame rates. This goal has been achieved, with the average city cell or neighbourhood now requiring only 60ms to generate and render. This means that the user can modify a city cell and view the results instantly. The generation of a complete city requires a little more time, around 1.6 seconds to create a test city of area 16Km<sup>2</sup> with 15,000 roads and 57,000 buildings. In this section we show the effects of the optimisations on the city generation times and briefly describe the most influential cases.

A performance summary of *Citygen* can be viewed in the table below. The generation times for each stage, with and without optimisations, are listed for comparison. The effects of the most influential optimisations are marked with a key, this character key points to a specific note for detail on each optimisation (see the captions below the table).

Generation Stage	Standard Execution Time	Optimised Execution Time
Primary Junctions	43.6ms	18.8ms†
Primary Roads	175ms	22ms†
Secondary Road Generation	771.8ms	149.8ms*
Lot and Building Generation	2050.7ms	415.6ms‡
Geometry Mesh Build	4468.6ms	1009.2ms†
Total	7509.7ms	1615.4ms†*‡

Figure 92: Performance Table: Samples averaged from a test run of five executed on Intel Core Duo 2.33GHz with an ATI FireGL V5200 (similar to X1600).

Optimisation Keys: \* Snap Algorithm Optimisations, ‡ Efficient Split Algorithm and † Inline Tangent Vector Calculations.

The Secondary Road Generation stage employs the snap algorithm extensively. However, the snap algorithm is an expensive operation and a number of optimisations were developed to improve its performance, a description of these optimisations can be found in section 4.3.3. In Figure 92, we can see the effect of these optimisations on Secondary Road Generation.

Here a five fold increase in performance over the standard snap algorithm can be observed when using the optimised snap algorithm. Clearly this performance benefit justifies the development and application of these the snap algorithm optimisations.

The Lot and Building Generation stage frequently calls the split algorithm. This algorithm is critical to the efficiency of lot subdivision. The operation of this algorithm is described in section 4.4.2. Improvements to the algorithm were made so that less memory allocations occur. This was achieved by using circular list structures in place of STL vectors. The performance increase that occurred with this optimisation was significant, the system can now generate lots and buildings at a rate four times faster than that previously experienced.

Tangent Vector Calculations are computed for every piece of geometry constructed. The tangent vectors are required by almost all of the advanced materials including relief mapping. The OGRE game engine, used in Citygen, supports the automated generation of tangent vectors. However, shared vertices were not supported, and as a result the entire geometry mesh needed to be relocated and re-indexed in order to generate the tangent vectors. This is a costly operation and should be avoided if at all possible. The solution used was to calculate the tangent vectors in-line as the faces are added to the mesh, thus avoiding the cost of relocation and re-indexing. This optimisation resulted in improved performance across several stages.

In this section we have shown the effects of our optimisations on overall city generation and on the individual generation stages. We have demonstrated that the optimisations developed provide a substantial performance boost to the system and are an essential component to enabling the short generation times that make interactive control possible. In the next section we will look at the integration of Citygen into industry standard development processes.

## 6.6 Development Integration

One the goals outlined for Citygen was to ensure that the system could easily slot into the development pipeline of the computer graphics industry. In the Implementation Chapter the accessible interface suitable for novice users was described. A camera model and point-and-click interface similar to those found in contemporary 3D authoring tools is used to make the application a familiar environment for those with experience in content authoring. In this section we analyse the usage data of 3D authoring tools and determine which tools can be used easily alongside the Citygen application.

