FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Artist-Controlled Modeling of Urban Environments

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computergraphik/Digitale Bildverarbeitung

eingereicht von

Johannes Scharl
Matrikelnummer 0325783

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Univ.Ass. Dipl.-Ing. Mag.rer.soc.oec. Dr. tech. Daniel Scherzer

Wien, 21.07.2010

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Erklärung zur Verfassung der Arbeit

Johannes Scharl

Antonsplatz 10/14-15
1100 Wien

"Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen – die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe."

Ort, Datum:

Unterschrift:

Abstract

Creating large-scale virtual environments for interactive applications such as computer games poses a demanding challenge for computer graphics. Urban environments are usually hand-crafted by artists using commercial 3D modeling software. For today's detail-rich games, this becomes less and less feasible. Procedural modeling techniques strive to help artists to create virtual worlds in less time.

In this thesis, I present a system that helps artists and game designers to plan, layout and model urban environments for games and other media. Methods are described to create street networks manually and procedurally and to edit them interactively at any time in the development process. A stable street tessellation technique is employed that is able to represent street segments as well as crossings connecting an arbitrary number of streets and that adapts to the underlying terrain. Furthermore, I propose a constraint based method to automatically populate a city with buildings from a set of existing building models.

Kurzfassung

Das Erstellen von weitläufigen virtuellen Umgebungen für interaktive Anwendungen wie Computerspiele stellt eine große Herausforderung für die Computergraphik dar. Solche urbanen Umgebungen werden meist per Hand von Level Artists mithilfe von kommerzieller Modellierungssoftware erstellt. Das ist jedoch für detailreiche, moderne Spiele kaum noch machbar, da es sich dabei um einen sehr zeitintensiven Prozess handelt. Prozedurale Methoden versuchen, Artists das Gestalten von virtuellen Welten zu vereinfachen.

In dieser Diplomarbeit stelle ich ein System vor, das Artists und Game Designern dabei hilft, urbane Umgebungen für Spiele zu planen und zu modellieren. Es werden Methoden beschrieben, wie ein Straßennetz manuell und prozedural erzeugt werden und zu jeder Zeit im Entwicklungsprozess bearbeitet werden kann. Weiters stelle ich eine Methode vor, mit der eine geometrische Repräsentation von Straßen generiert wird, die sowohl Straßensegmente als auch beliebige Kreuzungen darstellt und sich an das darunterliegende Terrain anpasst. Außerdem beschreibe ich ein System, das eine Stadt automatisch mit Gebäuden aus einem Set von bestehenden Modellen bestückt.

Contents

Chapter 1

Introduction

In today's video games and other media such as movies, large urban environments often play a central role. Titles like *Grand Theft Auto* or *Assassin's Creed* feature huge, highly detailed cities and other surroundings as can be seen in Figure 1.1. Usually, such AAA titles are created by a huge workforce[1].

Fig. 1.1: A screen shot of an historic city from the game *Assassins Creed*. Image courtesy of Ubisoft S.A.

Urban environments such as cities and villages are usually created by hand in commercial 3D modeling applications such as Autodesk Maya [Aut10] or custom level editors created by the developers. This involves modeling each building individually, creating street geometry manually, and placing buildings and other models in the scene. While this is feasible work for games with a linear, bounded level structure, creating

[1] In 2007, Ubisoft Montreal, the studio behind the Assassins Creed games, employed 1600 people.

an open world for large sandbox[2] games is a huge and tedious effort that involves many man-years of artist work. For modern applications that demand even larger and more detailed environments, this gets less and less suitable.

A promising approach that has emerged in recent years is to create content procedurally. Procedural modeling is a method to describe an object not as static data, but as a sequence of instructions needed to build that object. A simple example is the snowflake shown in Figure 1.2.

Starting with an initial triangle shape, each line is recursively replaced with a generator shape. After a few iterations, a very detailed snowflake is constructed.

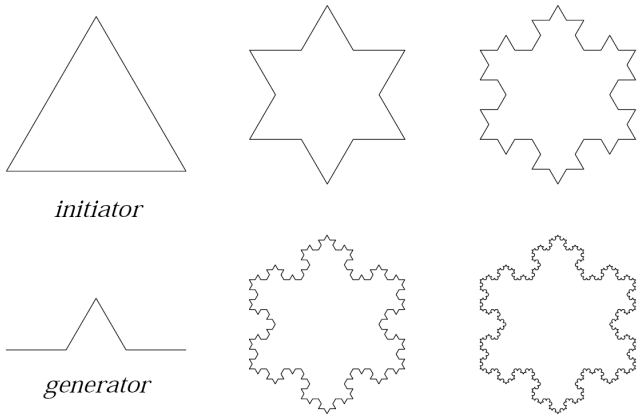

Fig. 1.2: Procedural construction of a snowflake shape. Image courtesy of P. Prusinkiewicz and A. Lindenmayer [PL91].

Game design is an iterative process. This means that such environments are often changed and revised during the development of the game. Until its release, a game passes through many phases, as described by Bethke et al. [Bet03]:

- *Concept Prototype*: shows very basic gameplay features.
- *First Playable*: An early version containing representative gamplay and assets.
- *Alpha*: Key gameplay functionality is implemented and contains the major features.

[2] A sandbox game is video game where the player can freely roam a large, virtual world.

- *Beta*: Feature and asset complete version of a game, three to four months before the game is released.
- *Game Release*: The final version of the game.

Usually, the game is tested by the quality assurance team after every single phase, and the feedback is incorporated into the further development. Examples of such changes may include:

- Moving a street because the walking distance for the player was too long or the player got lost.
- Restructuring a whole part of a city because a mission was cut or redesigned.

These are just two very specific examples of changes that can appear in the game development process, but they prove an important point: Urban environments can and will change often after they have been initially created. This means that the whole area affected by the change has to be re-modeled:

- Buildings and other models have to be moved manually.
- Street geometry has to be re-created by hand.
- Connections to other streets have to be updated.

All this involves even more time and work, because every step has to be done manually by an artist. A system or tool that would allow the designer to make some central changes and that updates the surroundings automatically would greatly reduce the time spent on revising the scene. This would give artists the time to focus on key assets even more and thus create a better user experience.

1.1 Typical Artist Workflow

While working at the game studio *Team Vienna* as a programmer, I interviewed coworkers experienced in game design. I asked them how a city environment for a game is usually planned, layouted and built. Especially helpful was Julian M. Breddy, lead level designer at Team Vienna. He was supervising the city design and creation for the game *7Million* and gave me valuable insight on how artists and designers approach the creation of a large urban environment for a game. A screen shot from 7Million can be

Fig. 1.3: A screen shot from the game *7Million*. Image courtesy of Koch Media GmbH.

found in Figure 1.3.

A typical design process works as follows - it is basically a top-down approach:

1. The first draft is a very coarse map that just color-encodes certain regions of the city (e.g. *little Italy*, *suburbans*, *slums*, *industrial park*) and the arrangements of certain districts (that are guided by game design/progress mainly).

2. For each major city part, a more detailed map is created that will show the positions of some landmarks (*stadium*, *office tower*, etc.) and some buildings or locations important for gameplay. No street layout is created yet, these maps contain just basic information about topology (parks, rivers, harbors, etc).

3. The city is divided into smaller parts: These are patches of a few hundred meters side length. For each of these sections, a layout is created that shows the approximate location of streets and buildings. Interestingly, creating this layout always begins with the landmarks (mostly one per section) mentioned earlier, the buildings and streets "grow" around the landmark.

4. Now, the creation process begins, every section is modeled in the editor. This is done by importing assets created by artists and placing them on the map. Streets are generated by drag-and-dropping texture tiles onto the ground, a very tedious process.

5. Local changes are made very often: sometimes buildings or the arrangement of buildings change, a road is narrowed, some gameplay elements are changed or cut completely, etc.

6. There might even be global changes necessary, although this is less frequent. Sometimes it turns out that walking distances are too long, or the space is too tight, etc. Redesigning a previously modeled section is very tedious: rearranging all texture tiles is a lot of work that could be greatly accelerated if those streets had been modeled procedurally.

A cutout of a typical game map is shown in Figure 1.4. Note the landmark annotations such as police stations, restaurants, etc.

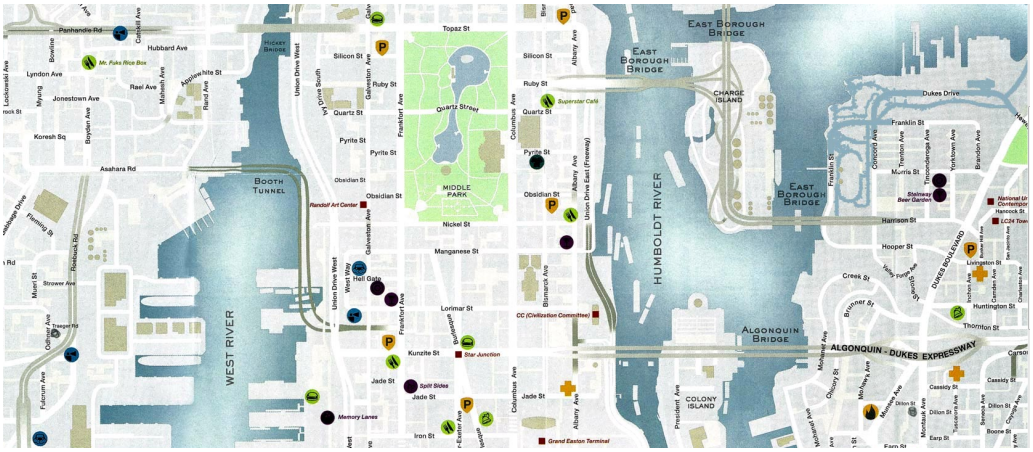

Fig. 1.4: A map from the game *Grand Theft Auto IV*. Image courtesy of Take-Two Interactive Software Inc.

Based on these insights and information, I created an editor that can be used by artists or designers for the whole design process. It supports all the described steps, from planning and layouting maps, creating the street network procedurally or manually to placing buildings and other models. Also, most details, including streets, can be changed easily and interactively at any time.

1.2 Scope of this Thesis

Based on the artist workflow described in Section 1.1, and limitations of similar, previously existing work that I will describe in Section 2.5, the following issues were the main focus for my thesis:

Planning and Layouting a City Map The user can import and display various maps in the application. This includes height maps for terrain generation as well as maps denoting gameplay elements, city zones or any other kind of spatial information, such as special points of interest. Additionally, markers can be set to identify important spots for game design or city layout, such as landmarks.

Street Network Creation Streets can be created procedurally or manually by placing nodes on the ground. Parameters can be defined for road growth, such as the road pattern, length, angles, etc. Also, it is possible to simply define some main roads and to apply the procedural growth in between.

Street Network Editing The user is able to edit the street network interactively at any time. This includes adding or deleting streets, moving junctions, etc. The affected streets are updated immediately.

Street Geometry Both manually and automatically created streets have a simple geometric representation. Creating complex parametric geometry is outside the scope of this work, but a stable tessellation is created that is able to represent street segments as well as crossings connecting an arbitrary number of streets. It is also important that the street geometry adapts to the underlying terrain.

Building Placement The user is able to place buildings and other models everywhere in the scene as well as to move, rotate and scale them. Additionally, it is possible to populate the city automatically with buildings using a constraint based system.

1.3 Contributions

The following list provides an overview of the major contributions:

- Existing applications and methods provide insufficient tools for artists that allow them to plan an urban environment before modeling it. In Chapter 3, I will describe techniques that help artists and game designers to plan and layout a city: height maps can be imported for terrain generation and maps can be projected onto the ground. This includes area maps, topology information, or maps denoting gameplay elements. Additionally, points of interest can be indicated with markers.

- Urban environments are mainly defined by their street network. Such a network forms the back bone of a city and determines its layout. Therefore it is the first thing that has to be generated when modeling a city. Usually a top-down approach is used, meaning that major roads are created first, because they define the main routes and districts in the city. Regions surrounded by major roads are then filled with minor roads, creating the finer structures of districts and neighborhoods. When using this approach, areas surrounded by major roads are usually located at the city center. This often leads to sparse regions at the outskirts of the city, where no minor roads can be created. In Section 4.2.5 an approach will be proposed that generates cities where minor roads are also generated in the outer regions.

- After the street network is generated, street geometry is created and the blocks in between are subdivided into building parcels. Tessellating the street geometry is straightforward in the case where all streets are in a single plane, but gets complicated if streets adapt to three dimensional terrain. In this case, junctions have to be kept planar. This is usually handled by forcing the junction geometry to be parallel to the ground plane, resulting in unnatural steps in steeper roads. Section 5.2 introduces a way to adjust these junctions in order to adapt them to the underlying terrain.

- Buildings are an essential part of every urban environment. Usually each model is hand-crafted in a modeling software and placed at its destination by a designer or artist. This approach is not feasible for larger urban environments. Usually only certain regions of the city are crucial for a computer game, while other regions do not contain a lot of individual detail. A method that fills these regions automatically can reduce the time needed to add building models to a road network.

  In Chapter 6, a method to select a building from a set of existing models that fits best for a certain building parcel and places it there, will be described.

- As mentioned in Section 1.1, the street network as well as the building models are often modified during game development. Chapter 7 describes various methods for editing the street network interactively. Changes in the street geometry as well as the surrounding models can be viewed in real-time.

1.4 Structure of this Thesis

The thesis is structured into different chapters as follows:

1. Chapter 2 gives an overview of previous methods used for urban modeling, including procedural techniques.
2. Chapter 3 explains how an artist can plan and layout urban environments in my application using image maps and markers.
3. Chapter 4 deals with the procedural creation of street networks and parcels and the necessary parameters.
4. Chapter 5 shows how street geometry is tessellated and adjusted to the underlying terrain.
5. Chapter 6 introduces a novel method to automatically populate an urban environment with previously modeled buildings using a constraint based system.
6. Chapter 7 explains the different methods that can be used to edit existing street networks and buildings.
7. Chapter 8 deals with implementation details and presents results on the basis of two urban environment examples.
8. Chapter 9 concludes this thesis and shows some possible future enhancements.

Chapter 2

State of the Art in Urban Modeling

Using procedural methods for urban modeling is a rapidly evolving field in computer graphics with many different applications, such as computer games, movies or urban planning. In the following, I will describe various approaches and aspects of urban modeling related to the methods I have employed in this work.

A detailed overview of various urban modeling techniques can also be found in a recent survey paper by Vanegas et al. [VAW+10].

2.1 Street Networks

The layout of an urban environment is based on its street network. Before actual street geometry can be generated, a street network has to be created either automatically or manually by a designer or an artist. Street networks are usually modeled as planar graphs. They specify position, orientation and length of street segments as well as junctions and often contain additional information, such as street widths, number of lanes, etc.

In the following sections, I will describe various previous methods used to create and edit street networks.

2.1.1 L-Systems

The work described in this thesis is based on previous procedural modeling methods that employ *L-systems*. L-Systems where originally developed as a formalism for plant modeling [PL91] such as the tree shown in Figure 2.1.

L-Systems are parallel string rewriting systems that consist of a starting string ω and so-called *production (or rewriting) rules*. Each production rule consists of a *predecessor* and a *successor string*. In each rewriting step, each character in ω is replaced by its successor string of a matching production

Fig. 2.1: Tree procedurally generated using an L-System. Image courtesy of P. Prusinkiewicz and A. Lindenmayer [PL91].

rule. See the following example of an L-System Lindenmayer used to model algae growth with production rules p_1 and p_2:

$$\omega : A$$
$$r_1 : A \rightarrow AB$$
$$r_2 : B \rightarrow A$$

The first 4 iterations of this L-System look like this:

$$n_0 : A$$
$$n_1 : AB$$
$$n_2 : ABA$$
$$n_3 : ABAAB$$
$$n_4 : ABAABABA$$

In order to create geometry from L-Systems, the resulting strings can be interpreted using a *LOGO-style turtle* [Ad86]. The basic idea of a turtle interpretation is that specific characters represent certain commands that are used to control the movement of the turtle:

- F: Move forward and draw a line
- +: Rotate to the right
- -: Rotate to the left

These commands can be used with the following L-System to create a dragon curve. Variables E and G represent the same turtle instruction as F, but have different production rules:

$$\omega : EF$$
$$r_1 : E \rightarrow E$$
$$r_2 : F \rightarrow F + GE$$
$$r_3 : G \rightarrow EF - G$$

The results of this L-System after 8 iterations is shown in Figure 2.2.

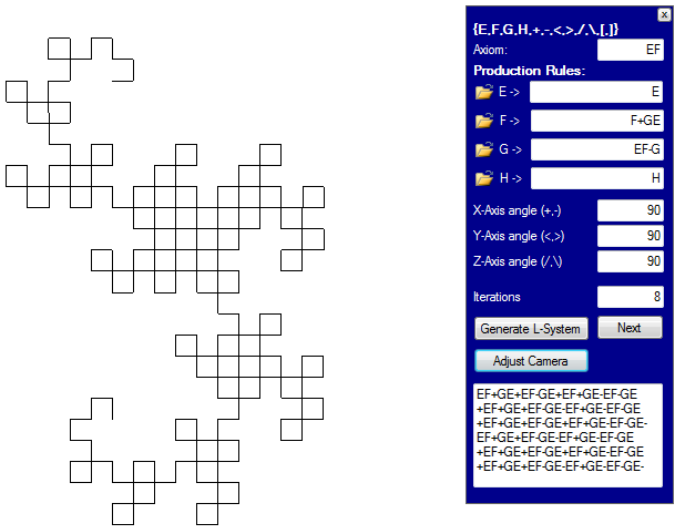

Fig. 2.2: Dragon curve after 8 iterations.

Using more complex commands and geometric representations, L-Systems can be easily used to create various 3D objects, such as the tree shown in Figure 2.3.

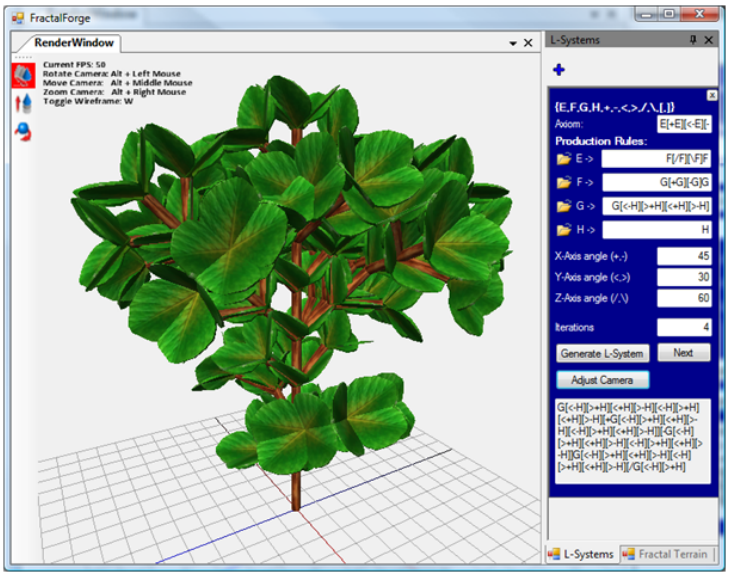

Fig. 2.3: Tree generated using a more complex L-System.

2.1.2 Using L-Systems to Create Street Networks

Měch et al. [MP96] introduced an extension to L-Systems that allows plants to communicate bidirectionally with their environment, called *Open L-systems*. The key concept of this method is the exchange of environmental information between plants and their surroundings, so that they influence each other. One example for this are tree roots growing in soil, absorbing water from it and following the gradient of water concentration [MP96]. This is shown in Figure 2.4.

To achieve this, communication modules of the form $?E(x_1, ..., x_m)$ are embedded into the string and used to both send and receive environmental information represented by the parameters $x_1, ..., x_m$ [MP96]. This is illustrated in Figure 2.5. Using this technique, a variety of novel plant behavior can be modeled, such as collisions between branches or the development of tree crowns competing for light.

Parish et al. [PM01] have extended this method to create city street maps using a set of production rules, calling their method *Extended L-Systems*. Street networks can be grown just like plants by developing new street segments and interacting with the environment to create crossings, avoid obstructed areas and steep slopes, etc. Instead of trying to set the parameters of the L-System inside the production rules, they proposed a 3-level hierarchy to evaluate parameters for a new street segment: First, an *ideal successor* is

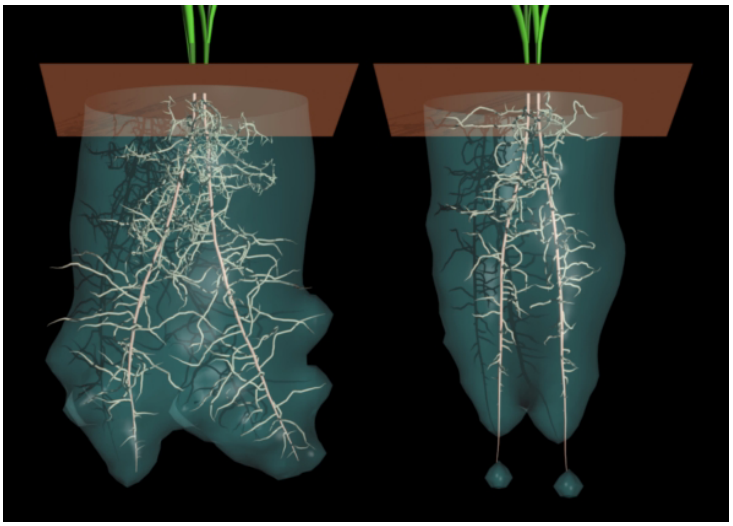

Fig. 2.4: Plant roots interacting with their environment using Open L-Systems. Image courtesy of R. Mech and P. Prusinkiewicz [MP96].

