

A Declarative Approach to Procedural Generation of Virtual Worlds

R.M. Smelik

About the cover

The rows of icons depicted on the cover illustrate the process of procedural generation as followed in this thesis. The first rows represent the generation of three types of features (a mountain, a tree and a building), from coarse user specification to a detailed geometric shape. The last row represents the crucial final step: the integration of separately generated features to form a consistent virtual world.

A Declarative Approach to Procedural Generation of Virtual Worlds

Proefschrift

ter verkrijging van de graad doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op woensdag 30 november 2011 om 10:00 uur

door

Ruben Michaël SMELIK

ingenieur technische informatica
geboren te Haarlem.

Dit proefschrift is goedgekeurd door de promotor:
Prof. dr. ir. F.W. Jansen

Copromotor:
Dr. ir. R. Bidarra

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof. dr. ir. F.W. Jansen	Technische Universiteit Delft, promotor
Dr. ir. R. Bidarra	Technische Universiteit Delft, copromotor
Prof. dr. A. van Deursen	Technische Universiteit Delft
Prof. dr. ir. A. Verbraeck	Technische Universiteit Delft
Prof. dr. R.C. Veltkamp	Universiteit Utrecht
Dr. B. Beneš	Purdue University, USA
Dr. ir. J.K. de Kraker	TNO



This research has been supported by the GATE project, funded by the Netherlands Organization for Scientific Research (NWO).

ISBN 978-90-8570-597-0

©2011, Ruben Michaël Smelik, Delft, All rights reserved.

Preface

The research described in this thesis has been performed at the Modelling, Simulation and Gaming Department of TNO in The Hague, in close collaboration with the Computer Graphics and CAD/CAM Group of Delft University of Technology.

It is part of a large national research program entitled *Game Research for Training and Entertainment* (GATE, see <http://gate.gameresearch.nl>). The goal of this program is to advance the state of the art in four different research themes: modelling the virtual world, virtual characters, interacting with the world, and learning with simulated worlds.

Within the GATE program, TNO and Delft University of Technology formulated a work package, consisting of two PhD projects, to research the automatic creation of virtual worlds. Simultaneously with Tim Tutenel (see [Tutenel 12]), I started my PhD project in the summer of 2007, aiming at contributing to this research goal in cooperation. My project was supervised by Rafael Bidarra, Klaas Jan de Kraker and Frido Kuijper.

Two concrete results of the fruitful cooperation can be found in this thesis: in Section 3.1, I have used Tim's semantic library to define the semantics of the objects in the virtual world, and in Section 4.2, we cooperated, together with Ricardo Lopes, to devise a method for the generation of consistent building models.

Ricardo is now continuing our research on virtual worlds at Delft, focussing on creating game worlds that automatically adapt to the player [Lopes 11].

Contents

Preface	v
1 Introduction	1
1.1 Procedural generation of virtual worlds	3
1.2 Problem statement	4
1.3 Research methodology	5
1.4 Research contributions	6
1.5 Framework overview	6
1.6 Scope of virtual worlds	7
1.7 Applications of the research	8
1.8 Thesis outline	8
1.9 Related publications	10
2 State of the art in virtual world generation	11
2.1 Procedural methods	12
2.1.1 Landscape	12
2.1.2 Rivers, oceans and lakes	14
2.1.3 Plant models and vegetation distribution	15
2.1.4 Road networks	17
2.1.5 Urban environments	19
2.2 Commercially available tools	23
2.3 Discussion	25
3 A semantic model for virtual worlds	27
3.1 Semantic modelling approach	28
3.2 Levels of abstraction in terrain features	30
3.2.1 Specification level	30
3.2.2 Structure level	31
3.2.3 Semantic object level	32
3.2.4 Geometry level	32
3.3 Incorporated terrain features	33
3.4 Layered structure	35
3.5 Discussion	36

4	Integration of procedural methods	37
4.1	Integrating procedural methods for terrain features	39
4.1.1	Interaction between the framework and procedural methods . .	40
4.1.2	Example of an integrated procedure	42
4.1.3	Integration limitations	45
4.2	Integrating procedural methods for semantic objects	45
4.2.1	Semantic moderator	47
4.2.2	Integration of procedures	49
4.2.3	Plan execution	50
4.2.4	Villa Neos: an example of a consistent building	51
4.3	Discussion	53
5	Virtual world consistency maintenance	55
5.1	Motivation for consistency maintenance	56
5.2	Basic notions	57
5.2.1	Landscape - feature interactions	58
5.2.2	Feature - feature interactions	59
5.3	Handling landscape - feature interactions	59
5.4	Handling feature - feature interactions	60
5.5	Example interaction scenario revisited	63
5.6	Discussion	63
6	User control in procedural modelling	67
6.1	Levels of modelling granularity	68
6.2	Declaring and maintaining high-level intent	70
6.2.1	Composing semantic constraints	71
6.2.2	Constraint evaluation method	72
6.3	Declaring the features of the virtual world	74
6.3.1	Procedural sketching	75
6.3.2	Iterative workflow	76
6.4	Refining feature specifications	78
6.5	Balancing user control and consistency maintenance	80
6.5.1	Element locks	80
6.5.2	Transition zone	83
6.6	Discussion	84
7	Prototype design and implementation	89
7.1	Prototype design	89
7.1.1	High-level components and flow	90
7.1.2	Integrated procedural methods	95
7.1.3	Configuration and templates	97
7.1.4	User interface design	98
7.2	Performance considerations	98

7.2.1	Use of GPU computing	99
7.2.2	Efficient data management	100
7.3	Creating the 3D virtual world	101
7.3.1	Generation of the 3D geometric model	101
7.3.2	Scene-graph organization	102
7.3.3	Rendering the virtual world	103
7.3.4	Exporting the virtual world	106
7.4	Discussion	107
8	Example modelling sessions	109
8.1	Modelling session 1: procedural sketching	109
8.1.1	Motivation	109
8.1.2	Walkthrough of modelling session	111
8.2	Modelling session 2: refining intent	112
8.2.1	Motivation	112
8.2.2	Walkthrough of modelling session	113
8.3	Modelling session 3: semantic constraints	115
8.3.1	Motivation	115
8.3.2	Walkthrough of modelling session	115
8.4	Discussion	117
9	Real-world application of the prototype	119
9.1	Case 1: Military training simulators	119
9.1.1	Motivation and objectives	121
9.1.2	Technical realization	121
9.1.3	Results	122
9.2	Case 2: Levee patroller	122
9.2.1	Motivation and objectives	122
9.2.2	Technical realization	123
9.2.3	Results	127
9.3	Case 3: Landscape	128
9.4	Discussion	129
10	Conclusions	131
10.1	Research contributions	131
10.2	Discussion of results	133
10.2.1	Current limitations	134
10.3	Recommendations for future work	134
	Bibliography	137
	Summary	147

Samenvatting	149
Curriculum Vitae	151
Acknowledgements	153

1

Introduction

Since the introduction of Pong, a simple and abstract game resembling tennis, in 1972, computer games have steadily become accepted as mainstream entertainment. People of all ages, races and gender regularly play games in their free time. Games have evolved into a large variety of genres, ranging from casual games, requiring only a couple of minutes of play time, to massive multiplayer online role-playing games, which can easily consume more than 20 hours per week. As a result of the popularity of games, this industry now rivals the film industry in terms of revenue.

Game technologies, both hardware and software, have progressed enormously, because of the competitive advantages gained by providing the cutting edge in gaming: high-resolution graphics, enhanced interactions, impressive environments, etc. When we compare, for instance, one of the earlier 3D games Wolfenstein (1992) to Skyrim (2011) in Figure 1.1, it is obvious that we have come a long way in the last two decades.

Because game technology now supports realistic, interactive environments on low-cost consumer hardware, the technology is also highly suitable to other domains. An example of this are serious games, which are games designed not for pure entertainment, but for learning specific skills, procedures or tactics, or cultural awareness or social change. Other applications of game technology include simulations, architectural renderings and interactive walkthroughs, movies, etc.

Games take place in a game world, generally called a *virtual world*: a 2D or, more frequently, 3D representation of an environment. Although earlier games were situated in abstract game worlds (called game levels), modern games increasingly often offer a highly realistic, albeit fictional outdoor environment. Examples include



Figure 1.1: The evolution of games in visual realism and level of detail: (a) Wolfenstein [id Software 92], (b) The Elder Scrolls V: Skyrim [Bethesda Game Studios 11].

historical urban environments, such as ancient Rome or Venice in Assassin’s Creed, or modern-day New York in the latest instalment of Crysis.

In the development of a game, an enormous amount of time and effort is spent on creating these virtual worlds. Creating a virtual world is a highly creative and iterative process, as the world not just serves as a backdrop for the game, but has a direct impact on the gameplay, experience and even the difficulty. In this process, game designers and artists work together to create a world that meets their functional and aesthetic requirements, while respecting the performance requirements dictated by the game engine and hardware platform. The iterative refinement of a virtual world from a rough sketch to its final polished version can take many months.

Unfortunately, designing a virtual world currently requires not only an artistic or creative effort, but also much routine modelling work. This is a result of the manual modelling methods and tools employed in the industry today. Working with such modelling tools is for the most part laborious and repetitive, and requires specialized 3D modelling skills. As a result, a large portion of the available budget is spent on low-level modelling work, which could have been better utilized to refine and polish the relevant gameplay aspects of the world.

Current modelling tools have another important drawback: the inflexibility of the virtual world models. Once completely constructed, these models are hard to modify by designers. Major changes in the world may result in the designer effectively having to start from scratch, and must therefore be avoided or worked around, hindering designers’ creativity.

As virtual worlds keep expanding each generation, manual modelling techniques are increasingly less able to keep up with their size, richness and detail. Therefore, the need for modelling paradigms that operate on a higher level of abstraction becomes more and more urgent. To alleviate the amount of mundane modelling work, it is natural to consider to employ automated content generation methods.

1.1 Procedural generation of virtual worlds

Procedural generation is an umbrella term for software algorithms that can (semi-) automatically generate a specific type of content (e.g., a 3D model of a tree) based on a limited set of user input parameters. Occasionally they are described as *data amplification* algorithms [Roden 04], as they convert a small amount of input data into a large set of more detailed output data. Procedural generation drastically reduces the amount of modelling effort required to create content. Furthermore, its output is often stochastic, varying somewhat with each run. This aspect can be exploited to create a variety of results using the same set of input parameters; e.g., a set of tree models, all of the same species and age, but each tree with a different branch structure.

Considering these advantages, procedural generation appears to be an attractive alternative to manual modelling that promises a high gain in productivity and a seemingly endless variation in content. It has been an active research topic for over thirty years already. In the domain of virtual worlds, this research has resulted in numerous high-quality procedures, each specific to a feature of the world, such as the landscape (i.e., the bare terrain), rivers, plant models and natural distributions of vegetation in a forest, road networks, the structure of urban environments, building façades and interior layouts (see Chapter 2).

However promising, currently, procedural generation is not directly a suitable alternative to manual modelling. There are three well-known open issues that apply to most procedural generation methods [Smelik 08]:

1. Procedural methods are often configured using a set of *unintuitive* input parameters, which can be hard to grasp and do not always have a clear, predictable effect on the output [Zhou 07, Gain 09].
2. Procedural methods typically provide limited support for *user control* [Esch 07, Šťava 08, Lipp 08]. To some extent, using a procedural method comes down to trial and error. In addition, because the runtime of these algorithms is frequently far from interactive, this process becomes even more cumbersome.
3. Procedural methods are often *specialized*, designed to generate one specific type of content. Integrating this content into a virtual world still involves a large amount of manual effort [Galin 10].

Above described issues explain why there is some reluctance in the industry to employ procedural generation methods. Especially the level of user control is inherently limited for procedural methods, as by definition manual modelling will offer more fine-grained control. On the other hand, such fine-grained control is not always needed; in fact, by offering solely low-level editing facilities, manual modelling of 3D virtual worlds has become quite complex and exceptionally laborious. Therefore, we can conclude that neither approach is in itself satisfactory for modelling the next generation of virtual worlds for games and other applications.

1.2 Problem statement

Considering the urgent need for a more efficient and accessible approach to the creation of virtual worlds, and the dilemma we identified in the previous section, we can now formulate the main research question of this thesis:

How can we improve the process of virtual world generation?

In order to refine our research question, we identify several requirements that need to be fulfilled by any approach intended to improve the current virtual world generation process:

1. The approach should provide a significant *productivity* gain with respect to manual modelling.
2. It should be *accessible* to non-specialist designers, entailing that its interaction method is intuitive and the approach reduces the complexity of modelling virtual worlds.
3. As modelling of virtual worlds is a highly iterative process, the approach should support this way of modelling by, amongst other things, providing a *short feedback loop* between edit action and effect.
4. The approach should offer sufficient *user control*, allowing designers to specify their intent at different levels of abstraction.
5. Using the approach, designers should be able to model complete virtual worlds, which are internally *consistent* (i.e., of which the incorporated features are not in conflict with each other). The burden of maintaining this consistency should not be left to the designers, as this would limit their flexibility and increase the modelling complexity.

To date, no such approach has been feasible, because of the identified open issues of procedural generation, and a lack of knowledge on how to fulfil the above requirements. In this thesis, we have therefore focussed on the following key questions:

1. The operation of a procedural method is complex to grasp without having considerable knowledge of its algorithm. Furthermore, the input parameters are often *unintuitive*, making it difficult to set their values properly, especially since there might be non-obvious dependencies between parameters. Simply put, procedural methods do not match a designer's way of thinking. How can we offer *accessible* user interaction with these procedures?
2. The most important drawback of procedural methods is the general lack of user control. How can we provide sufficient *control* and influence over procedural modelling?

3. As stated in Section 1.1, procedural methods are specialized, in the sense that they generate variations on one specific type of content. This content has then to be automatically combined, merged and adapted to fit into a consistent and plausible virtual world. Furthermore, as a result of each procedural operation on the virtual world model, the model might be in an inconsistent state. How can we ensure and maintain the *consistency* of the virtual world model, of which each feature is generated by a separate procedural technique?
4. Manual edit operations are typically very localized and do not require extensive computational processing. As a result, it is straightforward for manual modelling environments to support an iterative workflow with a short feedback loop. Procedural modelling operations, however, have a far broader scope and do often require a significant amount of computation. Besides, undoing the effects of a procedural operation is far more complex and has more repercussions than undoing a small manual edit. How can we support an *iterative workflow* for procedural modelling?

1.3 Research methodology

The methodology we have followed in our research is to validate our methods and results in practice, using prototype development and testing on the basis of modelling cases. We have implemented all research results in a single prototype modelling system, and used this prototype to obtain feedback on the results at specific milestones in our project. For this, we invited a diverse group of people to experiment with our tool and provide in-depth feedback on the usefulness of the declarative approach in general, the accessibility of the interaction methods, the quality of the generated results, and whether the amount of user control they experienced was sufficient for their typical modelling tasks. The group of people included game designers, graphical artists, game producers, researchers and students, and also people with no 3D modelling experience.

The feedback received was often valuable input for refining and adjusting our methods. To demonstrate the potential of our research, we discuss a number of example modelling sessions in Chapter 8. Furthermore, the prototype is applied in real-world cases, which continue to give us opportunities to receive additional feedback on our research (see Chapter 9). At some point, it would be important to also perform a formal user study and to accurately compare the research to existing industrial modelling methods. However, a proper formal user study leading to relevant and important results would require a higher level of maturity and general applicability of the research and prototype.

1.4 Research contributions

To address the identified problems, our main contribution to the field of procedural generation is an approach named *declarative modelling of virtual worlds*. This approach aims at improving the efficiency of designers, by allowing them to express their design intent more directly and at a higher level of abstraction. In other words, it lets designers concentrate on *what* they want to create instead of on *how* they should model it.

To realize the declarative modelling of virtual worlds, we have devised a *framework*, building upon established results on parametrized procedural generation, constraint solving and semantic modelling. The goal of our virtual world modelling framework is to enable designers to state their intent using simple, high-level constructs, which are then automatically translated into a matching 3D virtual world. The consistency of the virtual world is maintained using a semantically rich model of all its features and their relations.

Our research has the following key contributions:

1. a semantically rich model for virtual worlds (Chapter 3);
2. a structured method for integrating procedural techniques into a common framework, allowing them to be used in any combination (Chapter 4);
3. automatic consistency maintenance through generic methods for resolving interactions between the features in the virtual world (Chapter 5);
4. intuitive and accessible user interaction methods with user control at various levels of granularity (Chapter 6).

Besides, our framework has been implemented in a prototype modelling environment, called *SketchaWorld* (Chapter 7). The prototype results demonstrate the feasibility of declarative modelling of virtual worlds (Chapter 8), and SketchaWorld already been applied to several real-world cases (Chapter 9).

1.5 Framework overview

Figure 1.2 shows an overview of our framework for declarative modelling of virtual worlds. It depicts the role of the different research contributions incorporated in the framework, and the relation between them.

The semantic model for virtual worlds (1) provides a foundation for the framework. By providing the structured integration of individual procedural algorithms and techniques (2), the framework allows designers to generate the features that make up the virtual world; these features are in turn consistently integrated and maintained (3) as part of our semantic model. Intuitive interaction and user control are important requirements for our approach (4), and for this, the virtual world modelling framework provides user control at several levels of granularity.

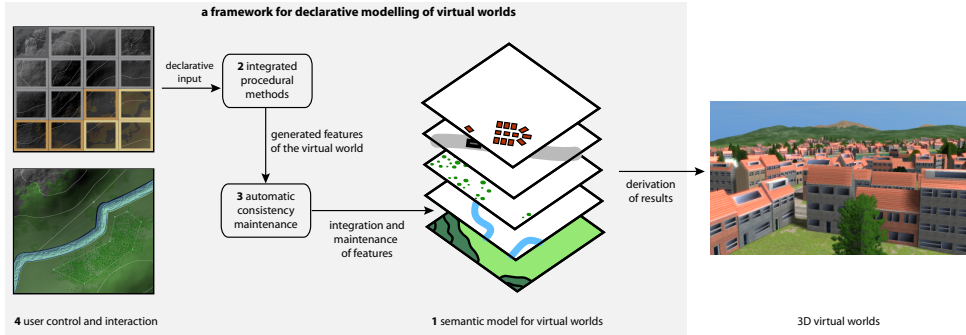


Figure 1.2: Overview of our framework for declarative modelling of virtual worlds, with research contributions indicated.

1.6 Scope of virtual worlds

In this thesis, we focus on the automatic creation of *geo-typical* virtual worlds. In contrast to *geo-specific* virtual worlds, *geo-typical* worlds imply no direct correspondence to a particular geographic location. However, they are very much like the real world, in the sense that they provide an environment that is both visually and functionally convincing. Geo-typical virtual worlds can match a specific region of the world (e.g., Western Europe, Middle East), by including only elements and arrangements typically found in that region. Furthermore, they can be set in a present, past or even a fantasy or futuristic setting.

Our choice for geo-typical worlds is based on the fact that they are often employed in both training and entertainment games. Entertainment game worlds are designed with the goal of providing a specific gameplay experience, and, as such do not strictly have to conform to any constraints imposed by a geo-specific correspondence. Even virtual worlds in games intended to represent real-world locations take such constraints liberally, and are more inspired by the locations than directly based on geographic information. As a result, typical virtual worlds in games are a good fit for our framework. An exception are abstract games, which, although they may be suitable for procedural generation, do not benefit much from our framework based on realistic features and semantics.

Similar reasoning holds for virtual worlds in training games. While entertainment games focus on gameplay experience, virtual worlds for training are designed with learning objectives in mind. As such, geo-typical worlds are a suitable match, as they can be designed to precisely fit these objectives. Even though we see the above two fields as the primary area of application of the research, the scope could be extended to virtual worlds used other applications, such as movies, architectural design, etc.

A further refinement of our scope is on the elements that we consider to be part

of the virtual world. We include many natural and man-made objects in a world, ranging from mountains to furniture, but exclude population (e.g., virtual characters, vehicles, creatures) or gameplay elements (e.g., events, area triggers, items, objectives). This is not to say that such game elements could not profit from the rich semantics encapsulated in the virtual world model.

Although our main focus is on geo-typical worlds, we also experimented with geo-specific aspects. In particular, we investigated how coarse geographic data can be incorporated as an inspirational basis for virtual world design (see Chapter 9).

1.7 Applications of the research

There are many possible practical applications of declarative modelling of virtual worlds. First, the approach can be applied throughout many of the phases of the game development process:

- During concept development, to inexpensively explore all kinds of virtual worlds and scenarios;
- For early testing of gameplay aspects before any actual game world has been designed;
- For rapid prototyping of virtual worlds, possibly resulting in a faster convergence of game world design;
- For creating the basis for the actual game world, possibly further refined using traditional methods.

However, our approach is not solely accessible to specialist game designers. Because of its easy to use interaction method, we believe that a much wider range of end users could benefit from it. For *commercial games*, virtual worlds in games are currently often predefined by the game developer, because of, amongst other things, the complexity of the tools used to create these worlds. However, using our approach, game modification enthusiasts might be able to more easily create new environments for their favourite game. For *serious games* in the training and instruction domain, our approach can help training instructors to create virtual worlds that better match their training scenarios. Taking it even further, as our virtual world modelling framework requires no special knowledge of 3D modelling of virtual worlds, basically anyone should be able to use it to create the worlds he or she imagines.

1.8 Thesis outline

Figure 1.3 presents an outline of this thesis, including the flow of chapters. The current issues with virtual world generation leads to our approach: declarative modelling

of virtual worlds. To realize this approach, existing procedural methods, surveyed in Chapter 2, are integrated in a common framework described in Chapter 4, which uses a semantic model for virtual worlds, defined in Chapter 3, as its basis. In the context of this framework, we particularly elaborate on automatic consistency maintenance of the virtual world model, as explained in Chapter 5, and user control through interactive methods, presented in Chapter 6. The framework is implemented in a prototype, named SketchaWorld, which is described in Chapter 7. The results of the prototype and its application to several real-world cases are discussed in Chapter 8 and Chapter 9, and conclusions with recommendations for further research are presented in the final Chapter 10.

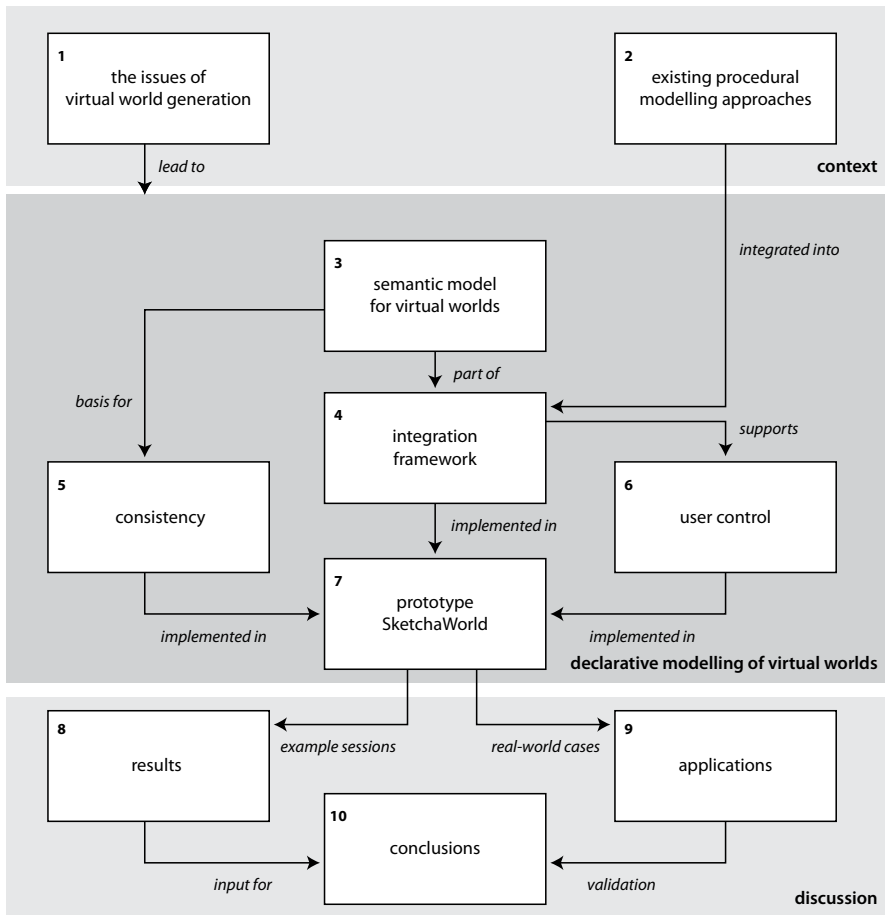


Figure 1.3: A visual overview of the structure of this thesis.

1.9 Related publications

Parts of this thesis were previously published as follows:

- Chapter 1: The motivation for the research was previously published as a proposal in [Smelik 08].
- Chapter 2: This chapter is an extended and updated version of our previous publication [Smelik 09a].
- Chapter 3: Section 3.1 is based on research by Tim Tutenel, which has been published in, amongst others, [Tutenel 10].
- Chapter 4: Section 4.2 results from joint research with Tim Tutenel and Ricardo Lopes, published as [Tutenel 11].
- Chapter 5: The consistency maintenance methods were previously published in [Smelik 11b].
- Chapter 6: The individual sections were based on material previously published as [Smelik 11a, Smelik 10c, Smelik 10b].
- Chapter 7: The implementation of some of our procedural methods was described in [Smelik 09b].
- Chapter 8: A modelling session similar to the one described in Section 8.1 was presented in [Kuijper 11].
- Chapter 9: Section 9.1 was more extensively described in [Smelik 10a].