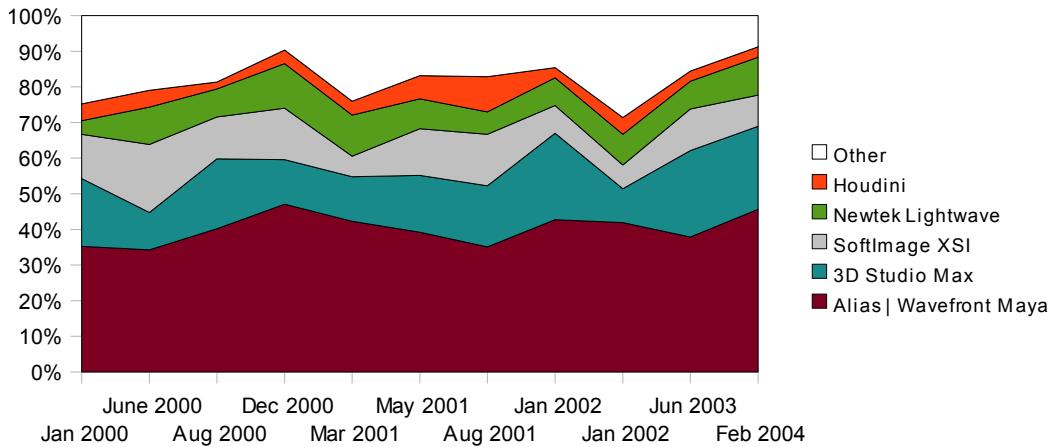


Figure 93: 3D Graphics Tools: Market Share

Figure 93 illustrates the market share of the leading 3D authoring tools. This data is acquired over a four year period from the specifications of job positions in the computer graphics industry. The table shown in Figure 94 displays which leading products support COLLADA, the export format used by Citygen. Using these two sets of data we can estimate the percentage of workers in the computer graphics industry that can use Citygen without any changes to their tool set or work practices. The result is a respectable 85%, the remaining 15% of workers can still use Citygen but may require an additional tool such as Deep Exploration to convert the model data to their own native format.

Product	Market Share	Built-in	Plug-in
<b>Maya</b>	42%	--	ColladaMaya
<b>3D Studio Max</b>	17.9%	--	ColladaMax
<b>SoftImage XSI</b>	12%	Yes	--
<b>Lightwave</b>	9.1%	--	LWCOLLADA
<b>Houdini</b>	4.7%	Yes	--

Figure 94: 3D Graphics Tools: Market Share

## 6.7 Analysis

In order to evaluate the results of the procedural city generation system we revisit the critical framework that was outlined in section 3.2.1. The same set of criteria that were used to evaluate existing city generation solutions are now applied to review the results of Citygen.

**Realism** – Citygen can accurately reproduce many complex and detailed road patterns that are found in modern cities. This is demonstrated in sections 6.2 – 6.5 where system output is

compared to real world examples. High level patterns are recreated accurately and quickly using the interactive road network editing facilities. Adaptive roads plot the route of each road to realistically fit to the terrain. Secondary road generation provides near identical matches for the Manhattan and Industrial examples. The Suburban example attains a very close resemblance but the planned nature of suburban road networks can pose some problems for the organic-like road growth algorithm. The Lot Subdivision stage identifies realistic lots without the limitations of shape of accuracy found in previous work. Buildings are basic but are designed to define structure for further generation. Overall the system is capable of generating cityscapes that closely match real-world examples with only a few weaknesses.

**Scale** – The city generation system is designed to scale. The graph algorithms partition the city into self-contained cells. This limits the negative effects of large scale networks on the efficiency of the generation process. True large-scale cityscapes covering an area of 16Km<sup>2</sup> with 15,000 roads and 57,000 buildings were tested in the performance section of 6.5 and did not pose any problems for the system.

**Variation** – The interactive road editor allows the user to specify any conceivable pattern in the high level road network. Using the parameter based secondary road generation a wide range of road networks can be constructed, varying from the rigid blocks of Manhattan to sprawling Suburbia. Building variety is limited to basic extrusions. Overall the system can produce a wide variety of road patterns and is unlimited in the style of output possible.

**Input** – Minimal input data is required by the system. No geo-statistical data or land usage maps are required. The only input needed is a terrain height map to place the city on. Users are responsible for defining the initial high level road network but this stage can be completed quickly using the interactive tools.

**Efficiency** – Citygen partitions the road network into cells, thus reducing the data-set for intersection tests. Several optimisations were also implemented, the snap algorithm discussed in section 4.3.3 proved to be the most influential. Only 1.6 seconds is required to create a test city of area 16Km<sup>2</sup> with 15,000 roads and 57,000 buildings. See Figure 92 for more details.

**Control** – An interactive road network editor is used for the primary road network. Real-time feedback and an intuitive interface provide a tactile system of control for the user. Using this system any high level road network pattern can be implemented quickly and easily. The adaptive road system is controlled via a few simple parameters and an overlay can be toggled

to view the sampling points and arcs. Secondary roads growth is again parameter based, although several pre-sets can be selected for frequently used patterns. To summarize the system provides an intuitive form of control for the high level road network but other generation stages are only controllable via input parameters.

**Real-time** – Complete cityscapes can be generated in near real-time, taking approximately 1.6 seconds. Individual neighbourhoods take significantly less time, at around 60ms per cell. Each cell batches geometry resulting in efficient rendering of the cityscape. Currently the system can be explored easily in real-time, running at around 50fps using the hardware configuration and cityscape described in Figure 92.

## 6.8 Final Output

In this final section of the results chapter we display a selection of screen-shots taken from the Citygen application. The city model shown in the screenshots contains over 82,000 buildings and 18,000 roads. The complete generation time for the city including adaptive roads, secondary roads and buildings is only 1.7 seconds using the same hardware as specified in Figure 92. In the next chapter, we shall provide some conclusions on the research.