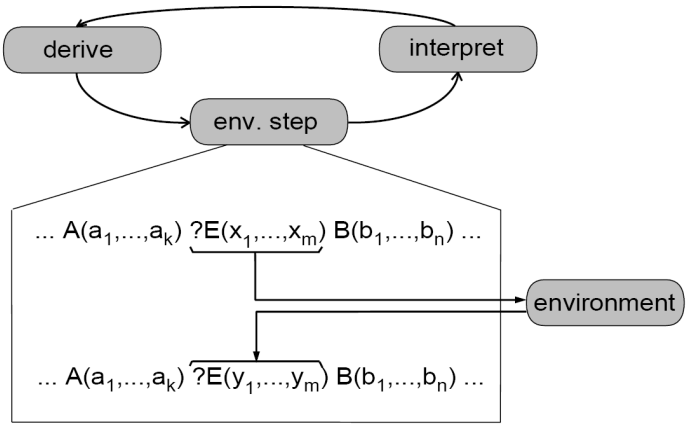

Fig. 2.5: Information can be exchanged between plants and their environment. Image courtesy of R. Mech and P. Prusinkiewicz.

created. This is a new street segment without any parameters assigned. For this new street segment, the *global goals function* is evaluated. The location and orientation of the street is set according to the superimposed street pattern and the local population density. After the initial parameters have been evaluated, the street segment is adapted to its local environment by calling the *local constraints function*: This function changes the location and orientation of the street according to the criteria such as the following [PM01]:

- check if the road segment ends in obstructed area
- check for intersections with other road segments
- snap new segment end points to existing junctions

Some of these constraints are illustrated in Figure 2.6. Extended L-Systems are capable of producing impressive results, such as the street maps shown in Figure 2.7

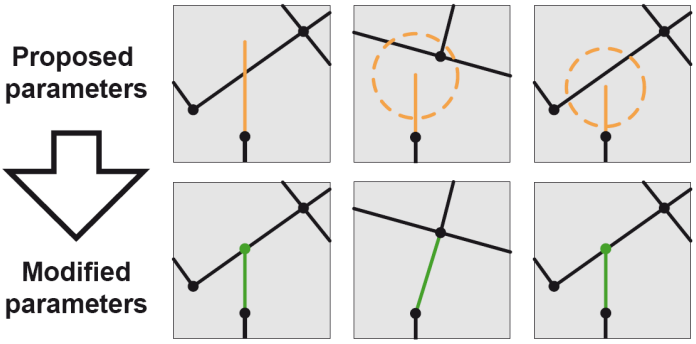

Fig. 2.6: Examples of local constraints. Image courtesy of Y. I. H. Parish and P. Müller [PM01].

2.1.3 City Hierarchy

Weber et al. [WMWG09] have extended this method to simulate a three-dimensional urban model over time. They define a city hierarchy to guide the creation of such networks:

A *street network* is a planar graph (V, E) with nodes V and edges E. A street consists of one or more edges $e \in E$, the *street segments*. Each street segment e connects two nodes $n(e)_1, n(e)_2 \in V^2$. Streets can be *major* or *minor* streets and have different *widths*. A facet in the planar graph (V_{major}, E_{major}) is referred to as *quarter*, that is an area surrounded by major roads. A facet in (V, E) (an area surrounded by any street) is called a *block*. Each block can be subdivided into *building parcels*. This hierarchy is illustrated in Figure 2.8.

To create a city obeying this hierarchy, major streets are created first, and spaces in between are filled with minor roads afterwards. This results in a planar street network and building blocks in between. A block consists, according to this hierarchy definition, of multiple building parcels, each defining the space a single building can take up. These parcels can be generated by repeatedly subdividing a building block.

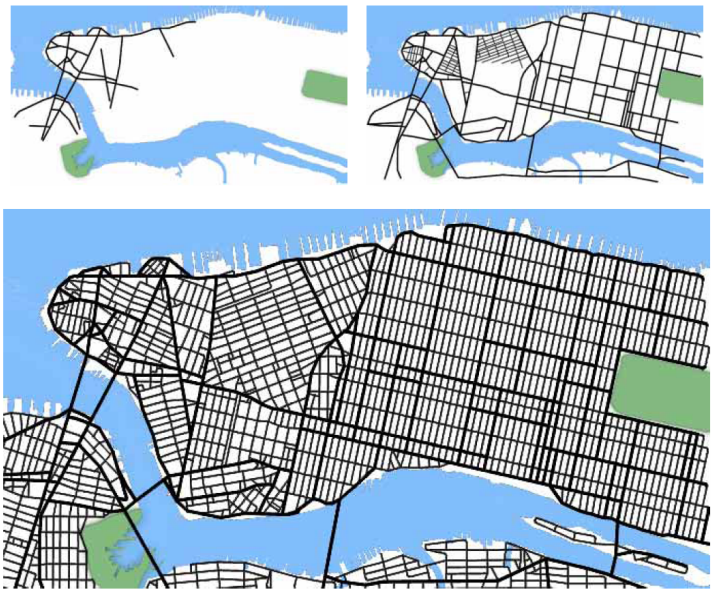

Fig. 2.7: Street network created using extended L-Systems. Image courtesy of Y. I. H. Parish and P. Müller [PM01].

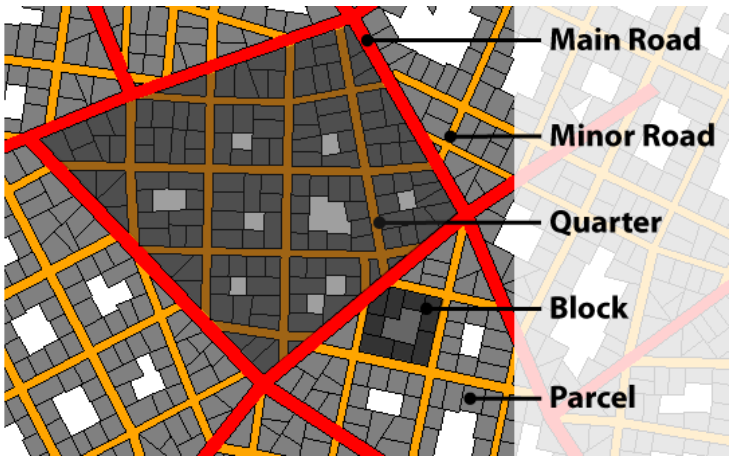

Fig. 2.8: City Hierarchy. *Main roads* are displayed red, *minor roads* are orange. *Quarters* are areas surrounded by major roads. *Blocks* are surrounded by any road and divided into *parcels*.

2.1.4 Other Methods to Create Street Networks

To allow more extensive user-control over the creation of street networks, Chen et al. [CEW+08] proposed an interesting method to interactively model and edit street networks without the usage of L-Systems. Instead,

tensor fields are used to guide the generation of streets. Tensor fields give rise to two sets of *hyperstreamlines*, one following the major eigenvector field and the other the minor eigenvector field [CEW+08]. Tensor fields are generated by a combination of user input, such as street patterns, and environmental information (terrain, obstructed areas, etc.) The user can modify the field by blending multiple fields together or modifying tensor values using a brush interface [CEW+08]. In a second stage, streets are extracted as hyperstreamlines and networks are modeled using a similar hierarchy as the one discussed in Section 2.1.3. Afterwards, the user can interactively edit the street network by again modifying the tensor field or by directly changing the created street graph [CEW+08]. Results of this method are shown in Figure 2.9.

Fig. 2.9: Street network created using tensor fields. Image courtesy of Chen et al. [CEW+08]

Another system that allows the user close control over the generation process was introduced by Kelly et al. [KM07]. Major roads can be interactively drawn and manipulated. Once a region is enclosed by major roads, minor streets are created automatically using an L-System similar to [PM01]. As discussed before, a variety of parameters can be changed and their effect can be viewed in real time. Additionally, a method is presented to generate simple buildings procedurally by extruding a building footprint from the previously calculated buildings lots.

A different approach to reconstruct a road network has been used by Aliaga et al. [AVB08]: They investigated the problem of synthesizing urban layouts from existing examples. This can be used for the design and extension of urban areas in a similar style as an existing location. Aerial-view images registered with street data from real-world urban environments are

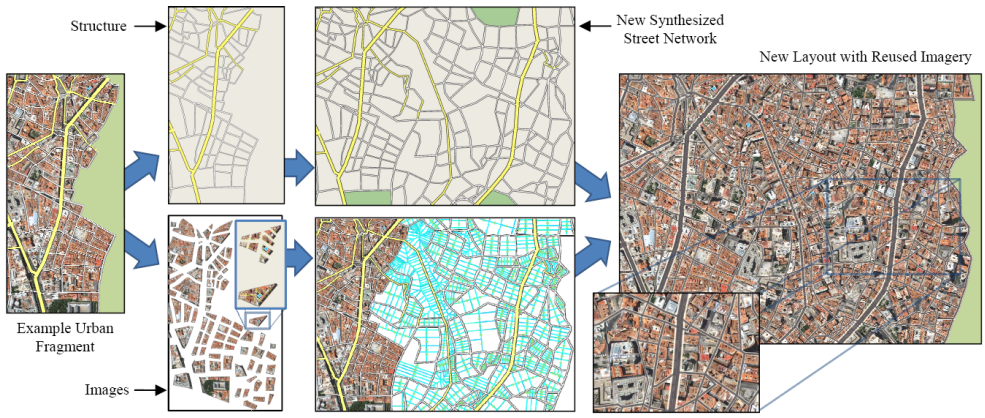

Fig. 2.10: Urban Layout synthesized from example images and structures. Image courtesy of Aliaga et al. [AVB08]

used to extract a street network and synthesize new layouts, as can be seen in Figure 2.10. This is done by creating a network of streets that attempts to follow the layout style given by the provided examples, extracting building blocks and parcels. Image synthesis is used to generate a new set of image fragments from the set of input aerial-view images. The system also allows several high-level editing operations such as joining, expanding, and blending example layouts. [AVB08].

2.1.5 Interactive Editing of Street Networks

Recently, Lipp et al. [LSWW10] presented new solutions for the interactive modeling of city layouts. They introduced transformation and merging operators for topology preserving as well as topology changing transformations based on graph cuts. Graph cuts stem from graph theory and were originally introduced by Ford and Fulkerson [FF62]: In a graph $G = (V, E)$ with a source $s \in V$, a sink $t \in V$ and capacities c_e for every edge $e \in E$, a *s-t graph* cut partitions all vertices into two subsets S and T with $s \in S$ and $t \in T$ [LSWW10]. According to the Max-Flow-Min-Cut-Theorem, such a cut is minimal if the sum of all edge capacities between the two partitions is minimal. By saturating all of those edges, the maximum flow in the given graph can be reached [FF62].

In image processing, graph cuts are used to merge multiple textures [KSE+03]. Lipp et al. use a similar method to merge two different urban layouts $\bar{U}_a$ and $\bar{U}_b$. A shared graph is created by copying the streets of $\bar{U}_a$ and $\bar{U}_b$ into a single data structure and calculating all in-

tersections. For $\bar{U}_a$, a source node is created, and a sink node is added to $\bar{U}_b$. User defined weights are used as capacities for the edges in the shared graph to compute the graph cut. After that, all streets that are not in the correct partition are deleted: Only the streets of $\bar{U}_a$ in S and the streets of $\bar{U}_b$ in T are kept. By adjusting the weights of the graph, the user can control how important certain regions of the urban layout are. More important regions are kept when merging them into less important ones [LSWW10]. Results of this method are shown in Figure 2.11.

Fig. 2.11: Renderings of two urban layouts merged using graph cut. From left to right: (1) Two separated layouts (2) merged scene (3) closeup of merged ares. Image courtesy of Lipp et al. [LSWW10]

Another editing method proposed in the same paper are *non-topological transformations*: Many operations on a street network (e.g. a junction moved by a small amount) do not change the topology of the streetgraph. In this case, it is sufficient to update the parcels in the blocks adjacent to the node [LSWW10]. This is done either by distorting the existing parcels or, if the area of the block was changed more than a certain threshold, creating new parcels by subdividing the block.

Parcel distortion can be achieved using mean value coordinates [Flo03]: These are calculated for every vertex of a parcel with respect to its containing block. If a node of this block is moved, the new parcel vertex positions can be calculated from these changes. A more detailed description of this method will be given in Section 7.2.

The city modeling system introduced by Lipp et al. allows very complex and powerful transformations on existing street networks, including the merging of multiple urban layouts. However, no real geometric representation for the streets is used, and street networks are limited to the 2D case of planar layouts. Transformations of networks that adjust to the underlying terrain are not discussed.

2.2 Street Geometry

The methods presented so far describe methods how to create and edit street networks but not how to create geometric representations of them. In the following section, I will discuss previous work that has focused on creating actual *street geometry*. I define the term *street geometry* as any tessellated geometric representation of a street, from very simple one-lane roads to complex geometry including sidewalks, lamp posts, roundabouts, etc.

In his master thesis, Hummel [Hum01] introduced a method to generate tessellated street geometry for large cities suited for real-time visualization. Input is provided in the form of static input files and consists of a terrain height map, the building footprints and the street graph. In a first step, street segments are processed into a set of points that serve as supporting points for the shape of the generated 3D geometry. Actual polygons are created by connecting these points and creating the final geometry using *Delauney-triangulation*. In the final step, the streets are textured.

While Hummel describes how street segments adapt to the underlying terrain by interpolating between height samples, the problem of how junctions should be modified to match the local features of the terrain is not addressed. Furthermore, junctions are limited to connect a maximum of 4 street segments.

Gebhart [Geb08] describes a system that helps artists to create tessellated 3D street networks. Streets are "drawn" by an artist or designer as skeleton lines in an editor. Based on these lines, detailed geometry consisting of multiple driving lanes, sidewalks and hard shoulders can be created and controlled using various parameters such as the number of lanes, lane widths, the radius of a curve, etc. Crossing geometry is created by adding osculating circles between the crossing streets. These curve segments have to be controlled manually by an artist and are not created completely automatically. Also, crossings are limited to two types:

- Intersections of two streets
- T-junctions, where one street segment ends at another street

Any other type of junction has to be hand-modeled as a roundabout, where the roundabout is a separate street connecting to the other streets via t-junctions [Geb08]. While this method creates highly impressive geometric street representations, it is not well suited for interactive editing of street

networks, because many crossings have to be recreated by hand every time the topology of the street graph changes.

The method I will describe in Chapter 5 to create street geometry is based on the work of Zimmermann [Zim07]. He presents a technique to construct a fully polygonal 3D street representation out of centerlines that are given by edges in the street graph. While this is trivial for single streets, Zimmermann also proposes a method to create geometry for crossings connecting an arbitrary number of street segments. This is done by finding the intersection points of the outer lines of the street segments and connecting them with the center point of the junction. This is illustrated in Figure 2.12. To create queuing zones next to the junction, the tessellation is altered so that street segments have an ending perpendicular to their direction. To achieve this, the intersection points have to be mirrored to the other side of the street [Zim07].

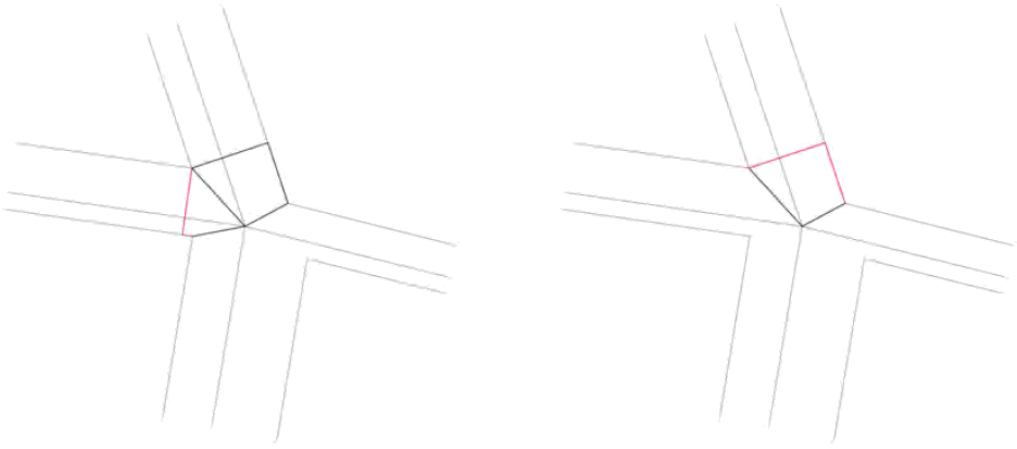

Fig. 2.12: Left: Junction geometry created by finding the intersection points of the outer street lines. Right: Intersection geometry with mirroring applied. Image courtesy of M. Zimmermann [Zim07]

While this method produces a good and reliable geometric representation of planar streets, it fails to address the problem of maintaining a stable and realistic tessellation for 3D streets that adjust to terrain levels of different heights.

2.3 Automatic Building Assignments

The problem of finding a model that fits into a certain environment has not been investigated very extensively. Kjølaas [Kjø00] presents a system that automatically places furniture into a given floor plan. To ensure valid room layouts and reasonable spatial allocations, he proposes a case-based approach. Templates are created by the user that represent different room types, such as dining, working or bedrooms. These templates are defined by drawing rectangular shapes inside room plans representing bounding boxes of the furniture. Since rooms and templates are restricted to rectangular shapes in the presented method, templates can be easily fitted into any given room by using *rotation*, *translation*, *flip* and *scale* operators [Kjø00]. If no transformation can be found without conflicts with doors or windows, furniture will be removed one by one and placed somewhere else. If no valid configuration can be found without removing the vital furniture of the template, it is discarded for this room. The presented system allows simple furniture population of indoor environments, but is very limited by its restriction to rectangular rooms and furniture bounding boxes.

In the field of automated building placement, a lot of work has been done in the direction of recognition and reconstruction of buildings from aerial images:

Jaynes et al. [JRH03] introduced a model-based technique for the automatic 3D reconstruction of building shapes. As input, digital elevation maps derived from optical stereo images registered to photographs of urban environments are used. Their method is divided into two phases [JRH03]:

1. First, a perceptual grouping algorithm detects building boundaries in the photograph using low-level features such as straight lines or image corners.

2. Then, rooftop shapes are determined from the corresponding regions in the digital elevation map. This is done by comparing surface orientation histograms for each model from a database of shape models to the elevation map. The best matching model is taken as the initial input for a robust surface fitting algorithm that refines the model parameters to get an approximation of the building shape.

This approach leads to the recognition and reconstruction of a wide variety of buildings [JRH03].

Suveg et al. [SV04] use a very similar approach to reconstruct buildings from stereo aerial images. The main difference from the method of Jaynes et al. [JRH03] is the use of models described as CSG trees and a metric that evaluates the found building shape in a final verification step.

Both methods can be used in the process of reconstructing existing cities, e.g. for mapping applications, but are not suited for artist-created urban environments due to their lack of interactivity.

2.4 Shape Grammar Applications

Recently, procedural generation of buildings and facades has been an active field of research, including the generation of facades using shape grammar. Shape grammars, first defined by Stiny et al. [Sti80, Sti82], are production systems that generate geometric shapes. Similar to L-Systems, production rules, here specifically called *shape rules*, define a *predecessor* and a *successor shape*. Starting with an *initial shape*, existing structures are refined according to the production rules. See Figure 2.13 for an illustration.

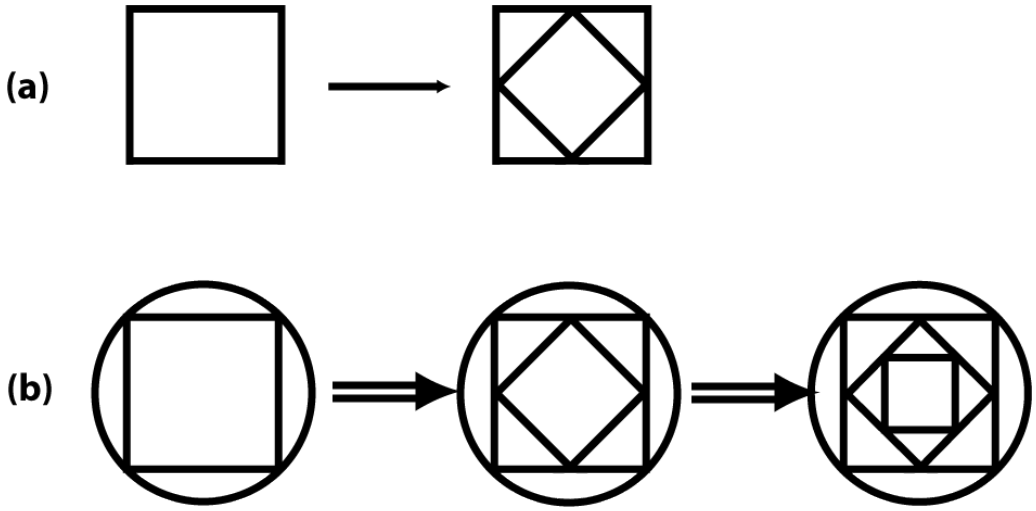

Fig. 2.13: Simple example of a shape grammar that inscribes squares in squares. (a) Shape rule, (b) generation of a shape after repeatedly applying the rule [Sti80].