2

State of the art in virtual world generation

This chapter gives an overview of the state of the art of procedural generation of virtual worlds. It discusses the relevant research and techniques as well as several notable commercial tools, serving two purposes:

1. Discussing the method of operation of procedural generation methods gives an idea of how these methods typically work. This is helpful for understanding the remainder of this thesis, as part of the current research is integrated into our framework;
2. By surveying the existing work, we obtain a more clear vision of its potential and current limitations.

In the Introduction, we already briefly touched upon the main open research issues of automatic creation of virtual worlds. This chapter explains the current situation in more detail, providing many examples of research of recent years. This gives a more accurate notion of the extent of the problems described in the Introduction.

The outline of this chapter is as follows. First, we survey procedural generation methods aimed at generating specific features of the virtual world, describing the prominent traditional methods and treating recent extensions made to procedures to increase their usability. Next, a number of commercially available procedural generation tools are discussed. Finally, we identify the main open issues we derive from this chapter, and relate these to our research contributions.

2.1 Procedural methods

Procedural modelling has been an active research topic for over thirty years, resulting in high-quality procedures for specific terrain features. This section surveys important procedural methods for generating the natural and man-made features of a virtual world (this section is in part based on [Smelik 09a]). This is followed by a discussion on extensions of these methods with regards to user control, modelling interactivity and integration of results. Note that we focus our discussion on methods for generating *features of the virtual world*, and thereby exclude other procedural content generation techniques, such as textures, sound effects, music, or very specific types of game levels, such as dungeons for role playing games or map generators for real-time strategy games. Although these are also examples of successful procedural approaches, excluding them here helps us to concentrate on the problem of automatic creation of 3D virtual worlds.

2.1.1 Landscape

The landscape (i.e., the bare terrain) is typically modelled on the basis of a height-map. A height-map is a 2D grid, where the value in each cell represents the elevation at that specific location. Because it is straightforward to map this structure to a 3D regular mesh, height-maps are often used as the basis of a virtual world model. Procedural generation of height-maps is one of the first topics explored at the inception of procedural modelling research, back in the 1980's. Today, there are many procedural algorithms for creating height-maps.

Among the earlier algorithms are the subdivision methods. A coarse height-map is iteratively subdivided, each iteration introducing a constrained amount of randomness to generate details. One of these subdivision algorithms is known as the mid-point displacement method, in which a new point's elevation is set to the average of its corners in a triangle or diamond shape plus a random offset [Miller 86]. The offset's range decreases at each iteration according to a parameter that controls the roughness of the resulting height-map.

Another class of methods for height-map generation is based on fractal noise generators [Mandelbrot 82, Fournier 82, Voss 85], such as Perlin noise ([Perlin 85, Perlin 02]), which generates noise by mapping each point in the height-map to a sampling point in a grid of random vectors, from which a noise value is derived using an interpolation scheme. Scaling and summing several layers of noise of increasing frequency into a height-map results in natural, mountainous-like structures. For a recommended textbook on fractal noise and height-map generation, see [Ebert 03].

Height-maps can be further transformed using common signal processing filters (e.g., smoothing) or simulations of physical phenomena, such as erosion. Thermal erosion diminishes sharp changes in elevation, by iteratively distributing material from higher to lower points, until the talus angle (i.e., maximum angle of stability for a material such as rock or sand), is reached. Erosion caused by rainfall (fluvial

erosion) can be simulated using, for example, cellular automata, where the amount of water and dissolved material that flows out to other cells is calculated based on the local slope of the elevation profile. Musgrave et al. treat both types of erosion [Musgrave 89, Musgrave 93], and Olsen discusses several speed optimizations with reduced but acceptable output quality [Olsen 04]. As an alternative to height-maps, Beneš and Forsbach introduce a structure more suited for realistic erosion algorithms [Beneš 01]. Their landscape model consists of stacked horizontal slices of material, each having an elevation value and material properties, e.g., density. This model is a trade-off between the limited but efficient height-map structure and a full voxel model. The model also allows for air layers, thereby it supports cave structures.

Noise-based height-map generation delivers results that are fairly random; users control the outcome only on a global level, often using unintuitive parameters. Several researchers have addressed this issue. Their methods vary in the type and degree of interactivity and level of control, from coarse to fine-grained.

Some of the proposed extensions provide a way to constrain the generation process in a non-interactive manner by new forms of user input. Stachniak and Stürzlinger propose a method that integrates constraints expressed as mask images [Stachniak 05]. It employs a search algorithm that finds an acceptable set of deformation operations to apply to a procedurally generated landscape in order to obtain a landscape that conforms to these constraints. However, this method is computationally expensive and far from interactive. Zhou et al. describe a technique that generates a height-map based on an example input height-map and a user line drawing that defines the occurrence of large-scale curved line features, such as mountain ridges [Zhou 07]. Features are extracted from the example height-map, matched to these curves and seamed in the resulting height-map. The resulting height-maps are convincing and have plausible transitions. Doran and Parberry propose a different constraint-based approach using agents, each creating a specific landform (e.g., coastline, beach, mountain) [Doran 10]. Although the method allows one to control the frequency of specific landforms, it does not offer any direct control on where they occur.

Saunders proposes a method that synthesizes a height-map based on Digital Elevation Models (DEM) of real-world terrain [Saunders 06]. A user draws a 2D map of polygonal regions, each of which is marked to have a certain elevation profile. The straight boundaries of the regions are perturbed and rasterized in a grid. A height-map is instantiated using a genetic algorithm, which selects DEM data that matches the requested elevation profile in each region. However, the generated transitions at the boundaries between regions are still rather abrupt.

Kamal et al. present a constrained mid-point displacement algorithm that creates a single mountain according to such properties as elevation and base spread [Kamal 07]. Belhadj introduces a more general system where a set of known elevation values constrain the mid-point displacement process [Belhadj 07]. Possible applications are interpolation of coarse or incomplete DEM's or user line sketches.

With the evolution of the Graphics Processing Unit (GPU) as a device for general

purpose parallel processing, interactive user control in height-map generation has become feasible. Schneider et al. introduce a setup in which the user interactively edits the height-map by painting greyscale images, which are used as the base functions of their noise generator [Schneider 06]. Using an efficient GPU-based hydraulic erosion algorithm, Štáva et al. propose an interactive way for users to modify landscape using several types of hydraulic erosion [Štáva 08]. To provide users with more control over the exact appearance of mountain ranges, Gain et al. introduce a sketch-based height-map generation method in which users sketch the silhouette and bounds of a mountain in a 3D interface, and the generator creates a matching mountain using noise propagation [Gain 09]. Using diffusion equations, Hnaidi et al. allow a designer to draw 3D curves that control the shape of the generated landscape [Hnaidi 10]. As a follow-up, Bernhardt et al. present an efficient CPU/GPU setup to present real-time feedback to designers using this method [Bernhardt 11]. Even more fine-grained control over the shape of mountains is provided by the interactive procedural brushing system introduced by de Carpentier and Bidarra. These GPU-based procedural brushes allow users to interactively sculpt a landscape in 3D using several types of noise [de Carpentier 09].

Besides interactive editing, the GPU is nowadays also applied for improving the efficiency of erosion simulations. While these algorithms add much to the believability of mountainous landscapes, they are also notoriously slow, having to run for hundreds to thousands of iterations. Promising examples of porting the algorithms to GPU include [Anh 07], [Štáva 08], and, more recently [Vanek 11].

An inherent limitation of height-maps is that they do not support rock overhangs and caves. Gamito and Musgrave propose a terrain model warping system that results in regular, somewhat artificial, overhangs [Gamito 01]. A more recent method by Peytavie et al. provides a more elaborate model with different material layers that supports rocks, arches, overhangs and caves. Their resulting 3D landscapes are visually plausible and natural [Peytavie 09].

2.1.2 Rivers, oceans and lakes

The topic of procedural generation of water bodies is somewhat under-addressed in the literature. However, several authors have proposed algorithms for generating rivers. Typical strategies for generating rivers can be divided into two categories: generate a river network as part of a height-map generation algorithm, or as a post-processing step on an existing height-map. For the former, a generated river network forms a basis from which a height-map is inferred. For the latter, a height-map is analysed to find potential stream routes from mountains into valleys.

Kelley et al. generate a river network as the basis of a height-map [Kelley 88]. They start with a single straight river and recursively subdivide it, resulting in a stream network. This network forms a skeleton for the height-map, which is filled using a scattered data interpolation function. The climate type and the soil material influence

the shape of the stream network.

Prusinkiewicz and Hammel combine the generation of a curved river with a height-map subdivision scheme [Prusinkiewicz 93]. On the river's starting triangle, one edge is marked as the entry and one as the exit of the river. In a subdivision step, the triangle is divided into smaller triangles, and the river's course from entry to exit can now take several alternative forms. The elevation of the triangles containing the river is set to be the sum of the negative displacements of the river on all recursion levels (resulting in a river bed); other triangles are processed using standard midpoint displacement. After eight or more recursions, the resulting river course looks reasonably natural. A downside of the method is that the river is placed at a constant elevation level, and thus carves deep through a mountainous landscape.

A more advanced approach that does not suffer from these limitations, described by Belhadj and Audibert, creates a height-map with mountain ridges combined with river networks [Belhadj 05]. Starting with an empty map, they place pairs of ridge particles at a particular high elevation and move them in opposite directions in several discrete steps. A Gaussian curve is drawn on the height-map along the particle positions of each iteration. Next, they place river particles along the top of the mountain ridge and let them flow downwards according to simple physics, comparable to hydraulic erosion. The remaining points in between ridges and rivers are filled with an inverse midpoint displacement technique. For this specific type of landscape, i.e., steep mountain ridges with valleys featuring a dense river network, the method is fast and effective.

The interactive method presented by Huijser et al. offers a very precise way to control a river curve, and define a lateral profile to be swept along this curve, resulting in a 3D geometric representation of the river [Huijser 10].

Except for rivers, procedural water bodies, such as oceans and lakes and their connections, stream networks, deltas and waterfalls, have received little attention to date. The forming of lakes is not considered at all. Oceans are commonly generated setting a fixed water level (e.g., 0 meter) or by starting a flooding algorithm from points of low elevation. Teoh also states that the research in this area is incomplete: several river and coastal features have not been addressed [Teoh 08]. He proposes fast and simple algorithms for river meandering, deltas and beach forming.

2.1.3 Plant models and vegetation distribution

Similarly to height-map generation, procedural vegetation is a classic research topic in the field of procedural modelling. It includes both procedures for generating 3D tree and plant models and methods for automatic placement of vegetation on a given landscape. The former can be used to quickly obtain a set of similar but varying plant models of the same species; the latter saves designers the laborious task of manually placing all these individual vegetation models to form e.g., a large forest.

Procedural plant models grow, starting from the root, adding increasingly smaller branches and ending with the leaves. They are typically generated on the basis of a

rewriting grammar, where a start symbol or shape is iteratively enriched by applying production rules, i.e., rules that replace an input symbol or shape by a string of new symbols or shapes. The Lindenmayer-system, or *L-system*, is a classical and often used example of such a rewriting system. Although L-systems rewrite strings of text, the resulting set of symbols can be interpreted in 2D and 3D by means of turtle graphics. L-systems have been successfully used to generate a wide variety of plant species [Prusinkiewicz 01]. Many extensions of L-systems exist, for instance *constrained L-systems* can restrict the growth of a procedural plant to a 3D bounding shape. Generated branches intersecting the bounds are simply pruned. This gives designer somewhat coarse control over the plant's final shape [Prusinkiewicz 90].

Defining an L-system that generates a shape that matches with one's intent is a challenging task, as the parallel and sometimes stochastic rule application and its growing nature make it hard to predict its exact outcome. Two new approaches deal with this issue in different ways. The first approach ([Šťava 10]) inverts the issue of designing an L-system, by letting designers provide the desired end result and automatically generating a matching L-system to generate this kind of result. Future work is to extend this method to 3D L-systems. The second approach ([Beneš 11b]) enables designers to draw constraining outlines, called *guides* for a growing L-system and provide connections between these guides. Once an L-system touches the edge of a guide, this communication system triggers the start of another L-system in a linked guide. The method supports a form of interactive modelling, by evaluating edit operations to guides and regenerating the corresponding L-systems accordingly.

Linterman and Deussen propose an alternative system to procedurally model plants, by placing plant components (e.g., a leaf) in a graph [Lintermann 99]. Connected components can be structured in sub-graphs (e.g., a twig). The system traverses this graph, generating and placing instances of the components in an intermediate graph that is used for geometry generation

Deussen et al. describe an ecosystem simulation model to populate an area with vegetation [Deussen 98]. The input of the simulation model is the height-map and a water map, several ecological properties of plant species, such as rate of growth, and, optionally, an initial distribution of plants. Based on this, and taking into account rules for competition for soil, sunlight and water, a distribution of plants inside an area is iteratively determined, running for several minutes. The simulation procedure results in a plausible distribution of plant species.

Another procedure for vegetation placement by Hammes is based on ecosystems [Hammes 01]. He uses elevation data, relative elevation, slope, slope direction and multi-fractal noise to select one of the defined ecosystems. Ground vegetation textures are generated at run-time, depending on the level of detail and the ecosystem. The ecosystem also determines the number of plants per species, which are then placed randomly.

The distribution of vegetation in an urban environment follows specific patterns, either through natural growth and competition, or by managed planting. Beneš et al.

present a system that incorporates the simulation of urban vegetation distribution within the design process of a city [Beneš 11a].

2.1.4 Road networks

Road networks for cities can be generated using a variety of methods, including pattern-based approaches, L-systems, agent simulations and tensor fields. The simplest pattern-based technique is to generate a dense square grid, as e.g., in the work of Greuter et al. [Greuter 03]. Displacement noise can be added to grid points to create a less repetitive network, however, the realism and variety of this technique is inherently limited.

A more elaborate method to create roads is by means of templates, as proposed by Sun et al. [Sun 02]. They observe several frequent patterns in real road networks and aim to reconstruct them. For each pattern, there is a corresponding template: a population-based template (implemented as the Voronoi diagram [Voronoi 08] of population centres), a raster and radial template, or a mixed template. To create the skeleton of the road network, highways are generated first using the pattern templates. Rules are applied to check their validity, e.g., when encountering impassable areas (e.g., oceans), roads are discarded or diverted. Next, high-ways are curved to avoid large elevation gradients. The regions they encompass are filled with a grid of streets. Although these patterns are indeed frequently observable in networks of real cities, their combination as presented in this method still seems somewhat artificial.

Similar to plant models, a road network can be viewed as a growing structure, and is thus a good fit for a rewriting system, such as an L-system. Parish and Müller use an extended L-system to grow a road network [Parish 01]. The L-system is goal-driven; its goals are population density (roads try to connect population centres) and specific road patterns, as for example the raster or the radial pattern. This L-system is extended with rules that have a tendency to connect new proposed roads to existing intersections and rules that check road validity with respect to impassable terrain and elevation constraints. Smaller streets are inserted into the remaining areas using a grid, but this could easily be extended with other patterns, as described in [Sun 02].

Glass et al. describe several experiments of replicating the road structure found in South African informal settlements using a combination of a Voronoi diagram for the major roads with L-systems or regular subdivision with and without displacement noise for the minor roads [Glass 06]. They were reasonably successful in recreating the observed patterns.

In contrast to the grammar- and pattern-based approaches discussed above, Lechner et al. introduce an agent-based approach, in which they divide the city into areas including not only residential, commercial and industrial areas, but also special areas like government buildings, squares, and institutions [Lechner 03]. They place two agents, named the *extender* and the *connector*, at a seed position in the virtual world. The extender searches for unconnected areas in the city. When it finds such an area that is located not too far from the existing road network, it seeks the most suitable

path to connect the area to the network. The connector agent starts from a certain location on the existing network and randomly chooses another spot on the network, within a certain radius. It determines the length of the shortest existing path between the two locations. If the travel time is considered too long, a direct road connection is added to the network. In their follow-up work, the authors extend this method with agents that are responsible for constructing main roads for fast connections through the city, and agents that develop small streets [Lechner 06]. This method gives plausible results, but a disadvantage is its very long running time.

Chen et al. propose interactive modelling of road networks by the use of tensor fields [Chen 08]. They define how to create common road patterns (grid, radial, along a boundary) using tensor fields. A road network is generated from a tensor field, by tracing the streamlines from seed points in the major eigenvector direction until a stopping condition is met. Next, along this traced curve new seed points are placed for tracing streamlines in the perpendicular (minor eigenvector) direction. Users can place new basis tensor fields, such as a radial pattern, smooth the field, or use a brush to locally constrain the field in a specific direction. Noise can be applied to make the road network less regular and thereby more plausible.

Previous methods give a designer little direct control over the trajectory of a generated road. Kelly and McCabe introduce the interactive city editor CityGen, in which a user defines the main roads by placing nodes in the 3D landscape [Kelly 07]. Regions enclosed by these roads can be filled with one of three patterns: Manhattan-style grids, industrial grown roads with dead-ends and organic roads as in e.g., North-American suburbs. McCrae and Singh present a method for converting sketched strokes to 3D roads that are automatically fit to the landscape [McCrae 09]. Their system also creates junctions and viaducts for crossing roads.

In the discussed methods, the influence of the underlying elevation profile is to varying degrees taken into account. Most methods take only basic measures to avoid too steep roads and roads through water bodies. Kelly and McCabe plan the precise path of their main roads between the user set nodes to have an even change in elevation as much as possible [Kelly 07]. An A*-based road generation method proposed by Galin et al. uses an elaborate cost function to encode the influence of slope, water bodies and vegetation on the trajectory of the road [Galin 10].

Still, for rough terrain this measure will not be adequate and the landscape needs to be modified to accommodate for the road. Early work by Amburn et al. already formulated the problem of fitting roads with terrain: on a coarse level, the road follows the elevation profile of the terrain, and on a fine level, the terrain must be modified to match locally with the road embankment profile [Amburn 86]. This specific integration problem was recently addressed by Bruneton and Neyret, who propose a shader-based system for real-time integration of Geographic Information Systems (GIS) vector features, such as road and rivers, into a DEM [Bruneton 08]. They create a road profile displacement texture based on footprint geometry, and integrate the profile by blending this texture with a height-map texture. The discussed

work by Galin et al. extends this by also removing any vegetation along the road [Galin 10].

2.1.5 Urban environments

The topic of procedural urban environments has received much attention in the last decade, starting with the work of Parish and Müller in 2001 ([Parish 01]).

Kelly and McCabe present an elaborate survey of several approaches for generating urban environments [Kelly 06]. A practical overview of the state of the art can be found in [Watson 08].

The common approach for procedurally generating cities is to start from a dense road network and identify the polygonal regions enclosed by streets. Subdivision of these regions results in building lots, for which different subdivision methods exist, see e.g., [Parish 01] or [Kelly 07]. To populate these lots with buildings, either the lot shape is used directly as the footprint of a building, or a building footprint is fitted on the lot. By simply extruding the footprint to a random height, one can generate a city of skyscrapers and office buildings. To obtain more complex and varied building shapes, several rule-based methods have been devised.

Greuter et al. generate office buildings by combining several primitive shapes into a floor plan and extruding these to different heights [Greuter 03]. Parish and Müller start with a rectangular floor plan and apply an L-system to refine the building [Parish 01]. Both approaches are most useful for relatively simple office building models. Coelho proposes an urban modelling process that is based on L-systems as well [Coelho 05]. This method generates a tree-like description of the overall scene structure from external data. L-systems are used to generate detailed building models that emerge from the abstract set of data.

Wonka et al. introduce the concept of a *split grammar*, a formal context-free grammar designed to produce building models [Wonka 03]. A split grammar resembles an L-system. However, whereas L-systems result in a string of symbols that need to be geometrically interpreted, split grammars explicitly associate a geometric shape to each symbol. In split grammars, a specific building style can be acquired by setting an attribute of the start symbol, which is propagated during the rewrite process. Within one building model, the style can differ per floor (e.g., an apartment building with shops on the ground floor). The method focuses mostly on generating coherent and believable façades for relatively simple shaped buildings. Larive and Gaildrat use a similar kind of grammar, called a *wall grammar* [Larive 06]. With this grammar they are able to generate building walls with additional geometric detail, such as balconies.

As a follow-up on split grammars, in 2006, Müller et al. introduced Computer Generated Architecture (CGA) [Müller 06], which is a *shape grammar* [Stiny 71] specifically designed for building façades. Shape grammars have been used and described before, especially in the architectural domain [Koning 81, Cagdas 96, Kwon 03]. Architects have described shape grammars as languages of design, supported by a vocabulary of shape rules. Shape rules are specified as spatial relations, where one or more shapes

on the right hand side of the rule is produced and replaces the symbol on the left hand side (which conditions when the rule can be applied).

The shape grammar of CGA allows for more freedom in modelling, including the possibility of creating roofs and rotated shapes. It typically starts with extruding a building lot polygon into a volumetric shape, which is divided into floors. The resulting façades are further subdivided, through shape rules, into walls, windows and doors. Variation can be created using conditional or stochastic rule application, shape parameters and random number generation.

Although the shape grammars in [Müller 06] can generate visually convincing building models, Finkenzeller and Bender note that they miss semantic information regarding the role of each shape within the complete building [Finkenzeller 08b]. They propose to capture this semantic information in a typed graph. Their workflow consists of three steps. Starting with a rough building outline, a building style graph can be applied to this model. This results in an intermediate semantic graph representation of the building, which can be modified or regenerated with a different style. In the last step, geometry is created based on the intermediate model, and textures are applied, resulting in a complete 3D building. In related work, Finkenzeller presents in more detail the generation of façades and roofs in this system [Finkenzeller 08a].

Yong et al. describe a method to create vernacular-style Southeast Chinese houses using an extended shape grammar [Yong 04]. The grammar is hierarchical and starts at the city level, whereas in other methods a shape grammar is applied to an individual building footprint. The grammar then produces streets, housing blocks, roads, and in further productions houses with components such as gates, windows, walls, and roofs. Through a number of control rules (defining, for instance, component ratio constraints), the validity of the buildings can be asserted. By applying this grammar system, a typical ancient Southeast Chinese town can be generated with plausible results, since the building style of these towns is very rigidly structured.

Müller et al. present a very different approach for constructing building façades [Müller 07]. Their method takes a single image of a façade of a real building as input, and is able to reconstruct a detailed 3D façade model, using a combination of imaging and shape grammar generation.

We can conclude that shape grammars are a versatile and often successfully employed method for automatic creation of building facades. However, defining a suitable shape grammar is complex and requires much experience and in-depth knowledge of its geometry derivation technique. Addressing this, Lipp et al. propose a more accessible shape grammar editing system, in which the effects of new rules are interactively visualized [Lipp 08].

As districts, blocks and parcels are defined by the city's road network, a typical method for a designer to influence the city structure is by manipulating the road network. Kelly and McCabe propose an interactive method to generate secondary roads and house blocks based on the primary roads the user manipulates [Kelly 07]. A similar system by de Villiers and Naicker [de Villiers 06] lets users create a road

network and city blocks using sketch strokes, and interprets a set of sketch gestures that modify the properties of the city blocks (e.g., population size, function). Lipp et al. present two graph merging operations for city road networks [Lipp 11]. The first technique is specialized for locally repairing the road network, after a designer has made a small change to a single road. The second approach merges two road network layers using a graph-cut technique. The first layer contains the part of the network that was changed by the designer, and the second contains a procedurally generated network. By using proper merge priorities for roads in the cut, they are able to merge both layers into one network with plausible transitions. This graph-cut merging technique can also be used to lock a subset of the road network.

Although the above city generation methods give fast and visually attractive results, the cities they generate often lack a realistic structure. New research incorporates existing urban land use theories and models in the generation process. Groenewegen et al. present a method that generates a distribution of different types of districts according to land use models of cities in Western-Europe and North-America [Groenewegen 09]. It takes into account a large number of relevant factors, including the historic core of the city and the attraction certain types of features (hillsides, oceans, rivers) have for e.g., industrial or high-class residential districts. Weber et al. use comparable models for a simulation of expanding cities over time [Weber 09]. Their procedural cities expand by growing road network into nearby available land. Their method is fast (about 1 sec. per simulated year) and interactive, meaning that the user can guide the simulation by changing roads or painting land use values on the landscape.

A dynamic system that combines geometric with behaviour modelling is proposed by Vanegas et al. [Vanegas 09]. Here, users paint statistical variables like employment density, which automatically leads to changes in the population distribution and, thereby, the city geometry.

To create a complete building, both its exterior façade and its interior must be generated. The procedural generation of building floor plans has been the focus of several researchers.

Rau-Chaplin et al. demonstrate that shape grammars can also be employed to generate floor plans [Rau-Chaplin 96]. In this case, shape grammars are used to create a *plan schema* containing basic room units. These individual room units are recognized and grouped to define functional zones like public, private or semi-private spaces. Individual functions are then assigned to each room, which are filled with furniture, by fitting predefined layout tiles from a library of individual room layouts.