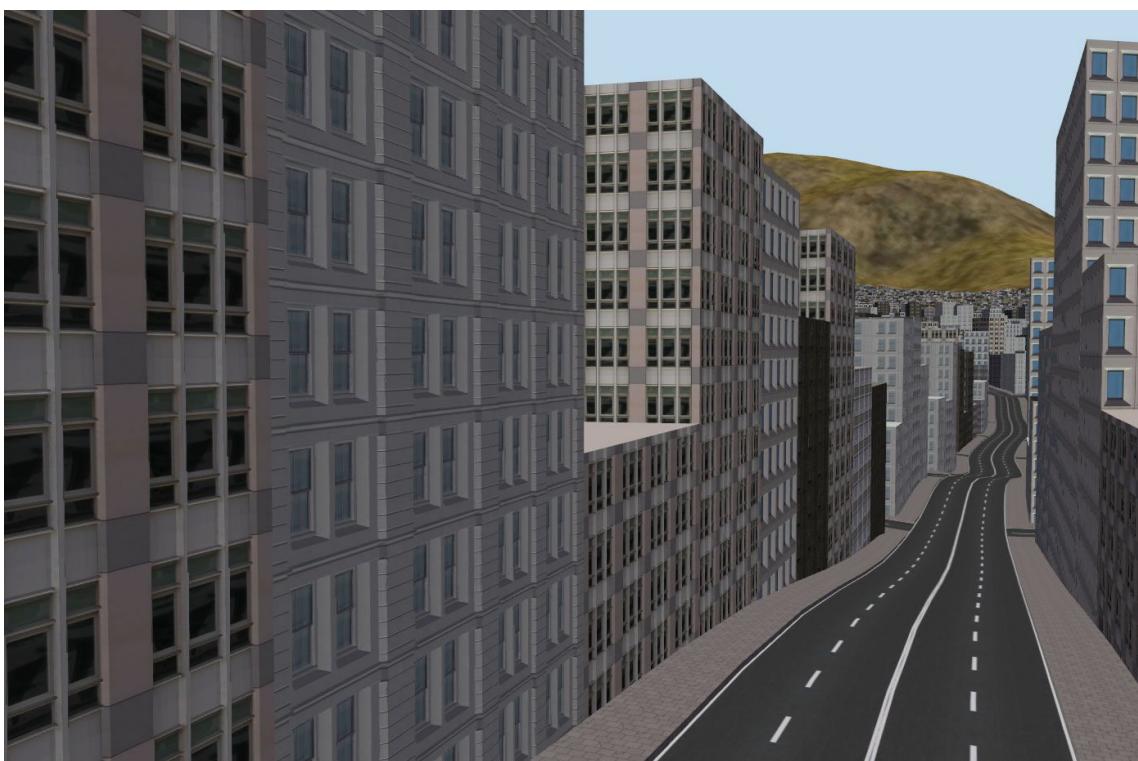
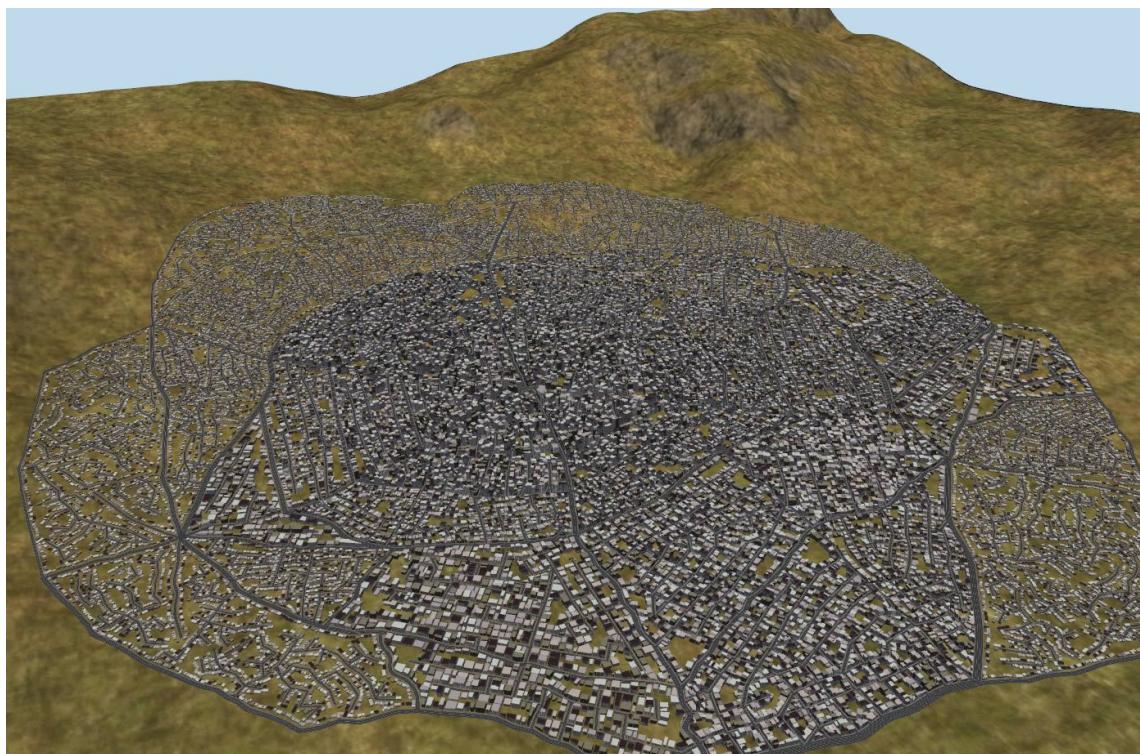