Parametric shape grammars are an extension of shape grammars that allow the use of parametrized shapes so that the context of already existing shapes can be taken into account.

Wonka et al. [WWSR03] and more recently Müller et al. [MWH+06] propose a shape grammar based approach to procedurally model buildings. They

extend the shape rule definition to contain one or more shapes and commands as the successor. Commands are macros creating new shapes or commands, like the following [LWW08]:

- Splitting a shape into multiple parts
- Repeating of shapes in a certain direction
- Component split: creates new shapes on components (edges or faces) of a shape

Repeated application of different rules including these commands creates a hierarchy of shapes by inserting the successor shapes as children of the predecessor shapes. This method allows to create very detailed and diverse buildings using a small set of shape rules, as can be seen in Figure 2.14.

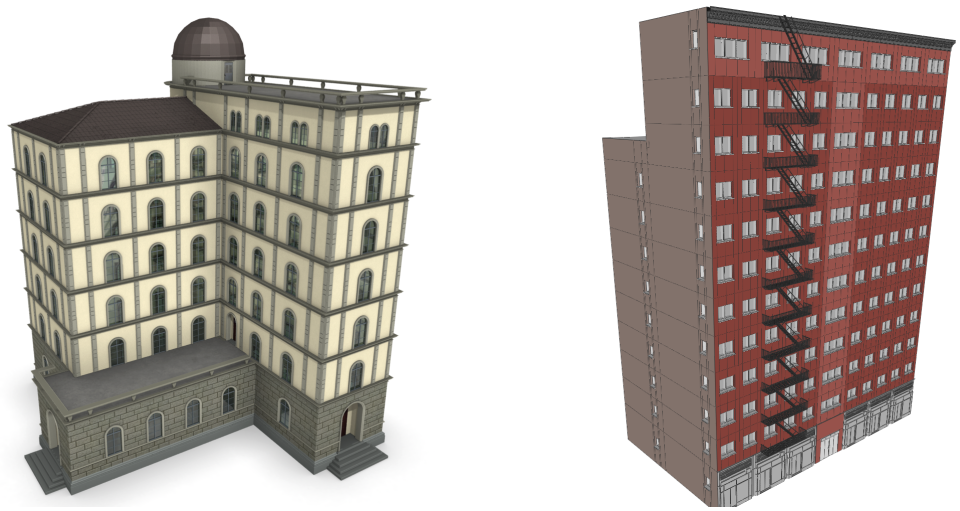

Fig. 2.14: Two examples of procedurally generated buildings using shape grammars. Image courtesy of Müller et al. [MWH+06].

The main limitation of this approach is that it is not very flexible: To add a little variation to the created model, for instance a single different window in a facade, the whole rule set has to be re-created to model the variation. This can quickly result in an explosion of the rule base.

Lipp et al. [LWW08] propose a solution to overcome this limitation using a visual workflow to shape grammar rules. This enables artists and designers to utilize shape grammars in an easy-to-use way. Additionally, Lipp et al.

introduced an extension to shape grammars that allows local modifications for created building instances. This is done by directly modifying the subtree of the created shape hierarchy interactively without exposing the hierarchy to the user. The modified nodes in the subtree are then marked to preserve the changes even if global changes in the set of shape rules are made and the shape hierarchy is re-created.

2.5 Commercial Applications

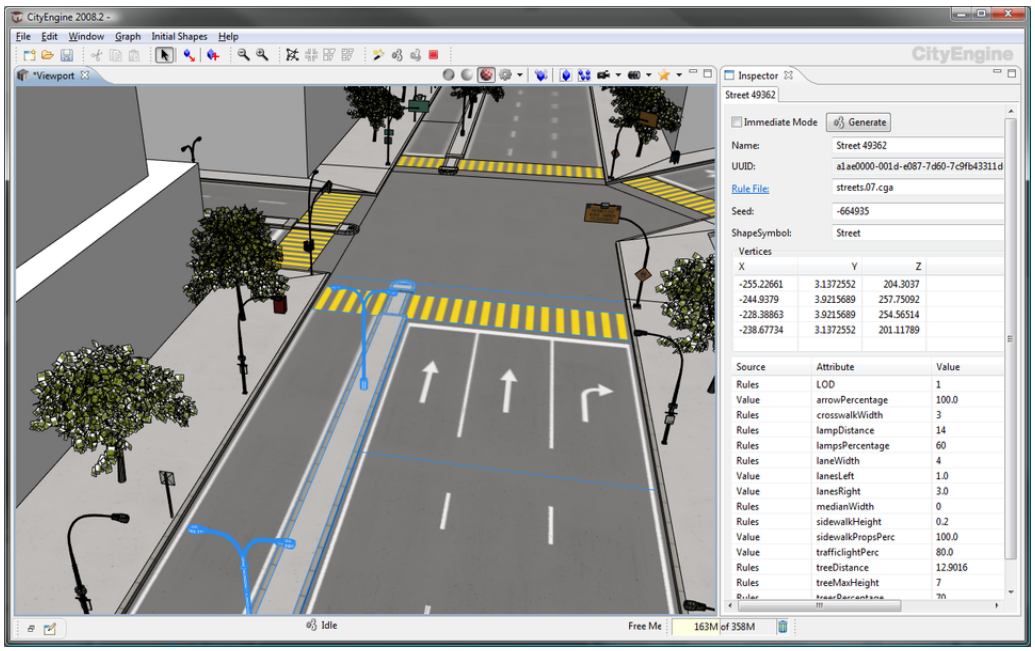

Fig. 2.15: Screen shot of the CityEngine application showing a complex junction modeled with shape grammar rules. Image courtesy of Procedural Inc. [Pro10]

The CityEngine [Pro10] is a commercial software package that is capable of procedurally generating cities. Input is provided in the form of controllable parameters and image maps: height maps can be used to model terrain, obstacle maps denote regions where no streets should be created and population density maps control the type and density of the streets and buildings in certain regions. The application implements many of the techniques discussed in the previous sections:

- Procedural street network and parcel creation using L-Systems as described in [PM01, WMWG09].

- Parametric street geometry creation similar to the method of Zimmermann et al. [Zim07], but also employing shape grammar techniques.
- Parametric modeling of 3D buildings using shape grammars as discussed in Section 2.4 [MWH+06, WWSR03, LWW08].
- Interactive editing of the street network by adding, moving or deleting streets.

The application is available for purchase and new features are added regularly. It is one of the best known and most powerful modeling solutions for urban environments. Screen shots of the application are shown in Figure 2.15 and 2.16.

At the time I started my thesis, the CityEngine had the following limitations, some of which have been made obsolete by recent preview versions:

- No possibility to directly modify the street geometry and view the results in real time: When adding, moving or deleting a street in the street graph, the affected geometry has to be removed and recreated manually.
- Sparse outer city regions: As mentioned in Section 1.3, minor roads are only created in quarters surrounded by major roads.
- Unnatural steps in steeper roads: To keep junctions planar, junction geometry is forced to be parallel to the ground plane.

These limitations were, among the insight won by the artist workflow described in Section 1.1, the main inspiration for the focus of my thesis. As mentioned in Section 1.3, I will introduce solutions to overcome them in the following chapters.

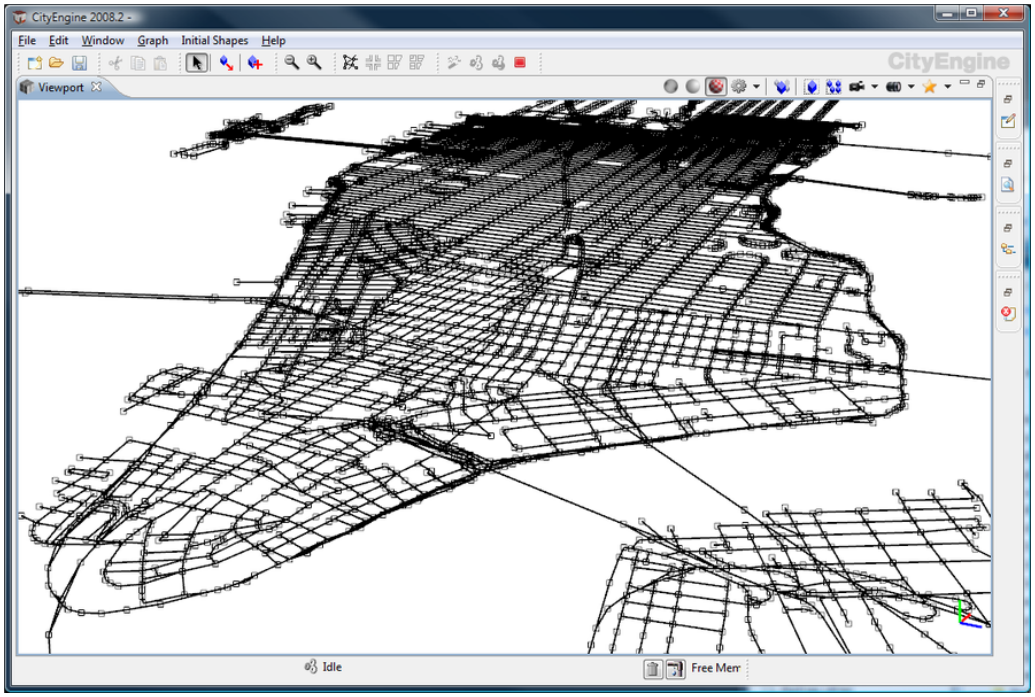

Fig. 2.16: Screen shot of the CityEngine application showing a large street network. Image courtesy of Procedural Inc. [Pro10]

Chapter 3

Planning and Layouting an Urban Environment

As discussed in Section 1.1, urban environments for games are usually planned using annotated maps. In the following chapter, I will introduce layouting tools that can be used to implement this artist workflow.

In Section 3.1, I will describe how image maps can be imported and used to model terrain. Image maps can also be used to illustrate regions of the city or gameplay related information, such as the player progress. This is shown in Section 3.2.

Points of interest in the scene can be denoted with markers without having to model the respective geometry. They will be discussed in Section 3.3.

3.1 Terrain

Most environments in games will be set in some kind of terrain, like mountains or hills. For instance, *Assassins Creed 2* is set in the hills of Tuscany. Terrain forms the basis of every game setting. The development of a street network is limited by too steep slopes, since streets usually do not run up or down a cliffy mountain. Instead, they form serpentines or follow the direction of the most even slope.

Similarly, buildings can not be placed on steep terrain without subsiding parts of the model into the terrain.

To model terrain, *heightmaps* are often used. A height map is a texture that stores elevation data as gray scale values in the interval $[0, 1]$. This texture can then be used to displace the vertices of a regular mesh by the stored values. The result is a tessellated terrain patch, as can be seen in Figure 3.1.

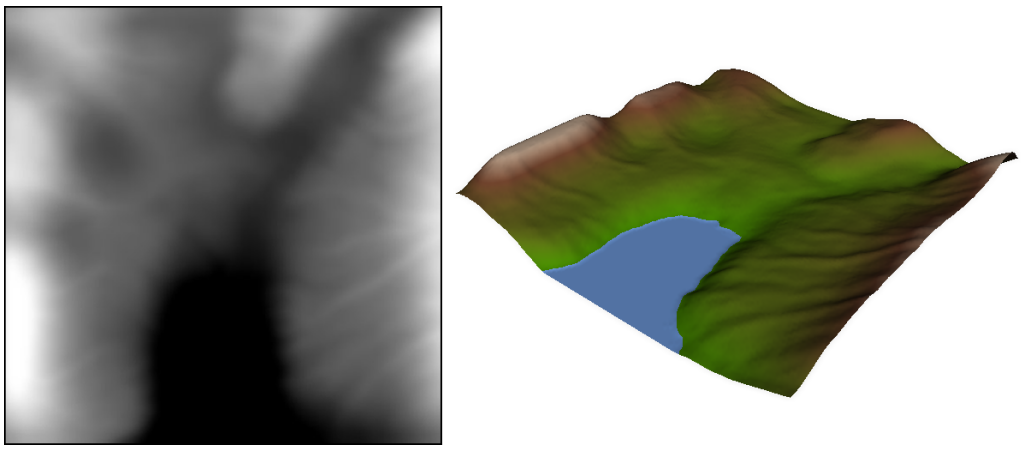

Fig. 3.1: Height map interpreted as terrain. Left: A gray scale image representing a terrain. Right: A regular mesh displaced by the height values stored in the texture.

To make the created terrain look more plausible, it can be rendered using a simple height dependent color scheme. Different colors simulate water, vegetation, timberline and finally snow. The result is simple but effective and produces believable coloring for terrain meshes.

3.2 Image Maps

Planning a city for a game usually involves the creation of many maps. These maps are used by the game designers for illustration purposes and can have different meanings:

- Maps showing the spatial layout of the city and indicating different regions of the urban environments (e.g. little Italy, suburbans, slums, industrial park)
- Illustration of the planned topology (parks, rivers, harbor, etc.)
- Plans to illustrate the player's progress, showing starting regions, gameplay targets, or places of certain missions
- Maps containing any other kind of spatial information, such as special points of interest.

Such maps are often put on different layers to combine and blend them together. For instance, it can be useful for a designer to see the planned player progress on top of a detailed spatial layout of the city's regions. Two examples of maps can be found in Figure 3.2.

Fig. 3.2: Examples of maps used in the planning process. Left: Map showing the quarters of *Liberty City*, a fictional city from the game *Grand Theft Auto IV*. Image courtesy of Take-Two Interactive Software Inc. Right: A tactical map with points of interest like spawn points or mission targets from the game *Call of Duty 4*. Image courtesy of Activision Publishing, Inc.

To help the game designer to layout an urban environment, maps can be displayed as overlays. They can be imported as images and selected to project down their content onto the terrain maps below. To display multiple maps on top of each other, they can be sorted by the user and blended together by controlling the transparency of each single map. It is also possible to hide projected image maps so that just the projection is visible. Figure 3.3 illustrates the method and shows some results.

3.3 Markers

As mentioned above, designers usually denote special points of interest in maps. Such maps can be inflexible to use if the position of these points is changed. Also, the maps may lack clarity if they are crowded with points. A better way to denote such points in the layout process is to use *markers*. Markers are billboard pointers that can be placed freely anywhere in the scene. To emphasize the importance of certain markers, they can also be scaled to different sizes.

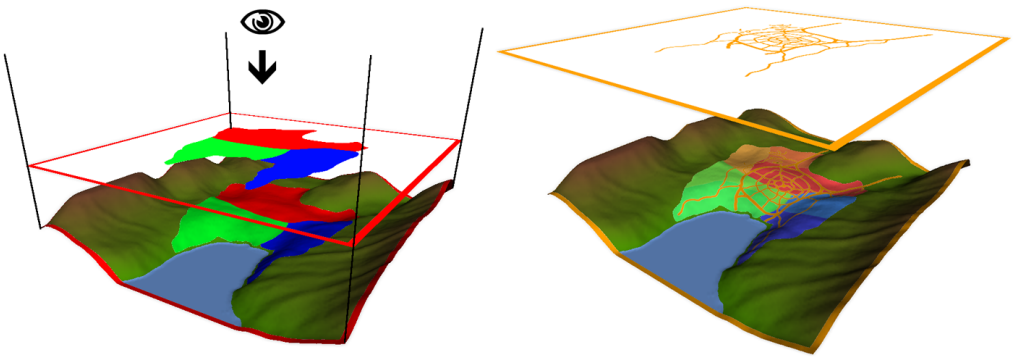

Fig. 3.3: Image projections. Left: A basic area map is projected onto a terrain using an orthographic projection. Right: Two (hidden) area maps are blended together with a third texture of a street network layout on top.

Points of interest highlighted by markers may be:

- The positions of special buildings, such as churches, office towers or landmarks.
- Starting positions of the player.
- Important mission targets or any other type of gameplay relevant position.

Markers are implemented as billboards [AMH02, p. 318-329] that always face towards the camera. They can contain user-defined icons to identify the denoted position. In Figure 3.4, three markers are shown displaying the planned positions of a windmill, a church and a lighthouse.

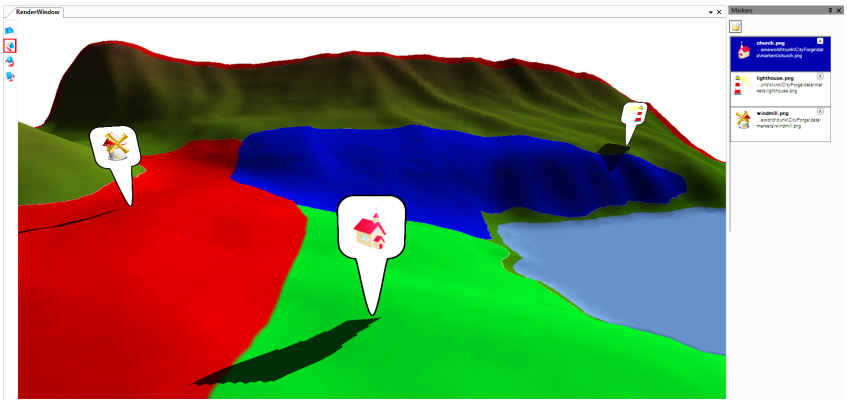

Fig. 3.4: Three markers denoting future positions of various buildings.

Chapter 4

Creating the Street Network

In this chapter, an algorithm is described that creates whole street networks or can be used to add streets to existing networks. This algorithm is based on the work of Parish et. al [PM01] and Weber et al. [WMWG09]. Streets are created using a system similar to extended L-Systems, although I chose not to implement a string rewriting system, but to apply the production rules directly to the street objects to avoid slow string operations, as proposed in [WMWG09]. The city hierarchy definition is the same as described in Section 2.1.3. I will first describe a set of important control parameters that are used for creating streets in Section 4.1. Finally, I will explain the algorithm in detail in Section 4.2.

4.1 Control Parameters

The creation of streets can be controlled with the following different parameters:

- Cities usually have different layouts. Streets in New York City are strictly rectangular, whereas Paris follows a loosely circular pattern. Cities with no superimposed pattern grow organically. In our system, major and minor roads can follow *different patterns*, as illustrated in Figure 4.1.
- Street *length*, *width* and *angles between adjacent street segments* can be controlled.
- To avoid street ends near existing junctions, a distance `snappingDistance` can be defined. If the distance between a street end and a junction is smaller than `snappingDistance`, the street end is snapped to this junction.
- A *height map* may be specified to create a terrain (see Figure 4.2). The streets will then adjust to this terrain. If the slope of a street segment

Fig. 4.1: Different street patterns. From left to right: (1) Grid/raster pattern (2) Radial/circular pattern (3) Organic pattern

between two positions A and B is larger than a user defined threshold `maxSlope`, the end B of the segment is rotated around A until it is plain enough, or removed if that is not possible.

- Urban environments may contain areas where no streets should be created, such as parks or water. Such areas can be denoted in an *obstacle map* (see Figure 4.2). This map is sampled regularly and the street creation algorithm will avoid any obstacles.
- Real cities are planned to meet the requirements of its inhabitants: Major roads connect centers of high population densities, while minor roads provide access to the major roads in populated areas. A *population density map* (Figure 4.2) can be set to control the development of major and minor streets.
- The average size of a building parcel can be controlled to adjust the parcels to the desired building size.

All of these parameters can be set using the user interface of our application.

4.2 Creating the Street Network

As explained in Section 2.1.3, to create a city obeying the used hierarchy, the algorithm is divided into two stages: (1) creating major streets and (2) identifying quarters surrounded by major streets and filling them with minor streets. Both stages can either be done automatically or manually by a designer as described in Section 7.1. In the following sections I will focus on the various aspects of automatic street network creation. This is basically an in-depth discussion on the algorithm introduced by Weber et.