On a different direction, Hahn et al. present a subdivision method tailored for generating office buildings on the fly [Hahn 06]. The initial building structure is split up into a number of floors. On each of them, further subdivisions are applied to create a hallway zone and individual rooms. A notable feature of this method is that, at runtime, floors and rooms can be generated or discarded based on the observer's

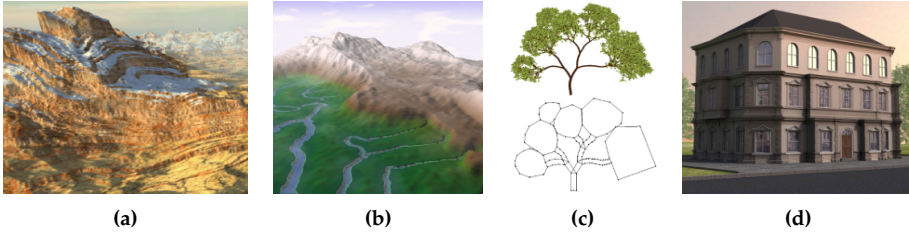


Figure 2.1: Examples of procedural methods: (a) landscape interactively created using procedural brushes [de Carpentier 09], (b) generated river network [Belhadj 05], (c) tree created using guided L-systems [Beneš 11b], (d) complex building façade [Finkenzeller 08b].

position. Re-using the same random seed in this procedure assures that discarded rooms can be properly restored.

Marson and Musse introduce a different room subdivision method, based on *squarified treemaps* [Marson 10]. Starting with a 2D building outline and a set of rooms with desired area and functionality, they recursively subdivide the outline into smaller areas, e.g., building shape, functional zones, rooms. In a post-processing step, corridors are automatically created to connect unreachable rooms.

Instead of starting with a building outline and rewriting or subdividing this space into rooms, Martin first composes a graph of the connectivity of individual rooms in a building, before transforming this graph into the spatial layout [Martin 06]. In this building graph, nodes represent the rooms and edges correspond to connections between rooms (e.g., a door). Public, private and stick-on rooms (e.g., closets, pantries) are gradually added to the graph by a user-defined grammar. This graph is transformed to a spatial layout, and for each node, a specific amount of “pressure” is applied to make the room expand to the desired size. Lopes et al. also propose an expansion-based method, which grows rooms in a geometric grid representing the building lot [Lopes 10]. The initial placement of room seeds is determined by a constraint solving algorithm that takes room adjacencies, connectivity and functional zones into account.

Tutenel et al. applied a generic semantic layout solving approach to expansion-based floor plan generation [Tutenel 09a]. In this approach, every type of room is mapped to a class in a semantic library and for each of these classes relationships can be defined. In this context, relationships will define room-to-room adjacency. In addition, other constraints can be defined as well, e.g., place the kitchen next to the garden, or the garage next to the street. For each room to be placed, a rectangle of minimum size is positioned at a location where all defined relation constraints hold, and all these rooms expand until they touch each other.

Charman gives an overview of constraint solving techniques that can be applied to room layout generation, if seen as a space planning problem [Charman 93]. The

proposed planner works on the basis of axis-aligned 2D rectangles with variable position, orientation and dimension parameters, for which users can express geometric constraints, possibly combined with logical and numerical operators.

More recently, Merrel et al. proposed a method for generating residential building layouts [Merrell 10]. Although the method eventually generates 3D buildings, its main focus is on floor plan generation. The authors use a Bayesian network, trained with real-world data, to expand a set of high-level requirements (e.g., number of rooms) into a complete architectural program (e.g., room adjacencies, area and aspect ratio). These architectural programs are then realized into the 2D shapes of the floor plans, through stochastic optimization over the space of possible building layouts. 3D models are generated from different style templates to fit the structure of the floor plan, including external windows, doors and roofs.

2.2 Commercially available tools

Similar to the procedural research methods we reviewed, commercial procedural tools often focus on a specific feature to generate. This section reviews a number of notable commercially available tools that employ procedural generation.

Numerous procedural tools exist for generating height-maps. From this large selection, we review three tools that have been around for several years: TerraGen, GeoControl and L3DT. We selected these because they have a wide user base and advanced editing capabilities.

TerraGen uses an elaborate network of nodes, where each node maps to an operation, such as noise generation, a filter or a mathematical function [Planetside 11]. A designer composes and configures the network in such a way that it generates the desired elevation profile. TerraGen delivers very impressive visuals, which have been used in several movies. However, to be able to use this tool effectively, background knowledge on mathematics and noise generation, and extensive experience with the tool is needed. Therefore it is most suitable for designers with extensive technical expertise, focussing on creating aesthetically pleasing landscapes.

GeoControl is a height-map editor that iteratively generates elevation data using a specialized subdivision algorithm [Rosenberg 11]. The process starts with a very coarse height-map and subdivides this using a fractal noise algorithm until the desired height-map dimensions are reached. Designers define the noise characteristics to be used in each subdivision step. Additionally, filters, such as erosion or smoothing, can be applied on top of this basic noise algorithm.

One feature of GeoControl is the *isoline*. Users define an isoline by setting the elevation value and the noise characteristics of the transition zone around the line. A mountain ridge with these properties is generated along this line that blends in with the existing height-map. GeoControl's isolines can, with practice, be used to draw height profiles that adequately match designers intent. Still, the modelling process of

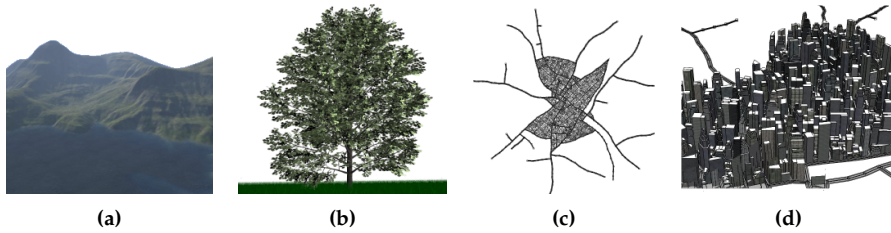


Figure 2.2: Examples of procedural tools: (a) A 3D render of a height-map generated using L3DT [Bundysoft 11], (b) A plant model generated using XFrog [Greenworks 11], (c) and (d) A road network and corresponding city generated using CityEngine [Procedural 11].

this tool can be quite complex and the quality of the results depends on knowledge of the effect of parameters and the dependencies between generation steps.

L3DT allows a user to design a height-map by drawing on a grid map using a brush [Bundysoft 11]. This brush actually consists of a set of generation parameters that are set by the user. These include the elevation, the amount of erosion, the roughness of the terrain, whether it is a source of water, and a climate profile. Each grid cell in the design map is automatically expanded to 64×64 height-map points in the resulting height-map by applying noise, erosion and water flooding algorithms. Climate profiles are used for generating a large texture that is draped on the height-map, by specifying, for each type of material (e.g., grass, rock) the conditions under which it can occur (e.g., elevation range, slope range, water level). After the height-map is generated, a scoring mechanism determines the placement of materials based on the climate profile. The resulting landscape texture looks very convincing, resembling a satellite image. From the mentioned tools, L3DT offers in our view the most accessible interaction method. However, again this tool is limited to generating height-maps and corresponding textures.

Another feature for which a number of successful tools have been developed is vegetation. XFrog is a procedural plant modeller [Greenworks 11], which is based on the previously discussed research [Lintermann 99]. SpeedTree is a commercial middleware package for modelling and rendering of large amounts of detailed vegetation, and is incorporated in many popular game engines [IDV 11].

The last feature category for which several commercial tools have been successfully launched in the last five years is procedural urban environments. CityEngine is a city generator based on the CGA shape grammar [Procedural 11]. The tool can be used to generate geo-typical urban environments, as well as geo-specific city models through a GIS data import. UrbanPad is an urban modelling tool where designers use a node-based interface to create rule templates for generating buildings [Gamr7 11]. In contrast to these two tools for urban environments, CityScape allows for interactive modelling using a mix of manual and procedural modelling, although it provides a

somewhat narrowly focussed set of procedural operations [PixelActive 11].

An interesting and generic approach to procedural modelling is provided by the Houdini tools [Side Effects Software 11]. With Houdini, designers create procedural generators out of small procedural building blocks. These building blocks are often basic mathematical or geometric operations. A visual editor is used to compose the individual building blocks and to connect their in- and outputs. A graph of primitive operations can be encapsulated into a single operation node, which allows designers to create reusable, high-level operations. Houdini is a versatile tool for creating new procedural methods in a visual way; to be used effectively, however, one must have advanced knowledge on how to design such a procedure.

2.3 Discussion

This chapter discussed procedural methods aimed at generating content for 3D virtual worlds. Procedural methods have a long history, which has delivered many high-quality results and continues to be an active research field. However, contrary to what one could expect, in practice their application is still very limited. Commercial tools that use procedural methods do exist, and in some domains they are even quite successful. Nevertheless, they are restricted to one particular type of feature and can be complex in use, requiring in-depth knowledge of the procedural technique used internally. All in all, we can say that the potential of procedural generation is not yet exploited to its fullest.

Fortunately, in recent years, the direction in procedural generation research is shifting towards user controllable and interactive procedures, thereby successfully addressing some of the problems of traditional procedural methods. We revisit the open issues of procedural generation, stated in Section 1.1, to discuss to what extent they have been addressed in recent research, and what our contributions are.

To effectively use a procedural method, one first has to obtain considerable knowledge of its inner workings. In particular, the input parameters of procedures are often *unintuitive*, and do not match a designer's way of thinking. A classic example is generating a height-map on the basis of Perlin noise: the input parameters of the procedure, such as the amplitude and persistence factor, are tied to the working of the algorithm, instead of directly related to the end-result. In recent years, newly proposed methods sometimes offer alternative and more intuitive input, such as sketch strokes ([McCrae 09]) and visual editing of production rules ([Lipp 08]). In this thesis, we substitute the unintuitive procedure parameters with sketched feature outlines and result-oriented attributes (Chapter 6).

Most of the traditional procedural methods we discussed more or less suffer from a lack of *user control*. Continuing with the height-map generation example, although the parameters influence the statistical properties of the resulting landscape, there is no way to control precisely where mountains or valleys are generated. As another example, the discussed river generation procedures ([Kelley 88, Prusinkiewicz 93,

Belhadj 05]) offer a designer no means to steer the course of the rivers. Rule-based procedural methods, such as L-systems and grammars, provide only *indirect* control: to alter the generated result, one has to modify the rule set used to create it. As procedural methods can be far from interactive, the modelling process becomes even more cumbersome.

In the last four years, much progress has been made in improving user control for specific features and methods. Often, more control can be offered by providing designers with *interactive* modelling. In Section 2.1, we already discussed a number of noteworthy examples. For height-map generation, there are several novel methods with improved user control, especially procedural brushes ([de Carpentier 09]), terrain sketching ([Gain 09]), and parameterized curves ([Hnaidi 10]). The work of Beneš et al. makes the definition and the execution of L-systems more controllable and accessible ([Štáva 10, Beneš 11b]). For road networks of cities, the intuitive modelling method of Chen et al. helps one to quickly define the desired road patterns ([Chen 08]), and the merging operations of Lipp et al. allow for fine-grained edit operations on such road networks ([Lipp 11]).

These methods have clearly contributed to the applicability of procedural generation research. However, they are primarily designed to improve user control for one specific procedural technique or type of feature. In this thesis, we will address the lack of user control by introducing several levels of granularity at which designers can influence the generation of complete virtual worlds. The edit operations on these levels, such as procedural sketching, are interactive, support iterative modelling, and aim to provide a suitable balance between user control and productivity (Chapter 6).

Procedural methods are specialized to generate one specific type of content. Fitting all generated content together into a complete virtual world involves a large amount of manual effort. Hardly any attention has been given to the integration of separate procedural methods into a virtual world modelling framework. This thesis presents a structured method for the integration of procedural modelling research (Chapter 4).

Once integrated, the consistency of all generated features of the virtual world has to be maintained during subsequent modelling operations. Specific for roads, methods have been proposed for generating road embankments ([Bruneton 08]), and the construction of bridges or tunnels to cross bodies of water ([Galin 10]). We present *generic* consistency maintenance methods to automatically handle interactions that occur between features throughout the modelling process (Chapter 5).

User control in procedural generation continues to be a challenging research topic. In particular, the seamless integration of manual edit operations with procedural regeneration of features remains an open issue. In Section 6.6, we describe some promising directions for continuing this research.

Our contributions to procedural generation research are combined in a framework for declarative modelling of virtual worlds. In the next chapter, we discuss a semantic model for virtual worlds as the foundation of the framework.

3

A semantic model for virtual worlds

In this chapter, we describe a semantic model for virtual worlds, as incorporated in our framework for declarative modelling of virtual worlds (see Chapter 1). The concepts in this model lay the foundation on which all subsequent chapters build. We discuss how the model is organized and what kind of relations exist between the entities in the virtual world. Using the semantic model, we can define these relations and automatically maintain the consistency of all objects in the virtual world, as explained in Chapter 5.

Most 3D modelling systems targeted at creating virtual worlds maintain a *geometric* model of the virtual world. In such a model, typically each entity is identifiable as a separate 3D geometric shape and can contain some additional information (e.g., a model filename, material definitions, animations, attached game scripts), or it can be part of a hierarchical structure such as a scene-graph. However, this kind of structuring of virtual worlds is fundamentally different from a *semantic* model.

In a semantic model, the type, role and relationships of an entity are explicitly represented. In a geometric modelling system, a tree could be represented as a triangle mesh, some material definitions and textures, and a position and orientation. This is all the information required to visually represent the tree, but it is far too limited to allow for any automated *reasoning* on trees. In contrast, a semantic representation of such a tree, as in our model, contains additional information, such as age and plant species, its preferences and effects on the local soil, and relates this tree to the forest it is part of and to its neighbouring plants.

The semantic model presented in this chapter is generic and easily extendable with new features and objects for the virtual world. As we will see in the subsequent

chapters, semantics play an important role in the maintenance of the consistency of a virtual world model, as well as for capturing designer intent.

3.1 Semantic modelling approach

This section briefly discusses the role of semantics in the creation of 3D virtual worlds, and how semantic modelling is featured in our research. As stated above, both manual modelling systems and procedural methods often lack a *semantic* representation of the features and objects, for which they create solely the geometry. There are many advantages in introducing semantics to enrich features and objects with additional information besides their geometric appearance. By having more information on an object's properties and role in the world (e.g., soil preferences of vegetation, functional properties of furniture), a procedure for automatically placing this type of object can take advantage of this information to generate a more plausible layout. Furthermore, by encoding object relations and constraints in these semantics, a procedure can automatically ensure and maintain the validity of a generated layout. Finally, in a mixed-initiative setting, a procedure can, for instance, suggest suitable locations for any type of object a designer wishes to place in the world.

Several semantic modelling approaches have been proposed, as surveyed in [Tutenel 08], including semantics applied in the context of CAD/CAM [Bidarra 00], Kallman's *smart objects* [Kallmann 98], object interactions by Peters et al. [Peters 03], and constraint-based interior layouts (see the research of Smith and Stürzlinger [Smith 01] and, later, Xu et al. [Xu 02]).

In [Bidarra 10], we describe the close relation between the research in this thesis and the semantic modelling approach by Tutenel et al. (see [Tutenel 12] for more details), and how both approaches complement each other. In particular, our model for virtual worlds borrows part of its structure from the semantic modelling approach.

At the core of the semantic modelling approach is the *semantic library* [Tutenel 12]. With this flexible library, one can define the relevant semantics of virtual objects in a particular domain of application. The basic structure of the semantic library is an ontology of physical objects, inspired by the WordNet database [Miller 95]. Over the years, the semantic library has evolved into a detailed and structured model for defining the semantics of any object in a virtual world.

The first application of the semantic library was in the domain of solving interior layouts [Tutenel 09a, Tutenel 09b]. A result of this layout method can be observed in Figure 3.1. The method was later generalized to arbitrary small-scale 3D scenes, for which a designer can define a semantic description of the scene, which is automatically translated to a layout problem that can be handled by the solver [Tutenel 10]. Besides its use at design-time, several runtime applications of semantics are supported and are currently being explored as well, such as the interaction with objects and the services they provide [Kessing 09], adaptive virtual worlds [Lopes 11].

The basic concept in the semantic library is the *entity*. The semantic library



Figure 3.1: An office scene automatically generated by the semantic layout solver of Tutenel et al. [Tutenel 10].

provides a hierarchical database of these entities, where for each entity its *semantics* are specified, i.e., all information that helps convey the meaning and the role of an entity in the virtual world. An entity can either be an abstract concept with no physical representation (e.g., a country), or a physical entity that has a concrete presence in the virtual world (e.g., a 3D geometric model).

For each entity, its semantics includes its *attributes*, *services* and possible *relations* with other entities. Attributes are name-value pairs that can have a specific unit of measurement (e.g., meter or kilogram). Services define the functionality an object provides at run-time; for instance, a soda vending machine can offer soda cans if money is inserted. Relations define dependencies between physical objects used, for instance, for automatic reasoning on object placement (e.g., the sofa has to face the TV [Tutenel 09a]).

Classes of entities can be inter-related by means of inheritance. Similar to object oriented programming, a class of entities can derive from parent classes and can be specialized in child classes. For instance, the class *tree* can be specialized in a specific species of tree, e.g., a *Chestnut tree*. A child class inherits the properties of its parent class, but can override or extend these. Naturally, all instances of physical entities are ultimately associated with some specific geometric model (e.g., a textured 3D model of a Chestnut tree), regardless of whether it is modelled by hand or procedurally generated on request. The entire knowledge base is represented and stored in a purpose-built relational database [Tutenel 12]. In this thesis, we have used the semantic library for the definition of the individual semantic objects (see Section 3.2.3).

3.2 Levels of abstraction in terrain features

Our semantic model for representing virtual worlds is based on the concept of *terrain features*, i.e., all relatively large, clearly identifiable entities in the virtual world. A terrain feature is a high-level concept that groups a set of concrete objects found in a virtual world. For instance, for a river feature, we consider the flowing water, bedding and banks, as well as any surrounding vegetation such as reed, to make up the river. Another clear example is a city, which is essentially a collection of buildings, roads, parks, lampposts, etc., that together form the abstract concept of a city.

Each terrain feature is defined at several *levels of abstraction*, giving structure to both its layout or topology and its objects. Procedural generation of a terrain feature can therefore be described as a top-down process, starting from a coarse input specification, and refining this specification from abstract structures to concrete objects in several phases, resulting in the complete terrain feature representation (see Chapter 4).

Figure 3.2 depicts an overview of the organization of a terrain feature. Each procedural refinement results in a new level of abstraction that further details the feature. The number of refinement phases varies according to the complexity of the feature, as a feature can have several sub-structures at the structure level. We will now discuss the different abstraction levels of terrain features in more detail.

3.2.1 Specification level

The specification level defines the high-level description of the feature, on the basis of which the procedural model can be generated. A terrain feature specification consists of a 2D coarse outline shape, which can be a point, a polyline (i.e., a sequence of consecutive line segments) or a simple polygon, defined at a specific location within the virtual world. The outline indicates where the feature is situated and provides an indication of its shape and structure.

Besides an outline shape, a feature specification comprises a set of *semantic attributes*. These attributes differ from typical parameters in that they describe properties the resulting procedural model should adhere to (e.g., the *width* of a river), instead of providing concrete input settings for the procedure employed to generate the feature (e.g., the *persistence* of a fractal noise generator). Therefore, we introduce a mapping of the attributes to a procedure's internal settings in such a way that the generated model matches the specification.

There are two main advantages of semantic attributes over procedure parameters:

1. As discussed in Section 1.1, the parameters of a procedural method are often cryptic and unintuitive for designers to work with, forcing them to a *trial and error* approach. With semantic attributes, designers have a far better understanding of what the effect of a specific value will be for the end-result. This makes our approach more accessible for novice users of procedural methods.

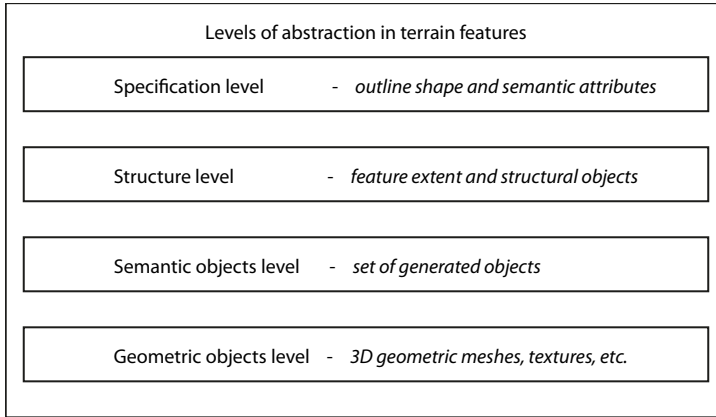


Figure 3.2: Overview of the levels of abstraction in a terrain feature.

- Using semantic attributes clearly separates the terrain feature specification from its generation procedure. Procedure parameters are specific to the actual algorithm used, while semantic attributes are independent of implementation details. This increases the flexibility of our framework by allowing any current procedure to be replaced by another. Of course, each newly integrated procedure has to map the semantic attributes to its internal settings (discussed further in Chapter 4).

A feature specification is thus the input for defining a new terrain feature. Typically, a feature specification results from direct interaction by a designer, as explained in Chapter 6.

3.2.2 Structure level

At the structure level, features only have an abstract representation of their internal structure, including a feature *extent*, representing the area of the virtual world affected by the feature, i.e., its footprint. In case of a forest feature, for instance, the extent defines the exact forest boundaries (see Figure 3.3).

For some types of features, additional elements are present at the structure level. These elements are used to layout the individual semantic objects within the feature extent, and to represent a feature's logical and functional structure. As an example, a city feature generates a distribution of district elements, which are used to determine the type and layout of the individual semantic building objects. Other than the extent, the data that forms the structure level of a feature is specific for each feature type.

Determining the structure level is the first step in the procedural generation of a terrain feature (see Chapter 4). Furthermore, it is at this level that maintenance of the consistency of the feature in relation to its surrounding features is performed (see

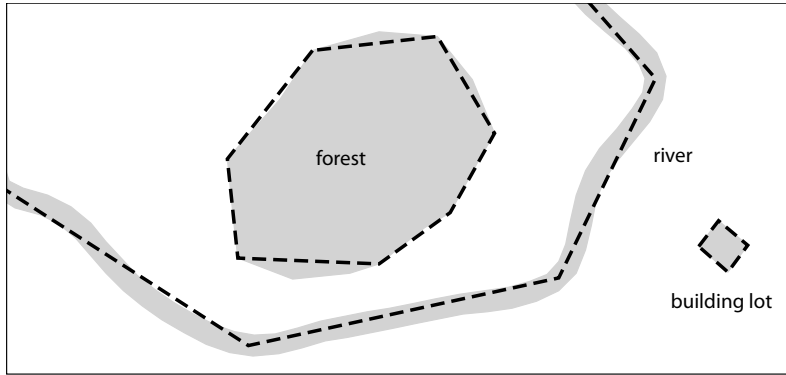


Figure 3.3: Examples of a specification outline (dashed line) and a feature extent (gray) of a forest, road and building lot feature.

Chapter 5). Complex terrain features, such as a city, can be composed of a hierarchy of levels of structure.

3.2.3 Semantic object level

The semantic object level of a terrain feature encompasses all individual semantic objects that will result in 3D geometry, e.g., all the individual tree objects in a forest feature. All objects are instantiated from a specific class in the semantic library and therefore inherit all the semantics incorporated in the entity hierarchy.

Relations between semantic objects are represented by connections (e.g., street connectivity) and placement constraints (e.g., minimum distance between objects). Terrain features can contain one to many thousands of semantic objects, and include a wide variety of object types. Typically, the same type of semantic objects can be found in different terrain features, for instance, both a forest and a city contain vegetation such as trees.

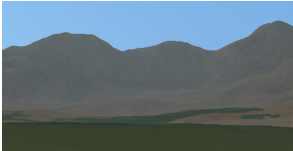
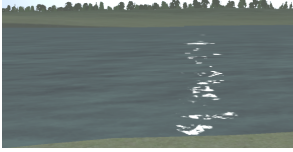
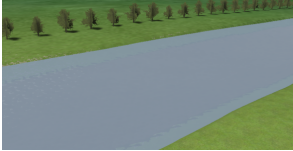
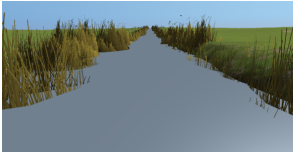
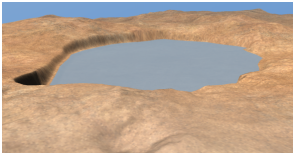
3.2.4 Geometry level



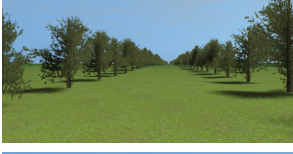





The geometry level defines all representations derived from the set of semantic objects that make up a terrain feature. The foremost of these representations are 3D models, used for the resulting visual representation of a virtual world. These 3D models and corresponding textures can either be selected from a library of hand-made models or be procedurally constructed. Other possible representations of terrain features at this level include 2D visualisations, digital maps, or additional representations for games and simulations, such as an AI path information database. Such representations are automatically derivable from semantic objects.

3.3 Incorporated terrain features

To test and evaluate our framework and, in particular, the model presented in this chapter, we have incorporated a substantial number of terrain features in our SketchaWorld prototype (see Chapter 7). These features range from very large features, such as mountains and cities, to small scale features, such as hedges and tree lines. For each feature, its specification and semantic structure have been defined, and the required generation procedures have been defined. Together, they allow for the creation of largely complete virtual worlds. In Table 3.1, we list the incorporated features and show a generated example of each feature. Furthermore, the table describes the feature specification that a designer can use to declare such a feature, with semantic attributes indicated in *italics*.

Table 3.1: Terrain features in the prototype SketchaWorld.

Name	Description of feature specification	Generated example
Landscape	Grid of ecotopes, where each cell contains elevation ranges and soil material definition.	
River	Polyline outline, <i>width</i> , <i>depth</i> .	
Canal	Polyline outline, <i>width</i> , <i>depth</i> .	
Ditch	Polyline outline, <i>width</i> , <i>depth</i> .	
Lake	Polygon outline, <i>depth</i> .	