Figure 95: City Screenshot 1



*Figure 96: City Screenshot 2*



*Figure 97: City Screenshot 3*

# Chapter 7

## Conclusions

In this chapter we conclude the thesis by reviewing our initial aims and outlining the achievements of the project. There is also a project assessment, a discussion of future research and finally a review of the applications with a number of conclusions.

### 7.1 Summary

This thesis began with the context, motivation and scope of the research. The content creation problem facing the computer graphics industry was discussed and the field of procedural techniques was introduced as a potential solution. The additional benefits of procedural generation in computer graphics were also outlined. Furthermore the scope of the project was refined to the application of procedural techniques to construct an urban environment model suitable for real-time rendering. Next, an overview into the field of procedural techniques was included. This described the properties of procedural techniques and identified the key properties of effective procedural generation algorithms. To gain an insight into what makes procedural techniques effective, additional studies focused on the operation and results of successful established procedural algorithms such as Fractals, Perlin Noise, L-Systems and Tiling. A review of the existing research on procedural city generation was carried out and each approach was evaluated against a standard set of criteria: Realism, Scale, Variation, Input, Efficiency, Control and Real-time optimisations. The approaches reviewed included the real-time system by Greuter et al.[2003], the L-systems based CityEngine by Parish and Müller [2001] and the agent based approach by Lechner et. al.[2004]. From the analysis the relative strengths and weaknesses of each approach was identified. It was determined that no system fulfilled the aims of our research and thus further validated the unique goals of our

project. Following the review, the design of our system – *Citygen* – was presented. The process of city generation was decomposed into its constituent components. Primary roads were constructed using adaptive roads. Secondary roads were generated using a growth based algorithm in combination with a snap algorithm. A lot subdivision algorithm was applied to identify lots and basic buildings were constructed with advanced materials to simulate more geometry. The implementation of an accessible, interactive and real-time application titled *Citygen* was described. All major components and their role in the system were outlined. Key technologies such as the built-in game engine, graph data structures, GUI libraries and geometry export were also discussed. In the results section we demonstrated the system in action, showing: the construction of the primary road network, the application of adaptive roads and the generation of secondary road patterns. We also provided a review of system performance, measuring the effectiveness of the optimisations developed along the way, and followed with a short feature calculating the accessibility of the system to those in the graphics industry.

## 7.2 Achievements

Before assessing the major achievements of the project we first revisit the original goals as defined in Chapter 1. That was to develop a city generation system capable of producing the required geometry, materials and textures to model a cityscape. Specifically the system should procedurally generate a city model that is:

- Realistic – of similar appearance to a real city, in particular the road networks must reproduce or emulate a number of distinct styles found in real city road networks.
- Large Scale – the size of the generated model must be comparable to that of an actual city, not a town or small urban settlement. A minimum building count of 50,000.
- Detailed – a level of detail comparable to that of a modern games title: with geometry suitable for real-time rendering but detailed enough to maintain realism.

In addition, the system should operate so that it is:

- Accessible: easy to use for novices, no reliance on expertise and minimal input data.
- Interactive: automatic but can facilitate manual and tactile control.
- Real-time: minimal generation time for an expansive range of applications.

Finally, some practical objectives were to:

- Develop a portable multi-platform application that can serve as a complete integrated workspace for procedural city generation.
- Implement a system that can be rapidly developed and easily extended.
- Export data easily to fit into the pipeline of tools used in the graphics industry.