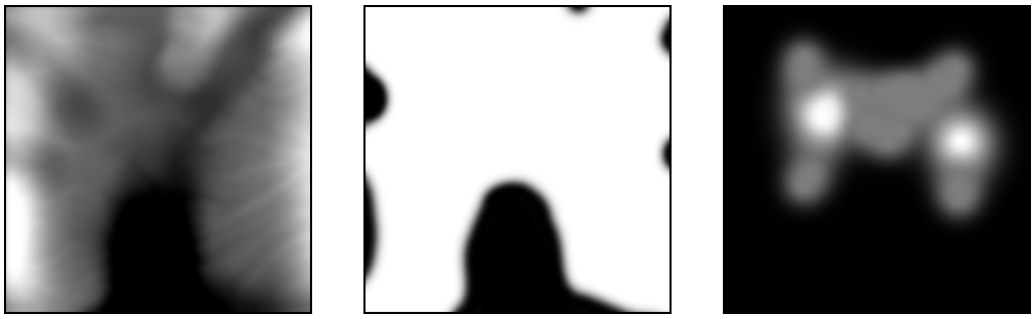

Fig. 4.2: Input maps for a bay area environment. From left to right: (1) Height map, (2) Obstacle map, (3) Population density map.

al [WMWG09]. In Section 4.2.5, I will propose an extension for this method to create more realistic city boundaries.

4.2.1 Seed Street Segment Creation

The L-System used to create street networks needs an initial street segment as input. Production rules can be applied on this input segment to further develop the road network.

Therefore, the very first step in the city creation algorithm is to create an initial major street segment. This seed street can then be expanded and new streets can be created as branches. The position of this first segment is dependent on some input parameters, mainly the input maps. Since realistic street growth should start in a densely populated area, the density map (if supplied) is sampled to find the positions with the highest population density. These samples are weighted with their distance from the center of the map, so that there is a better chance of creating the first segment near to the center. The first segment is placed at the position with highest value $v = d \cdot delta_c$, where d is the population density at this position, and $delta_c$ is the Euclidean distance to the center of the map, normalized to the texture coordinates [0, 1]. If no density map was supplied, d is constant. The initial street segment is created at the position with the highest value v. If the street segment could not be created (if the position is obstructed or at a too steep slope), the position with the second highest value v is chosen. This is repeated until a valid position is found.

To fill a quarter with minor streets, another seed street is needed inside this quarter. A seed street for a quarter is created as follows: At first, the seed street segment is created at the barycenter of the polygon surrounding

the quarter. If this is not possible (e.g. because of obstruction, or if the polygon is concave and the center is outside of the polygon), the following algorithm is used to find another position:

```
While no valid seed street segment found:
  pick a random node v in the surrounding polygon
  pick a random node w != v in the surrounding polygon
  take a random position p between v and w
  try to create a street segment at p
```

After a few iterations, the loop is canceled to avoid infinite loops in case no seed street can be found (e.g. if the whole quarter is obstructed).

4.2.2 Street Growing Strategies

Created streets with at least one initial street segment can be iteratively expanded. They can either grow or create new streets by branching. As proposed in [PM01] and discussed in 2.1.2, a 3-step system is used to develop a street in each iteration:

1. A production rule is chosen to grow a street or to create a branch. This production rule returns a new street segment called the *ideal successor*. This new street segment is either added to the existing street at the beginning or the end, or is the first segment of a new street.

2. The parameters for this segment are evaluated by the *global goals* function. This mostly effects the orientation of the new street segment.

3. The *local constraints* that limit the creation of the new street segments are determined. If a segment cannot be changed to fit all constraints, it will be deleted. These constraints will be described in more detail in the next section.

Creating a new street segment can happen in two ways: Either a street segment is added to an existing street, or a new branch is created. This can be modeled by the following three different production rules:

- *Appending Production Rule*: Creates a new street segment at the end of the given street

- *Prepending Production Rule*: Creates a new street segment at the beginning of the street

- *New Branch Production Rule*: Creates a new street as a branch of the given street. A random node of the existing street will be picked as the crossing where the new street will be created. It is possible to limit the number of streets per crossing. This production rule will pick a node where this number has not been reached.

All of these rules will return a new street segment as ideal successor. The parameters for this segment will be set according to the following global rules:

Street Patterns

The algorithm will adjust the new segment to follow the superimposed road pattern. This includes mostly road length and orientation. In the following, the patterns illustrated in Figure 4.1 will be explained:

Basic (organic) pattern This is the street pattern found in most historic European city centers. Instead of developing the city according to a certain layout, roads and buildings were mostly built where space was available.

Such a "pattern" is achieved by setting the segment length according to the parameters `length` and `lengthDeviation`. The resulting length $\texttt{segmentLength} = \texttt{length} \pm \mathit{random}(0, \texttt{lengthDeviation})$. The same is done with the street width. The orientation is changed in relation to the previous segment by a random angle between $[0, \texttt{maximumAngle}]$.

Grid/raster pattern This pattern is used in many cities in the United States, most notably in Manhattan.

New streets will be classified as longitudinal or latitudinal according to their initial orientation. The main directions are defined once for major streets and once per quarter for minor streets. The orientation of the streets will vary very little, and new street branches are created exactly orthogonal to the existing street.

Radial pattern Some European cities like Paris or Vienna follow a radial pattern. Streets can be divided into two different categories: radial streets that run from the city centers outwards, and tangential streets that form a concentric circle around the center.

Creating streets according to this pattern is not as easy as with the previous designs. The street length will be assigned very similarly to the basic pattern, but the segment orientation is more complex, since newly created segments have to be re-oriented to become either a radial or a tangential segment. Otherwise, the created street network will not resemble

a radial pattern.

First, it is determined if the new street segment is oriented more radial or tangential to the center of the radial pattern.

- If it is radial, it is re-oriented to point inwards or outwards from the center as can be seen in Figure 4.3.

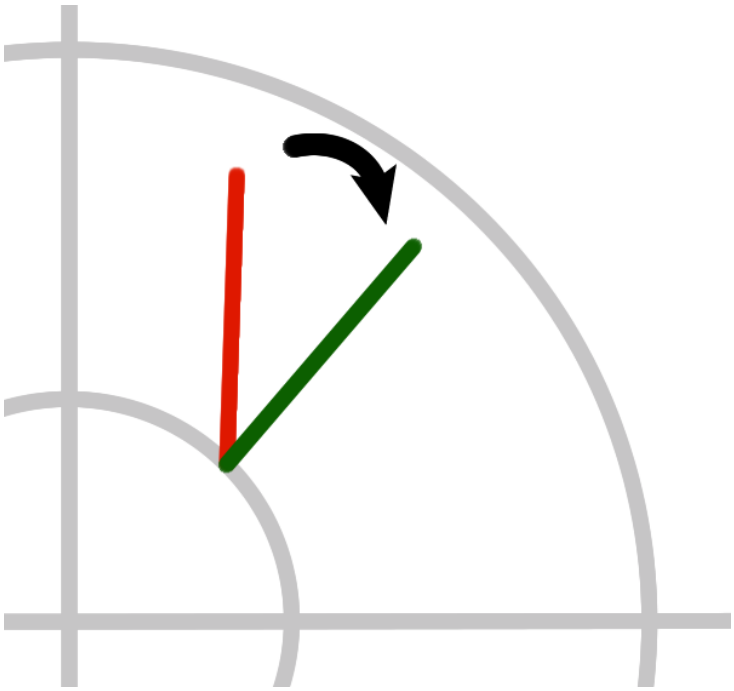

Fig. 4.3: A street segment in a radial/circular pattern is re-oriented to become a radial street. The pattern is given in gray, the original street segment is displayed in red and green after the re-orientation.

- If it is more tangential, the segment's orientation has to be changed so that it forms a tangent to the pattern center. The starting point P_1 of the street segment has to remain unchanged to keep the connection to the existing street. Therefore, the end point P_2 of the segment has to be moved. Its new position can be calculated by intersecting two circles C_1 and C_2, where the center of C_1, $center_{C_1} = center_{rp}$ (the center of the radial pattern) and the radius $radius_{C_1}$ is the distance from $center_{rp}$ to P_1, while the center of C_2, $center_{C_2} = P_1$ and the radius $radius_{C_2}$ is the distance from P_1 to P_2. This is illustrated in Figure 4.4.

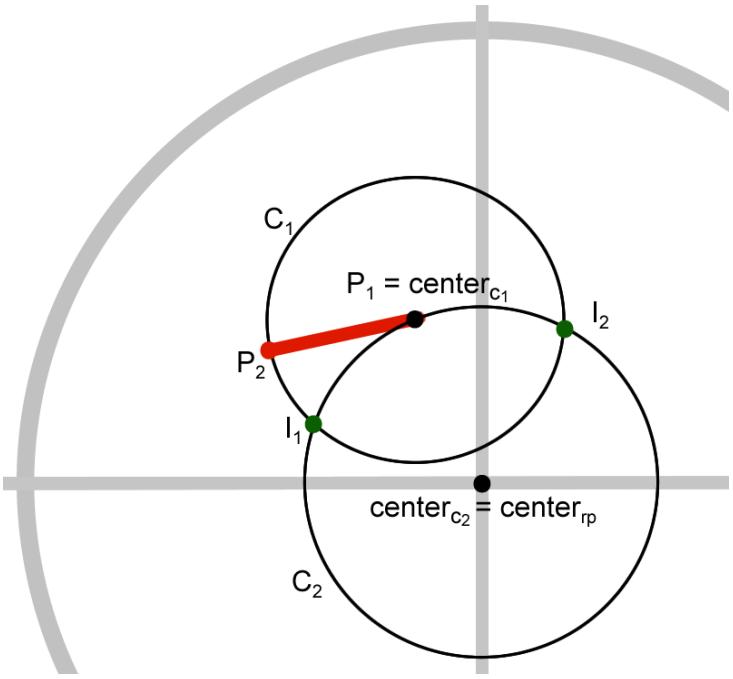

Fig. 4.4: Adaption of a tangential street segment to fit a circular pattern. The pattern is given in gray, the street segment (P_1, P_2= is displayed in red. In this case, P_2 will be set to the found intersection point I_1 so that the street segment is a perfect tangent to the center of the radial pattern.

The intersection points of the two circles can easily be calculated as described in [MST89]. Note that there is no intersection point if $radius_{C_2} >= radius_{C_1}$, in that case the street segment is discarded. If $radius_{C_2} < radius_{C_1}$, there will be two intersection points. P_2 is set to the one where the orientation of the new segment changes less.

Population Density

If the current street is a major road, the algorithm will try to connect centers of population with this street. For this purpose, the population density map is evaluated to find peaks of population density. As described in [PM01, p. 303], *"Every highway road-end shoots a number of rays radially within a preset radius. Along this ray, samples of the population density are taken from the population density map. The population at every sample point on the ray is weighted with the inverse distance to the road end and summed up. The direction with the largest sum is chosen for continuing the growth."* An illustration is shown in Figure 4.5.

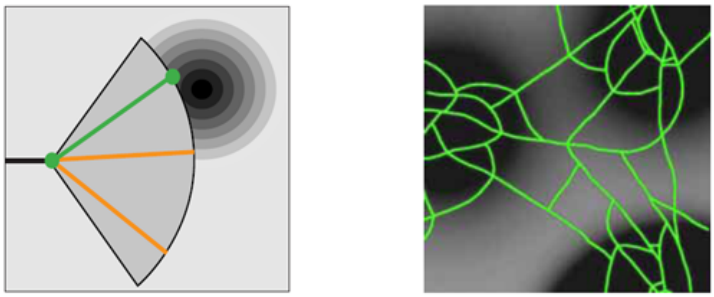

Fig. 4.5: Main streets will connect centers of high population density. Left: ray shooting to find next population peak. Right: Street Network connecting population centers. Image courtesy of Parish et al. [PM01].

4.2.3 Local Constraints

As mentioned in [PM01, WMWG09], local constraints are used to adapt the new street segment to its local environment. Segment length and orientation may change within certain boundaries. These boundaries can be predefined, such as minimum street length or maximum rotation angle change. If it is not possible to find valid parameters, the segment is discarded. The following rules have been implemented, some of them are shown in Figure 4.7:

Slope correction Streets can not follow every slope. If they are to steep, they would be impossible to use. If the slope of the new segment is more than a user defined threshold `maxSlope`, the street segment is rotated until the slope is below `maxSlope`.

Obstacle avoidance Streets should not be created inside obstructed areas such as parks or water. This can be controlled by providing an obstacle map that is sampled at regular positions along the new street segment. If the segment runs through an obstructed area, the system tries to shorten the segment. If the new segment length is below a user defined threshold `minimumSegmentLength`, the segment is rotated until a valid alignment is found.

Intersection In order to create junctions with other roads, the new street segment is intersected against other street segments in its local proximity using a simple line/line intersection test as described in [AMH02, p. 127-129]. The two segments only intersect if the intersection point lies within both segment bounds. If an intersection is found, the segment is shortened and ends at the intersection point.

Extension intersection To avoid road ends close to other streets, new street segments are extended by a certain factor and tested for intersections. If such an intersection is found, the segment end is set to the new intersection point. If not, the segment length is reset to its original value.

Snapping Snapping is used to prevent close-by crossings of streets. If the distance between the node at the end of the new segment and a node in its surroundings is below a user-defined threshold, the new node snaps to this position.

Clip to mask In order to allow the user to limit the creation of new streets to a certain region, it is possible to define a closed polygon that is used to clip the new street segments at the borders of this polygon. A simple inside/outside test like the odd-even rule as described in [HB04, p. 127-129] is used to determine if the new street segment is intersecting the mask polygon. If so, the segment is clipped to the mask.

Remove duplicates Sometimes street segments are created very close to already existing streets. If they are nearly parallel and the distance of the two segments is below a user-defined value `minimumStreetDistance`, the new segment is discarded. The two street segments are roughly parallel if the following inequality yields true:

$$|v_{a_n} \cdot v_{b_n}| > 0.9$$

Where v_{a_n} and v_{b_n} are the normalized direction vectors of the two street segments. As described in [Wei10], the distance of a point $\mathbf{x_0}$ to a line specified by two points $\mathbf{x_1} = (x_1, y_1, z_1)$ and $\mathbf{x_2} = (x_2, y_2, z_2)$ can be calculated as follows:

$$t = -\frac{(\mathbf{x_1} - \mathbf{x_0}) \cdot (\mathbf{x_2} - \mathbf{x_1})}{|\mathbf{x_2} - \mathbf{x_1}|^2}$$

$$d = \frac{|(\mathbf{x_0} - \mathbf{x_1}) \times (\mathbf{x_0} - \mathbf{x_1})|}{|\mathbf{x_2} - \mathbf{x_1}|}$$

Where d is the distance of the point to the line and parameter t describes the position between $\mathbf{x_1}$ and $\mathbf{x_2}$ that is nearest to the point $\mathbf{x_0}$. An illustration can be found in Figure 4.6.

This distance is calculated for both nodes of the new street segment to other line segments in its proximity. If the distance of both nodes is less than

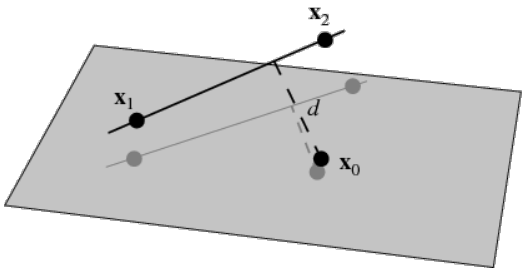

Fig. 4.6: Distance of a point $\mathbf{x_0}$ to a line specified by two points $\mathbf{x_1} = (x_1, y_1, z_1)$ and $\mathbf{x_2} = (x_2, y_2, z_2)$. Image courtesy of Wolfram Research, Inc. [Wei10].

`minimumStreetDistance`, the system uses the calculated parameters t to determine if the two line segments are actually close-by. If this is the case, the new street segment is discarded.

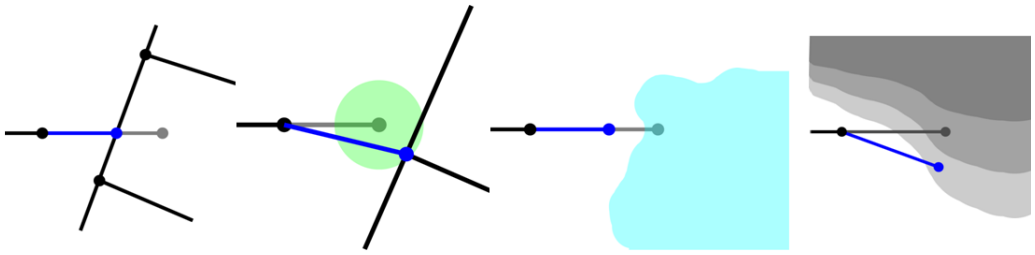

Fig. 4.7: Selection of local constraints. Proposed street segments are drawn in gray, corrected ones in blue. From left to right: (1) Intersection (2) Snapping (3) Obstacle avoidance (4) Slope correction

4.2.4 Quarter Identification

As defined by Weber et. al [WMWG09], areas surrounded by major roads are called quarters (see Section 2.1.3). To fill all quarters in a street network with minor streets, they have to be identified. If the street graph is interpreted as a graph, quarters are equivalent to faces in the planar graph (V_{major}, E_{major}). A simple example is given in Figure 4.8.

To find all faces in the graph, the system uses a planar face traversal algorithm similar to the one implemented in the boost C++ libraries [Riv10]. Facets can be found by traversing each edge in the graph and visiting the next edge in counter-clockwise order. To speed up this process, for every node n in the street graph data structure, the nodes that are connected to

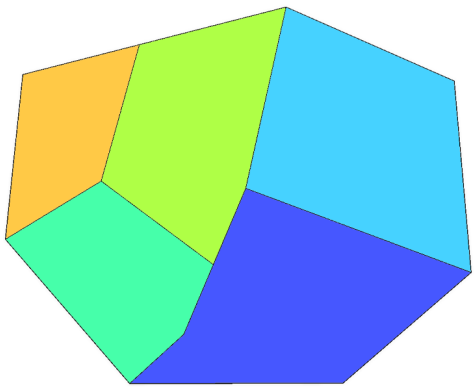

Fig. 4.8: A simple graph with five differently colored facets.

n by a street segment are stored. The pointers to these nodes are ordered counter-clockwise as illustrated in Figure 4.9

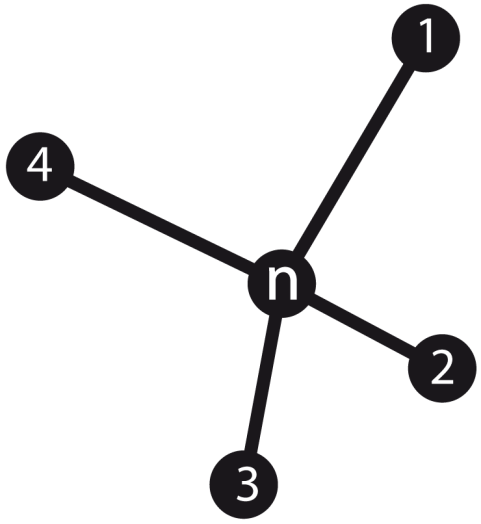

Fig. 4.9: Data structure used to store node adjacency information: For each node n, pointers to all adjacent nodes are stored in counter-clockwise order.

Using this data structure, the algorithm works as follows:

```
For every two connected nodes n and m in street graph:
  travelEdge(n, m)
  travelEdge(m, n)

travelEdge(n, m)
  e = getEdge(n, m)
```

```
if e has not been visited: // new face
  create new face f
  while (n not visited from e)
    add n to f
    temp <- n
    n <- m
    e <- edge to next node connected to n after temp
    m <- e.end
```

This algorithm starts with visiting each edge in the planar graph once in every direction. If this edge has not been traversed in this direction yet, a new face is created and the current edge is added to this face. From the end of the edge, the next edge in counter-clockwise order is visited. This is repeated until a node is visited from an edge the second time - then the face is complete.

Of course, the outer face of the planar street graph is also found, but it does not represent a city quarter. Therefore it is discarded. It is crucial that the data structure describing which nodes are connected to a node is kept up-to-date and correct, even minor errors in this list can cause the algorithm to fail detecting all quarters. This is ensured by encapsulating the methods to add, insert and delete a street segment to the street graph data structure and maintain the list of nodes there.

The very same algorithm can be used to identify building blocks in the street network by applying it to the whole street graph (V, E) instead of just the planar graph (V_{major}, E_{major}).

4.2.5 City Boundaries

One of the main problems of the system is that quarters - facets in the planar street graph - have to be surrounded by main roads. That means that no quarters (and therefore no minor streets) can be created outside of the city borders with the algorithm discussed so far, as shown in Figure 4.10.