Name	Description of feature specification	Generated example
Levee	Polyline outline, <i>levee type</i> .	
Forest	Polygon outline, <i>vegetation species, density, age</i>	
Tree line	Polyline outline, <i>vegetation species, spacing, age</i> .	
Hedge	Polyline outline, <i>hedge type, height, width</i>	
Field	Polygon outline, <i>field type</i> .	
Road	Polyline outline, <i>type, profile</i>	
Building	Polygon outline, <i>building type</i>	
City	Polygon outline, <i>population size, city type, city centre type, districts</i> .	

It is relatively straightforward to incorporate a new type of terrain feature in terms of specification and semantic structure. The total complexity and implementation effort very much depends on whether a new, dedicated procedure to generate the feature has to be devised.

To give some concrete examples of the abstraction levels of terrain features, we consider the forest and city features. In Table 3.2, the level hierarchy of these two features is schematically represented. The forest feature has a non-complex structure level. The city feature has a more complex and hierarchical structure level, with concepts such as districts and blocks, which corresponds to how cities are structured in the real world, but for which no concrete semantic or geometric objects are generated.

3.4 Layered structure

A virtual world consists of a wide range of very diverse types of objects, ranging from mountains to street lights. All these semantic objects have to be organized in a convenient and logical manner. A coarse categorization of the elements at the semantic objects level is provided by a layered structure [Smelik 08, Smelik 10a]. Structuring elements in layers is common practice in Geographic Information Systems (GIS). Similarly, Parish and Muëller used several predefined layer maps (e.g., height-, water-, and obstacle-maps) for procedural city generation [Parish 01]. Each semantic object is situated on one of the predefined layers of the virtual world model. We distinguish five layers, stacked as follows:

1. Urban layer: e.g., cities, districts, blocks, houses, factories
2. Road layer: e.g., highways, major roads, streets, paths, bridges, road signs, street lights
3. Vegetation layer: e.g., natural forests, planted tree lines, agricultural fields
4. Water layer: e.g., rivers, canals, lakes, ditches, oceans
5. Landscape layer: elevation profile and soil material

This specific organization has been chosen because of the semantic similarity of and relations between the objects within each layer. Although there are many alternative categorizations possible, it has been our experience that this categorization matches both the data and the generation procedures well. The categorization is not unlike layers found in GIS. All layers are initially empty, except for the landscape layer. Its extent encompasses the complete virtual world and provides the foundation on which all features and objects are placed.

Level	Forest	City
<i>specification</i>	outline, density, species, age	outline, population size, city type, centre type, districts
<i>structure</i>	forest extent	city extent, districts, transportation network, blocks, parcels
<i>semantic objects</i>	trees and plants	roads, buildings, vegetation, light poles, etc.
<i>geometric objects</i>	3D vegetation models, leaf textures, etc.	3D buildings meshes, wall textures, etc.

Table 3.2: Examples of the levels of abstraction in a forest feature and a city feature.

3.5 Discussion

This chapter introduced a semantic model for virtual worlds, as employed in our framework for declarative modelling of virtual worlds. It structures terrain features in several levels of abstraction, from a coarse user specification to concrete 3D geometry. This structuring is very convenient for the procedural generation of the features, because it separates the procedural model (e.g., a road network) from the actual procedural technique used to generate it (e.g., an L-system). This allows us to integrate procedural techniques in a more flexible manner (see Chapter 4).

Furthermore, using the semantic library, objects that make up a terrain feature are enriched with information on their functionalities, services and roles, allowing for a variety of applications other than visualisation. The semantic library also eases the extensions of object descriptions, such as introducing new attributes and object relations.

The semantic model introduced here is dedicated to representing features of virtual worlds, and perhaps not suitable for other procedural models. Limitations of the concrete implementation of our model include the choice of a height-map representation for the landscape, as this popular representation excludes overhangs and complex underground structures.

The semantic model for virtual worlds and the concepts introduced in this chapter provide the foundation for our framework, and hence will play a role in all subsequent chapters.

4

Integration of procedural methods

The semantic model for virtual worlds, described in the previous chapter, provides a solid foundation for our framework for *declarative modelling of virtual worlds*. This chapter focuses on methods for the integration of individual procedural algorithms and techniques. In the subsequent chapters, we will complete the presentation of our framework by treating two other essential aspects: consistency maintenance (Chapter 5) and user control (Chapter 6).

As concluded in Chapter 2, for many features of 3D virtual worlds, suitable and mature methods have been proposed that are able to generate plausible results. However, to date, they exist as standalone results. Too little attention has been given to the *integration* of these methods in a common framework, providing designers with the means to use them in combination. As stated in the introduction, we believe that the lack of integration is one of the main causes for the reluctance in the industry to employ procedural methods. It is simply not so useful to generate a feature, such as a road or river, in isolation, as the form, structure and details of a feature all depend on the context in which it exists.

The framework presented in this thesis specifically aims at integrating procedural methods to facilitate automatic creation of virtual worlds. Such procedures can be employed to generate terrain features or complex semantic objects (see Section 3.2). Integrating an existing standalone procedural method in our framework has clear benefits, both for its author and its end-user:

1. The framework provides a solid platform for research on procedural methods to be applied and evaluated in the context of complete virtual worlds.

2. Integrated procedures can be replaced as new research results become available, without needing to modify the framework or the semantic model for virtual worlds.
3. The integration of procedural methods allows designers to use them in any combination, without concern for the integration of their results, or for maintaining their consistency during consecutive procedural operations.

This chapter describes the structured integration of procedural methods. We discern the integration of procedural methods at two levels of abstraction (as defined in Section 3.2): the *feature* level and the *semantic objects* level. The procedural generation process of the two different levels follows the same principles depicted in Figure 4.1. For both levels, our framework steers a number of integrated procedural methods to generate content (a feature or a complex semantic object) matching input specifications; this content is combined into a semantic model, of which the consistency is automatically maintained, ensuring that the elements that make up the model are in harmony and without conflicts.

The generic process in Figure 4.1 is instantiated at the two different levels as follows. For terrain features, each type of feature maps to a specialized integrated procedural method. The framework employs the procedural method to generate the feature according to its specification. All generated features are combined in the semantic model for virtual worlds (Chapter 3). The consistency of this model is automatically checked and maintained by our virtual world consistency maintenance mechanism, explained in the next chapter. In Section 4.1, we describe the integration of procedural methods for generating terrain features, illustrating this with an example of an integrated procedural method for generating river features.

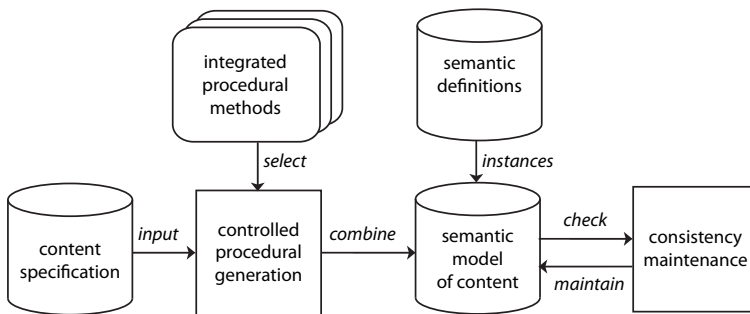


Figure 4.1: Overview of the generic procedural generation process from specification to generated content in our framework.

For semantic objects, we use a combination of procedural methods, each of them generating a particular element of the object. These elements are combined in the

semantic model of the object, which is defined using the semantic library (see Section 3.1). The consistency of the object model is maintained by a semantic moderator. In Section 4.2, we present our integration method for generating (complex) semantic objects. This method was devised as part of a case study on procedural generation of consistent building models, but it could very well be applied to other types of semantic objects.

4.1 Integrating procedural methods for terrain features

Figure 4.2 shows a simplified view on the procedural operation to generate a terrain feature, from user specification to terrain feature, generated in phase (a), and, finally, to its derived 3D representation (phase (b), see Chapter 7 for details). Note that the operation corresponds to a single iteration of generating one terrain feature. In an iterative modelling workflow, such an operation will typically be executed several times on the basis of (slightly) modified user specifications (see Chapter 6 for details). The overview in Figure 4.2 will be refined during the course of this section.

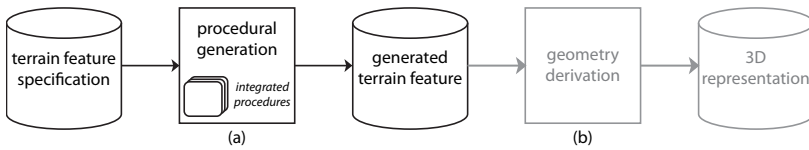


Figure 4.2: Simplified overview of the procedural operation to generate a terrain feature from a user specification, after which a 3D representation can be derived.

Once a feature, such as a river or a lake, has been specified, we proceed to generate its structure and objects accordingly using procedural methods (Figure 4.2 (a)). Each type of terrain feature employs one or more specific procedures from the procedural methods integrated in the framework. These procedures in some cases had to be newly created specifically for this purpose. However, in many cases it is possible to base or inspire a procedure on the large volume of published procedural methods. Amongst the reasons is the fact that, as concluded in Section 2.3, those methods already cover many of the possible features and objects. Using existing research has the advantage that one can build upon proven and accepted methods, without having to reinvent the wheel.

Integrating a procedural method requires some amount of implementation effort, as it has to properly implement the framework's interface and correctly employ the virtual world model. In this section, we give insight in the challenges of integrating an existing, standalone procedural method in the framework. We follow this section with an example of integrating a procedure for generating river features. Finally, there are some limitations to what kinds of methods are suitable for integration, which we will discuss thereafter.

4.1.1 Interaction between the framework and procedural methods

In the first refinement the procedural generation operation, shown in Figure 4.3, the procedural generation phase is split up in two parts: generate feature structure and generate feature objects.

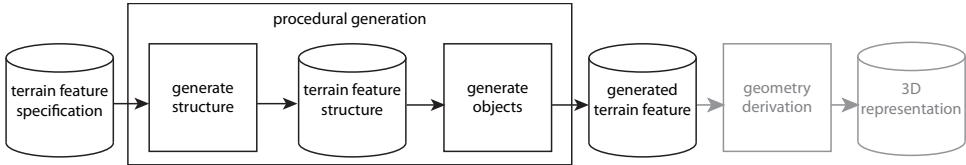


Figure 4.3: Refined view of the procedural operation to generate a terrain feature.

To be integrated in our framework, a procedural method has to adhere to an interface, and support a number of functions. First of all, it has to be able to generate a specific type of feature according to its specification. Secondly, the feature generated should be made up of semantic objects, as defined in the virtual world model (see Chapter 3). Lastly, it should fulfil a small number of additional requirements imposed by the consistency maintenance mechanism (see Chapter 5) and user control (see Chapter 6).

The interface that the framework uses to interact with a procedural method consist of the following list of functions:

- *generateStructure*: generate the extent and structure of a terrain feature, given a feature specification;
- *generateObjects*: generate and place all semantic objects that make up the terrain feature, within the generated extent and according to the feature's internal structure;
- *restructure*: adapt or regenerate the feature's structure, subject to additional requirements imposed by the consistency maintenance mechanism, explained in Chapter 5;

As follows from the interface definition above, the responsibility of a procedural method is only to generate a terrain feature matching the specification. Other responsibilities, such as maintaining consistent relations with other features, or embedding the feature in the landscape, are encapsulated within the feature's semantics. This eases the integration of new procedural methods, or replacing existing ones.

Making a standalone procedural method suitable for integration can be challenging. In general, such a procedure typically does not operate on the basis of a feature specification, but generates content according to the settings of a number of algorithm parameters. For enhancing a procedure to generate features according to specifications, two steps can be discerned:

1. The procedure must be able to generate a feature structure matching the specification outline;
2. A mapping from the semantic attributes of the specification to the parameters of the procedure has to be defined.

Generation of a feature structure matching the specification outline

The specification of a terrain feature includes a coarse 2D outline shape. As explained in Section 3.2.1, this shape can be a point, polyline, or a simple convex or concave polygon, positioned somewhere in the virtual world. These shapes serve as a guide for the procedure to generate the structural level of the feature.

In our framework, we do not strictly enforce the feature extent to fit exactly to the specification outline. We believe that strictly enforcing this would limit the procedures in their ability to generate variations in results. Instead, a procedure is responsible for generating a structure that a user would reasonably expect, while enriching it with plausible procedural details.

Of course, in giving the procedure the freedom to interpret the outline shape liberally, we introduce a new challenge. It is not really possible to ensure that an integrated procedural method will always generate a structure according to user expectations, as different users will have different expectations and an exhaustive test of possible outlines is not feasible. Therefore, in the process of integration of a procedure, much user feedback and parameter tuning are typically needed. Designers clearly profit from this effort, by having control over where and how a terrain feature is generated, without having to understand the inner workings of the procedure.

A minimal example of what a procedure could do is to refine the specification outline shape to better match the landscape's local elevation profile, according to some criteria such as maximum slope, preferred elevation ranges, etc. However, being integrated in the framework, a procedure has the semantic model for virtual worlds at its disposal. As a result, the procedure can consider other nearby features and objects, and the local soil profile. A procedure that takes the local environment into account often results in more plausible and interesting results. For instance, a forest generator might take nearby bodies of water into account, as well as the soil material, to determine the distribution of vegetation. Most standalone procedures generate a feature using a specific algorithm or technique, such as L-systems, which often supports the introduction of new rules and constraints. This can be used to encode the effects of the local environment, as provided by the semantic model, on the feature structure.

Defining a mapping of semantic attributes to procedure parameters

The second step for integrating a procedure is to map the values of semantic attributes (part of the feature specification) to the proper settings of the parameters exposed by

the procedure. Recall from Section 3.2.1 that a semantic attribute describes a desired result from the user perspective. On the other hand, a procedure parameter has an influence on the algorithm's inner working, although the relation of this parameter's value to the end result is often not so obvious. As a result, providing a suitable mapping from semantic attribute to a procedure parameter can be rather challenging; especially considering that each semantic attribute may need to influence on several procedure parameters.

As described in Chapter 2, procedure parameters are typically cryptic and have non-obvious interdependencies. On top of that, often only a limited range of the possible values for a parameter leads to plausible results being generated. Similarly to the above considerations for tuning a procedure to match specification outline shapes, by exposing the user to semantic attributes instead of procedure parameters, the burden of finding good settings for procedure parameters is alleviated for designers.

Using the defined mapping from semantic attributes to procedure parameters, the procedure will generate results that match user expectations for the semantic attribute value. Some of these user expectations expressed through attributes are straightforward to map and generate, e.g., *this river should be 50 meters wide*, while others leave some room for interpretation, e.g., *this forest should be very dense*. The bottom line is that the mapping will need to be carefully tuned according to user feedback.

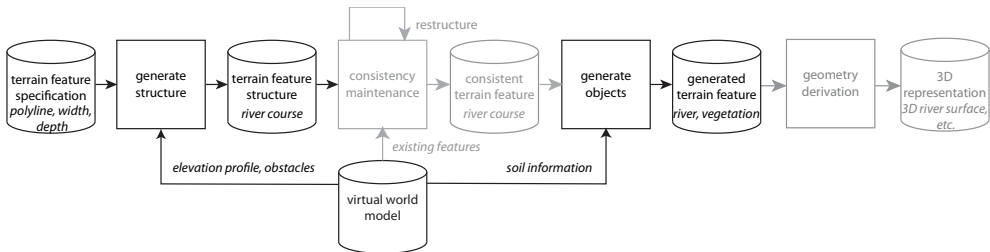


Figure 4.4: Detailed view of the procedural operation generate a terrain feature, annotated for the river feature example.

4.1.2 Example of an integrated procedure

As an example of a procedure integrated in our framework, we consider the procedural method used to generate the *river* feature. Figure 4.4 further details the procedural generation operation, with annotations (in italics) of the specifics for the river feature example. The figure shows that the virtual world model is employed in all the main phases, for instance, a river's structure very much depends on the elevation profile of the landscape. We will now describe how two of the three interface functions are implemented for the river feature example. The *restructure* function, part of the consistency maintenance mechanisms, will be explained in more detail in Chapter 5.

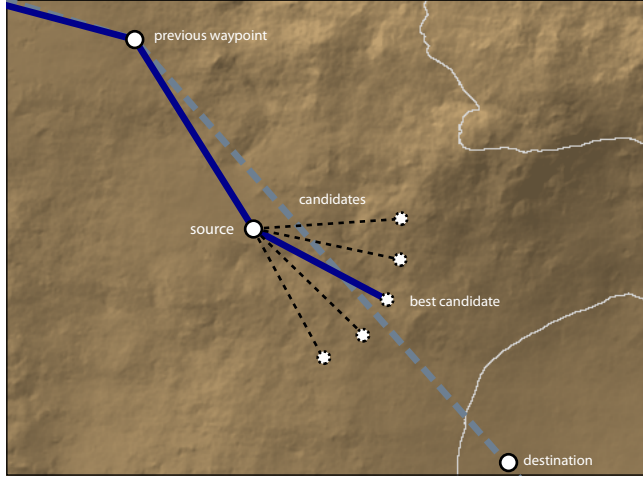


Figure 4.5: Visualisation of river path plotting step, where each candidate is evaluated based on elevation, curvature and feature crossing.

The *generateStructure* function generates a river matching the river feature specification. In Section 3.3, we saw that this specification consists of a polyline outline and two semantic attributes: the *width* and *depth* of the river.

Matching the specification outline to generated results is, in the case of a river, relatively straightforward. The control points of the polyline are interpreted as waypoints of an iterative path planning procedure, visualised in Figure 4.5. The main aspect to consider in this procedure is the local elevation profile, as the river needs to flow downhill, while visiting the user-specified waypoints.

The actual river path is determined by iteratively finding a sub path for each pair of waypoints \bar{p}_{src} and \bar{p}_{dst} in the specification. It starts from the highest elevated point, either the start or end point of the polyline. The partial procedure is outlined in Algorithm 1. It is a generic path planner, extended from the road generation procedure of Kelly and McCabe [Kelly 07]. The planner is steered by scoring each proposed candidate locations.

Each iteration of the procedure generates n new candidate points \bar{p}_{can} on a circle with range r_{step} at an interval $[\alpha - \alpha_{dev}, \alpha + \alpha_{dev}]$, where α is the angle from \bar{p}_{cur} towards \bar{p}_{dst} and α_{dev} is an angle range of deviation from α (e.g., 36°). Each of these generated candidates is scored according to the following weighted sum of scores:

Algorithm 1 Iterative path planning procedure

```

for all subsequent points  $(\bar{p}_{src}, \bar{p}_{dst})$  in  $\bar{p}_{start} \dots \bar{p}_{end}$  do
  while !reached do
     $\alpha = \text{computeAngleToDst}(\bar{p}_{src}, \bar{p}_{dst})$ 
    // generate candidate intermediate points
     $c = \text{generateCandidates}(\bar{p}_{src}, n_{can}, r_{step}, \alpha, \alpha_{dev})$ 
    for all  $\bar{p}_{can}$  in  $c$  do
      // compute candidate suitability score
       $s_{elevation} = \text{getElevationScore}(\bar{p}_{can}, \bar{p}_{src}, \bar{p}_{dst})$ 
       $s_{curve} = \text{getCurveScore}(\bar{p}_{can}, \bar{p}_{src}, \bar{p}_{dst}, \alpha_{dev})$ 
       $s_{obstacle} = \text{getObstacleScore}(\bar{p}_{src}, \bar{p}_{can})$ 
       $s[\bar{p}_{can}] = w_{elevation}s_{elevation} + w_{curve}s_{curve} + w_{obstacle}s_{obstacle}$ 
    end for
    // select best candidate based on score
     $\bar{p}_{best} = \text{selectCandidate}(c, s)$ 
    if  $\|\bar{p}_{dst} - \bar{p}_{best}\| \leq r_{snap}$  then
      // snap to destination waypoint
      reached = true
       $\bar{p}_{best} = \bar{p}_{dst}$ 
    end if
    // start from selected candidate in next iteration
     $\bar{p}_{src} = \bar{p}_{best}$ 
  end while
end for

```

$$s_{elevation} = \frac{\bar{p}_{cur}.z - \bar{p}_{can}.z}{\|\bar{p}_{can} - \bar{p}_{cur}\|}$$

$$s_{curve} = 1 - \cos^{-1}\left(\frac{\|\bar{p}_{dst} - \bar{p}_{cur}\|}{\|\bar{p}_{dst} - \bar{p}_{can}\|} \cdot \frac{\|\bar{p}_{can} - \bar{p}_{cur}\|}{\|\bar{p}_{can} - \bar{p}_{dst}\|}\right) / \alpha_{dev}$$

$$s_{can} = w_{elevation}s_{elevation} + w_{curve}s_{curve} + w_{feature}s_{feature}$$

The terms $s_{elevation}$ and s_{curve} denote the scores for local elevation difference and river curvature, respectively. The feature score term $s_{feature}$ is negative if the current segment crosses the extent of another feature with would prevail in a conflict over this overlapping extent (explained in Chapter 5). Figure 4.5 visualises an iteration of the river path planner algorithm, where, in this case, \bar{p}_{best} was chosen because of its relatively steep local slope downhill.

The currently chosen weights reflect the importance of the elevation constraint for a river ($w_{elevation} = 0.7$, $w_{curve} = 0.1$, $w_{feature} = 0.2$). If the candidate with the highest score s_{can} does not adhere to the constraint of monotonously decreasing elevation, the elevation value of the candidate is forced to an elevation slightly lower

than its predecessor.

The width and depth of the river varies along its course. After this path has been determined, the river width is computed at each intermediate point, based on the *width* attribute and modified by the local slope. Next, the river's local depth is computed as a function of the local width and the *depth* semantic attribute.

The *generateObjects* function generates the concrete river object, by sweeping a Catmull-Rom spline interpolation of the path. Taking width and depth variations at intermediate points into account, we can obtain the 3D shape of this river. It also places a small amount of vegetation, such as reed, along the river banks. The river feature has only a limited amount of semantic objects, and its *generateObjects* function is therefore straightforward, once the structure has been determined. Other features, such as a city, can have a far more elaborate *generateObjects* function.

4.1.3 Integration limitations

There are some limitations to the kind of procedural methods that can be effectively integrated in the framework. Not all procedural methods will fit as well as others. Two main factors can hinder the successful integration of a procedural method in our framework: the method's performance and its ability to constrain its output.

Interactivity is an important factor in user control, although this does not require methods to run in real-time. Through asynchronous execution of procedural methods, such operations can take several seconds of computation time, without hindering or frustrating a designer. Obviously, there is a limit to this, as any method running longer than 10 seconds is likely to seriously hinder the iterative modelling workflow the framework provides. This means that some of the evolutionary optimization, simulation or agent based approaches that typically run non-interactively, might be inappropriate for incorporation in our framework, or would require prior optimization effort to run more interactively.

A second limitation is that a procedural method should adhere to a terrain feature specification, and, specifically, its outline shape. While this seems obvious, there are many existing methods that grow unconstrained structures, or which, in their current design, cannot be effectively guided by an outline. Attempting to integrate such a procedure could entail the need to extensively modify its internal algorithms, or lead to the conclusion that, again, this method is not a good fit for our framework.

4.2 Integrating procedural methods for semantic objects

The previous section described our method for integrating procedural methods for generating terrain features. This section focuses on complex semantic objects, such as intricate buildings with detailed façades and interior layouts. We present a method for integrating procedures to generate the parts of complex objects in combination (this section is in part based on [Tutenel 11]).

At the semantic object level of abstraction, features consist of concrete objects such as trees, streets, etc. To generate these objects, again, we typically employ procedural methods. Relatively basic semantic objects are generated as part of the generation procedure of the corresponding terrain feature and do not require additional complex procedures. For more intricate objects, such as buildings, there are a number of specialized procedural methods that can be used to generate parts of them, e.g., floor plans [Merrell 10, Lopes 10], building façades [Wonka 03, Müller 06], furnished layout [Tutenel 09a, Merrell 11, Yu 11], etc.

For such complex procedural objects, it can be more efficient to use several existing procedural methods in combination, instead of devising a new and dedicated solution for generating a complex object in one go. Of course, it entails that the individual procedures have to be integrated in some way.

This section describes an integration method for semantic objects. Although we have focused on buildings, the method is generic at its core, and can be applied to other complex objects, e.g., infrastructures as airports, harbours or train stations.

The integration method has some parallelism with our method for integrating procedures to generate features of the virtual world. In particular, the integrated procedures again employ a common semantic model to enable them to generate consistent and plausible results, following the schema depicted in Figure 4.1.

Our integration method aims at generating *consistent* buildings, which we define as buildings exhibiting two characteristics:

1. *complete* buildings, i.e., buildings consisting of not only a façade, but also interiors, stairs, furniture, etc. The main challenge is to find and implement suitable procedures to generate all that content;
2. *congruent* buildings, i.e., buildings with plausible elements (walls, windows, etc.) in harmony and without conflicts. The main challenge here is that most current procedural methods generate just one type of building element, without taking into account the remaining elements.

In this method, the individual procedures share a common semantic model and implement a communication interface, through which they are led to collaborate in the generation of buildings. With this setup, we can integrate and combine relevant procedural methods to generate all sorts of buildings, with an efficiency comparable to a dedicated procedural method. Like our integration method for terrain feature generation, the versatility in reusing and combining existing procedural methods brings about a number of advantages. First of all, developers can focus on local specialization, i.e., concentrate on improving individual generation methods for a building element, without considering the integration of its output into the complete building. Secondly, replacing outdated methods for specific building elements, or experimenting with new ones, becomes much easier, increasing development flexibility.