After reviewing the original goals we can now assess our system with these in mind.

The system applies a number of techniques to generate the geometry, materials and textures needed for a complete city model. Adaptive roads emulate the problems that difficult terrains can cause and, like the real world, force route plans to be compromised and fit to the environment. The result is a more realistic road path that introduces characteristics from the terrain and ensures that roads fit cohesively into the world. Secondary road generation operates in the enclosed regions of the main road network. This has been identified by several authors as a key boundary between neighbourhoods which can each possess their own distinctive style[Alexander et. al. 1977][Lynch 1960]. A growth based algorithm is employed to generate roads within the regions and a parameters sets are used to recreate several distinct styles taken from the real world. By using an editable primary road network with adaptive roads and self contained cells, characteristics can be specified on a high level for a large area. As a result, it is easy for users to create large-scale urban areas with distinct characteristics without the need to micro-manage finer details. *Citygen* enables the generation of urban geometry on a true city scale. Geometry is created for a limited number of assets such as roads, footpaths and buildings. This geometry is more basic than that found in the most modern games but it is suitable for real-time rendering and enables the display of the full city at interactive frame rates. Extra detail is accomplished via the use of advanced materials that simulate additional geometry. The most up-to-date techniques are used, some of which are only present in the most recent gaming titles. Techniques like relief mapping, parallax occlusion mapping and displacement mapping help increase the visual fidelity of the output model. These techniques applied in *Citygen* enable the procedural generation of a realistic, large-scale and detail city model.

The city generation system is operated via a unified interface with a live 3D view of the world. Any novice can use the point and click interface which is designed in a similar fashion

to mainstream 3D authoring tools. No input data, such as geo-statistical data or image maps are required. The primary road network can be constructed rapidly using an simple and intuitive interface that features *Auto-snap*. This mechanism aids the user by suggesting proposed actions and enforcing the integrity constraints of the graph. All neighbourhoods are automatically detected and the parameters for each cell can be viewed in the property inspector. Users can modify parameters and view live updates. An adjacency and dependency model is used so that only the portions of the city that need updating are regenerated after each modification. Using this system, any component of the city can be modified at any stage in the generation process and the changes are propagated through in real-time maintaining interactivity. Road network nodes can be grabbed and manipulated with real-time generation results, thus enabling an element of tactile control for the user. The generation algorithms applied in the system are designed to be computationally efficient and exploit multiple threads of execution. This effort is required to produce near real-time speeds for complete city generation and expands the range of applications for the system. All of these features combine to provide an accessible, interactive and real-time city generation system.

*Citygen* is a single application that serves as an integrated workspace to operate procedural city generation. A built-in game engine is included and the system has been tested on multiple platforms. Rapid development was accomplished by using established libraries and extensibility is guaranteed by adhering to open standards and transparent file formats. Export data fits easily into the graphics industry pipeline by using a popular open exchange format.

### 7.3 Project Assessment

This project accomplished its initial aims by completing the development of a procedural city generation system. The core aims of generating a city model that is realistic, large-scale and detailed were fulfilled. The success in each of these particular criteria is difficult to determine. Realism, has been analysed in the results section and it can be seen that a number of different patterns are represented and panoramic views over the cities appear realistic. Scale, is not in question, the system can easily generate cities with over 50,000 buildings and there is no hard limit on the maximum number it can handle. The only core aim left in doubt is that of detail. Evaluating detail can be subjective but if we look back to the original aims we can see that the detail level should be 'comparable to that of a modern games title'. Using this as a guide we can conclude that the city models generated by *Citygen* are on par with

many modern titles and superior to some genres such as flight simulators, etc. But, we must acknowledge that our solution could have produced better results to equal the detail found in every genre of modern gaming.

The operation of our system and implementation of the standalone city generation tool, *Citygen*, have been most successful. The resulting application meets the original aims, it is an accessible, interactive and real-time tool that can enable developers to pick it up immediately and begin creating their own cities. The system that has been developed is cross-platform and employs open XML standards to import and export data thus ensuring extension is possible.

## 7.4 Future Work

In this section, a number of areas for improvement and future research are outlined.

### Probability distribution

Random generators are used as part of the road growth and lot subdivision algorithms. The random values acquired are used to apply the deviation values that accompany parameters. Currently a uniform probability distribution is used in the *Citygen* application. This is adequate for many applications but it is not ideally suited to recreating the complex patterns found in cityscapes. A simple and obvious example of a parameter that can benefit from a different probability distribution is the '*building height*' parameter. In a real city it is evident that most buildings are of a similar height with only a few skyscrapers peeking out. If a normal distribution was used then this trait and many others could be replicated in the *Citygen* application.