I propose the following solution for this problem: A convex hull around the city is calculated using for instance the *Graham-Scan* algorithm [Gra72]. We use the street nodes V_{major} of the planar graph (V_{major}, E_{major}) as points for the finite set. This yields a new set of line segments surrounding the street network. But using these line segments directly to detect quarters would result in very artificial looking city borders. Therefore, the calculated

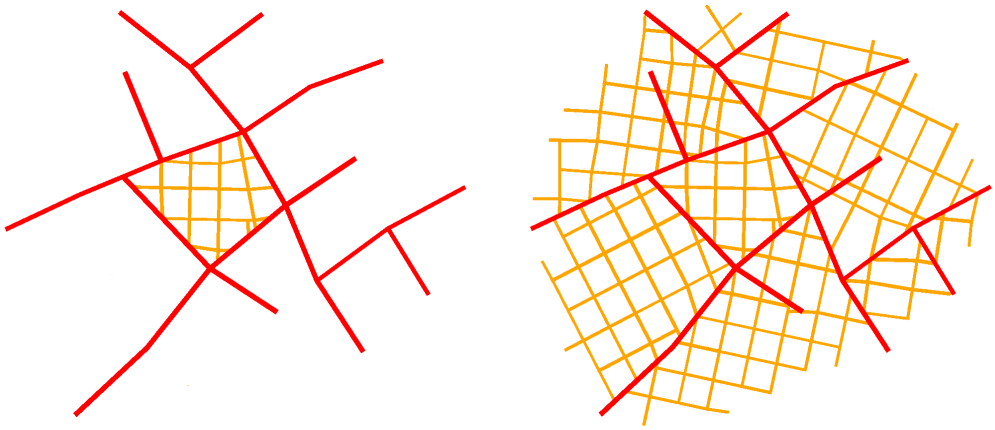

Fig. 4.10: Left: In previous methods, no detailed neighborhoods are created in the outskirts of the city. Right: Outer regions created with the new algorithm.

convex hull is 'bulged' to create a more realistic city outline: First, the barycenter C for the convex hull is calculated. Then, each line segment of the convex hull is subdivided into 4 new segments, creating 3 new nodes. Each of these nodes is displaced away from C by a random value, forming an irregular, not necessarily convex city outline. The resulting hull can be used to find city quarters, including quarters at the city borders not completely surrounded by major streets.

After the quarters have been filled with minor streets using the procedural algorithm described in the last sections, the convex hull is removed.

4.2.6 Building Parcel Generation

As mentioned in 4.2.4, the same facet finding algorithm used to identify quarters can also be used to find enclosed building blocks in the street network. But the facets found in the street graph do not directly correspond to the areas the parcels can take up. Since each street has a certain width, subdividing the facet directly would cause intersections between parcel and street geometry.

Therefore, a polygon shrinking algorithm is used to calculate the correct outline of the building blocks. Aicholzer et al. [AAAG95] introduced a method to calculate a *straight skeleton* for a polygon that can be used to shrink a polygon by a fixed offset for each edge (as opposed to linear scaling, where a polygon is just resized around a certain point). I modified this method so that different offsets can be used for each edge. This allows the

calculation of a stable and accurate boundary of the real block polygon.

Once the block footprint is found, building lots can be created using a simple subdivision scheme as described in [WMWG09]. The algorithm recursively splits the building lot polygon until the area of each resulting polygon is below a user defined threshold `maxArea`. If the area of a polygon is larger than `maxArea`, it is split using the following steps:

- Collinear edges of the polygon are merged
- The longest side of the polygon is selected
- A split line is calculated perpendicular in the middle region of the selected side
- The polygon is split along this line

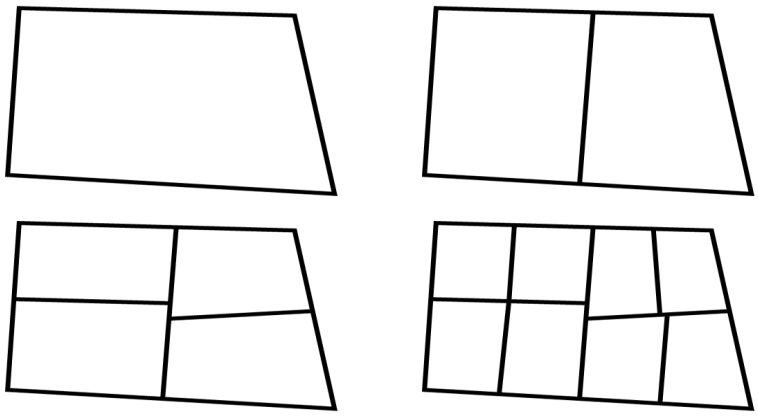

Fig. 4.11: This figure illustrates how a block is recursively split into parcels until each parcel is below the area threshold `maxArea`.

The algorithm is illustrated in Figure 4.11. The user can define various parameters:

- float `maxArea` will control the size of the created building lots
- float `subdividingEdgeDerivation` controls the randomness of the splitting line position
- bool `deleteInnerLots` specifies if lots without street access should be deleted

The complete street creation process is illustrated in figure 4.12.

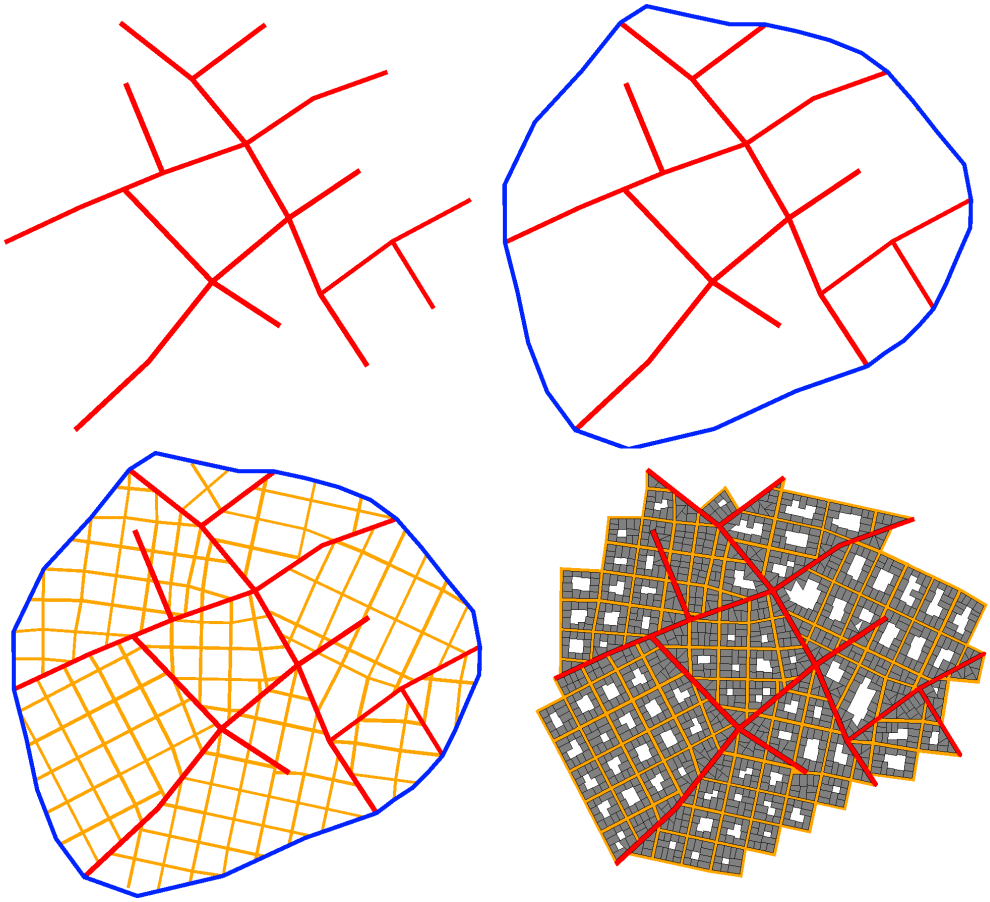

Fig. 4.12: This figure illustrates the steps necessary to create a street network. Top left: Major streets are created (organic pattern, red). Top right: The convex hull is calculated and bulged (blue). Bottom left: Minor streets are created inside the quarters (grid pattern, orange). Bottom right: The final street network with building parcels, convex hull and dead end roads removed.

Chapter 5

Street Geometry Tessellation

Street tessellation is not the main scope of my thesis, so I opted to implement a simple, but stable geometrical representation of the street network that enables simple shading of streets that can adjust to the slope of the underlying terrain.

Tessellating a single street is straightforward, but some problems arise at junctions: If each street was tessellated and rendered independently, discontinuities and z-fighting would appear where two streets meet. Therefore a special geometric representation for junctions is needed that allows junctions to connect an arbitrary number of street segments. Most previous work like [Geb08] limits junctions to connect a maximum of four orthogonal streets. But in general, real street layouts are usually more complex, like the crossing shown in Figure 5.1 that connects five street segments. One of the main requirements of the street tessellation was that such cases must be handled correctly without user interaction.

Another problem that arises is how streets that adjust to multiple height levels should be tessellated so that junctions are planar, but do not form unnatural steps on slopes. I will first describe my method for street tessellation in case of streets that are coplanar in Section 5.1. In Section 5.2, I will discuss how I solved the problem of non-coplanar street networks. In Section 5.3, a simple method to texture the created street geometry will be described.

5.1 Geometry for Planar Streets

The street network is represented as a planar graph of edges that connect to each other at junctions. To draw this network, we need to construct a fully polygonal street representation. The original edges serve as *centerlines* for the street geometry.

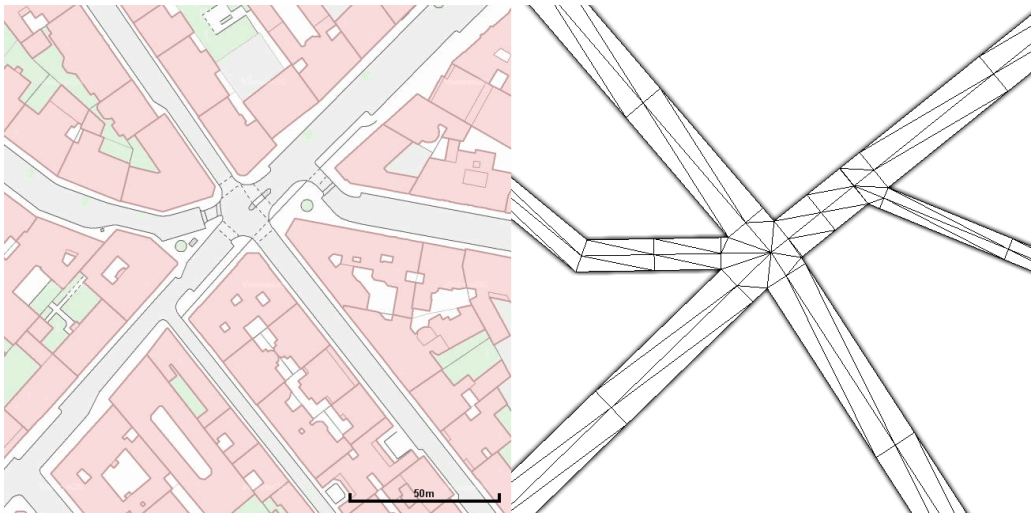

Fig. 5.1: Complex junctions connecting more than 4 street segments. Left: A crossing in the 4th district in Vienna (©ViennaGIS). Right: A similar junction modeled in my application.

The geometric representation of a street network consists of *junctions* and *street segments*. Each street has a certain `streetWidth` that was set in the street network creation process. The point where two street centerlines meet is called the *center point* of a junction.

To create a polygonal representation of the street, I use a similar approach as the one discussed in [Zim07]: we offset lines from the centerline on both sides by $\frac{\texttt{streetWidth}}{2}$. These offset lines are called *street outlines*. Street outlines of two adjacent street segments intersect at the *corner points* that define the corners of the junction and separate the junction geometry from the street segment geometry. To create the junction geometry, the two corner points of one end of a street segment together with the junction center point form a triangle. This triangle is called a *street head*.

A junction consists of n street heads, where n is the number of street segments adjacent to the junction. This is illustrated in Figure 5.2. In cases where two junctions are too close to each other, and junction geometries overlap, the two center points are merged to create a stable junction geometry.

5.2 Geometry Displacement for 3D Streets

The method described in [Zim07] creates good results as long as all streets are in one plane. If streets are not coplanar, numerous problems arise:

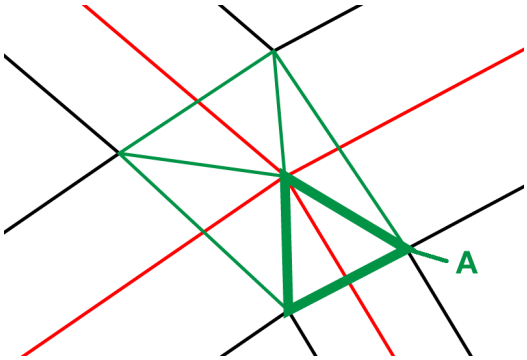

Fig. 5.2: This figure illustrates how a junction is defined by 4 street heads. The street centerlines are displayed in red and meet at the center point. Street outlines are colored black. Street heads are shown in green, the lower right street head is highlighted for clarity (A).

The junction geometries described in Section 5.1 need to be flat, otherwise streets will twist unnaturally. In the following, I will describe a method that nestles street segments and junctions to the terrain underneath while preserving the planarity of junction geometry.

To create flat junction geometry, initially all the vertices of the junction geometry are placed at the same height as the center point of the junction. This leads to unaesthetic and unrealistic steps and extreme slopes, as can be seen in Figure 5.3. These steps can be smoothed by moving the junction geometry into the tangent plane to the terrain surface at the junction center point. This can be done by calculating how much the normal of the terrain is rotated against the up vector. Based on that, a rotation matrix is calculated around an arbitrary axis that is later applied to every vertex of the junction. As a result, the whole junction geometry is rotated into the tangent plane of the terrain surface. An illustration can be found in Figure 5.3.

Unnatural steps in steep streets are avoided this way, but a certain lateral grade is introduced. This lateral grade is limited by the longitudinal slope of all other streets adjacent to this junction. This is acceptable for interactive applications such as games, since the maximum allowed longitudinal slope (12%) is not much higher than the maximum allowed lateral grade (8%)[1].

I follow the approach in [Zim07], where street segment ends are modified so that they connect to the junction at a line perpendicular to the direction

[1] In Austria; this may be different in other countries and for mountain roads.

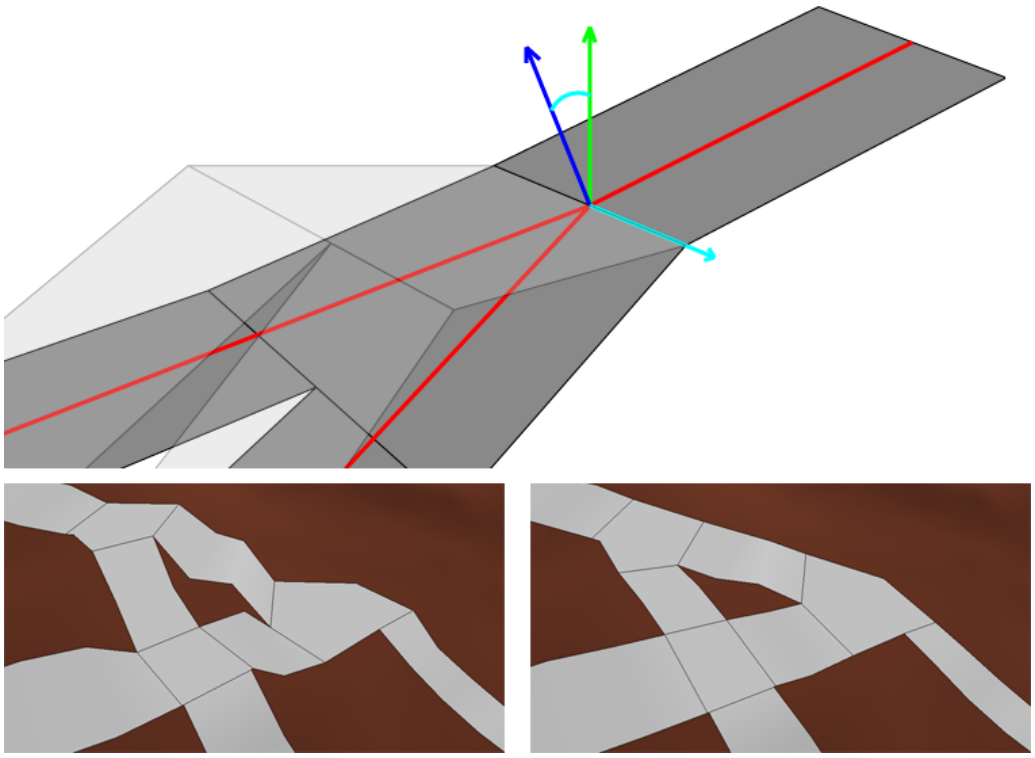

Fig. 5.3: Smoothing of junction geometry. Top: The initial geometry is displayed transparently. It is then rotated into the tangent plane of the terrain surface. Bottom, left: Geometry before rotation. Right: After rotation.

of the segment. Refer to Figure 5.4 for an illustration.

This is done for the following reasons:

- If the two corner points of a street head do not lie on a line perpendicular to the street segment direction, the street may twist unnaturally or become uneven.
- The connecting line separates junction and street segment geometry. This can be used to texture junctions differently than roads (as discussed in Section 5.3). In the future, queuing zones or crosswalks could be created at these lines.

This method produces simple and stable results, as can be seen in Figure 5.5.

In between the street heads, the geometry of the street segment is subdivided regularly and displaced according to the terrain height to nestle against

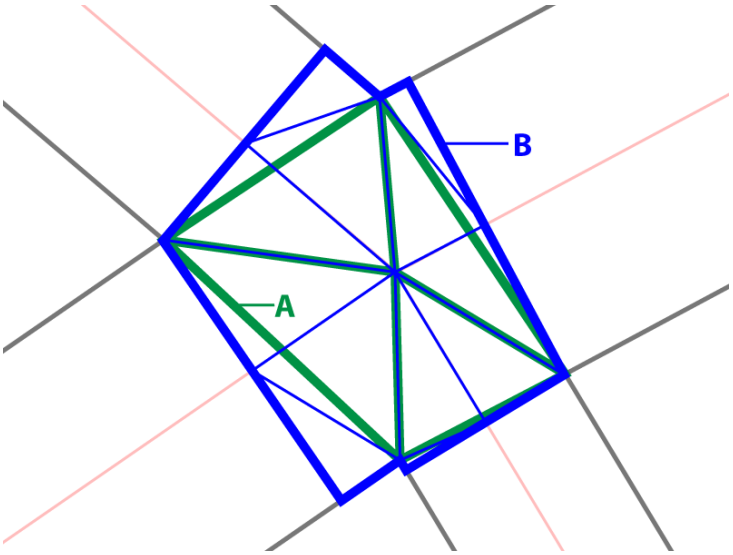

Fig. 5.4: Street ends are modified to create connecting lines perpendicular to the street segment direction. The original junction geometry is shown in green (A), the modified geometry in blue (B).

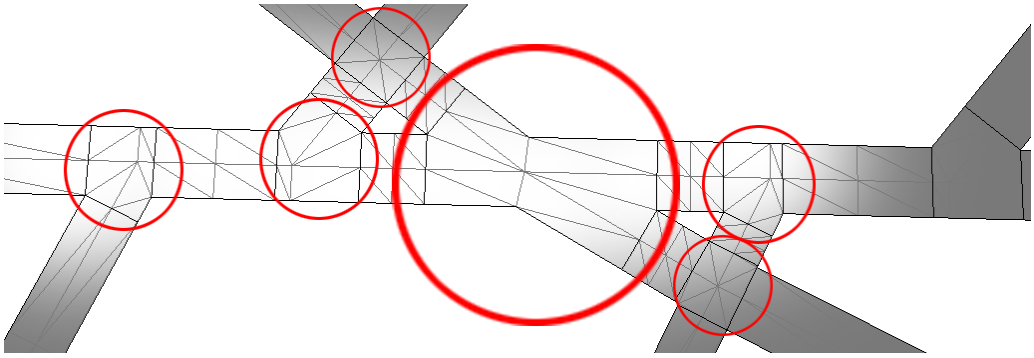

Fig. 5.5: Even complex junctions are tessellated correctly.

the underlying terrain surface. Street geometry is also displaced away from the terrain by a configurable offset to avoid z-fighting and overlaps between road and terrain geometry.

This may cause the street to appear floating above the terrain, so the tessellation algorithm adds side faces to the geometry that intersect with the underlying terrain. A screen shot of a tessellated street with side faces can be found in Figure 5.6

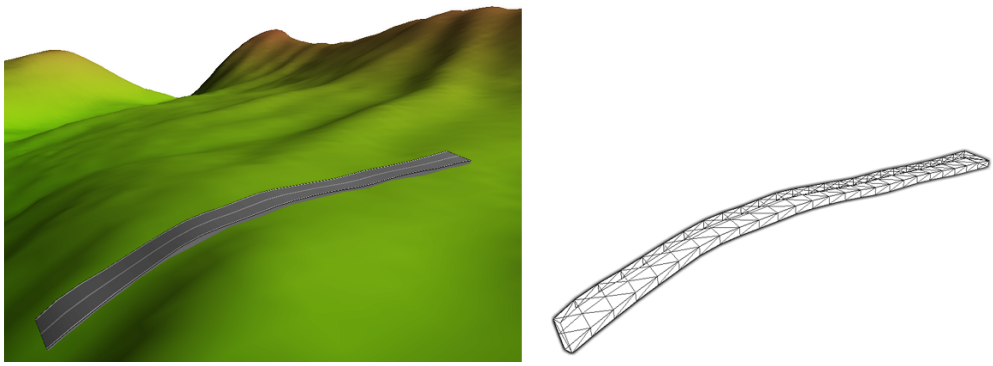

Fig. 5.6: A single tessellated street segment with side faces. Left: Shaded geometry with underlying terrain. Right: Wire frame view of the street segment with hidden terrain.