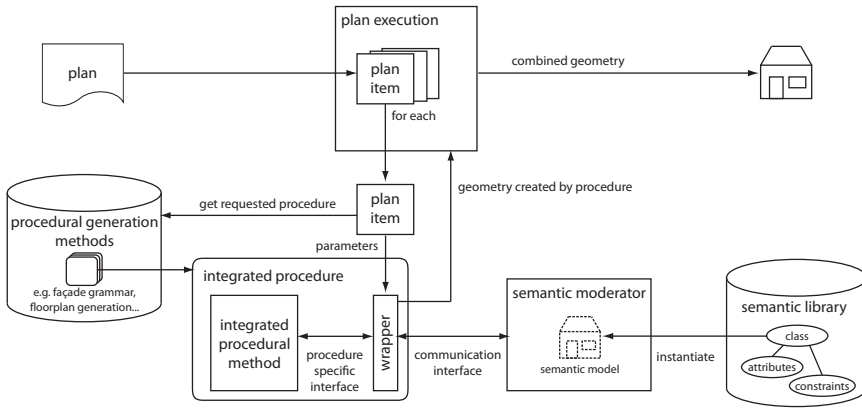


Figure 4.6: Integrating procedural methods for generating consistent semantic objects: *semantic moderator* (with *semantic library* and *communication interface*), *integrated procedures*, *wrappers*, *plan* and *plan items*.

Figure 4.6 outlines the architecture to support this integration method. The various procedural methods are made available through a wrapper interface and are invoked according to a plan. The semantic moderator, in turn, helps prevent conflicting situations, managing the communication with the procedures, and providing them with building advice. Below, we explain this setup in detail.

4.2.1 Semantic moderator

Typically, each procedural method is able to generate one specific element of a building (e.g., façade, floor plan, furniture, lot shape), but mostly without much regard for other building elements. Therefore, a major challenge for the integration is to watch over the consistency of a building, either avoiding or properly handling any conflicts arising among building elements. For this, we establish a *semantic moderator*, which shares relevant building information with the individual procedures, so that they can make good and timely decisions, in order to avoid conflicts, i.e., inconsistent results. We distinguish three categories of conflicts between building elements:

- *geometric* conflicts, occurring when building elements that should not intersect each other, overlap in some way. For example, façade windows should not intersect inner walls, furniture should not obstruct inner doors, etc.;
- *functional* conflicts, occurring when building elements with incompatible roles are associated. For example, bathrooms should not have the same type of window as bedrooms;

- *exclusion* conflicts, occurring when a *required* unique building element is placed in such a way that it becomes impracticable in the resulting building, and has to be removed from it. For example, a required fireplace should only be placed on one of the possible locations where it has a feasible path to the (façade or roof) chimney.

The semantic moderator is responsible for watching over the consistency of the building by examining and approving the requests of each integrated procedure. For this, it maintains a semantic model of the building, which represents all its elements, including their attributes and constraints. Each of these semantic building elements is an instance of a physical object described in Section 3.1, and therefore carries all its semantics.

An integrated procedure can resort to the semantic moderator in a number of ways in its generation algorithm, which we now briefly describe (see [Tutenel 11] for more details).

- *Register a building element* A procedure can register a new building element with the moderator. The moderator can either *approve* the registration, meaning that the new building element is deemed valid for integration, or *reject* it, meaning that the element causes a conflict, in which case the procedure should retract the conflicting element. For each successfully registered building element, the moderator instantiates the corresponding semantic element and inserts it in the semantic building model.
- *Register a constraint* Besides new building elements, integrated procedures can also register new *constraints* with the moderator, to be satisfied between two building elements. A variety of different constraint types can be defined, enforcing e.g., connectivity, proximity, adjacency or non-adjacency between elements.
- *Inquire about a building element* Procedures can *inquire* the semantic moderator about registered building elements, for instance to determine which room is adjacent to a given exterior wall, which rooms share an interior wall, what the function of a particular room is, etc. Inquiries can also be used to verify beforehand whether a candidate building element would be approved as valid by the moderator.
- *Select valid positions for a building element* A procedure can provide the moderator with a list of candidate locations for a building element, requesting it to *select* a given number of valid locations for that element. This query is typically used for specific types of building elements that need to be placed once (or any fixed number of times) in the building, such as an external ventilation unit, satellite dish or chimney. It is a useful advice for procedural methods that do not allow backtracking, and is particularly suited to avoid *exclusion* conflicts.

Using the above functionality of the semantic moderator, procedures are indirectly made aware of the results of each other's actions. By registering, inquiring and selecting, procedures are provided valuable building advice, to which they can react and thus prevent the occurrence of *geometric*, *functional* and *exclusion* conflicts.

4.2.2 Integration of procedures

We describe the steps required integration of new procedures, and the impact of the integration process on each procedural method. The two main implementation steps that need to be taken are (i) implementing a wrapper interface for the procedure, and (ii) modifying the procedural method to include the proper semantic moderator queries, described above.

The purpose of the procedure wrapper is to provide access to the functionality of the moderator using a generic interface, as shown in Figure 4.6. Such a wrapper only needs to be implemented once for each procedural method, regardless of the number of other procedures or the type of building being generated.

Furthermore, the wrapper allows integrated procedures to be notified of elements generated by another procedure. For this, the moderator uses a notification mechanism that informs all procedures of changes to the semantic building model. A procedure can handle specific notification events, triggering an action when another procedure registers a specification building element. For example, a texture generator can create an appropriate wallpaper when an inner wall is registered by a floor plan generator. Procedures can filter out irrelevant notification events; e.g., a façade generation method typically does not need to know the positions of all the furniture placed by a layout procedure. As a result, introducing more procedures will not necessarily have a large impact on the computational complexity of the building generation.

The final functionality of the wrapper is to handle the conversion between a procedural method's specific shape representation (i.e., data structure, coordinate system, etc.) and the common shape format used by the semantic moderator.

Of course, a specific wrapper can include more functionality relevant to the procedure. After communicating with the semantic moderator, a procedure might need to perform additional actions. Typical examples include: (i) deciding what to do when an element cannot be registered, or (ii) immediately selecting a position and creating a building element after getting a number of marked locations for this element.

Minor alterations will need to be made directly in the procedure. At least, the wrapper methods need to be invoked, for instance elements need to be registered with the moderator before they are definitively placed. Still, the implementation of the wrapper interface is the most important step required for the successful integration of a new procedure. After a procedure's wrapper is implemented in the correct way and the mentioned alterations to the procedure have been performed, that procedure becomes and remains correctly integrated, regardless of changes to, and replacements of, other procedures.

4.2.3 Plan execution

The method described so far enables procedures to collaborate, through their wrappers, in the generation of buildings. However, the invocation of the various procedures still needs to be orchestrated in such a way that they constructively work together, i.e., following the correct steps in the appropriate order. The order of invocation of procedures often has an influence on the end result, and we therefore need to have sufficient control over this.

To support this degree of control, we introduce the concept of a *plan*. As follows from Figure 4.6, the individual steps in a plan, or plan items, are executed sequentially, invoking the corresponding procedures through their wrappers.

Plans are simple documents where one can declare which procedures should be used, how to use them, and in which order. One can create separate plans for generating different building types using the same set of procedures. Primarily, a plan dictates the sequence in which procedures are invoked, and also provides the input each procedure requires. By varying the input and procedure execution sequence, we can define different building types. For example, using different values for the style and start shape input parameters of a façade grammar results in different building façades. Multiple executions of the same plan typically result in variations of the same type of building, since most procedural methods are stochastic in nature.

In particular cases, a straightforward one-time sequential invocation of a set of procedures can be sufficient for generating a consistent building. This is especially the case for situations where the constraints and dependencies between the building elements produced by the different procedures are fairly loose. An example is generating the façade of a one-floor building after the complete creation of a floor plan. If the only constraint is to avoid geometric conflicts between e.g., windows and interior walls, then their sequential invocation can create a multitude of consistent building variants.

However, for the vast majority of buildings, stronger dependencies are present and step-based execution of procedures is needed for consistent results. For example, a façade generator creating a multi-storey building might need to wait for the generation of one floor plan to complete, before resuming with the next floor's façade. Plans can include step-based execution of procedures if the wrapper functions are implemented to support it. Note that, although procedures can execute in a step-wise fashion, that is not enough to support backtracking, i.e., undo or redo a step of a specific procedure that turned out to yield an unsuitable configuration. The main reason for this is that to support backtracking in our integration method, every procedure should support it. This would be an unreasonable demand, since it would exclude many valuable procedural methods.

Plans are also responsible for another mechanism: sharing and passing building elements from one procedure to the next, to allow for further detailing by the latter. The moderator distributes the semantic elements representing the building elements among procedures, according to the plan. An example of this are building elements

produced by a floor plan procedure: after registration, floor plan elements could be passed to a shape grammar to detail their geometry or texture. The plan specifies and controls how registered elements are passed to other procedures.

4.2.4 Villa Neos: an example of a consistent building

This section illustrates the potential of our integration method for semantic objects by means of an example of an automatically generated consistent building. For this, we have selected, implemented and integrated the following procedural methods:

1. To procedurally generate the exterior of our buildings, we integrated the CGA shape grammar proposed by Müller et al. [Müller 06].
2. For generating floor plans, we integrated our grid-based procedural floor plan generation method [Lopes 10].
3. For furniture placement, we integrated our semantics-based layout solver [Tutenel 09a].

For each procedure, we created a specific wrapper to communicate with the semantic moderator. This wrapper provides the necessary calls and notifications for registration of building elements and inquiries for building advices. Furthermore, it provides conversion of generated geometry from the procedure specific model (e.g., 3D geometry, a 2D grid of tiles, etc.) to the common model used in the semantic moderator, ensuring the registered element's scale, orientation and location are coherent.

In each procedure, the usage of the wrapper had to be implemented as well. For instance, for the floor plan and interior layout procedures, registration calls or inquiries are added at specific points in the algorithm. For the shape grammar procedure, we provide each call as a shape operation (registration) or function (inquiry). They are written within a grammar definition file as part of the normal shape derivation rules. This made the interaction with the moderator easy and more intuitive, e.g., within a conditional rewriting rule we can inquire whether deriving the current shape to a window is allowed here, and if not, rewrite it as a plain wall segment instead.

The building plan can determine not only when but also to what extent each procedure is executed, allowing for interleaved, step-by-step execution of procedures. For this, *break* and *continue* calls were added to the wrapper for each procedure. A break call can have procedure-specific parameters. For instance, for a shape grammar procedure, a break point can be placed at a specific shape symbol, halting execution when that symbol is about to be derived.

Our example depicts a modern and luxurious Greek holiday villa. This villa has two floors, the second smaller than the first one because of a large open balcony. Inside, an interior staircase connects both floors. The building plan of the example is

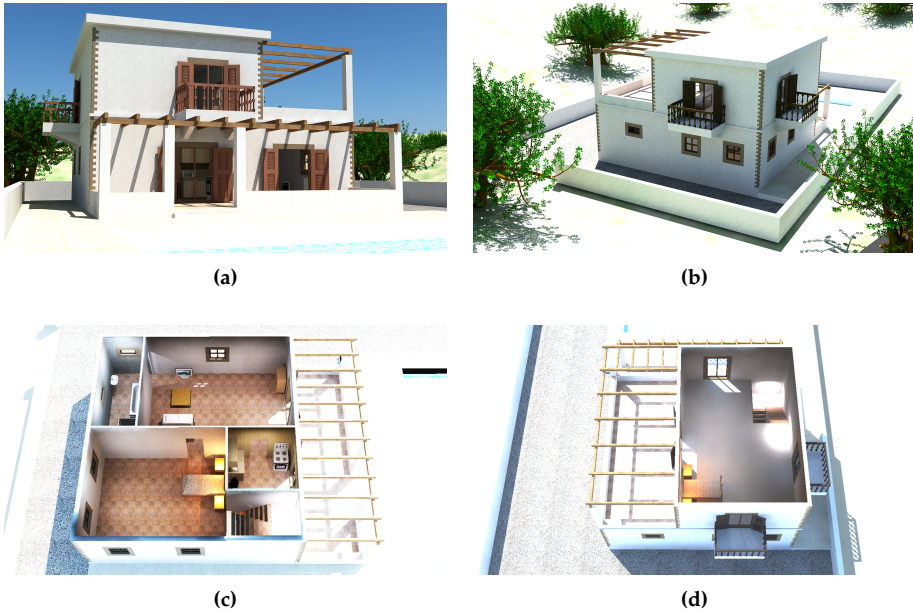


Figure 4.7: Example of Villa Neos, a Greek holiday two-storey luxurious villa: (a) front view with veranda and pool, (b) back view with different types of windows depending on adjacent rooms, (c) first floor with several rooms, (d) second floor with balconies, terrace and staircase.

quite straightforward, consisting of five consecutive steps (executed by the procedures in brackets):

1. Create coarse volumetric building shape (shape grammar);
2. Layout the villa's first floor (floor plan);
3. Layout the villa's second floor (floor plan);
4. Detail the complete building including its façade (shape grammar);
5. Place appropriate furniture in each room (furniture).

One of the results generated by this plan is shown in Figure 4.7. Figure 4.7 (a) and (b) shows the veranda and second floor balconies from different angles. Note the staircase connecting both floors in Figure 4.7 (c) and (d). The staircase shaft is determined by the shape grammar during the creation of the coarse building shape (step 1). It is then registered to the moderator as a semantic object of class *staircase*. In steps 2 and 3 of the plan, using an inquiry, the staircase is obtained and passed to first and second floor plans as a “room” that is treated as *fixed* during the layout

process (see [Lopes 10] for details). In this way, we ensure that the staircase placement is congruent between two floors. In the final step, the furniture procedure creates interior layouts based on the semantic room elements. Figure 4.7(c) and (d) show that the automatically placed furniture matches well with the function of the rooms.

4.3 Discussion

This chapter presented an important part of our framework for declarative modelling of virtual worlds: the structured integration of procedural methods. We discussed how the framework presented in this thesis forms a flexible platform for integrating procedural methods. The integration of procedural methods is performed at two levels of abstraction: (i) integrating procedural methods for each type of feature to generate virtual worlds, (ii) integrating procedural methods for each set of elements to generate complex objects.

The procedural generation process of these two different levels follows the same basic principles. Our framework steers a number of integrated procedural methods to generate content matching input specifications; this content is combined into a semantic model, of which the consistency is automatically maintained.

The common procedural generation process is realized in a slightly different way for both levels. In Section 4.1, we detailed the process of the generation of a terrain feature, from user specification to derived 3D geometry. All these features are combined in the semantic model for virtual worlds (Chapter 3). At each intermediate phase, the integrated procedure has the virtual world model at its disposal to generate a plausible structure and object layout.

The integration method is straightforward and has a clear communication interface. The main challenge of integrating a procedural method for a specific feature type is to adapt the procedure to generate features matching user specifications, which consist of a guiding outline shape and result-oriented semantic attributes. This is often an iterative process that requires much user feedback.

In Section 4.2, we explained the process of generating a complex semantic object using several integrated procedural methods, each suited for one type of element. The execution of these methods, integrated as procedures, is dictated by a plan. A semantic moderator coordinates and advises procedures towards the goal of generating consistent results.

For integrating a procedure, the creation a procedure wrapper is required, and the registration of generated elements and inquiries with the moderator need to be implemented in the procedure. After integration, the individual procedures still execute their original algorithms, while communicating results with the semantic moderator helps them to prevent the shared semantic model from reaching an irreversible invalid state, where required elements are misplaced or excluded.

Regarding the extendibility of our framework, we have focussed on the integration of new terrain features, semantic objects and procedural methods. Other concepts,

such as the organization of a terrain feature in levels of abstraction, explained in Chapter 3, and, in particular, the specification as the basis of every feature, are essentially fixed. As a result of these choices, both global procedural operations (for instance, a procedure that adds vegetation to the entire virtual world) or unconstrained generation procedures (e.g., an unconstrained city growth simulation) might not match the framework's design.

Although we believe that the currently integrated procedures for generating terrain features successfully demonstrate the working of our framework, they often do not represent the state of the art of the procedural generation research. Especially for urban environments, as discussed in Chapter 2, there exist many new approaches that are more elaborate and generate more detailed and plausible models. It would be interesting to further evaluate the framework with the integration of such new procedural methods.

The integration method for semantic objects was originally devised for generating consistent buildings. Although we have not explored this, the method and its setup are rather generic. As such, we expect that it could be applied to other complex semantic objects, for which it is beneficial to combine and coordinate several specialized procedures to generate a consistent result. Potential examples of this include complex infrastructure, such as airports and harbours.

Again, and even more so for the integration method for semantic objects, not all procedures are a good fit. This stems from the fact that here the procedures strongly depend on each other's generated output. In other words, if two procedural methods do not naturally fit well together, you can hardly make them fit any better regardless of the amount of integration work put in it. Consider the example of a floor plan generation method which creates rooms individually and assembles them to form a new building shape. If this unknown building shape needs to fit inside the building lot shape, which could have been generated by another procedure, many modifications might be necessary to assure that the results of those two procedures fit. Another complicating factor might stem from differences in capabilities of procedures, for instance when integrating a furniture generator that only supports rectangular rooms with a floor plan generator that produces arbitrary room shapes.

The integration of procedural methods presented in this chapter contributes to an important requirement for realizing the declarative modelling approach (see Section 1.2, requirement 5). In the next chapter, we discuss how to automatically keep the semantic model of virtual worlds, generated by these integrated procedures, in a consistent state.

5

Virtual world consistency maintenance

Virtual world features are not only to be generated according to designer specifications, but they also have to be properly embedded in the world in order to form a consistent and plausible environment. Each feature introduced to a virtual world typically *interacts* in some way with the existing features nearby, and vice versa. Examples of this behaviour include a feature adapting itself to local elevation constraints, affecting a nearby feature (e.g., a city competing with a forest for building space), modifying the local elevation profile for creating e.g., a road embankment or a river bedding, or forming some sort of connection, e.g., a junction or bridge.

One can imagine the amount of tedious manual modelling work when the responsibility of handling these interactions and keeping the virtual world consistent is left to the designer, as is currently the case for all manual and most procedural modelling systems. In our framework, we are able to maintain the consistency in an automated manner because, as we described in the previous chapter, the *semantics* of terrain features and their relations are encapsulated in the virtual world model.

This chapter describes the generic handling methods with which the consistency of the virtual world model is maintained (this section is in part based on [Smelik 11b]). We first give a motivation for consistency maintenance, by means of an example scenario. We then more precisely define interactions and introduce relevant concepts related to handling these interactions. Next, generic methods are presented to deal with these interactions in a generic manner. After this, we show how these methods work in practice, by returning to the example scenario previously described. We conclude with a discussion on the main advantages and limitations of our methods for consistency maintenance. This discussion continues in the next chapter, where the

repercussions of automatic consistency maintenance on user control are identified and addressed.

5.1 Motivation for consistency maintenance

We start our discussion on consistency maintenance with an example scenario where, using a typical manual modelling system, a designer creates a highway in a virtual world. This example serves to illustrate the different kinds of interactions that occur among entities in a virtual world. Furthermore, it shows how these interactions can be handled.

1. A designer creates a major road, running through a hilly landscape. After plotting the road's path and creating its surface geometry, the designer has to fit the road in the existing environment. Because of the roughness of the terrain, it is unsuitable for direct placement of the road. Therefore, the designer first has to create an embankment in the landscape's elevation profile along the path of the road. Cut and fill operations have to be performed to obtain a suitable smooth profile on which the road is placed. This can be seen as an interaction between the road and the landscape. The interaction is handled through a local modification of the landscape to fit the profile required by the road.
2. At some point, the road's path runs through a forest. As a result, several trees are situated either on the road's surface or too close to the verge. The designer is required to remove those trees obstructing the path. This is a typical example of two features competing for space. The conflict interaction is handled here by removing part of the forest to make room for the road (i.e., letting one feature prevail over the other).
3. Further on, the road intersects a river. In order to create a safe crossing, the designer manually inserts a suitable bridge model and connects the road to the bridge. Here, we also see two features interacting; however, the solution to this interaction is less conflicting: neither loses any ground, instead they *connect* at the crossing (i.e., they cooperate to overcome the interaction).
4. Later on, the designer decides to replace a rolling hill with a steeper, rough mountain. The road's path that previously ran on that hill now becomes blocked, as the slope has become too steep. The designer adapts the road's path to run around the mountain. Unfortunately, the embankment has to be partially restored and recreated for this. This is an example of the landscape interacting with the road, forcing it to be restructured by adapting its path.

In the above example, after each modelling step, the designer had to perform a number of laborious operations in order to re-establish the consistency of the virtual

world model. Now, consider the effort it would require designers, if, at a later point, it is decided to remove the road. The embankment would have to be removed, restoring the original landscape, the trees removed from the forest would have to be reinserted, the bridge no longer has any function, etc. This is just one modelling scenario where clearly some form of automatic consistency maintenance would be very helpful.

Some attention has been given to interactions in the literature (see Chapter 2 for details). Bruneton and Neyret present a generic solution for landscape modifications, such as road embankments, based on feature footprints [Bruneton 08], but not for other types of interactions. Galin et al. [Galin 10] include interactions with the landscape and with water bodies, and the formation of connections (bridges and tunnels) in the cost function for their path planner. However, their procedure is specific to major roads. In procedural methods for modelling urban environments (e.g., [Parish 01, Lechner 06, Weber 09]), the typical approach is to require input maps for the generation process, such as a height-map and “obstacle”-map (based on features such as rivers). These input maps then limit the buildable area of the city.

Although these methods successfully address the subset of interactions they focus on, they can be seen as limited in scope or feature-specific. Therefore, the *generic* approach to interactions presented in this chapter can be considered a novel contribution to procedural generation of virtual worlds.

5.2 Basic notions

We now introduce and define a number of basic concepts related to consistency maintenance in order to make our discussion on interaction handling more precise.

Recall from Section 3.2.2 that all terrain features have an *extent* e , representing the area of the virtual world affected by the feature, i.e., its footprint (Figure 5.1 (a)). At instantiation stage, a feature’s extent is empty. A feature can request to modify its extent by issuing a *claim*. A claim includes a claim extent, regarding the new area the feature wants to reserve for its use. An issued claim can be dismissed, or partially or fully granted. The extent of the granted claim replaces the feature’s current extent.

An *interaction* is the effect that occurs as two terrain features of the virtual world influence each other, through overlapping feature extents. The interaction extent e_i is defined as the overlap in the feature extents of the two interacting features. The terrain feature f_a that initiates the interaction (when it is created or modified) is denoted the *active* feature. f_a interacts with one or more *passive* features, denoted as f_p , already part of the virtual world model. Any feature f_p can interact with the active feature f_a at several disjoint extents e_i .

Interactions must be handled in order to maintain the consistency of the virtual world model, as they always have an effect on, at least, one of the features involved. An interaction can be handled either through a conflict or through a connection.

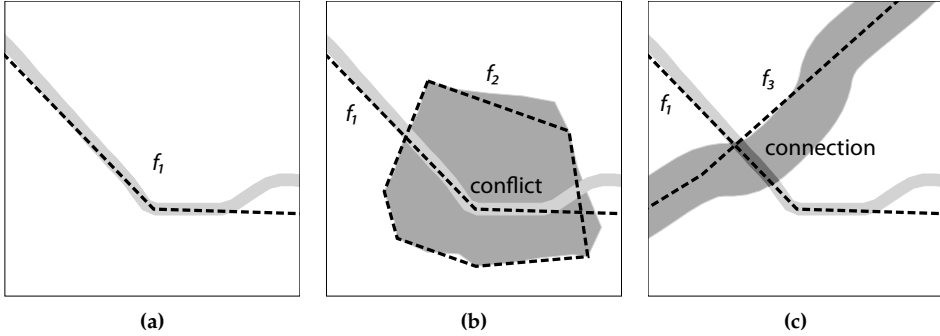


Figure 5.1: Feature extent: (a) the extent of a feature f_1 is shown in light gray (specification in dashed line), (b) a new feature f_2 loses conflict to f_1 , resulting in a partially granted claim, (c) overlapping extent (dark gray) shared between f_1 and f_1 by means of a connection.

A *conflict* is the interaction handling method by which the losing feature f_{lose} , either f_a or f_p , loses the interaction extent to the winning feature f_{win} . If $f_{lose} = f_a$, this means that its issued claim is only partially granted, i.e., its extent will not include the interaction extent e_i (Figure 5.1 (b)); if $f_{lose} = f_p$, its feature extent loses e_i . As a result, feature f_{lose} has to procedurally restructure itself within its new extent.

A *connection* is the interaction handling method by which f_a and f_p agree to share the interaction extent e_i . For f_a , this means that its issued claim is granted, at least for the extent e_i . For f_p its feature extent remains intact (Figure 5.1 (c)).

These definitions apply to all combinations of interactions between entities of the virtual world model. Our virtual world consistency maintenance is instantiated for effective handling of two common types of interactions: (i) between the landscape and a feature and (ii) between two features.

5.2.1 Landscape - feature interactions

The landscape plays a special role in our virtual world model in the sense that it forms its omnipresent basis to which all other terrain features attach. Therefore, the landscape does not compete with features to claim extent, as it is the “extent provider” for all other features. As such, the landscape, f_p by definition, cannot lose any part of its extent to another feature f_a , but it can form connections with features to share parts of its extent.