Additionally, if specialised distributions were acquired from city statistical data then the generation system would be able to more accurately model real cityscapes with little additional computation required. Any code changes required to the *Citygen* application are trivial as the system has been developed to use the boost random generator libraries which support a number of built-in probability distributions and allow the use of custom distributions [Boost 2007]. However, more research is required to find and analyse statistical data from existing cites that can be used to validate this theory.

## GPU Execution

In the *Citygen* system the generation of a cityscape is accomplished in stages via a hierarchy of components. Throughout the city generation process, many of the components can be computed in parallel. The most substantial generation time is spent within the city cells. Each of these cells are designed as independent work units that can be allocated and shared among a number of threads to exploit multi-core processors. Currently the number of threads used in the application is limited to the number of cores on the host machine. It is no secret that increased parallelism and multiple cores are the predominant design trends in advancing CPU processing power.

Additionally, immediate gains can be made by exploiting an existing piece of parallel programmable hardware present in any gaming computer. With GPGPU (General Purpose GPU) programming it is possible to offload massive, specialised, parallel computation workloads onto the GPU for efficient calculation [Trendall and Stewart 2000]. With the advent of DirectX 10, Geometry Shaders have emerged as a new method to generate geometry on the GPU [Glassenberg 2006]. These changes in the graphics industry point to a determined move towards procedural techniques and in particular parallelisable procedural techniques. Thus, the next logical step forwards for research is the optimisation of suitable procedural generation algorithms, such as our implementation of city generation, for execution on the GPU.

## Real-time Rendering

Currently, the *Citygen* system executes the procedural generation of every city component and region in advance and a single level of detail is supported. However, using the native graph structures within the *Citygen* application, proximity and region data can be utilised to provide optimisations for real-time rendering. A technique is proposed, titled *City Cell Paging*, which is a method of geometry paging for cities. Similar to terrain paging algorithms the world is divided into regions and only the regions near to the camera need to be viewable and the remaining regions can be pre-emptively generated and loaded on demand using an adjacency model. To accomplish this *City Cell Paging* scheme, an efficient method to calculate the proximity of geometry and a structural partition of the city is required. The road graph data structures present in *Citygen* already contain the required adjacency data and can be used to access neighbouring regions directly without the need for expensive calculations. Each city produced by *Citygen* is procedurally generated in a series of regions and an inherent

hierarchical structure of regions composed of cells, blocks and lots is present. The adjacency model and hierarchical partition form the basic building blocks for such a scheme but additional research and development is required to implement the proposed system.

### **Building Generation**

The building materials applied in *Citygen* are selected from a very limited set. There are no technical constraints on the number of materials used, beyond the standard graphics memory limitations. The only reason for limited textures are the time requirements of their authoring. Material property sets can be edited in the *Citygen* application so user edited textures and relief maps can be easily added. An obvious improvement could be gained in the realism and variety of the output if larger, more varied or professionally authored texture sets were used.

The building geometries displayed in the *Citygen* application are primitive shapes defined by an extrusion of the proposed building footprint. These shapes provide basic visualisation for buildings but were primarily designed to provide a bounding volume that could be used later by a dedicated building generation component. Further research is required to develop a building generation system or to integrate an existing generation component.

### **7.5 Conclusions**

In this final section we outline the applications of our system and draw a number of conclusions on the success of our research. The procedural city generation system we have developed may not immediately solve the content creation problems facing the industry, but it can play a part by fulfilling a useful role in the construction of an urban model. *Citygen*, our standalone application, is ready to use and provides a tool for content authors to begin creating their own cities. The level of visual detail required can vary depending on the application and for this reason our tool is likely to be used in different ways. In cases where high complexity models are required: *Citygen* can be used to generate a base model, this can be loaded as a background or imported into a 3D authoring tool like Maya, via Collada, where more detail can be added. For applications where real-time rendering is a concern or where a lower complexity model is desired the system can be used to generate a complete city with user specified materials and terrain. Additionally, for applications such as flight simulators, where a large amount of low complexity models are required, the generation component could be linked into the users game engine and used to generate cityscapes on demand from concise definitions.

The approach taken by our research has proved to yield a useful system that is easy to apply to the procedural generation of cities. The operation of *Citygen* is accessible and familiar to those working in the content creation field of the graphics industry. Obviously the generation system is not without faults and we have outlined a number of areas where future research would be beneficial. However, *Citygen* demonstrates that procedural techniques can create realistic cityscapes suitable for real-time rendering and proves that interactive and tactile control for procedural city generation is both possible and desirable. Finally, as we look to the future, procedural techniques will undoubtedly become more predominant as researchers and developers encapsulate the behaviour of increasingly complex phenomena into code.