5.3 Street Texturing

To make it more visually appealing, generated street geometry is textured. This is done by using a road texture: every street has two lanes, independent from its width. The lanes are separated by a double line, the hard shoulder is bordered by a single line. The used input texture and some results can be seen in Figure 5.7.

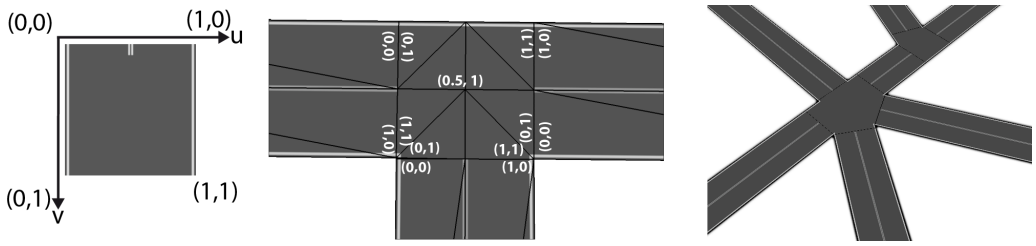

Fig. 5.7: Street texturing. From left to right: (1) Input texture with (u, v) coordinate system. (2) Simple T-junction, texture coordinates of vertices are shown in white. (3) More complex junction with texturing.

The texture contains the two hard shoulder lines and, around texture coordinate $(0.5, 0)$, the double line between the two lanes. For street segments, texture coordinates $(0, 0) - (1, 0)$ are used. Starting from the connecting line perpendicular to the segments, the junction geometry is textured using coordinates $(0, 1) - (1, 1)$. This ensures that the double lines at the center of the street will not be drawn in the junction area. Refer to Figure 5.7 for an illustration.

Chapter 6

A Constraint Based System to Populate Cities with Buildings

In today's games, urban environments are usually created by hand by artists using commercial software packages. Even in open sandbox games like *Grand Theft Auto* all buildings are placed manually by artists. This can take several man-years for larger environments. Of course certain regions of a city or village have to be hand-crafted to perfectly suit the game design, e.g. a vicinity where an important mission takes place in special buildings and scripted events matched to the surroundings guide the progress of the player.

But placing buildings in a larger environment outside of such gameplay related areas is a huge and tedious effort. Since it is not necessary to dictate the placement of every single building, a tool that places buildings in such locations would greatly reduce the time artists and designers have to spend modeling surrounding areas and would allow them to concentrate on the sites that are important for the attractiveness of the game.

Therefore, one of the main contributions of my thesis is a technique to automatically assign buildings to parcels from a set of previously modeled buildings. After the street network creation process described in Section 4, building parcels of various size and shape have been created by subdividing buildings blocks enclosed by streets, most of them being rectangular. I propose a method that selects the "best fitting" model for each parcel from a set of existing buildings. I define "best fitting" as the model that occupies most of the building parcel, while satisfying various constraints, such as not protruding from the parcel. These buildings can be created using a commercial modeling software like Autodesk Maya, or may be procedurally generated. They can be loaded into the application with various importing tools, and grouped together in a pool. An example of a set of previously modeled buildings is shown in Figure 6.1.

Fig. 6.1: A selection of 20 pre-modeled buildings.

The rest of this section is structured as follows: First, I will discuss some characteristics of building models in Section 6.1. Then I will describe my new method for selecting the "best fitting" building for a parcel in Section 6.2.

6.1 Building Properties

Each house has a footprint that can be defined as the convex hull of all vertices in its geometry projected onto the ground plane. This gives a good estimate for the area the building will occupy on a parcel. If this area exceeds the area of a certain parcel, the building will not be considered any further (constraint 1).

A building consists of sides that have to face a street side, e.g. because there is a door that needs street access. Also, there are sides that must not face a street, e.g. because there is a plain brick wall on this side. I will refer to them as *street access sides* and *inaccessible sides*. These sides pose constraints to the system that have to be met. *Street access sides* have to be aligned and placed next to streets to guarantee direct street access (constraint 2), and *inaccessible sides* have to point away from streets so that they will not be directly visible (constraint 3). All other faces of the building may or may not face a street (see Figure 6.2).

All these properties are stored as meta information in a XML file for each building model.

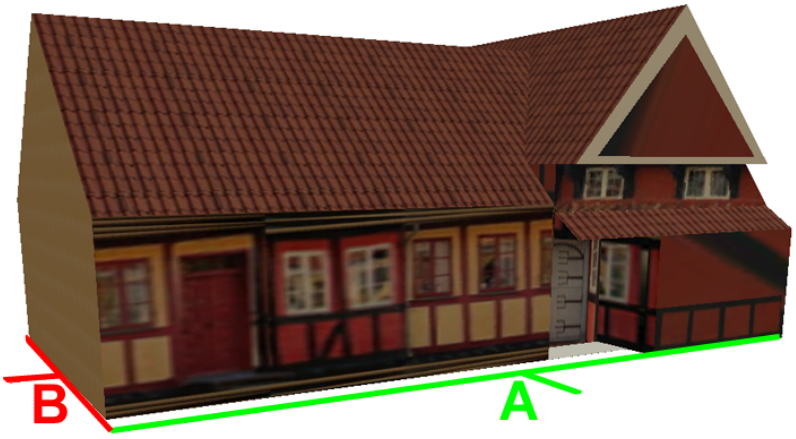

Fig. 6.2: A simple building model. *Street access sides* are displayed by green lines on the ground plane (A), red lines denote *inaccessible sides* (B).

6.2 Selecting a Building

The building with the largest footprint that meets all the criteria described above will be selected as the "best fitting" building for a parcel.

The set of previously modeled buildings is stored in a list ordered by footprint area size from largest to smallest. This list is enumerated for each parcel. All models that have a larger footprint area than the current parcel are discarded. Also, models that contain more *street access sides* than the parcel has adjacent street sides are not considered, because it is not possible to align these models correctly. For every remaining building, a series of transformations and tests are applied. The first model that passes all the tests is chosen for the current parcel. This guarantees that the building that occupies most of the parcel area and that meets all constraints is selected. The process is illustrated in Figure 6.3.

1. The building footprint is moved into the center of the parcel.

2. The largest *street access side* of the footprint is aligned to the largest side of the parcel that is adjacent to a street to get an initial alignment for the building. This suffices for most of the buildings, since many of them only have one front side that needs to face the street. If all sides are correctly aligned after this step, step 3 can be omitted.

3. Rotate the building footprint until all *street access sides* face a street and all *inaccessible sides* do not. A side faces a street if it is nearly

parallel[1] and a ray cast perpendicular from its center directly hits a street.

4. Move the footprint as close as possible to any street adjacent to the parcel. Buildings usually adjoin directly to streets, but a minimum distance can be configured in the user interface.

5. Check if all points of the footprint are inside the parcel.

6. If any of the above tests failed, repeat the process with the next smaller building. If a valid solution is found, assign the building to this lot considering the found transformations.

If the parcel is located on a slope, the building is moved down so that every vertex is on the ground or beneath. If the slope of the gradient of the parcel exceeds a user defined threshold `maxParcelSlope`, the parcel is discarded and no building is assigned.

The selected building is placed on the parcel and can be further modified in the same way as if it was placed manually. That means that it can be translated, rotated, scaled or even deleted as described in Section 7.3. This way, a designer or artist can make adjustments to the building placement if he/she is not satisfied with the results of the automatic assignment.

In Figure 6.4, a small suburban street network is displayed where buildings have been assigned to the parcels using the proposed method. The whole scene was created in less than 3 minutes. All the buildings were created by *Mogens Bregnbaek* and are taken from *Google 3D warehouse* [Goo10a]. Google 3D warehouse is a platform to share 3D content, including models that are intended for use in Google Earth [Goo10b]. At the time of this writing, models uploaded to Google warehouse could be used and distributed freely, at least for non-commercial use.

All buildings displayed in Figure 6.4 are part of the *Aeroeskoebing* collection. Aeroeskoebing is a town on the small island of *Aeroe* in Denmark. They were selected because they share a common design and nicely resemble typical buildings in European suburbs and villages.

[1] I chose a max. deviation of $\pm$ 30 degrees

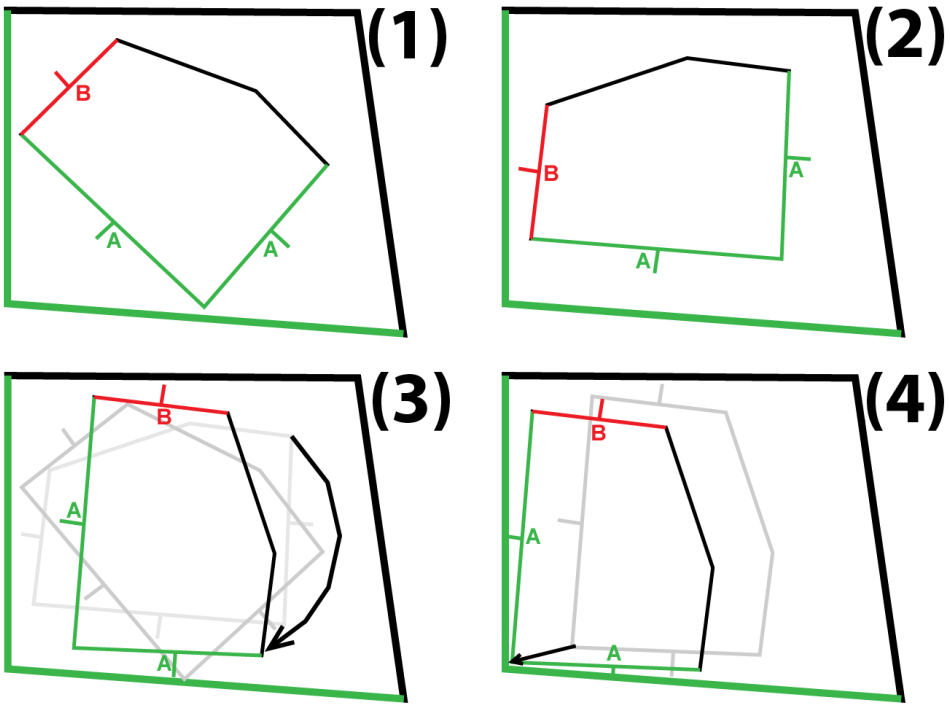

Fig. 6.3: Steps for fitting a building footprint into a parcel. The outer polygon illustrates the parcel, the inner one the building. *Street access sides* and parcel sides adjacent to a street are colored green (A), *inaccessible sides* red (B). (1) The footprint is moved into the center of the parcel. (2) The largest *street access side* is aligned with the largest parcel side with street access. (3) The footprint is rotated. (4) The footprint is moved close to the streets.

Fig. 6.4: Buildings assigned automatically to parcels using a constraint based system.

Chapter 7

Interactive Editing of Urban Scenes

7.1 Interactive Street Sketching

As described in Section 4.2, street networks can not only be created automatically using L-Systems, but streets can also be placed manually by the user in my system. This is crucial for artists and designers if they want direct control over the street network layout. Of course it is also possible to just create single streets manually and the rest of the city procedurally. Streets can also be added to a previously created road network.

Manual street creation is realized via *interactive street sketching*: First, the user has to set some parameters such as street width and type (major or minor) or control maps such as the terrain map used to displace the streets. Then the street geometry can be drawn interactively by clicking at a position in the render window. A street graph node is created and placed at this position. The world space position of the node is calculated by *picking* as described in [AMH02, p. 557-564]: A ray is cast from the camera position through the unprojected screen space position where the mouse was clicked. This ray is then intersected with the terrain geometry to evaluate the exact position the user selected. If no terrain map is provided, the ray is intersected with the ground plane and the node is placed at the intersection point. Street sketching is shown in Figure 7.1.

After each click, a new street segment is created. The end point follows the mouse cursor while it is over the render window. This way, the user is able to see exactly what geometry will be created when placing the next node at the current position. The geometry of the current segment is updated continuously while the cursor is moved.

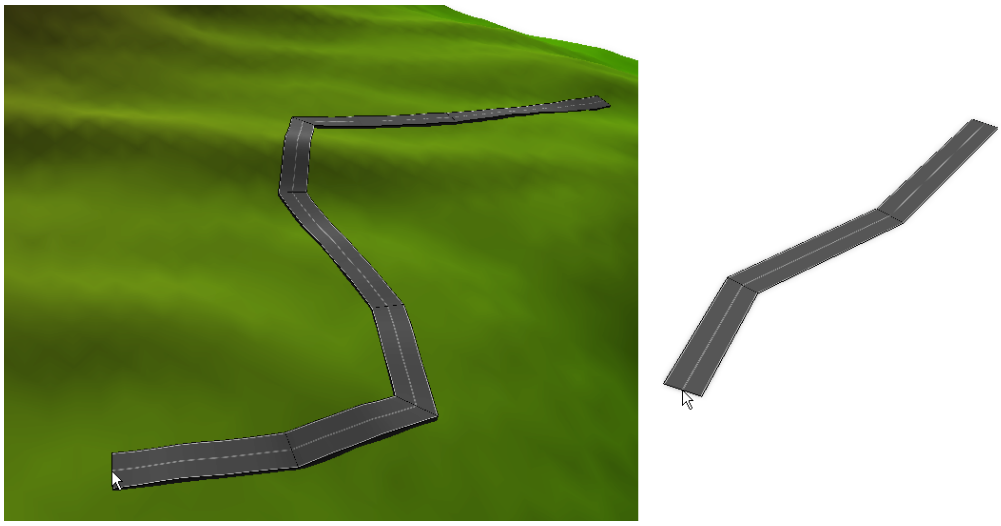

Fig. 7.1: Interactive street sketching; street nodes can be placed using point-and-click operations. Left: A street created on terrain; Right: Without terrain.

Instant intersections If the current street segment intersects an already existing segment, it is shortened to end at the intersection point. As soon as a new node is created, this node is also inserted into the existing street and a new junction is created and its geometry instantly tessellated. Refer to Figure 7.2 for an illustration.

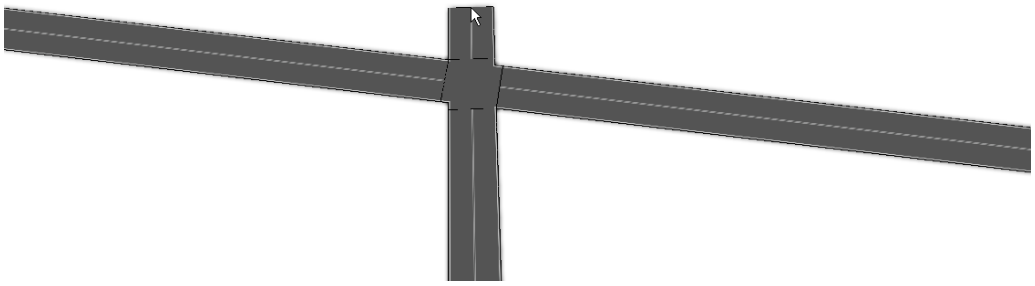

Fig. 7.2: Junctions and the corresponding geometry are created on-the-fly when the currently drawn segment intersects an existing street.

Snapping To prevent close-by intersections and overlapping junction geometries, the end point of the currently sketched road segment is snapped

to existing nodes in the street graph if the distance of the new and the existing node is below a user-defined threshold `snappingDistance`.

Street deletion Street segments can also be deleted manually. There are two modes to delete road geometry:

- The user can select a single street segment by picking it in the render window. This time, the ray is intersected with the *bounding volumes* of the street segments (or junctions for the next mode) [AMH02, p. 616-619]. If an intersection is found, the ray will be further tested directly against the segment geometry using a *ray-triangle test* as described in [AMH02, p. 578-582]. If the ray intersects a triangle, this segment will be selected and can be deleted in the user interface. Just this segment will be removed, adjacent nodes will be preserved as long as the segment is not a dead end.

- Similar to picking a segment, the user can also select a node or junction. If a node is deleted, all adjacent street segments will be removed too.

Topology changes To enable the *topology preserving transformations* discussed in Section 7.2, parcels and buildings are assigned to the street graph facets (i.e. blocks) that enclose them. Adding or removing street segments may change the topology in the street graph. Facets can be split or merged and parcel/building assignments may become invalid.

In this case, parcels and building models that were assigned to a no longer existing block are reassigned. This is done with the following steps:

- Collect all "abandoned" parcels and buildings in an array.

- Calculate all shrinked blocks in the street network as described in Section 4.2.6.

- For each parcel/building to reassign, test if it is completely contained in one of the shrinked blocks.

- If it is contained in a block, reassign it to the corresponding parcel.

- If it is not contained in any block, remove the assignment. The position of that object is now fixed and will not change with the surrounding geometry as described in Section 7.2.

- If a parcel or building is now protruding out of a shrinked block, it will intersect a newly created street segment. Therefore, this object has to be removed. This is illustrated in Figure 7.3.

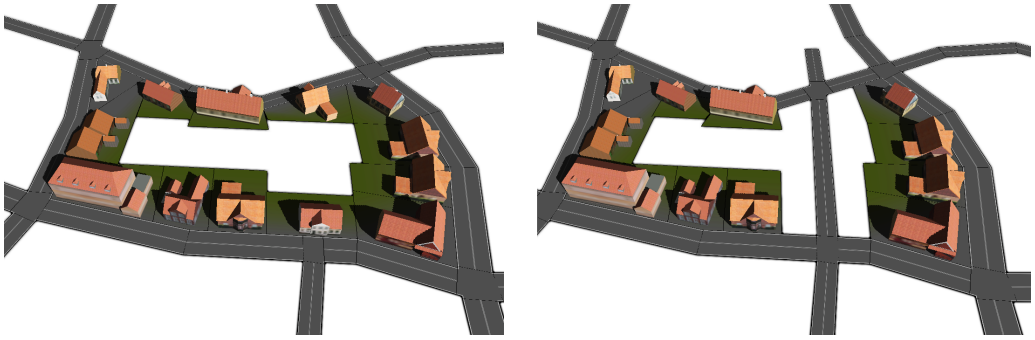

Fig. 7.3: Block Splitting. If the topology of the street graph changes, existing parcels and buildings are instantly reassigned to new building blocks. Left: Original block with parcels and houses. Right: After adding a new street, the block is split, objects intersecting the road are removed.

7.2 Topology Preserving Transformations

Interactive street sketching as described in Section 7.1 enables the user to create street networks fitting his requirements. But usually, urban environments are not used "as-is" right after they have been created, they are changed and modified very often. Therefore, it is desirable for an artist or designer to be able to directly edit street networks including adjacent parcels and models without removing a street and remodeling it again.

Inspired by the non-topological transform proposed by Lipp et. al [LSWW10], I implemented a similar method to interactively edit the street network without changing the topology of the street graph. This can be done by selecting any node or crossing (again by picking) directly in the render window and dragging it around. The new position of the node is calculated in the same way as described in Section 7.1. Adjacent street segments and their geometric representations are updated instantly. In addition, the adjacent parcels and buildings are moved with the node. In the following, I will describe in more detail how this is done.

Mean value coordinates Parcel vertices and the position of models inside a closed building block are stored as *mean value coordinates* as proposed by

Floater [Flo03]. Mean value coordinates are a generalization of barycentric coordinates that allow to express a vertex v_0 as a convex combination of its neighboring vertices v_1,...,v_k using a set of weights λ_1,...,λ_k such that [Flo03]:

$$\sum_{i=1}^{k} \lambda_i v_i = v_0$$

$$\sum_{i=1}^{k} \lambda_i = 1$$

"Let α_i, $0 < \alpha_i < \pi$, be the angle at v_0 in the triangle $[v_0, v_i, v_{i+1}]$, defined cyclically. Then the weights

$$\lambda_i = \frac{\omega_i}{\sum_{j=1}^{k} \omega_j}, \omega_i = \frac{tan(\alpha_{i-1}/2) + tan(\alpha_i/2)}{||v_i - v_0||}$$

are coordinates for v_0 with respect to v_1,...,v_k." [Flo03, p. 21]

These mean value representations are used to store parcel and building positions with respect to the nodes of their enclosing building block. This way, if a position v_i of a block polygon v_1,...,v_k changes, the updated parcel vertices and building positions can easily be calculated using the stored weights λ_1,...,λ_k with the formula introduced above:

$$\sum_{i=1}^{k} \lambda_i v_i = v_0$$

This calculation is very fast (as opposed to the calculation of weights, which has to be done just once) and can be used to modify the contents of a building block at interactive frame rates. Figure 7.4 shows some results.

Updating the street geometry When moving a node in the street network, the tessellated street geometry has to be updated in real time. Tessellating the whole network would take too long, it would be impossible to edit streets interactively. Therefore, each street segment is stored in a separate *vertex buffer*. This results in many more draw calls than if a single buffer had been used for the whole network, and therefore a significant overhead. On the other hand, this way streets can be culled away if they are not visible using standard *view frustum culling* [AMH02, p. 363-365].