A landscape *connection* entails that f_a attaches to the landscape. The attachment of f_a at e_i results in the landscape to be constrained to an elevation and soil material profile p . This profile must be integrated in the landscape. Although a profile may be applied outside the feature’s extent (e.g., to obtain smooth transitions), the application extent should not overlap another feature’s extent, because that would affect the

landscape connection of that feature. In case a connection could not be formed for (part of) f_a 's extent, the interaction results in a conflict where f_a loses the interaction extent e_i .

5.2.2 Feature - feature interactions

Interactions between two features f_1 and f_2 are handled based on a set of *feature priorities*. In particular, the priority $p_{claim}(f)$ defines a numeric priority value of feature f for claiming an extent e for exclusive use. The values of the priorities p_{claim} are unique per class of terrain feature. However, for a specific feature instance f , the priority value of the feature class can be overridden with a new value.

Handling interactions through a *conflict* between feature f_1 , with $p_{claim}(f_1)$, and feature f_2 , with $p_{claim}(f_2)$, entails that the feature with the highest $p_{claim}(f_i)$ is the interaction's winning feature, f_{win} , the other feature is the losing feature, f_{lose} . In case of equal priority, the passive feature f_p is defined as the winner f_{win} .

Handling interactions through a *connection* introduces a semantic connection object, which is a semantic object that links two interacting features f_1 and f_2 at the extent e_i . It allows f_1 to share the extent e_i with f_2 . A priority $p_{con}(f_1, f_2, e_i)$ defines the preference for connection over conflict for feature f_1 and feature f_2 at extent e_i . The actual priority value is 0 if no connection is defined between the two feature classes of f_1 and f_2 . Otherwise, the defined base value can be modified by a context-specific factor (e.g. depending on the properties of e_i). The resulting value is compared to a defined threshold, below which the connection is rejected.

5.3 Handling landscape - feature interactions

Before discussing the handling method for multiple features interacting with each other, we analyse how we handle the basic interaction between the landscape and a terrain feature. As defined above, a change to a region of the landscape affects any terrain feature which extent overlaps with the region. This notion is captured in the landscape - feature interaction handling method, outlined in Algorithm 2.

As follows from this algorithm, the outcome of the landscape interacting with any feature f over extent e_i is that f needs to adapt its structure. However, a feature decides to what extent to restructure (if at all) which depends on its particular semantics and the scope of the changes to the landscape. Typically, drastic changes in the elevation profile will probably cause features like roads or cities to strongly restructure, whereas changes in soil material will probably affect vegetation features the most.

Recall from Section 4.1 that the restructure operation is part of the interface for integrated procedural methods. Depending on the implementation of the feature's associated generation procedure, restructuring could entail that a subset of the objects of the feature is removed from the respective terrain layer(s) and that part of the

Algorithm 2 Landscape - feature interaction handling method

```

// handle all interactions between landscape and terrain features
//  $e_i$  - extent of the landscape that is changed
handleLandscapeInteractions(extent  $e_i$ ):
  // find all affected terrain features
   $F = \{ f \mid f \in \text{features}, e_i \cap f.\text{extent} \neq \emptyset \}$ 
  // find all their dependent terrain features,
  // i.e., features involved in connections or conflicts with an  $f \in F$ 
   $F = F \cup \{ f_2 \mid f_2 \in \text{features}, \exists f_1 \in F : f_2.\text{dependsOn}(f_1) \}$ 
  sort  $F$  according to highest  $p_{\text{claim}}(f)$ 
  // let each affected feature  $f$  handle the interaction
  for all feature  $f$  in  $F$  do
    // restructuring results in a modified extent for  $f$  through connections and conflicts
    restructure( $f, e_i$ )
  end for

```

feature's granted extent is abandoned. Such a restructuring can have consequences for a set of other features, all of which have a *dependency* relation with f . Features are considered dependent on a feature f either if they have lost a conflict with f , or if they share extent with f through a connection. As a result of the changes to feature f 's extent brought about by its restructuring, these dependent features can now potentially reclaim extent previously lost to f , or might no longer need to share their extent with f . Because of this, even though their extent does not overlap with the changed region of landscape, they can be indirectly affected by the change in the landscape.

For the set of directly and indirectly interacting features F , the algorithm successively handles interactions ordered by priority p_{claim} , as its value provides a good heuristic of the impact a feature will have on other features.

Features always form a connection with the landscape, meaning that, for each type of feature a landscape connection has been defined. The connection of a feature f with the landscape attaches f to the landscape by imposing a desired profile. Although features typically adjust their structure to the existing landscape profile, they can still pose additional requirements to the local elevation and soil material profile. For instance, a building could require an area of flat terrain. Through the feature attachment, these local requirements are fulfilled.

5.4 Handling feature - feature interactions

Terrain features may compete with each other to claim extent for their own use. Therefore, a generic interaction resolution method has been devised to handle interactions arising between terrain features, based on the notions discussed above. The feature

Algorithm 3 Feature - feature interaction handling method

```

// handle all interactions between a feature and existing features
//  $f_a$  - feature which has made a new claim
//  $e_a$  - extent claimed by  $f_a$ 
handleFeatureInteractions(feature  $f_a$ , extent  $e_a$ ):
     $f_a$ .extent =  $f_a$ .extent  $\cup$   $e_a$ 
    // find all interacting features with overlapping extent
     $F = \{ f_p \mid f_p \in \text{features}, e_a \cap f_p.\text{extent} \neq \emptyset \}$ 
    sort  $F$  according to highest  $p_{\text{claim}}(f_p)$ 
    // handle all feature interactions
    for all feature  $f_p$  in  $F$  do
        handleInteraction( $f_a, f_p, e_a \cap f_p.\text{extent}$ )
    end for

// handle an interaction between a pair of terrain features
//  $f_a$  - active feature
//  $f_p$  - passive feature
//  $e_i$  - disputed extent
handleInteraction(feature  $f_a$ , feature  $f_p$ , extent  $e_i$ ):
    // determine whether connection can and should be formed
    if share( $f_a, f_p, e_i$ ) then
        // interaction is handled through a connection,  $f_a$  and  $f_p$  share  $e_i$ 
        connect( $f_a, f_p, e_i$ )
    else
        // interaction is handled through a conflict
        if  $p_{\text{claim}}(f_a) > p_{\text{claim}}(f_p)$  then
            // passive feature loses extent  $e_i$ 
             $f_{\text{lose}} = f_p$ 
        else
            // active feature's claim for  $e_i$  is dismissed
             $f_{\text{lose}} = f_a$ 
        end if
         $f_{\text{lose}}.\text{extent} = f_{\text{lose}}.\text{extent} \setminus e_i$ 
        //  $f_{\text{lose}}$  has to restructure, avoiding to use  $e_i$ 
        restructure( $f_{\text{lose}}, e_i$ )
    end if

```

- feature interactions handling method is outlined in Algorithm 3, and operates as follows.

An active feature f_a makes a claim for extent e_a . The claim for e_a is temporarily granted. The claim is made once the structure of f_a has been generated by its specific procedure (see Section 3.2.2). At this point, no semantic objects have yet been placed, but the precise extent the feature will want to claim is already known. Note that the structure is not only generated initially when the terrain feature is first introduced to

the virtual world model, but also regenerated after changes to its specification.

The set of interacting passive features with extents overlapping e_a is found, and sorted according to priority p_{claim} . Each interaction is then handled in this order. Interactions with passive features with higher priorities are handled first, as these features will most likely have the highest impact on f_a 's claim.

Sharing the extent e_i (which is the overlap of the extents of f_a and f_p) is a cooperative solution to the interaction; this is, if possible, often preferable. For this, a connection between the feature types of f_a and f_p must have been defined. Of course, connections cannot be defined for every possible pair of feature types, as there might not be a sensible real world equivalent (e.g., between a lake and a forest).

Still, to be able to choose the connection solution, it must have a higher priority than a conflict solution. This may not be the case in specific scenarios where a given connection might be deemed too costly or impractical.

In order to incorporate the decision between connection and conflict in the algorithm, a generic function *share* is used. This function is defined in the following way:

$$share(f_a, f_p, e_i) = \begin{cases} \text{true,} & \text{if } p_{connect}(f_a, f_p, e_i) > threshold(f_a.type, f_p.type) \\ \text{false,} & \text{else} \end{cases}$$

$$p_{connect}(f_a, f_p, e_i) = \begin{cases} p_{connect}(f_a.type, f_p.type)c(f_a, f_p, e_i), & \text{if connection defined} \\ 0, & \text{else} \end{cases}$$

The priority $p_{connect}$ is composed of a fixed base value defined for the two feature types, modified by a function c . The result of this function is feature type and context specific. The resulting priority value $p_{connect}$ is compared to a threshold value defined for the connection type to decide whether to share extent, i.e., to choose a connection over conflict. Of course, this function could be elaborated to make it more context-aware.

The task of creating a concrete connection is specific to the type of connection and the types of the two features involved. Examples of connection objects that can be generated to form the connection include bridges, tunnels, road junctions, estuaries, etc. Furthermore, for some pairs of terrain feature types, several alternative connection types may be defined, of which one is chosen using a feature-specific decision mechanism. This choice can be changed upon user request.

If no connection solution is possible, the priorities p_{claim} for f_a and f_p are compared in order to determine which feature f_{lose} is losing the extent e_i . As these priority values are unique per feature type, p_{claim} induces an ordering among all feature types. As an example, in the current implementation of our prototype, we use the following (descending) ordering: landscape, lake, river, road, building lot, city, ditch, field, hedge, tree line, forest. However, the interaction handling methods do not depend on

this specific ordering. Furthermore, the defined ordering can be changed at any time, if so desired.

If the interaction is handled through a conflict, the lower priority feature f_{lose} has to restructure itself to its modified extent. It is up to the feature's generation procedure to decide how far it needs to restructure the feature. The best results are often obtained by performing a full regeneration within the updated extent. However, this can be somewhat costly. Therefore, for minor interactions, partial updates could be preferable. For instance, a conflict between a ditch and a forest, where the forest loses the overlapping extent, is handled through removing all trees within that extent, instead of regenerating the forest.

The interaction handling loop terminates when all interactions have been handled. However, it is aborted if, at some point, the extent granted to f_a becomes empty. In this case, the active terrain feature simply cannot be placed in the virtual world.

5.5 Example interaction scenario revisited

We now return to the four examples of interactions, introduced in Section 5.1, in order to analyse how they are handled in our framework. Figure 5.2 (a) shows the major road running through hilly landscape. In this scene, we see the result of an interaction between the road feature and the landscape. The road's path avoids steep slopes and connects to the landscape by forming an embankment. In Figure 5.2 (b), the road has to run across a forest, which is handled through a conflict where the forest feature loses part of its extent, clearing the path for the road. The following interaction, in Figure 5.2 (c), is handled cooperatively, as the road and river can share the interaction extent by forming the connecting bridge structure. Finally, the designer changes the landscape from hilly to mountainous, which has drastic consequences for the road, river and forest feature. Automatically, through restructuring, the consistency of the virtual world is restored (Figure 5.2 (d)).

5.6 Discussion

The main advantage of automatic consistency maintenance, introduced in this chapter, is that it removes a huge burden from the designer. Being freed from the responsibility of fitting all features together and keeping the world consistent, the designer can more freely experiment and make major changes to the layout of the world. Just as procedural generation can alleviate the amount of low-level modelling tasks for a designer, automatic consistency maintenance can relieve this designer from all the mundane corrections to be performed after most changes.

Furthermore, from a feature integration point of view, the *generality* of these methods has a clear advantage: introducing a new feature to the framework does not

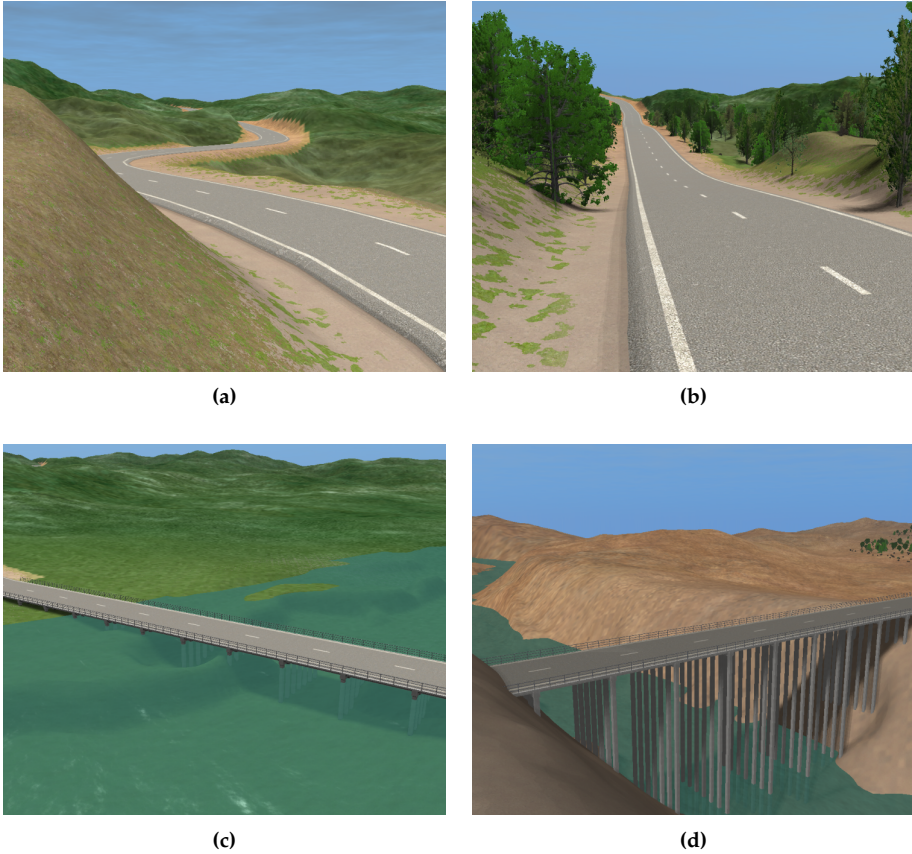


Figure 5.2: Results of interactions in Section 5.1: (a) road - landscape, (b) road - forest (conflict), (c) road - river (connection), (d) feature restructuring because of drastic changes to the landscape.

require designing interaction resolution methods for each of the framework's incorporated features. Instead, only a small interface has to be implemented, as discussed in the previous chapter, to let the feature handle e.g., a demand for restructuring. For a new feature, connections can optionally be defined. Although the mechanism for defining connections is generic, the connection object to be generated is logically specific for each pair of feature types, as it will vary in type per case.

The additional execution time for consistency maintenance is in typical modelling cases hardly noticeable. However, generating or removing a large, high priority feature with many dependent features can result in a cascade of many conflicts and connections to be resolved. This is especially the case for landscape - feature interactions, where a large change to the landscape (e.g., changing a valley to a moun-

tainous landscape) can have a drastic effect on a large set of features due to feature dependencies.

The amount of interactions that need to be handled can be alleviated by using proper heuristics (e.g., sorting interactions according to priority) and conflict avoidance. During procedural generation of a feature, conflicts can be avoided by querying the predicted outcome of a claim by that feature for a specific area, and avoiding that area if a conflict would arise. This querying mechanism can also be exploited to, for instance, obtain a convincing path for a road that avoids impassable areas, without explicitly defining in the road's generation procedure what kind of situations or features to avoid. Implementing conflict avoidance is not a requirement for a feature and its procedure to be successfully integrated in this framework, but, of course, it does increase the performance of consistency maintenance. Currently, we use conflict avoidance for some of the major terrain features, such as the city feature.

As we can use our interaction handling methods to detect and predict the effects of modelling operations, it could be interesting to identify operations that have a very large and potentially undesirable effect on other features. This information could then be used to inform or warn the designer performing the operation. Of course, such operations are not destructive, as a designer can always restore the previous state by undoing the operation afterwards (see Chapter 6).

In a few cases, our generic methods might result in a non-optimal handling of an interaction, in terms of efficiency. This means that a dedicated procedure for handling that specific interaction scenario would involve less operations and, as a result, less computation time. However, such an ad-hoc consistency maintenance implementation would require a specific procedure for each combination of pairs of all n feature types, resulting in $n \times n$ consistency maintenance procedures, which would seriously hinder the integration of new features and procedures in the framework.

The decisions resulting from our interaction handling methods are *binary*: either the extent is shared or not, in which case one of the feature loses the extent. This can be seen as a minor limitation, as it does not support a smooth transition between conflicted features, which could perhaps be introduced if we were to define a gradient along the conflicted extent. However, in practice, this problem is not really noticeable. Transitions between terrain features are, if desired, typically implemented in the feature's generation procedure. For instance, close to the border of a forest's extent, the tree density declines, regardless of whether the border is part of the original forest outline or results from a conflict.

An inherent limitation of our consistency maintenance mechanism is that it can only deal with interactions in 2.5D. This is a direct result of feature claims being defined as 2D polygons, and to some extent, of the structure of the landscape layer as a height field. This limitation currently excludes consistency maintenance of e.g., complex, multi-level underground structures. In practice, this is rarely problematic or limiting; 2.5D is the *de facto* standard in many domains, for instance GIS-based virtual world modelling. A straightforward extension to address this limitation is

to define several discrete layers of landscape, of which a feature can claim extent. A more generic solution would be to extend the structure of claims from 2D polygons to 3D bounding volumes.

Although designers can influence the consistency maintenance mechanism by manually setting claim priorities, either per feature type, or by overriding e.g., the priority of a specific feature instance (e.g., a historical wood within a city), we think this level of control is too coarse to accommodate all desirable modelling scenarios. The next chapter will discuss more fine-grained mechanisms for constraining the automatic consistency maintenance, including the ability to protect features or objects from modifications.

6

User control in procedural modelling

The previous chapters introduced important aspects of our framework: its semantic model for virtual worlds (Chapter 3), the structured integration of procedural methods (Chapter 4), and the automatic consistency maintenance mechanism (Chapter 5). However, its facilities for user control have not yet been adequately addressed.

This chapter identifies and addresses several challenges and open issues related to the integration of user control in procedural generation. One of the main obstacles for introducing procedural generation into mainstream virtual world modelling is that, so far, procedural methods offer designers either inadequate or little control to specify their intent. Our approach, aimed at bridging this gap between procedural and manual modelling, is to introduce a variety of modelling operations that are neither fully procedural nor conventional low-level manual editing. We present these facilities, categorized according to different levels of granularity. As with all automated processes, there is a tension between user control and automatic consistency maintenance. Addressing this tension, we discuss facilities for designers to influence the automatic process, so that they can limit unwanted restructuring of generated elements of the virtual world. We conclude the chapter by summarizing the limitations on user control in our framework, and identifying some promising directions for further research.

6.1 Levels of modelling granularity

When considering to use either procedural methods or manual modelling, currently a designer has to choose between high productivity and full user control. However, both productivity and user control are desirable for modelling a virtual world. Still, the level of granularity of user control that is required depends on the phase in the modelling process, or the scale on which a designer is operating. At the start of a new project, designers are defining the lay of the land in broad terms. At this point in the modelling process, fine-grained user control is not necessary and becomes more of a hindrance. Modelling exact details is not important, as designers are still in the process of refining their idea or vision on the virtual world. In fact, procedurally generated results can serve as suggestions or inspiration. What is important is that modelling operations are fast and have a short feedback loop, so that a designer can iterate on the design and try many alternatives before committing to one.

Once the basic layout of the world has been decided, it will be refined until it meets the requirements, both functional (i.e., related to gameplay or training goals) and aesthetic (related to the look and feel). Typical aesthetic refinements operate on a very small and detailed scale, and need fine-grained user control. Functional refinements, however, are often more high-level and can be done efficiently using more coarse-grained tools. Therefore, procedural methods are not only useful for generating and filling an initial model of the virtual world, but also in later refinement steps.

During the iterative modelling process, there are often certain high-level requirements that need to be preserved. These requirements can, for example, relate directly to gameplay objectives or stem from the training scenario for which the virtual world is being designed. It would be cumbersome to preserve such high-level requirements with fine-grained tools. These requirements would ideally be maintained automatically.

It seems clear that designers should not have to face a dilemma between productivity and user control, i.e., relinquishing all control to use procedural generation, or sacrificing productivity in order to have complete control. Each phase in the modelling process necessitates its own level of user control, and in many phases procedural generation can be used in some form to increase productivity as much as possible. For effective modelling of virtual worlds, a mix of facilities is necessary, operating on different levels of granularity.

For a designer it is, of course, most efficient to work through the modelling phases in a linear fashion, starting with specifying the high-level layout for the virtual world and ending at polishing and tweaking individual 3D models. However, virtual world design is a creative process. Consequently, the desired end result is not envisioned in detail beforehand and will repeatedly change as new insights and ideas come up during modelling. Therefore, it should be possible for designers to interleave actions from different levels.

Considering the modelling process outlined above, we discern several relevant

levels of granularity on which a designer can exercise control over the procedural generation of a virtual world:

Macro At the macro level of modelling granularity, designers define requirements that should hold regardless of what features the virtual world is composed of. Designers declare high-level requirements by defining what kind of constraint should be maintained within an area of the virtual world. An example of such a constraint is a line-of-sight, which requires a certain feature in the virtual world to be clearly visible from an observation location. Designers define these constraints once and expect them to be automatically maintained throughout the modelling process. In case of a line-of-sight, any obstructing feature that is introduced in the extent is automatically influenced by the constraint.

Coarse At this level, designers specify large scale terrain features such as mountain ranges, rivers and cities. Terrain features are specified by outlining their coarse shape and by setting a limited number of intuitive, semantic feature attributes. These attributes hold uniformly throughout the feature's extent, as discussed in Chapter 3. Based on this specification, a specialized procedure incorporated in the framework, is executed to generate a matching terrain feature, composed of semantic objects. Further editing at this level entails either changing the shape or position of the specified outline, or setting a different value for one of the feature attributes. These high-level changes are addressed by regenerating the feature based on its modified specification. Indirect changes to the feature can also be a result of interactions with other features, causing a restructuring of the feature or of its connections to other features.

Medium At this level, designers further refine their specification of intent for a particular terrain feature. These relatively large refinements to features typically involve a substantial amount of changes. Feature refinements are also specified quite coarsely, leaving to the generation procedure the responsibility to further detail its effects. A refinement is specified by outlining a sub-area of a terrain feature, and by denoting a new value for a semantic attribute within that area. This attribute is interpreted by the feature's generation procedure to locally update the feature to match with the refined intent.

Fine The fine level deals with refining individual semantic objects within a feature. Editing on this level involves displacing the object or modifying one of its attributes. This level typically involves little procedural generation. The main challenge here consists of preserving the fine-grained edit operations, where possible, whenever the corresponding terrain feature is regenerated.

Micro On the lowest level, designers manipulate geometric meshes, assign textures to 3D models, etc. This is the level on which most manual modelling systems operate.

As discussed above, it is important for a designer to be able to freely mix these levels throughout the modelling session. For this, the consistency maintenance mechanism presented in the previous chapter is very helpful, continuously keeping the model in a valid state. However, at the finer levels of control, it can sometimes be necessary to limit the influence of automatic consistency maintenance.

For any level, a short feedback loop is required to support iterative modelling, meaning that even high-level procedural operations with a large scope of application should run reasonably fast, i.e., nearly interactive. As mentioned in Chapter 4, this puts some limits on the kind of operations that can be supported within this framework.

In the remainder of this chapter, we will discuss how these levels of user control are featured in our framework for declarative modelling of virtual worlds. Most of the mentioned levels are fully implemented in our SketchaWorld prototype, described in the next chapter. However, some important research issues remain open in this area. At the end of the chapter, we identify these and indicate directions for continuing the research on user control in procedural modelling.

6.2 Declaring and maintaining high-level intent

On macro level of control, we focus on capturing *high-level designer's intent* (this section is in part based on [Smelik 11a]). An example of such intent is to have a clear line of sight between two locations in a virtual world. In the design of entertainment game worlds, the purpose could be to have a *vista point*, where players have an impressive view on the city they are going to visit next. In a serious game for e.g., military training, it could be to have a suitable overwatch position on a hill to support a friendly unit on patrol in the valley.

In current manual modelling systems, such intent cannot be made explicit. As a result, a designer has to manually preserve the intent throughout the modelling session, which makes experimentation or exploration of alternatives more cumbersome. In the context of procedural generation of virtual worlds, this kind of declarative intent is not only to be *captured* and translated into procedure parameters, but it also needs to be automatically *maintained* throughout the modelling process.

In Chapter 2, we already reviewed some noticeable prior work controlled generation of content. Examples of such research include controlled generation of elevation profiles, e.g., using 2D user-drawn imagery [Zhou 07] and height-map elevation constraints [Stachniak 05]. A more recent method introduces guides to constrain procedural structures based on L-systems [Beneš 11b]. The main drawback of these solutions is that they are not easily extensible to other procedural techniques or different features of the virtual world.

In our framework, we manage high-level intent as *constraints* imposed on the generation procedures. Maintaining intent by means of constraints has already been



Figure 6.1: A *line of sight* constraint for a bridge with visibility obstructed by a city and a forest: (a) overview of situation before, (b) after evaluation of the *line of sight* constraint composed of two feature constraints, resulting in a clear view on the bridge.

successfully applied in different fields within procedural generation, for instance, in the 2D platform game level generator of Smith et al. [Smith 10].

In this section, we introduce an extendable mechanism capable of expressing and automatically maintaining high-level intent over a specified area of the virtual world and all terrain features within this area. The intent is expressed through the concept of *semantic constraints*. Examples resulting from the application of semantic constraints to virtual worlds include tight mountainous passageways forming choke points, lookout spots with an unobstructed line of sight over a designated area or valleys with limited access, all of which can have great impact on e.g., gameplay or training value. Furthermore, by supporting constraint composition and context awareness, semantic constraints enable designers to express their high-level intent in an accessible way.