# References

- AICHHOLZER O., FRANZ AURENHAMMER, ALBERTS D. AND GÄRTNER B. 1995. A Novel Type of Skeleton for Polygons. *Journal of Universal Computer Science*, Graz University Of Technology, 752-761.
- AKENINE-MÖLLER T. AND ERIC HAINES 2002. *Real-Time Rendering*, A K Peters, Ltd., Wellesley, MA, USA.
- ALEXANDER C., ISHIKAWA S. AND SILVERSTEIN M. 1977. *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series)*, Oxford University Press, New York, NY, USA.
- AMD 2007. *Accelerated Computing Solution that Breaks Teraflop Barrier*. [http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51\\_104\\_543~116238,00.html](http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543~116238,00.html). Last accessed: 22 Aug 2007.
- BARNESLEY M. 1988. *Fractals Everywhere*, Academic Press Professional, Inc., San Diego, CA, USA.
- BOOST 2007. *Boost C++ Libraries, free peer-reviewed portable C++ source libraries*. <http://www.boost.org>. Last accessed: 14 Oct 2007.
- EBERLY D. 2005. *The Minimal Cycle Basis for a Planar Graph*. Geometric Tools, Inc..
- EBERT D., MUSGRAVE K., PEACHY D., PERLIN K. AND WORLEY S. 2003. *Texturing & Modelling - A Procedural Approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- FÁBIO POLICARPO, MANUEL M. OLIVEIRA AND JÓAO L. D. COMBA 2005. Real-Time Relief Mapping on Arbitrary Polygonal Surfaces. In *Symposium on Interactive 3D Graphics and Games*, ACM SIGGRAPH, 155-162.
- FARBRAUSCH 2007. *.debris - Nominee for Scene.org Awards 2007 in the Category of Best Demo*. <http://www.theprodukkt.com/kkrieger>. Last accessed: 15 Nov 2007.
- FELKEL P. AND OBDRZALEK S. 1998. Straight Skeleton Implementation. In *14th Spring Conference on Computer Graphics (SCCG'98)*, 210-218.
- GAMASUTRA 2002. *New Middleware Tool For Rendering Trees*. [http://www.gamasutra.com/php-bin/news\\_index.php?story=1314](http://www.gamasutra.com/php-bin/news_index.php?story=1314). Last accessed: 5 Apr 2006.
- GLASSENBERG S. 2006. DirectX Graphics: Direct3D 10 and Beyond. In *Proceedings of WinHEC 2006*, Microsoft Corporation.
- GREUTER S., PARKER J., STEWART N., AND LEACH G. 2003. Real-time procedural generation of 'pseudo infinite' cities. In *Proceedings of GRAPHITE 2003*, ACM Press, 87-95.

- GREUTER S., STEWART N. AND LEACH G. 2004. Beyond the Horizon: Computer-generated, Three-dimensional, Infinite Virtual Worlds without Repetition. In *Image Text and Sound Conferance 2004*, RMIT Publishing.
- HART E. 2002. 3D Textures and Pixel Shaders. In *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*, Wordware, Plano, Texas.
- INTERACTIVE DATA VISUALIZATION INC. 2006. *SpeedTree RT*. <http://www.speedtree.com>. Last accessed: 22 Aug 2006.
- INTERNATIONAL SCENE ORGANIZATION 2004. *Scene Awards 2004*. <http://scene.org/awards.php?year=2004>. Last accessed: 10 Aug 2006.
- LECHNER T., WATSON B., WILENSKY U. AND FELSEN M. 2003. *Procedural City Modeling*. Northwestern University.
- LEFEBVRE S. AND NEYRET F. 2003. Pattern Based Procedural Textures. In *ACM-SIGGRAPH Symposium on Interactive 3D Graphics (I3D)*, ACM Press, 203 - 212.
- LINDEN W. AND SCHACHINGER E. 2002. *Fractals - Computersimulations*. [http://itp.tugraz.at/LV/wvl/Comp\\_Simulationen](http://itp.tugraz.at/LV/wvl/Comp_Simulationen). Last accessed: 10 Sep 2007.
- LINDENMAYER A. 1968. Mathematical models for cellular interaction in development, Parts I and II. In *Journal of Theoretical Biology*, Vol. 18, No. 3, 280-315.
- LLUCH J., CAMAHORT E. AND VIVÓ R. 2003. Procedural multiresolution for plant and tree rendering. In *AFRIGRAPH '03: Proceedings of the 2nd international conference on Computer graphics, virtual Reality, visualisation and interaction in Africa*, ACM.
- LYNCH K. 1960. *The Image of the City*, Cambridge: MIT Press, Cambridge, MA, USA.
- MANDELBROT B. 1982. *The Fractal Geometry of Nature*, W.H. Freeman & Co., New York, NY, USA.
- MÜLLER P. 2006. *Pascal Muellers Wiki - Procedural Modeling of CG Architecture - The CityEngine*. <http://www.vision.ee.ethz.ch/~pmueller/wiki/pmwiki.php/CityEngine>. Last accessed: 8 Jun 2006.
- NVIDIA 2007. *GeForce 8800 Frequently Asked Questions*, Nvidia. [http://www.nvidia.com/object/8800\\_faq.html](http://www.nvidia.com/object/8800_faq.html). Last accessed: 6 Nov 2007.
- O'Rourke J. 1998. *Comp.Graphics.Algorithms - Frequently Asked Questions*. <http://www.faqs.org/faqs/graphics/algorithms-faq/>. Last accessed: 10 Nov 2007.
- OGRE 2007. *OGRE (Object-Oriented Graphics Rendering Engine) is a scene-oriented, flexible 3D rendering engine*. <http://www.ogre3d.org>. Last accessed: 6 Sep 2007.
- PAETH A. 1995. *Graphics Gems V: IBM Version*, Academic Press, Inc., Orlando, FL, USA.
- PANDROMEDA. 2006. *Mojo World Applications*. Published by Pandromeda Software.
- PARISH Y. AND MÜLLER P. 2001. Procedural Modeling of Cities. In *Proceedings of ACM SIGGRAPH 2001*, ACM Press, 301-308.