Fig. 7.4: Topology Preserving Transformations. Left: Two building blocks with parcels and building models in their initial state. Right: The same blocks after the street at the center has been moved to the left and the right street even more to the right.

If a node is moved, all adjacent street segments have to be re-created. But this is not always sufficient, since the intersection points between these segments and their adjacent segments at the neighboring nodes may have changed during the process. Therefore, these segments of *second grade* have to be updated as well. This is shown in Figure 7.5.

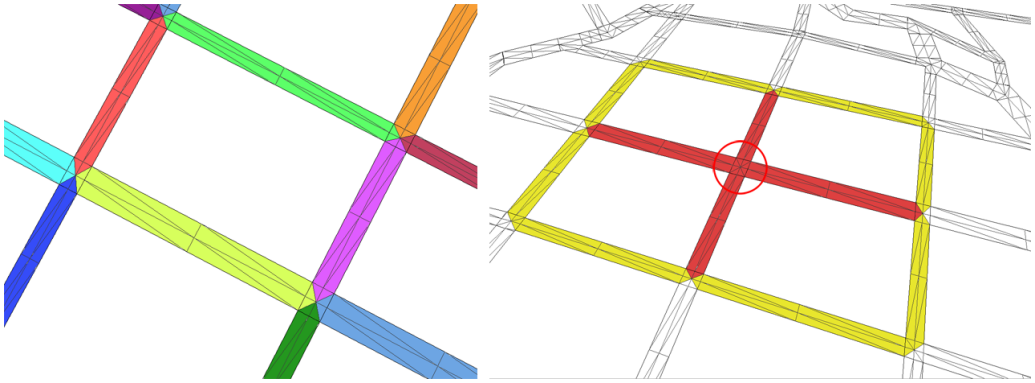

Fig. 7.5: Street Geometry updating. Left: Every color represents a single street segment vertex buffer. Right: If the junction marked with a red circle is moved, the surrounding segments are updated. Directly adjacent segments are colored red, segments of second grade are colored yellow.

7.3 Building Transformations

Not only will it be necessary to modify the street network in the game development process, but it should also be possible to place, rotate, and scale every model including buildings. For this purpose, standard 3D manipulators are used. When I interviewed experienced game designers and artists

about how such editing operations should work, everybody insisted that I should use a user interface known from modeling software, such as Autodesk Maya [Aut10]:

Translation manipulators are visualized as a 3D coordinate system. The user can click at one of the axes and pull and push it back and forth in the axis direction. The model is then translated into the corresponding direction. Models can also be translated with one additional degree of freedom by moving a plane defined by two of the axes.

Rotation is only possible around the vertical Y-axis, since buildings should not be tilted. For this manipulator, a circle is displayed around the model that can be dragged to rotate it.

Scaling works very similar to translation. Building models can be scaled around single axes and planes. Additionally, a uniform scaling manipulator can be used that does not distort the model.

Hover mode This is a transformation mode not available in Autodesk Maya. Models can be dragged around and placed at the current terrain position. This is implemented using the same picking code as explained in Section 7.1 and the building is placed at the intersection position. This is a very convenient tool for fast and accurate placing of buildings on terrain, parcels or street geometry.

Illustrations of all manipulators can be found in Figure 7.6.

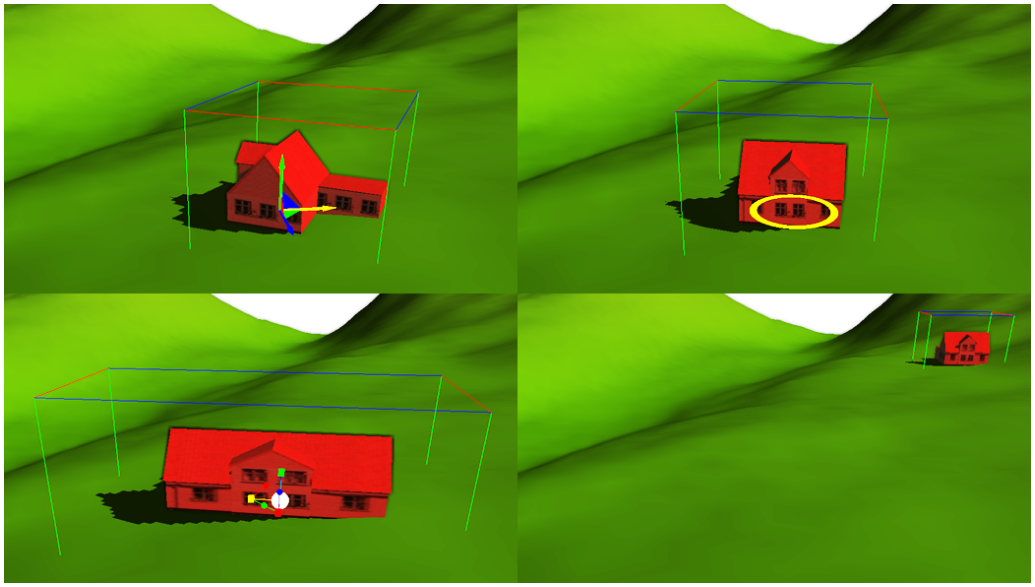

Fig. 7.6: Model transformation manipulators. Top left: Translation. Top right: Rotation. Bottom left: Scaling. Bottom right: Hover mode.

Chapter 8

Implementation and Results

8.1 Implementation Overview

The introduced system was implemented as a *C# application*. The XNA framework [Mic10] was used for rendering and to simplify various tasks. XNA includes an extensive set of libraries specific for game development. While normally the XNA runtime has to be installed first to run a software that makes use of it, all required assemblies are included in my application. All the software requires is Windows and the DirectX Runtime.

Using the XNA framework has various advantages over using native DirectX or OpenGL that have allowed me to focus on the central issues of my thesis:

- XNA provides complete model loading functionality. Building models can be imported using built-in methods without writing a custom importer. FBX and X formats are supported out of the box and extensions for other formats such as Collada or OBJ exist and can easily be docked to the application.

- XNA already contains a simple shader framework that can be used right away. While it does not feature advanced effects such as shadow mapping, it is easy to extend and has saved me a lot of time otherwise spent on implementing basic shader functionality.

- Many math helpers and data structures are available. Besides vector, matrix and quaternion classes, intersections between rays, planes, bounding spheres, boxes and frustums can be calculated. This made it easy to implement view frustum culling or object picking in a couple of hours.

8.2 Code Reuse

Most of the algorithms presented in this thesis were implemented in a separate library and decoupled from any rendering framework or user interface. This includes the following features:

- Procedural street network creation
- Block shrinking and parcel subdivision
- Street geometry tessellation
- Automatic building assignments

All these functionalities can be used via an interface from any other .net application. The library uses data structures such as a street graph including nodes and edges, vector and matrix types or parcels. These data structures are defined as interfaces that must be implemented by the user of the library. To allow the instantiation of classes implementing the data type interfaces, the *abstract factory pattern* is used.

Abstract factory is a pattern used to insulate the creation of objects from their usage. An interface is defined that provides methods to create instances of abstract data types. The factory class implementing this interface is responsible for instantiating the objects. This way, the algorithms in the library have no knowledge of the concrete type and only deal with the abstract types defined by the provided interfaces. The declaration of the *IAbstractDatatypeFactory* interface can be found in Listing 8.1.

This allows to reuse the code in various other applications and decouples the algorithms from the application. Most of these techniques were actually reused by Lipp et al. in their related work [LSWW10].

8.3 User Interface

The user interface of my application can be seen in Figure 8.1 and is divided into three parts:

Render window This is the central element of the user interface, where the current scene is displayed. Here, the user can select elements (buildings, streets, junctions, markers etc.), place, move and delete them. The camera is controlled in the same way as in Autodesk Maya [Aut10]: While pressing

```
1     public interface IAbstractDatatypeFactory
2     {
3         IVector2 CreateVector2(double x, double y);
4         IVector3 CreateVector3(double x, double y,
              double z);
5
6         INode CreateNode(double x, double y, double
              z);
7         IWay CreateWay(IStreetGraph father);
8         IParcel CreateParcel(IStreetGraph father,
              double verticalOffset);
9
10        IStreetGraph CreateStreetGraph();
11    }
```

Listing 8.1: Interface for abstract data type factory

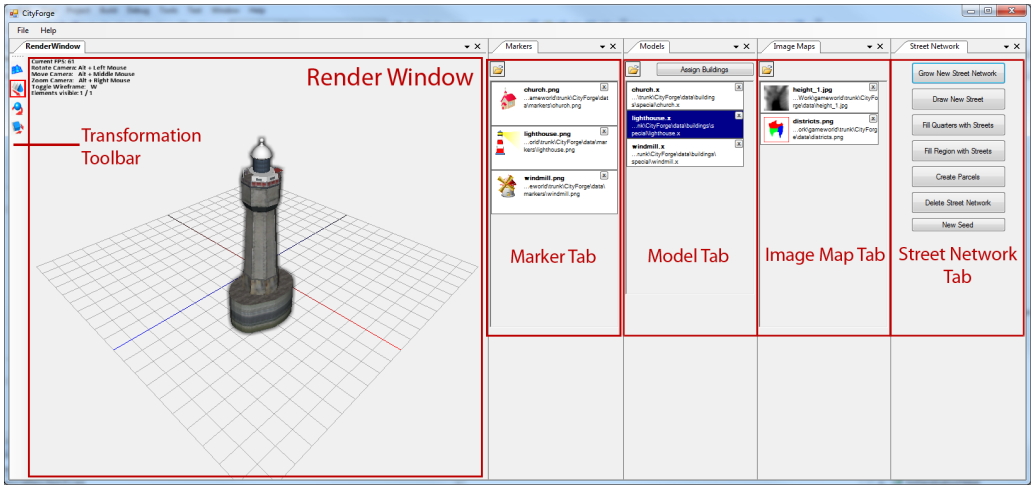

Fig. 8.1: The user interface of my application. The four panels at the right are usually placed behind each other and have been rearranged for illustration.

the "Alt" Key, the camera can be rotated by holding the left mouse button, translated by pressing the middle mouse button, and zoomed in and out using the right button.

Transformation tool bar The tool bar is placed left of the render window. Here, the user can select the current transformation mode (described in Section 7.3):

- Hover mode
- Translation
- Rotation
- Scaling

Element panel The element panel is divided into four tabs. Usually these tabs are placed behind each other and the user can switch between them, but they can also be rearranged as shown in Figure 8.1.

The marker, model and image map tab work very similar: Elements can be loaded from a file (images or 3D models) and imported into the application. They are displayed in a list together with their attributes, such as the transparency value. From this list, they can be dragged into the render window and placed in the scene. The model tab also contains a button to open a dialog used to assign buildings as discussed in Chapter 6.

The street network tab differs from the other tabs and contains buttons to create streets and street networks. Apart from drawing streets manually, there are three ways to create streets automatically:

- Create a whole street network using L-Systems. This can be used to create a street network without placing any street by hand. The dialog is shown in Figure 8.2.
- Fill the quarters of an existing network with minor streets. This can be useful if a rough layout of the city was created by a manual sketch of major roads. The finer structures of the quarters are then created automatically.
- Create streets inside a mask polygon drawn by the user to grow streets in a bounded region only.

The street network tab also contains buttons to create parcels and to completely delete a street network.

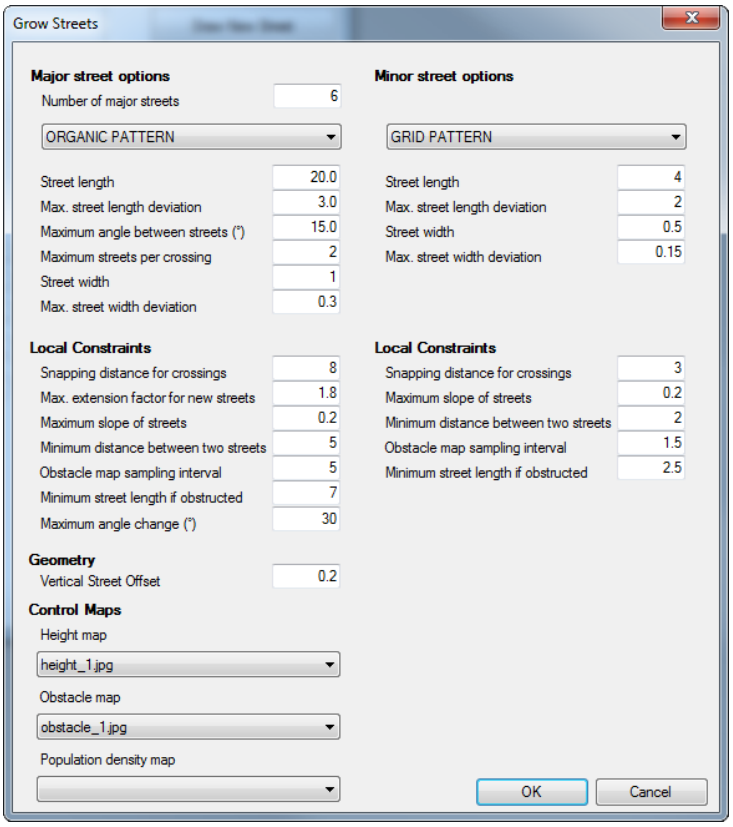

Fig. 8.2: The grow streets dialog. Parameters for the major streets are shown left (organic pattern), parameters for the minor streets on the right (grid pattern). The user can also select control maps, such as the terrain or obstacle map.

8.4 Rendering Improvements

Building models and other elements are hard to place in a scene without any depth clues showing the exact position of the object in 3D space. Two improvements were implemented to enhance the depth impression in the rendered scene:

Shadow maps All objects cast and receive shadows. The shadow map is updated after each modification of the scene (e.g. moving a building) and resized to contain all elements of the created environment.

Depth darkening A post processing shader is employed that darkens areas in screen space where depth values change strongly. This creates a dark "glow" around the silhouettes of objects that helps to distinguish them from

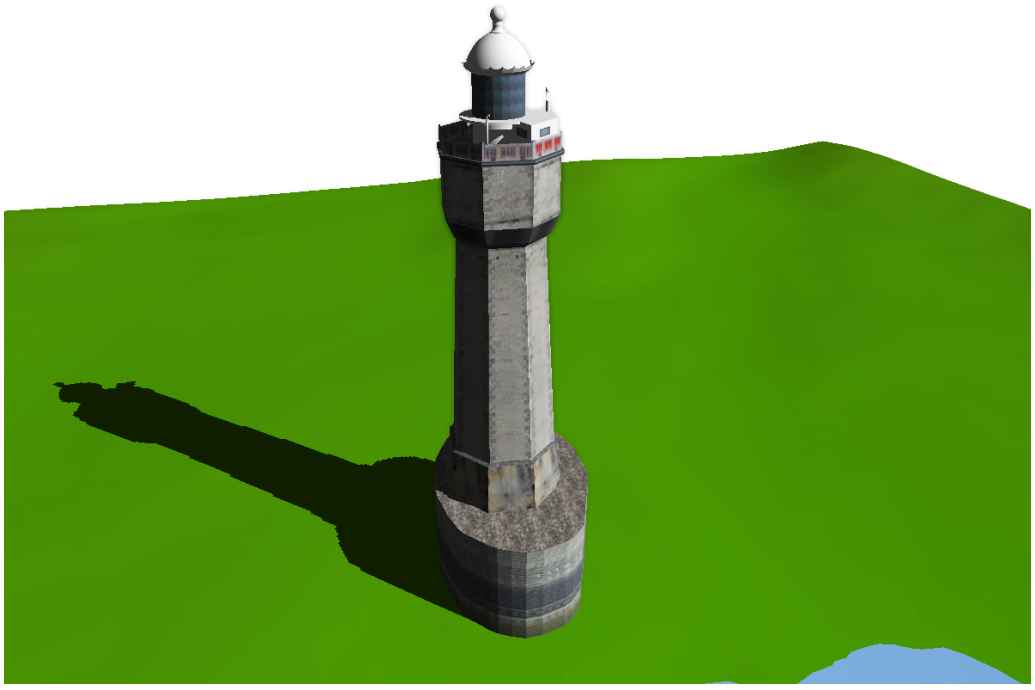

Fig. 8.3: Rendering Improvements: Shadows are cast by scene elements onto other objects, and a depth darkening effect outlines silhouettes.

the background.

A screen shot that shows both effects can be found in Figure 8.3.

8.5 Persistency

To save and load a modeled city scene, it can be serialized to and deserialized from an XML file. To implement this functionality, I used the C# serialization framework. The attributes (e.g. the used texture map, the position, etc.) of all objects are saved to a file. This file can be loaded later on and the objects are created with the serialized attributes. Pointers between objects are handled by saving its hash code[1] with each object. This hash code can then be referenced from other objects to store a pointer.

[1] A hash code is a unique identifier for an object.

8.6 Results

In the following sections, I will present two different (sub)urban environments that were created with the methods described in this thesis. I will also show performance measurements of various operations carried out in these scenes and discuss the tasks and time that were necessary to build them.

8.6.1 Bay Area Village

The first scene is a village near a bay. It was layouted using an area map that denotes three different town districts: On one side of the bay, a residential district is arranged around a church. On the opposite side there is a more suburban area with a lighthouse at the water front. An agricultural area is set in the back-country with some windmills in between. The positions of the church, the lighthouse and the windmills was indicated with markers in the layout process. Figure 8.4 shows the layout process and the creation of the agricultural area. Screen shots of the other quarters and some close-up views are displayed in Figure 8.5. Performance measurements taken in this scene are shown in Table 8.1.

Task	Count	Time
Procedural street creation[1]	34 streets, 155 segments	4.21s
Street network tessellation	63 streets, 261 segments	2.17s
Parcel creation	226 parcels, 56 blocks	0.09s
Building assignments	223 buildings	1.48s
Junction manipulation[2]	12 segments tessellated, 17 building parcels updated	0.18s

Tab. 8.1: Performance benchmarks of my system for the village scene.

8.6.2 Virtual New York City

The second showcase scenario is a city scene that resembles *Manhattan*. To layout the city, a simplified aerial image of New York City was used. Central Park and the Statue of Liberty were flagged with markers. Again, the major

[1] Only the streets inside the three quarters were created procedurally.

[2] A junction node moved by the user. The junction is adjacent to 4 street segments and 4 building blocks with a total of 17 building parcels that are updated. Together with the 2ndgrade segments, 12 segments have to be tessellated anew.

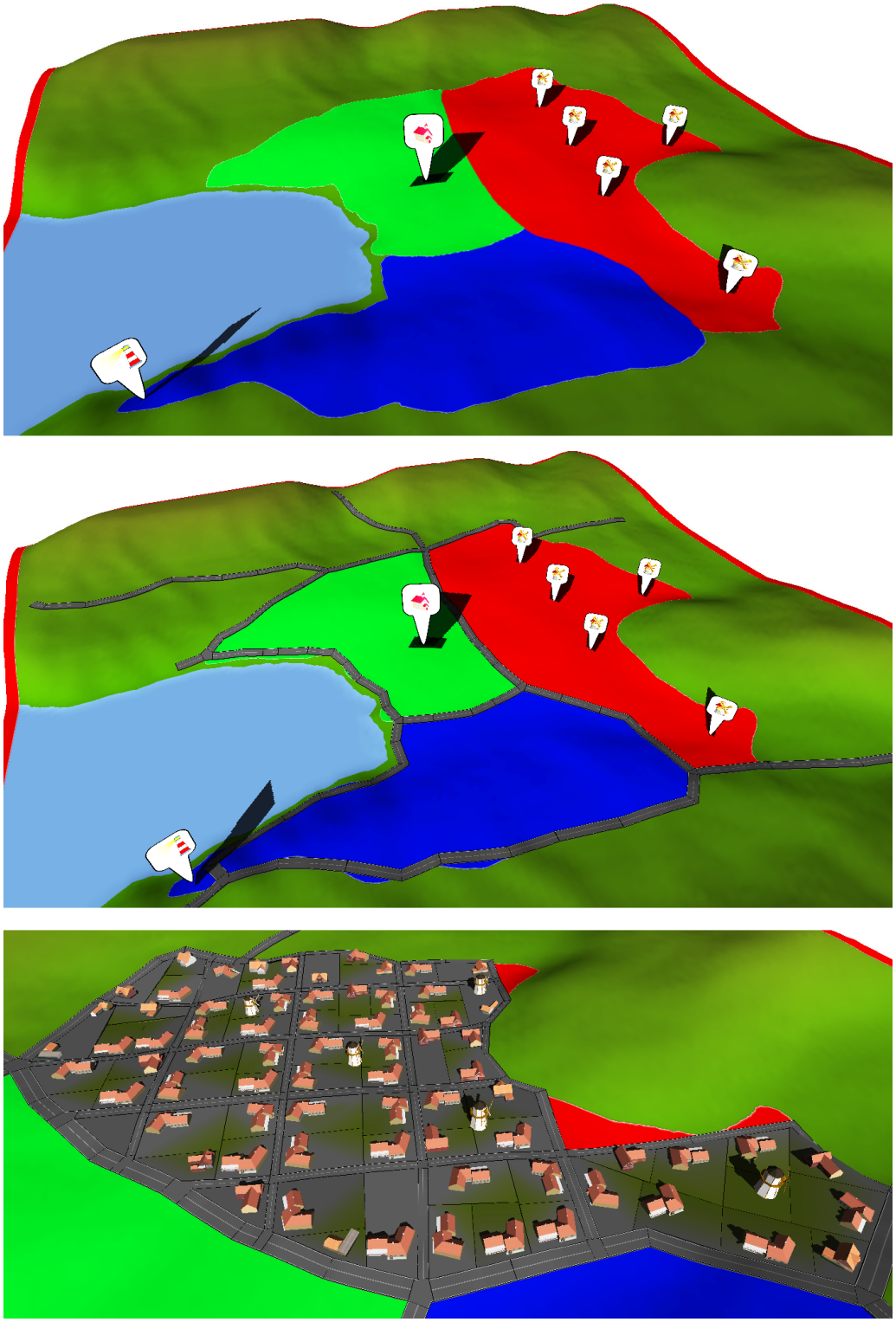

Fig. 8.4: Layouting process of the bay village scene. From top to bottom: (1) Initial terrain mesh with projected area map and markers (2) Manually sketched major roads (3) Agricultural area with parcels and buildings; streets have been generated procedurally.