6.2.1 Composing semantic constraints

A semantic constraint is a control mechanism imposed on the generation process in order to satisfy explicit designer’s intent over a specific area. We use the concept of an *extent*, as we defined for terrain features in Chapter 3, to denote this specific area of the virtual world. A semantic constraint adapts to the current context of its extent, i.e., the local terrain and nearby features. The constraint is re-evaluated when the terrain is modified or whenever a new feature is introduced within the constraint’s extent. Because of this, the virtual world remains consistent with the designer’s intent. As a result, designers can start with specifying the high-level requirements of their world and provide additional detail later on.

To define its behaviour, a semantic constraint can be composed of several sub-constraints, called *feature constraints*. Semantic constraints are abstract, high-level constructs, which convey the vocabulary that is directly used by designers to express their intent. Depending on the context of the extent, semantic constraints automatically apply a subset of their feature constraints.

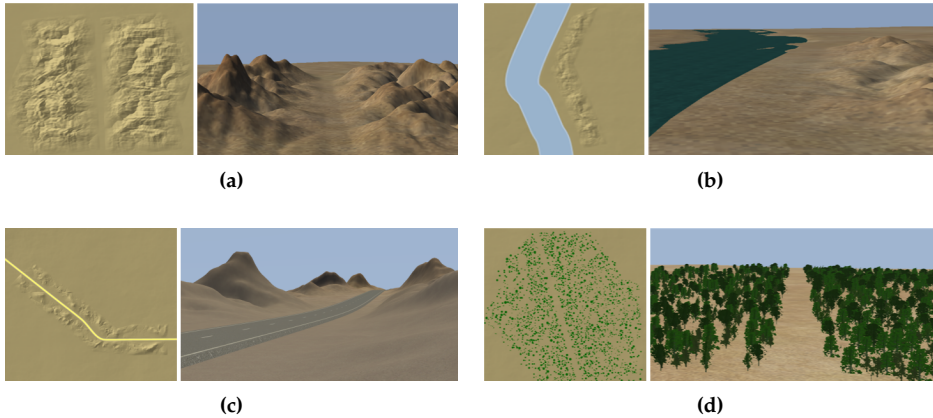


Figure 6.2: Results of different application schemes of a *choke point* in the context of: (a) bare terrain, (b) a river, (c) a road, and (d) a forest.

Feature constraints are specialized to operate on a single type of feature, such as a city or a forest. They are mapped to low-level operations to achieve a specific result, like limiting the height of vegetation within a designated area. The feature constraints of a semantic constraint are independent of each other, but, together, are configured to fulfil the common goal.

To give some insight in the composition of constraints, let us consider a *line of sight* constraint applied to a virtual world. The evaluation of this semantic constraint can affect terrain, vegetation and urban features, if present within the constraint's extent. Figure 6.1 presents a concrete example of a *line of sight* constraint, where the *line of sight* constraint is composed of two feature constraints that affect the generation of the city and the forest.

6.2.2 Constraint evaluation method

Context detection is the analysis of a constraint's extent to derive a specific application scheme. An *application scheme* is a set of instances of feature constraints, suitable for the context (see Figure 6.2). As a semantic constraint is aware of which feature constraints it is composed of, context detection embeds the process of deriving an application scheme in the semantic constraint. Each instance of a feature constraint in such a scheme is linked to a single terrain feature. This enables the feature to inform the constraint regarding changes to its state.

An *association relationship* is a relation between a semantic constraint and features that are not necessarily in its extent. This relation links features that are not the object of constraint application, yet provide context. As an example we describe a *route*

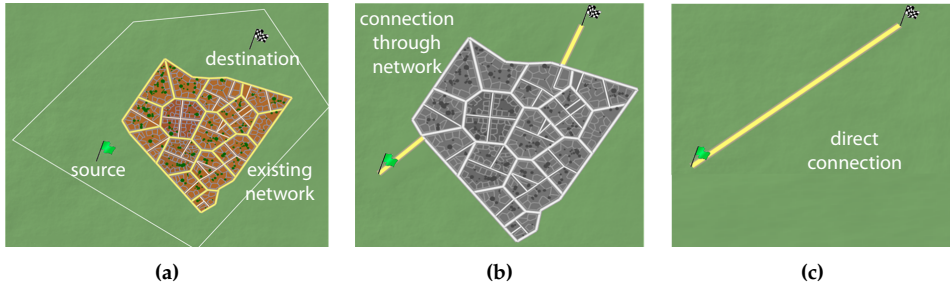


Figure 6.3: Constraint association of a *route constraint*: (a) *route definition* between two locations (flags), (b) association with existing network, (c) removing this network results in a direct connection.

constraint, which enforces that there is a route between two locations in the virtual world. For this, the evaluation of this constraint takes the existing road network into account (see Figure 6.3). In case there is no connection in the area, or the local road network provides only part of the route, the constraint evaluation has to introduce as many new roads as needed to connect the two locations. Therefore, the existing road network is in association with the route constraint. Using this association relationship, the constraint is notified in the event of removal of any existing roads, resulting in a re-evaluation of the context and proper handling of the situation.

The evaluation of a semantic constraint can result in an application scheme consisting of numerous feature constraints. As a result, for a given terrain feature, several semantic constraints can impose multiple low-level constraints. To manage all these different feature constraints, a feature maintains a stack of applicable constraints (see Figure 6.4). By mapping the constraints in this stack to corresponding operations, we obtain a sequential list of operations that are performed during the generation of the feature to satisfy the semantic constraints. Through a sequential analysis of a feature's constraints, we check their consistency and handle any conflicting constraints. This analysis can result in changing a feature constraint's parameters or cancelling its current application.

An important aspect of the consistency analysis is handling interactions between semantic constraints. We reuse our generic method for feature interactions, described in the previous chapter, to handle constraint interactions in the same manner. As such, for the interaction handling process, priorities have been defined for each type of semantic constraint. In case an active constraint issues a claim for extent that overlaps with a passive constraint's extent, the constraint's type priority value determines whether the claim is granted. An example of this mechanism can be seen in Figure 6.5, where a *line of sight* constraint is placed over an existing *choke point* constraint. The *line of sight* constraint claims an extent overlapping with the *choke point*'s extent. As it has a higher priority than the *choke point*, the claim is granted. The *choke point* constraint adapts to this by reapplying itself to its modified extent, resulting in a consistent

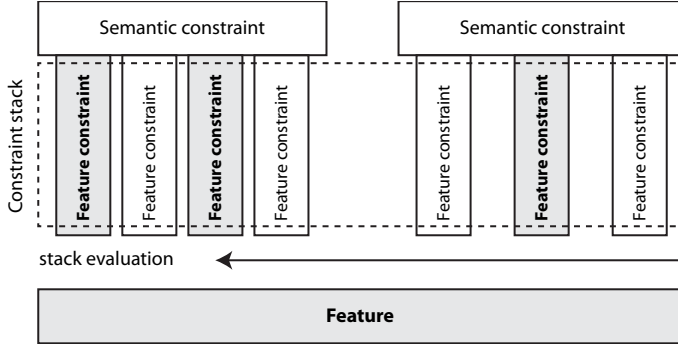


Figure 6.4: Schematic view of the creation and evaluation of a constraint stack.

coexistence of the two. Note that the outcome of this interaction is what we described in the previous chapter as a *conflict*. The concept of *connections*, defined as the other possible outcome of an interaction, can also apply here. Two compatible constraints could share extent by defining a connection that adheres to the requirements of both.

As an example of a semantic constraint, we give an outline of the implementation of the *line of sight* constraint. Based on the input observer and observation locations, we calculate a *view plane*. This view plane consists of all required lines of sight, starting at the observer, which have a view on the observation area, essentially providing a threshold height for each location within the constraint's extent. To enforce the line of sight, we need to modify the height of both the landscape and all features in it. For the landscape profile, we calculate a scale factor s as $\min(\frac{H(x,y)}{h(x,y)})$, where $H(x,y)$ is the threshold height value of the view plane and $h(x,y)$ is the original elevation value at that point. Scaling the elevation in such a way induces unnatural transition artefacts; we therefore use a blending approach to create a more smooth transition. The blended result is defined as $H'(x,y) = \text{lerp}(s * h(x,y), h(x,y), d(x,y))$, where $d(x,y)$ is a linear interpolation factor based on the distance from the direct line between observer and observation area.

6.3 Declaring the features of the virtual world

The main method of interaction with our framework is designated *interactive procedural sketching* (this section is in part based on [Smelik 10c]). This interaction method provides designers with user control on the identified coarse level, at which designers declare the features of the virtual world. The most important requirements for procedural sketching are ease of use and modelling speed, although they have to be balanced with precision and control. One of the solutions is to provide a *short feedback loop*, where designers can quickly examine the results of their modelling oper-

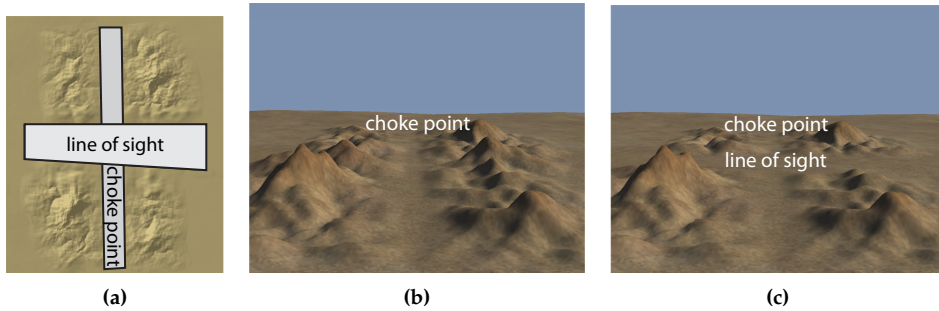


Figure 6.5: Constraint priorities: (a) 2D view of areas granted to a *choke point* and *line of sight* constraint, (b) before and (c) after specification of the *line of sight* constraint on top of the *choke point* constraint.

ation and act accordingly. The feedback loop is combined with support for iterative procedural modelling, allowing any procedural operation to be undone and redone while guaranteeing the exact same outcome. Procedural sketching allows designers to interactively build up the complete virtual world by sketching the coarse outlines of its features. In this section, we describe this interaction method and its interactive workflow in detail. In the next section, we introduce feature refinements to support more fine-grained precision in specifying the features of the virtual world.

6.3.1 Procedural sketching

Procedural sketching provides designers with easy to use tools to declare their intent. This is mainly achieved by creating a 2D digital sketch; i.e., a rough layout map of the virtual world. Procedural sketching provides two interaction modes: landscape and feature modes. This distinction is made because it is convenient and natural to specify the landscape and features in a different way: defining a landscape's elevation and soil material profile both match a raster-based approach well, while features typically have sharp contours and, as a result, are better specified by a vector drawing.

Landscape mode

Designers paint a top view of the landscape by colouring a grid with ecotopes (an area of homogeneous terrain). These ecotopes encompass both elevation information (elevation ranges, terrain roughness) and soil material information (sand, grass, rock, etc.). The grid size is adjustable and the brushes used are similar to typical brushes found in image editing software, including draw, fill, lasso, magic wand and transition pattern brushes (e.g., from ocean to shore).

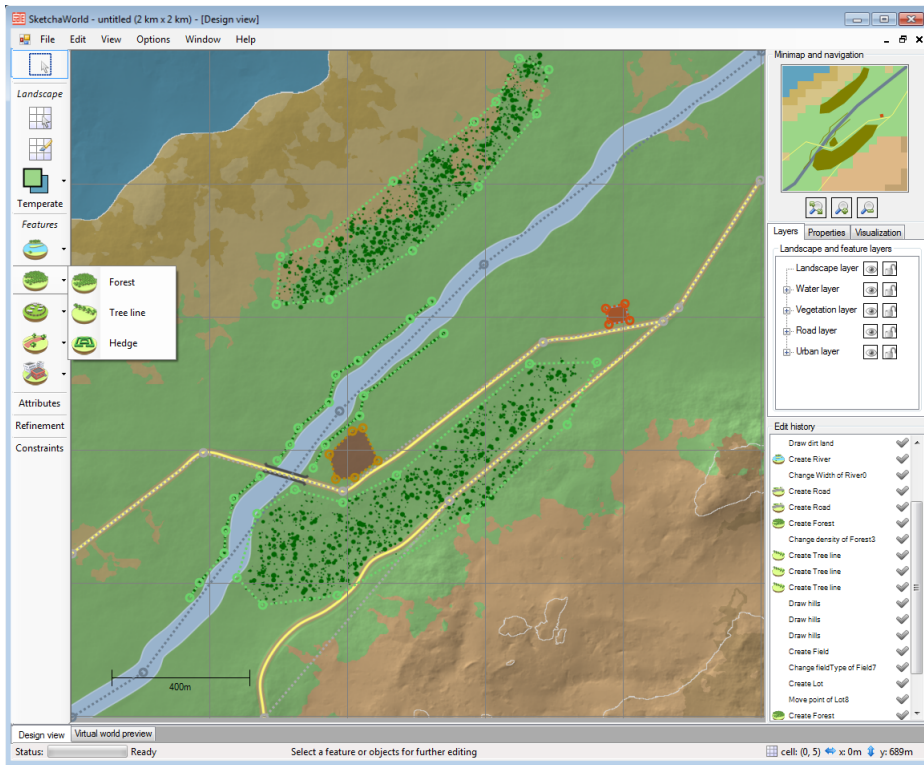


Figure 6.6: User interface for procedural sketching (feature mode), also showing editing tools (left hand) and navigation, layers, and edit history (right hand), as implemented in the SketchaWorld prototype.

Feature mode

Designers specify features like forests, lakes, rivers, roads, and cities on the landscape using vector lines and polygon tools. This resembles vector drawing software: placing and modifying lines and polygons is done by manipulating control points.

6.3.2 Iterative workflow

Each sketched feature specification is procedurally expanded to a corresponding terrain feature, using the integrated procedural methods associated with this feature (see Chapter 4). To directly see the effect of an edit action on the virtual world model (e.g., drawing ecotopes, rerouting the path, modifying the shape of a feature, removing a feature), users sketch on a 2D top view of the generated virtual world. This view updates immediately as new results are generated. Depending on the interaction

mode, an overlay is displayed representing relevant elements of the design. Figure 6.6 shows the user interface for procedural sketching in feature mode, as implemented in our prototype. By keeping the user interface and interaction modes simple and clear, we strive to make procedural sketching accessible to people without special modelling expertise.

A *short feedback loop* between a designer's edit action and the visualization of the generated results is essential to allow designers to model virtual worlds iteratively. This requires each edit action to be executed separately and the results of an action to be displayed immediately. Such an interactive setup allows designers to quickly see the effect of their edit operations and work towards the desired end result.

Several challenges have to be overcome in order to provide designers with an interactive and iterative workflow. Although improvements in hardware and new approaches such as GPU computing significantly alleviate the execution time of procedural methods, operations affecting large regions or requiring complex algorithms (e.g., city generation) may still execute at non-interactive rates (e.g., several seconds). Therefore an asynchronous setup is implemented, explained in the next chapter. It separates the *user interaction* and the actual *execution* of edit actions. This allows designers to continue working on the virtual world without being hindered by the execution of complex procedural operations.

The ability to undo and redo any modelling action is one of the other main requirements for an iterative modelling workflow. For this purpose, the familiar edit history is provided. Because of memory constraints inherent to modelling large virtual worlds, it is far more efficient to implement undo and redo by (partial) regeneration, instead of storing all intermediate modelling states. At the expense of some additional time recomputing the previous state, designers are provided with unlimited undo and redo facilities. Furthermore, because designers expect regenerated results to be exactly the same as before when redoing an action, we need to carefully manage the sequence of random numbers generated by procedural methods. The state of the random number generator (i.e., its starting seed and position in the random sequence) is saved and restored for each edit action that involves procedural operations.

As it is the main interaction method for declarative modelling, interactive procedural sketching is often used for defining the largest part of the virtual world. Its level of granularity is fine enough for designing all major features in the virtual world.

In order to test the usability of procedural sketching, we organized a number of informal sessions with Dutch game design professionals during the course of the research project. In these sessions, after a brief introduction of our research, we invited them to experiment with the prototype and to provide feedback. They were enthusiastic about our approach and saw potential in its use for rapid prototyping of game worlds. However, they consistently desired finer-grained control to be offered alongside procedural sketching. Their feedback led us, among other things, to further improve our interaction methods with a new level of user control, which we call feature refinements.

6.4 Refining feature specifications

When declaring a feature using procedural sketching, a designer can set a specific value for any of its semantic attributes (see Section 3.2.1). Such a value is intended to hold uniformly throughout the feature, although it is certainly possible for a procedure to introduce some variation on this attribute in the generated result. An example of this is a river feature's *width* attribute. As we saw in Section 4.1.2, the procedural generation method takes the specified width as a base value, on which it introduces slight variations, e.g., depending on the local slope.

Setting a semantic attribute of a feature specification can be considered a somewhat coarse measure; for example, setting a forest to be densely populated results in a similarly thick forest throughout. Often a designer may want to be more precise in the specification of intent, for instance, to create an open spot within this impenetrable forest.

Feature *refinements* allow designers to specify areas, within a feature's extent, in which they want to provide a different local setting for one of the feature's semantic attributes. Examples of possible applications of feature refinements include the creation of a ford in a river at a specific location, or modelling a zone within a forest where mainly young (newly planted) trees grow.

Perhaps the most valuable aspect of a refinement is that it becomes part of the feature's specification, instead of just being applied as a post-processing step on the generated feature. One of the advantages of this is that the refinement is preserved even if the corresponding feature is regenerated, for instance, because it was moved or its outline was modified. In such a way, feature refinements provide user control on a *medium* level of granularity. A designer is still specifying what a procedure should generate, instead of directly manipulating the generated content.

By definition, the shape on which the feature refinement operates is a subset of the feature extent. For most refinements, this comes down to an interval of a polyline, or a polygon. Figure 6.7 shows two shape variants of feature refinements. By defining refinement shapes relative to the origin of the specification's outline, their shape is preserved when the corresponding feature specification is transformed, e.g., translated.

The transition of an attribute's value is not always abrupt, as it could need a smoother change in order to yield plausible results. For this, we can define a *fall off* range and function per semantic attribute type. For instance, for an open spot in a forest, we typically want an abrupt change, while for a change in vegetation species, the transition should be more gradual.

During generation, the procedural method associated with a feature queries the values of the feature specification's semantic attributes to steer its algorithm to produce results matching the intent. To ease the definition of feature refinements, we encapsulated the evaluation of feature refinements in the feature specification. As a result, minimal changes are required for existing procedural methods to support refinements. A procedure queries the feature specification for the local value of a

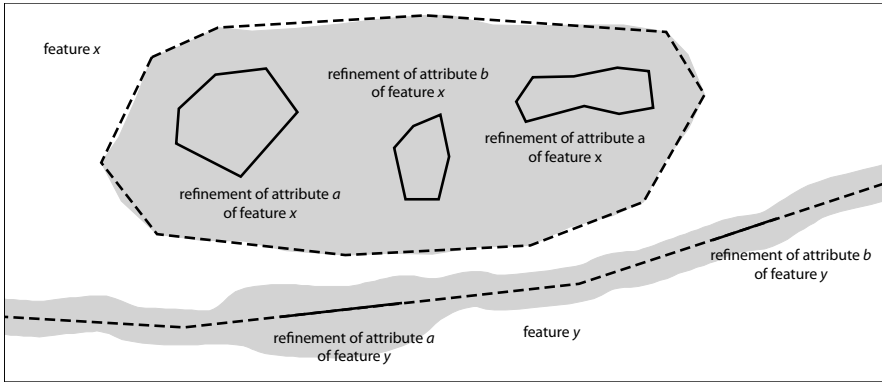


Figure 6.7: Area and line-interval refinements (solid lines) defined on features x and y (specifications in dashed line).

semantic attribute at a *specific location* within the feature extent. The evaluation of such a query takes all relevant refinements and their transitions into account. Among other examples, this makes it convenient for a river generation procedure to determine how the river's width should vary along its course, or for a forest generator to determine the local density the forest should have.

To determine the value of semantic attribute x at location l , for which a refinement r has been defined within the vicinity of r , we first compute the *fall off* f . The fall off f is in the range $[0 \dots 1]$ with 0 denoting outside the refinement r and 1 denoting within the refinement extent. We obtain a relative distance d based on l and r 's fall off range. Evaluating d in the fall off function set for r results in the fall off f .

We can use the fall off f in several ways, depending on the type of semantic attribute x . If the attribute is a numerical value, we can simply interpolate between the value set for r and the feature's uniform value for x . In other cases (e.g., vegetation species), we use f as a probability the refinement r will be chosen over the uniformly defined attribute value.

Figure 6.8 shows an example of two feature refinements applied to a forest feature. The first refinement adjusts the *density* attribute from relatively dense to an open spot. The second refinement adjusts the local vegetation species from a default collection of deciduous trees to only Pines and Spruces. The example shows how easy it is to perform feature refinements to create local variation in a feature.

The novelty and main advantage of the feature refinements discussed in this section is that they offer a new way to specify intent at a medium level of granularity, where one is still operating on the intent specification instead of on modifying generated results. As a result, it combines very well with integrated procedural methods, which can typically support these refinements with minimal changes to their implementation. All in all, feature refinements offer designers a new and convenient level of user control.

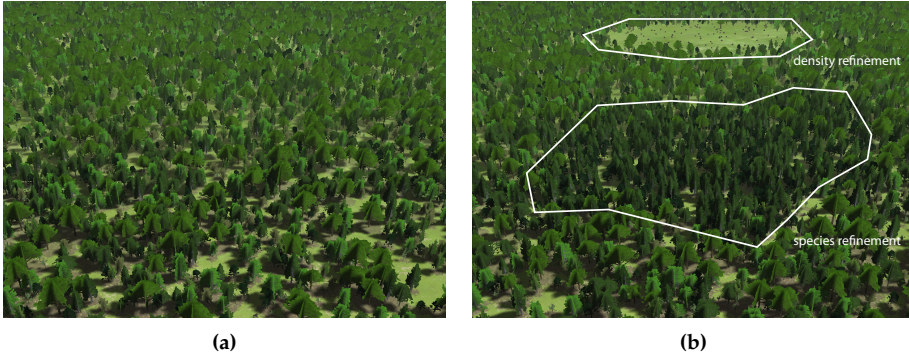


Figure 6.8: Example of the application of two feature refinements to a forest feature, with *density* set to “open” and *species* to Pines and Spruces: (a) uniform density and species, (b) refined density and species.

6.5 Balancing user control and consistency maintenance

Automatic consistency maintenance, as described in Chapter 5, enables designers to model a virtual world in a more efficient and accessible way. However, as with more or less all automatic mechanisms, it limits user control to some extent. Especially in the context of fine-grained edit operations, these limitations on user control can become problematic, to the point where designers are no longer able to fully realize their intent. In [Smelik 10b], we summarized the main issues that arise when mixing high-level procedural operations with more fine-grained edit actions, and the tension that exists between user control and automatic maintenance procedures. One of the avenues of further research indicated there is the concept of *locking* an element, preserving it from any modification.

In this section, we describe how this locking concept, which is a well-known and extensively used operation in vector graphics and image editing software, can be applied to procedural modelling of virtual worlds. We discuss how locks can be employed in order to enable more control over automatic consistency maintenance, and what are some of their inherent difficulties and open issues.

6.5.1 Element locks

In this section, we present two (partial) classifications of locks. The first one according to *scope*, i.e., what aspects does a lock entail, and the second one according to *target*, i.e., to which elements of the virtual world can it be applied.

Intuitively, if one locks an element, one would expect it to be impossible to modify this element, in any way. However, as locking is a concept that stems from manual modelling, the realization of such a lock is exclusively aimed at disallowing any user

modifications on this element. In the context of procedural modelling, a lock can have more aspects involved and is, therefore, far more complicated. For example, a lock on an element could entail any combination of the following aspects:

1. manual and procedural user edit actions on this element are prohibited;
2. the extent of the element cannot be modified in any way;
3. no new connections with this element are allowed (see Section 5.2);
4. the landscape within the extent of this element cannot be modified in any way.

Not all combinations of the possible lock aspects described above, will always make sense or be useful for a designer; for some of these combinations it is not even possible to ensure the consistency of the virtual world model. Here, we describe several types of partial locks that we consider valuable and helpful in the context of designing virtual worlds. We selected these partial locks based on the usefulness we foresee for using this partial lock scenario within our approach, as well as the intuitiveness for designers to understand what the effects of such a partial lock are.

1. *Full lock* - Lock all aspects, resulting in an element that cannot be modified in any way.
2. *User edit lock* - Prohibit any changes made by a designer to the locked elements, but allow possible regeneration.
3. *Consistency maintenance lock* - Although designers can still directly modify the element, it will not be changed by any automated process.
4. *Cooperative lock* - The element is locked for modifications, but still accepts connections to other elements.

In the context of our semantic model for virtual worlds, we could consider the following elements as suitable candidates for locking:

1. an *area* of the virtual world, including the landscape and all features and objects within that area.
2. a complete *terrain feature* (e.g., a river);
3. the *landscape*, possibly restricted to a certain bounding area;
4. a (set of) *semantic object(s)* belonging to one or more features (e.g., some trees, buildings or streets);

Area locking

An area lock locks all features, objects and the landscape within the given area of the virtual world. A specific complexity of an area lock is caused by the fact that it can contain a subset of a feature's extent, which is then locked, while the rest of the extent is free to be regenerated or modified. To preserve a plausible world, zones that provide a natural transition along the boundaries of the locked part of a feature are especially important in the context of area locking. We discuss transition zones in the next section.