- PERLIN K. 1999. *Making Noise*. Based on a talk presented at GDCHardCore on Dec 9, 1999. <http://www.noisemachine.com/talk1/index.html>. Last accessed: 25 Mar 2006.
- PERLIN K. 1985. An Image Synthesizer. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, ACM, 287-296.
- PLANETSIDE 2004. *Terragen is a scenery generator capable of photorealistic results*. <http://www.planetside.co.uk/terragen>. Last accessed: 10 Aug 2006.
- PRUSINKIEWICZ P. AND LINDENMAYER A. 1990. *The Algorithmic Beauty of Plants (The Virtual Laboratory)*, Springer-Verlag, New York, NY, USA.
- SENSIBLE SOFTWARE 1987. *Shoot'Em-Up Construction Kit (SEUCK)*. Published by Outlaw.
- SIDE EFFECTS 2005. *Houdini*. Published by Side Effects Software Ltd.
- SONY COMPUTER ENTERTAINMENT 2007. *COLLADA, XML-based schema to make it easy to transport 3D assets between applications*. <http://www.collada.org>. Last accessed: 08 Aug 2007.
- SPITZER J., GREEN S. AND NVIDIA CORPORATION 2003. Noise and Procedural Techniques. In *Proceedings of Game Developers Conference 2003*, GDC.
- STINY, G. 1980. Introduction to shape and shape grammars. *Environment and Planning B*, Pion Ltd, 343-361.
- SUN J., XIAOBO Y., BACIU G. AND GREEN M. 2002. Template-based generation of road networks for virtual city modeling. In *VRST-02*, ACM Press, 33-40.
- TOMOMICHI KANEKO, TOSHIYUKI TAKAHEI, MASAHIKO INAMI, NAOKI KAWAKAMI, YASUYUKI YANAGIDA, TARO MAEDA AND SUSUMU TACHI 2001. Detailed Shape Representation with Parallax Mapping. In *ICAT 2001 (The 11th International Conference on Artificial Reality and Telexistence)*, The University of Tokyo, 205–208.
- TRENDALL C. AND STEWART A.J. 2000. General calculations using graphics hardware, with application to interactive caustics. In *Eurographics Workshop on Rendering*, Springer, 287-298.
- URBANLAB 2006. *www.chil.us = Chicago, Illinois, USA*. <http://www.chil.us>. Last accessed: 21 Jul 2007.
- WONKA P., WIMMER M., SILLION F. AND RIBARSKY W. 2003. Instant Architecture. In *Proceedings of ACM SIGGRAPH 2003*, ACM Press, 669-677.
- WRIGHT W. 2005. The Future of Content - An Introduction to Spore. In *Game Developers Conference*, CMP Game Group.
- WXWIDGETS 2007. *wxWidgets (formerly wxWindows), a widget toolkit for creating graphical user interfaces*. <http://wxwidgets.org>. Last accessed: 16 Oct 2007.