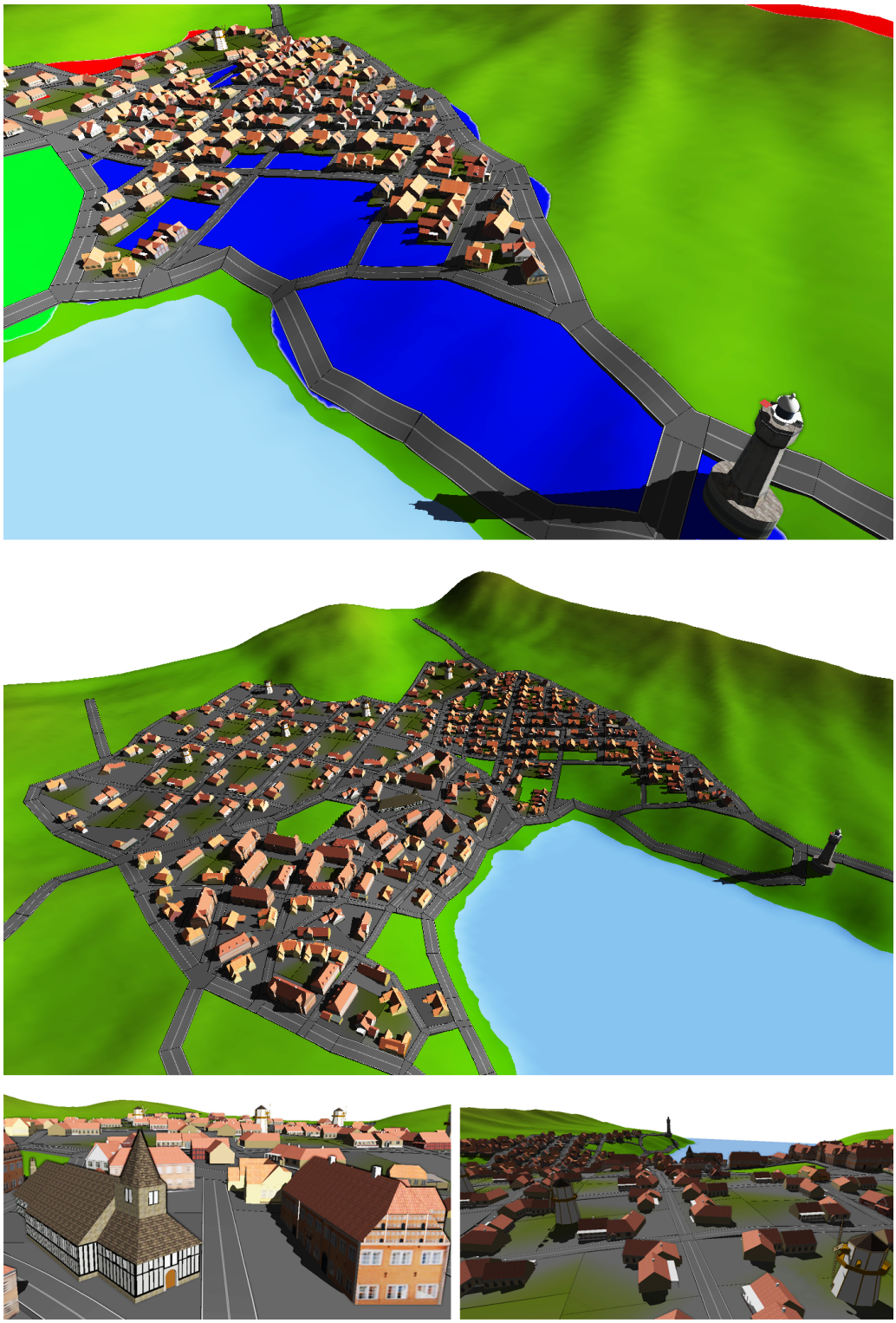

Fig. 8.5: Bay village results. From top to bottom: (1) suburban quarter with lighthouse; note that no parcels have been created on the steep slope towards the water; (2) Overview of the village; the residential district is front, the agricultural area behind; (3) Close-up views show details like the church or windmills.

streets were created by hand, and the minor streets were generated procedurally using a grid pattern typical for Manhattan. Screen shots of the scene can be found in Figure 8.6. Table 8.2 shows benchmarked measurements.

Task	Count	Time
Procedural street creation[1]	43 streets, 239 segments	1.21s
Street network tessellation	51 streets, 282 segments	4.47s
Parcel creation	132 parcels, 116 blocks	0.10s
Building assignments	112 buildings	0.54s
Junction manipulation[2]	12 segments tessellated, 4 building parcels updated	0.17s

Tab. 8.2: Performance benchmarks of my system for the village scene.

8.6.3 Modeling Effort

Another aspect worth evaluating is the time needed to plan and create a virtual environment using the system described in this thesis. Table 8.3 lists the tasks that were necessary to create the scenes presented above. Modeling urban settings with similar level of detail as the shown examples usually takes days if the designer uses a traditional 3D modeling software. With my application, urban environments can be built from scratch in a couple of hours. Please also bear in mind that the building preparations listed in Table 8.3 have to be made just once, then these buildings can be utilized in any future scene without additional overhead.

[1] Since the minor streets were created all at once and not separately for each quarter, the generation was much faster than in the previous example.

[2] A junction node moved by the user. The junction is adjacent to 4 street segments and 4 building blocks (with one building parcel each) that are updated. Together with the 2ndgrade segments, 12 segments have to be tessellated anew.

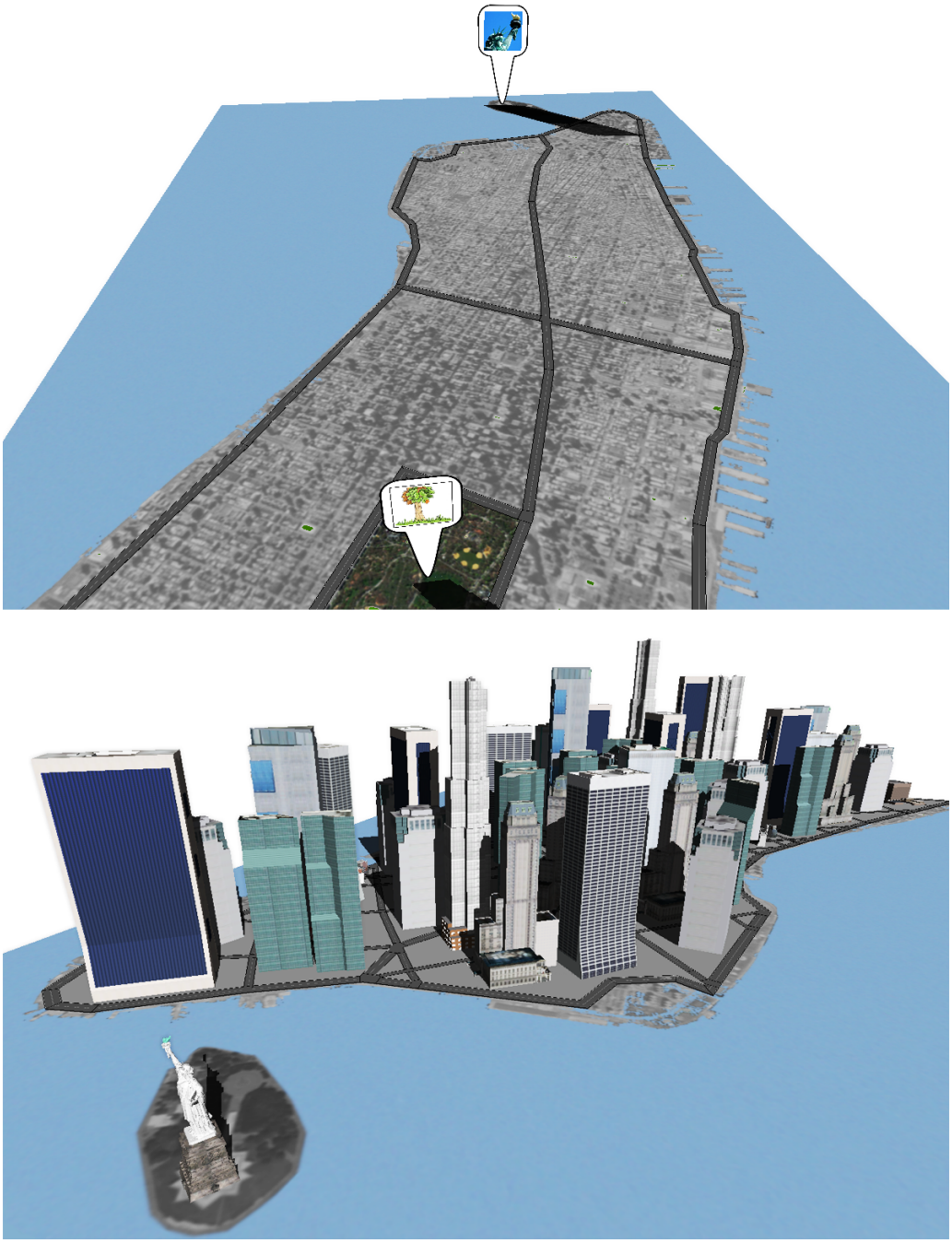

Fig. 8.6: Results from the Manhattan scene. Top: The initial city map with markers denoting Central Park and the Statue of Liberty. The outline of the city and some major roads were created manually. Bottom: A screen shot from the finished scene.

Task	Description	Bay area	Manhattan
Building preparations	Identify and mark *street access sides* and *inaccessible sides* for each building	60min	30min
Planning and layout	Create and draw height maps, area maps, textures, obstacle maps and markers; import all assets, configure scene, place markers	90min	60min
Creating street skeleton	Sketch the major streets to define the main street layout	30min	20min
Street network	Procedurally generate the minor streets inside quarters, adjust generated street network, parcel creation	20min	15min
Editing and modifications	Final adjustments: place special buildings, adjust single building assignments to personal wishes, modify street network if necessary	30min	20min
Total		230min	145min

Tab. 8.3: Tasks and approximate time necessary to create the bay area and Manhattan scenes.

Chapter 9

Conclusion

9.1 Summary

In this thesis, I presented a system that can be used by designers and artists to interactively plan, layout and create urban environments for games. Creating such environments poses great challenges for the developers of modern games.

The main scopes of my thesis were guided by limitations of previous methods for procedural city generation as well as ideas deduced from an interview with a professional game designer experienced in creating virtual worlds for games. In the following, I will describe the main contributions and improvements of my thesis over existing systems:

Planning and layout Existing applications and methods provide insufficient tools for artists that allow them to plan an urban environment before modeling it. The system I presented supports various techniques that simplify the work of artists and game designers: height maps can be imported for terrain generation and maps can be projected onto the ground. This includes area maps, topology information, or maps denoting gameplay elements. Additionally, points of interest can be indicated with markers.

Street network creation While the procedural generation of street networks greatly reduces the time needed to build a virtual city, it minimizes the artist's ability to control the created environment in every detail. In the presented application, streets can be drawn interactively using a sketch-based approach. Procedural street generation can be employed accurately in specified regions or quarters without limiting the artist's possibility to create streets according to their wishes. Additionally, I introduced a new method to create minor roads at outer city regions where previous methods fail to find

enclosed quarters and fill them with streets. Procedurally generated street networks have a more natural layout using this method.

Street geometry improvements The geometric representation of streets described in previous methods is usually limited to connect only three or four street segments. The method proposed by Zimmerman [Zim07] is capable of tessellating junctions with more than 4 segments, but the created geometry does not adjust to the local features of the terrain. Based on this work I described a method that is able to create junctions with any number of adjacent segments and proposed a novel tessellation technique that adapts to the underlying terrain without creating unnatural steps or steep slopes.

Interactive editing In the development process of a game, the virtual environment will be changed and revised very often. In previous work, the created street geometry can not be edited directly. If at all possible, the initial street graph has to be changed and the tessellation of the affected street segments must be triggered manually. The application described in this thesis allows to modify the street geometry directly by adding, moving or deleting streets. The involved geometry is updated interactively.

Building placement A major contribution in this thesis is the introduction of a constraint based system that automatically places buildings in a city. Without this method, every building has to be positioned manually by the user. This is a very tedious and time consuming process. Various building properties are taken into account to place a building on a parcel, such as its footprint size and sides that must or must not face a street.

As a whole, the methods described in this thesis form a system that enables and supports artists and game designers to create virtual urban environments in a very short time. The application provides support for all aspects of the development process, from the planning of the city to its realization and revision.

9.2 Future Work

Since this thesis covers many different methods used to create virtual environments, there are various possibilities for improvement and future work. Some of them are listed below:

Curved street segments The algorithm used in this thesis to create street networks is based on previous methods that employ L-Systems. The street graph created with this technique consists of nodes and straight edges between them. However, real streets are often curved. While this can be modeled with many short segments, it would be beneficial to investigate possibilities to create curved street segments. This could for instance be done using a spline-based approach.

Detailed street tessellation The street geometry created in my system is quite simple. To create more detailed street representations, shape grammar methods could be used. Street details such as multiple lanes, crosswalks, sidewalks and lamp posts could be added this way.

More sophisticated building placement The constraint based building assignment algorithm could be improved in many ways:

- An image map similar to a height map could control the building height for different regions.
- Another map could be used to manage the type of buildings: A building could be of a certain type (residential, industrial, suburban, etc.), and each type would be associated with a certain color in the map. This information could be used in the building assignment process to automatically create districts with a different look and feel.
- In one building assignment pass (buildings can be assigned at any time, only empty parcels are populated automatically) the same set of buildings is used for every parcel. Therefore, it mainly depends on the parcel properties (size, shape, street access) which building is selected. If the parcels are very uniform, a few buildings will be assigned very often. To account for that, the algorithm could be adapted to consider often used buildings less for future assignments.
- The building assignment process does not take neighboring buildings into account. In most larger cities, adjacent buildings directly connect to each other and form a common facade. To model such a behavior in my system, the algorithm must take the adjacent parcels into account and select building models that fit together.

List of Figures

Bibliography

[AAAG95] O. Aichholzer, F. Aurenhammer, D. Alberts, and B. Gärtner. A novel type of skeleton for polygons. *Journal of Universal Computer Science*, 1(12):752–761, 1995.

[Ad86] H. Abelson and A. diSessa. *Turtle Geometry*. MIT Press, 1986.

[AMH02] T. Akenine-Möller and E. Haines. *Real-Time Rendering 2nd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2002.

[Aut10] Autodesk, Inc. Autodesk maya, www.autodesk.com/maya, 2010.

[AVB08] D. G. Aliaga, C. A. Vanegas, and B. Beneš. Interactive example-based urban layout synthesis. *ACM Trans. Graph.*, 27(5):1–10, 2008.

[Bet03] E. Bethke. *Game development and production*. Wordware game developer's library. Wordware Pub., 2003.

[CEW+08] G. Chen, G. Esch, P. Wonka, P. Müller, and E. Zhang. Interactive procedural street modeling. *ACM Trans. Graph.*, 27(3):1–10, 2008.

[FF62] L. Ford and D. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[Flo03] M. S. Floater. Mean value coordinates. *Computer Aided Geometric Design*, 20(1):19–27, 2003.

[Geb08] G. Gebhart. Automatisches Generieren von 3D-Straßensystemen. Master's thesis, Vienna University of Technology, March 2008.

[Goo10a] Google, Inc. Google 3D warehouse, sketchup.google.com/3dwarehouse, 2010.

[Goo10b] Google, Inc. Google earth, earth.google.com, 2010.

[Gra72] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Inf. Process. Lett.*, 1(4):132–133, 1972.

[HB04] D. Hearn and M.P. Baker. *Computer Graphics with OpenGL, 3rd Edition*. Pearson Education, Inc., Upper Saddle River, NJ 07458, 2004.

[Hum01] G. Hummel. Modellierung von Straßen für Echtzeitvisualisierung, 2001.

[JRH03] C. Jaynes, E. Riseman, and A. Hanson. Recognition and reconstruction of buildings from multiple aerial images. *Comput. Vis. Image Underst.*, 90(1):68–98, 2003.

[Kjø00] K. A. H. Kjølaas. Automatic furniture population of large architectural models. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, January 2000.

[KM07] G. Kelly and H. McCabe. Citygen: An interactive system for procedural city generation. In *Fifth International Conference on Game Design and Technology*, pages 8–16, 2007.

[KSE+03] V. Kwatra, A. Schödl, I.A. Essa, G. Turk, and A. F. Bobick. Graphcut textures: image and video synthesis using graph cuts. *ACM Trans. Graph.*, 22(3):277–286, 2003.

[LSWW10] M. Lipp, D. Scherzer, P. Wonka, and M. Wimmer. Interactive modeling of city layouts using layers of procedural content. 2010. Submitted.

[LWW08] M. Lipp, P. Wonka, and M. Wimmer. Interactive visual editing of grammars for procedural architecture. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–10, New York, NY, USA, 2008. ACM.

[Mic10] Microsoft Corporation. XNA Framework, www.xna.com, 2010.

[MP96] R. Měch and P. Prusinkiewicz. Visual models of plants interacting with their environment. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 397–410, NY, USA, 1996. ACM.

[MST89] A. E. Middleditch, T. W. Stacey, and S. B. Tor. Intersection algorithms for lines and circles. *ACM Trans. Graph.*, 8(1):25–40, 1989.

[MWH+06] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. Van Gool. Procedural modeling of buildings. *ACM Trans. Graph.*, 25(3):614–623, 2006.

[PL91] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants*. Springer Verlag, 1991.

[PM01] Y. I. H. Parish and P. Müller. Procedural modeling of cities. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 301–308, August 2001.

[Pro10] Procedural Inc. CityEngine, www.procedural.com, 2010.

[Riv10] R Rivera. Boost libraries, www.boost.org, 2010.

[Sti80] G. Stiny. Introduction to shape and shape grammars. *Environment and Planning B*, 7(3):343–351, 1980.

[Sti82] G. Stiny. Spatial relations and grammars. *Environment and Planning B: Planning and Design*, 9(1):113–114, January 1982.

[SV04] I. Suveg and G. Vosselman. Reconstruction of 3D building models from aerial images and maps. *ISPRS Journal of Photogrammetry and Remote Sensing*, 58(3-4):202 – 224, 2004.

[VAW+10] C. Vanegas, D. Aliaga, P. Wonka, P. Müller, P. Waddell, and B. Watson. Modeling the appearance and behavior of urban spaces. *Computer Graphics Forum*, 29(1):25–42, 2010.

[Wei10] E. W. Weisstein. Point-line distance 3-dimensional., mathworld.wolfram.com/point-linedistance3-dimensional.html, 2010.

[WMWG09] B. Weber, P. Müller, P. Wonka, and M. Gross. Interactive geometric simulation of 4D cities. *Computer Graphics Forum*, 28(2):481–492, April 2009.

[WWSR03] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky. Instant architecture. *ACM Trans. Graph.*, 22(3):669–677, 2003.

[Zim07] M. Zimmermann. Procedural construction of streets. Technical report, ETH Zürich, 2007.

Acknowledgments

I would like to send my most sincere thanks to the following people:

My coworkers at *Team Vienna*, especially *Julian M. Breddy* for giving me valuable insight into the development of urban environments for games.

The *Institute of Computer Graphics* for providing the advisory support for my thesis, especially *Daniel Scherzer*, *Michael Wimmer* and *Markus Lipp*.

Sabine Meyer for proof-reading this thesis.

The *Faculty of Computer Science* for granting me a scholarship to work on this thesis, and especially *Michael Wimmer* and *Werner Purgathofer* for supporting my application for the scholarship.

And finally, *my family*, especially *Irene Kubiska*, who had to endure me during the crunch times while working on this thesis.