Feature locking

A feature lock implicitly enforces this type of lock on all its objects, to preserve its currently generated structure. One might expect a feature lock to be less complex than an area lock, as a feature is a discrete element, and requires thus no transition zone. However, this is not necessarily the case. Firstly, a feature may have already formed connections with other features. Secondly, a feature may have lost extent to other features, which are not necessarily locked as well, leading to problematic scenarios. For instance, consider a road that crosses a river. It has formed a connection with this river using a bridge. Imagine this road feature is fully locked, and the river is removed. Obviously, the connection needs to be removed as well, but this would mean a change to the road feature. Not only is the bridge replaced by a road segment, the road's path was modified locally to align with the bridge (i.e., to have a straight ramp onto the bridge). Now that the river is gone, this modified path might no longer be sensible.

As a second example, consider a river flowing through a city, that is split up into two clusters (i.e., the city feature lost part of its extent to the river). If the river is removed, and the city, cannot procedurally restructure because of its locked state, an empty strip of land in between the two city clusters would be left in the former river bed.

A straightforward approach to deal with this issue is to let a lock on a feature entail a lock on all associated features. In such cases, however, using a feature lock becomes very restrictive for a designer, strongly diminishing its usefulness.

A better approach is to view a feature lock as a variant of the area lock, where the locked area coincides with the extent the feature originally intended to claim, thereby also encompassing the extent lost in conflicts with associated features. This prohibits associated features to be removed and locks only the required areas of these features. In the city example, the river's path through the city would be locked, whereas changes to the river outside the city would be still allowed. Note that, as for area locks, the landscape is also partially locked within the feature's extent.

Landscape locking

An area of the landscape can be locked for modifications using the *landscape lock*. Any landscape mode editing within the locked area is prohibited. To ensure continuity in the landscape profile, a transition zone has to be established around the locked landscape area. Again, following the same reasoning as above, we can see this as a variation of the area lock. A cooperative landscape lock would still allow new features to be placed within this area by forming connections.

6.5.2 Transition zone

If we were to keep an element of a virtual world locked, while other elements closeby change because of user modification and regenerations, unnaturally sharp, drastic or mismatching transitions would likely occur between the locked element and its surroundings. This means that the problem of locking does not only involve preserving some elements throughout a modelling session, but also to always guarantee a smooth and plausible transition between them and the surrounding unlocked content.

Within this transition zone, unlocked elements close to the locked elements will have to connect or adapt in such a way that the transition is natural and continuous. Of course, what is considered natural depends on the type of element that is locked. For instance, a locked area of landscape will require a continuous elevation profile within the transition zone, and a locked partial road network will require plausible connections to its surrounding network. In a general way, we can consider the transition as a zone where the *lock strength* (i.e., a value denoting whether the element cannot change, might change or should change) smoothly falls off.

An important category of transition zones is the transition *within* a feature, resulting from a lock applied to part of this feature. An example of this is an area lock that partially overlaps a feature's extent. In this case, part of the extent of a feature is locked, while the rest of its extent could be modified or regenerated.

A transition zone within a feature can be seen as a *generalization* of the fall off defined for feature refinements. Not only should such a transition zone result in a gradual change in a feature's semantic attributes, it should also preserve the *connectivity* within the feature in a plausible way. The difficulty of this type of transition zone is that the desired form of transition varies per feature. For instance, a transition zone within a forest entails a gradual change in density, species, age, etc. from one part to the other. For a river, the transition zone acts as a buffer to create a plausible connection (curvature, width, depth) between the locked river course to the unlocked course. Within a city, partial road networks need to match up within the transition zone, but also a plausible transition in building types, style needs to be provided.

As follows from these examples, for each type of terrain feature, one needs to devise and implement a non-trivial procedure to generate a plausible transition zone. Previous work has provided a number of dedicated solutions for city road networks [Aliaga 08, Lipp 11], but, for other features, no solutions exist to date. This hinders

the applicability of area locking in procedural modelling of virtual worlds.

Fortunately, reoccurring patterns exist in the transition zones for the different features. A first category are feature generation procedures that can be seen as *object placement* problems. For this category, a transition zone entails a plausible mix of locked objects and newly generated objects. In general, we can use a fading lock strength as a gradient to obtain such a mix. Of course, each feature has its own specific set of requirements and constraints that need to be fulfilled for the generated object distribution, such as the competition model for vegetation in a forest.

A second category are features that can be abstracted as a network of *connections*. For this category, a transition zone is essential to obtain plausible connectivity between the locked part of the network and the regenerated network. In general, this can be seen as a graph merging problem, which has been successfully shown for city road network in the recent work of Lipp et al. However, features again have very specific requirements on what constitutes a plausible transition. For instance, a major road or a river has many additional constraints regarding curvature and its elevation and lateral profile. This means that the mentioned graph merging approach would not be successful here.

Although we identified several common properties here, the specific requirements posed by features have forced us to make our current implementation to a large part, feature-specific. We consider it is interesting and important to continue this research in order to find more generic, i.e., less feature- and procedure-specific, solutions to the problem of generating transition zones. The scope of the research should be further extended to the interpolation of any combination of procedural models, generated by different techniques. For this, generic interpolation schemes could be devised by analysing the common properties and reoccurring patterns in procedural models. Such schemes would make it possible and convenient to define and maintain transition zones *between* different features and objects.

6.6 Discussion

One of the main obstacles for getting procedural techniques into mainstream virtual world modelling is that they offer designers either little or inadequate control to specify their requirements. In the early days of procedural modelling research, the concepts user control and procedural generation were almost mutually exclusive: for quick results, a procedural method could be used to generate vast amounts of content, but if one also wanted to have control over this content, manual modelling was the only alternative. As a result, designers for the most part still rely on conventional modelling systems, which require enormous manual efforts, but at least offer proven editing facilities to experiment with.

In the last few years, research in procedural modelling has focussed more and more on the integration of user control in procedural methods, resulting in a variety of novel, partial solutions to this problem (see Chapter 2). However, the goal of a

hybrid modelling method that combines the strengths of procedural and manual approaches is not yet fully realized, and will require more research attention. We expect that as user control in procedural modelling continues to improve, so will its practical application in mainstream virtual world modelling.

In this thesis, we have made a contribution to this ongoing research with new levels of user control and intuitive interaction methods. According to their level of granularity, we identified a variety of modelling operations that are neither fully procedural nor conventional low-level manual editing. We believe that supporting this kind of operations is essential to bridge the gap between procedural and manual modelling. This chapter identified these levels of granularity in user control, and showed how all are embedded in our declarative modelling approach.

We introduced *semantic constraints* for providing high-level control over the procedural generation process of complex virtual worlds. Our constraint definition and evaluation method allows for flexible composition and extension of semantic constraints, which then in turn adapt to their context and are automatically and consistently maintained.

The coarse-grained level of user control in our declarative approach is provided by *procedural sketching*, an easy to use interaction method that significantly increases the usability of procedural modelling techniques. The short feedback loop between each edit action and its generated results makes it easier and more intuitive for both experts and non-specialist designers to create complete virtual worlds, thus increasing their productivity.

To help further specify designer intent, we introduced the simple but effective concept of *feature refinements*. Feature refinements offer designers edit operations of medium granularity, thereby providing another novel and very convenient level of user control.

We identified the tension between user control and consistency maintenance that arises on the level of fine grained edit operations. The ability to lock an area, feature or object in the virtual world allows designers to limit the influence of, in some cases undesired, automatic consistency maintenance. For the successful integration of locking, transition zones between locked and unlocked content are very important. Although plausible transitions would apparently seem very much dependent on the type of feature, we concluded that there are only a limited number of transition forms. With this, the implementation of locks becomes less feature-specific and somewhat more straightforward.

Several challenges and open issues associated with user control remain. The examples of semantic constraints currently integrated in our framework can be considered a proof of concept. Future work could focus on introducing new and more elaborate semantic and feature constraints, defining connections between compatible constraints, and incorporating more constraint application schemes to cover a wider range of possible contexts and terrain features.

For fine-grained user control, an important open issue is how to preserve edit operations that a designer has manually performed on objects of a procedurally generated terrain feature. The motivation for preserving these actions is twofold: they state, in more detail, the designer's intent for that specific feature, and they can equate to a large amount of designer modelling effort. We see three possible approaches for solving this problem that could be explored:

1. Encoding the manual changes in the specification of the procedural model, in such a way that regeneration of this model automatically yields content matching the manual changes. In this approach, we interpret manual edit actions as fine-grained refinements of intent, and handle them in a way that resembles the feature refinements explained in this chapter.
2. Attach manual edit actions as a sequence of operations to be re-applied after each regeneration. The challenges include that the edit actions need to be mapped to every new local situation, and that some evaluation is required to assess whether it still makes sense to reapply those manual operations.
3. Lock all objects involved in a manual edit action to preserve the exact situation during subsequent regeneration of the corresponding feature. Although challenges for locking individual objects are similar to challenges for locking areas of the virtual world, in the sense of transition and connections with its regenerated surroundings, the solutions need to be more fine-grained. The definition of an area lock's extent is coarse and the interpolation from locked to regenerated content is performed across a continuous transition zone; in contrast, a lock on a single object, e.g., a building in a city, has a discreet and sharp transition. This results in little room for generating suitable connections, such as street connectivity, etc.

Each of these approaches has specific advantages, but none of them is likely to work well in all circumstances. Therefore, a mix of these three approaches could very well be needed to optimally deal with all the different types of edit actions and possible modelling scenarios.

A major part of the challenge is not only to find technical solutions for solving complex modelling situations, but to involve the designer in the decision process. Firstly, automatic solutions often require to interpret the designer's intent for a specific edit action, in order to ascertain its importance for preservation, etc. If several alternative solutions to a situation exist, it will not always be possible to choose the "best" one automatically. It is therefore necessary to involve the designer in certain choices, without completely disrupting the interactive modelling workflow, by e.g., giving him too many or unintuitive options to choose from.

Furthermore, we need additional facilities for designers to precisely express their intent. These would allow us to derive the priority of preserving certain features, objects or situations during procedural (re-)generation. On the higher levels of

modelling, designers can already express intent through semantic constraints, feature specifications and feature refinements. On the fine-grained level of editing objects, one straightforward way for a designer to assign priority to an object, is to lock it. But there might be other, more implicit, operations, such as grouping a set of objects, from which one can derive this intent.

Lastly, in order to be usable, even a procedural modelling environment needs, to some extent, to be *predictable*. There are many possible technical solution to improve its predictability, e.g., choosing default solutions and providing designers the option to change them, allowing unlimited undo and redo for all combinations of manual and automatic operations, offering a preview of the effect of an operation, and warning a designer if an operation has large side-effects (through, for instance, consistency maintenance).

In this chapter, we discussed common properties of transition zones for different features of the virtual world. This research scope should be further extended to the interpolation of any combination of procedural models, generated by different techniques, by means of generic interpolation schemes. Such schemes would make it far more convenient to define and maintain transition zones, connections and fall off regions for new types of features and objects, and, most likely, could also be applied in other domains within the field of procedural modelling.

In conclusion, we expect that the integration of user control in procedural modelling will remain a challenging and important research topic in the coming years.

7

Prototype design and implementation

Our framework that supports the declarative modelling approach, consisting of a semantic model for virtual worlds (Chapter 3), integrated procedural methods (Chapter 4), consistency maintenance mechanisms (Chapter 5) and user control facilities (Chapter 6), has been implemented in a prototype named *SketchaWorld*. The results and real-world application of this prototype are discussed in the subsequent chapters.

This chapter discusses relevant design and implementation aspects of such a complex system. We start by describing the high-level design in components and flow of execution, from user interaction to procedural operations. Next, we discuss the implementation of the currently integrated features and procedural methods.

One of the implementation challenges for such a modelling system is maintaining acceptable performance and interactivity, especially for larger virtual worlds. We explain some implementation measures and strategies for obtaining the desired performance, and indicate some potential improvements.

On the basis of the semantic representation of the virtual world, we automatically derive a 3D virtual world model. We therefore also detail how we create and render this 3D virtual model, and how additional data and models can be derived for use in other applications.

7.1 Prototype design

This section discusses the high-level design of our SketchaWorld prototype system. We identify the important components within this prototype and examine the flow

of operations that follow a user interaction. Two of the main design goals of the prototype are (i) the integration of procedural methods, and (ii) support for extension of the virtual world model with new features and objects. Chapter 4 discussed the integration of procedural methods on a conceptual level. Here, we will give a more practical overview on how to integrate new procedures, terrain features and objects. Two additional design criteria for the prototype were configurability and accessibility. We briefly discuss how and to what extent the prototype can be configured and customized, and take a look at the user interface, which we have tried to keep simple and straightforward to use.

7.1.1 High-level components and flow

The SketchaWorld prototype is rather large and complex, as illustrated in Table 7.2. Describing its complete design in e.g., class and sequence diagrams is out of scope for this thesis. Fortunately, it has an understandable and straightforward structure at a high-level of components and concepts.

Languages	Code contributors	Source files	Lines of code
C#, C++, GLSL and OpenGL	17	±2000	±350000

Table 7.2: Current statistics of the SketchaWorld prototype implementation.

Prototype organisation

The organisation of the prototype in components and concepts is presented in Figure 7.1. The prototype is structured using the Model-View-Controller (MVC) pattern. The MVC pattern, which separates the presentation of an application from its model (defined as the data model and the logic that manipulates this data), is useful in designing applications that have a relatively complex data model or logic.

The *view* layer contains both the graphical user interface and the conversion module. The graphical user interface is composed of a 2D view (see Figure 6.6) and a 3D preview of the virtual world, which is rendered in a separate thread of control. The conversion module derives the visual representations of the data model, i.e. our semantic model for virtual worlds, required for the 2D and 3D view components, or exports it for use in an external application.

At the *controller* layer, designers use the interaction tools to select an edit action, which is to be executed on the model. All previous edit actions are stacked in the edit history, which can be used to request to undo or redo any of these actions. The user interface thread handles all user interactions.

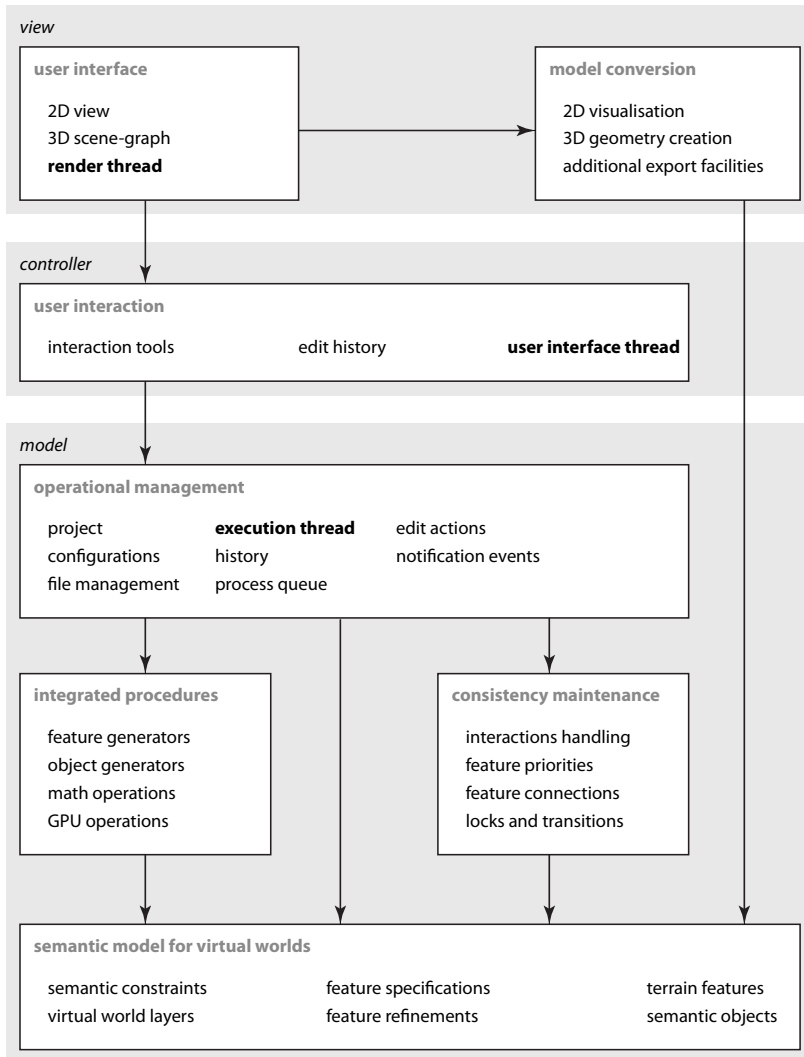


Figure 7.1: Overview of components and dependencies in the SketchaWorld prototype.

The *model* layer is divided in four core components. In *operational management*, we find the typically required concepts for any content authoring program, including the current virtual world project, the many configuration settings of all components featured in the prototype, and file management for external 2D and 3D content, saving and loading projects and generated results, etc. Furthermore, it contains the main structures required for processing user edit actions. As mentioned in Section 6.3.2,

edit actions are executed asynchronously on a separate execution thread. The process queue is the subset of actions that need to be executed in sequence in order to match the history's current state.

The *integrated procedures* component include all integrated procedural methods and algorithms for generating features and objects, as well the shared libraries for mathematical and GPU operations.

Another core component of the prototype is responsible for the *consistency maintenance* mechanism (see Chapter 5). In this category, we find all defined feature priorities, as well as the available types of feature connections. Furthermore, locks and transition zones are represented here (see Section 6.5).

In the component *semantic model for virtual worlds*, the levels of terrain features are represented (Chapter 3), as well as the different types of semantic constraints.

Processing edit actions

Figure 7.1 gives a static overview of the organisation of the prototype. To get a better insight into this prototype, we need to examine its high-level execution flow. Figure 7.2 presents a simplified diagram explaining the flow of operations from user edit action to the display of the generated results.

The asynchronous workflow is realized by three separate threads of control: the user interface and interaction thread, the edit action execution thread, and a thread for rendering the 3D virtual world. In the example depicted in Figure 7.2, a designer has just performed an edit action (a), e.g., a modification of the specification outline of a certain terrain feature. For this, the user interface thread creates a new edit action (b), with id 3 and the required information on the action. This action is appended to the history (c), which is immediately visible to the designer. Furthermore, the action is enqueued (d) in the process queue. Control is returned to the user interaction, which means that the designer can continue to edit the virtual world. In the situation depicted here, the execution thread simultaneously polls the process queue (e) and retrieves action 1, which is executed (f) using some of the integrated procedures and operations. The action results in modifications of the virtual world model. These modifications trigger notification events (g, h) that are received by the user interface thread, which updates the 2D view (i), and by the render thread. The latter, using the geometry creation module, updates the scene-graph (j), after which the thread proceeds to render it.

Operational management

Using Figures 7.1 and 7.2, we have discussed the general structure of the prototype and the generic flow of handling user edit actions. From the four categories in the *framework* in Figure 7.1, *operational management* is not treated in any of the previous chapters. Operational management can be seen as the “engine” of the framework. As such, its tasks include supervising the virtual world model and its features. To

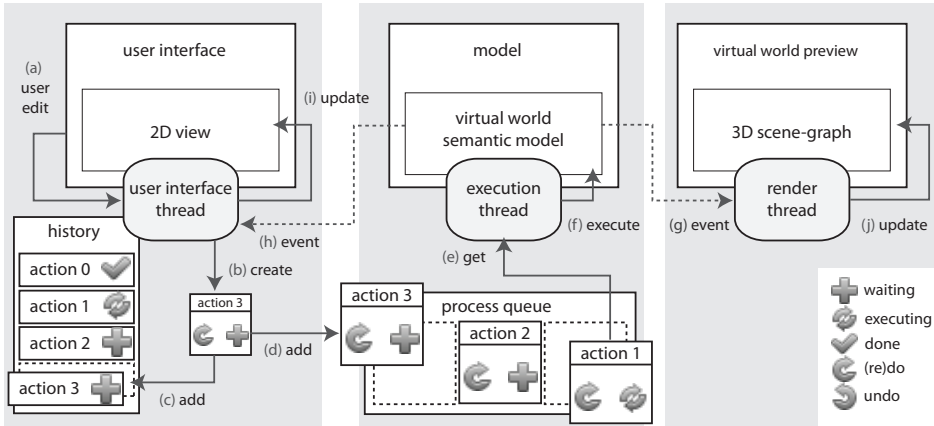


Figure 7.2: Diagram showing the asynchronous workflow of the execution of user edit actions.

give some insight into what this responsibility entails, we discuss as an example its method for handling requests for feature removal or regeneration. There are many possible causes why a terrain feature has to be removed or regenerated:

1. designers can directly execute actions to remove or regenerate a selected feature;
2. a complex edit action may request several procedural operations, including a feature regeneration step. In particular, changes to the landscape often require the regeneration of all features within the modified extent;
3. feature interactions may cause a feature to restructure. Depending on the implementation of the associated procedural method, this could be implemented as a full regeneration step.

Algorithm 4 outlines the methods for removing a feature and regenerating one or more features. An important step for both methods is to analyse the *dependencies* of the feature to be removed or regenerated. These dependencies stem from feature interactions, i.e., connections and conflicts (see Section 5.2). The dependent features are recursively collected, starting from the feature f_r to be removed or regenerated. Connections are two-way dependencies, therefore we include all features with connections to f_r . In case of conflicts, one feature lost part of its extent to another. As such, conflicts are one-way: if f_r successfully claimed part of another feature's extent, the latter is considered a dependency when f_r is removed or regenerated. Note that although the same concepts apply to connections with the landscape, we handle it separately by obtaining the extent of the landscape that was modified as a result of feature connections.

Algorithm 4 Feature regeneration handling method

```

// remove a feature and regenerate dependent features
//  $f_r$  - feature to remove
removeFeature(feature  $f_r$ ):
  // find the extent of landscape  $f_r$  has connected to
   $e_m = f_r.getSharedLandscapeExtent()$ 
  // recursively find all dependent features that need to be regenerated
   $F = f_r.getDependentFeatures()$ 
  // remove the feature
   $f_r.remove()$ 
  // regenerate dependant features
  regenerateFeatures( $F, e_m$ )

// regenerate a set of features
//  $F$  - features to regenerate
//  $e_m$  - extent of modified landscape
regenerateFeatures(features  $F$ , extent  $e_m$ ):
  // recursively find all dependent features that need to be regenerated
   $F += F.getDependentFeatures()$ 
  // expand  $e_m$  with landscape connections from features in  $F$ 
  for all feature  $f_r$  in  $F$  do
     $e_m += f_r.getSharedLandscapeExtent()$ 
  end for
  // remove all features in  $F$ 
  for all feature  $f_r$  in  $F$  do
     $f_r.remove()$ 
  end for
  // restore landscape in  $e_m$ 
  generateLandscape( $e_m$ )
  // regenerate the features in  $F$ 
  sort  $F$  according to highest  $p_{claim}(f_r)$ 
  for all feature  $f_r$  in  $F$  do
    generateFeature( $f_r.specification$ );
  end for

```

As we can see in Algorithm 4, to remove a feature f_r , we collect its dependencies and use the regeneration method to restore its surroundings in a consistent state. For regenerating a set of features F , we determine the total extent of landscape that was modified, which we restore before regenerating all features based on their specification. Sorting the set F based on feature priority is a simple optimization to minimize the amount of interactions during regeneration, but has no effect on the end result. Note that the algorithm presented here is somewhat simplified for clarity; for instance, we have omitted the fact that the feature or an area overlapping the feature's extent can be locked, and the feature can have relations with feature constraints.

7.1.2 Integrated procedural methods

In Chapter 4, we discussed our method for the integration of existing procedural methods to generate terrain features. To validate this approach, we implemented and integrated procedural methods for all terrain features currently incorporated in SketchaWorld. Their generation procedures are often based on (combinations of) existing procedural methods, but they have been modified to fit in the framework, i.e., to better consider their surroundings and context, and also to implement the interface described in Chapter 4. After implementation of the procedure interface, such a procedure is registered with the prototype to provide the generation method for a specific terrain feature.

Table 7.4 indicates for each of the features the basis for its generation procedure. As follows from the table, for a number of features, we based the generation procedure on methods from the literature. For instance, for the road feature, we based its procedure on the path plotting algorithm by Kelly et al. [Kelly 07], but there are other alternatives available. The algorithm by Kelly et al. iteratively finds a smooth path between a set of control points of a polyline defined on an elevation map. It prefers an even change in elevation from start to end, while guaranteeing all control points to be visited and the path to deviate only within a limited range from the specification. This algorithm was extended to avoid unacceptably sharp turns and slopes, to connect to existing features, such as rivers, if necessary, and to avoid potential feature conflicts with negative consequences for the road. Although the procedure uses a scoring mechanism to determine a path, it is not optimizing a cost function, as for instance the A*-based path finding method by Galin et al. [Galin 10], and therefore is not guaranteed to find an optimal path in all cases. However, this has the advantage that the procedure typically runs more interactively while staying close to the coarse path sketched by the designer, thereby providing more fine-grained user control. Still, it would be interesting to incorporate the method of Galin et al. to compare the results of both. Furthermore, especially for city generation, there exist a number of recent state-of-the-art approaches (e.g., [Vanegas 09, Chen 08]) that, if integrated in this framework, would likely improve the quality of the generated virtual worlds.

For some features, a custom procedure was devised, because no suitable procedure was directly available. Often, these procedures are fairly straightforward, as it was not the focus of our research. As an example of a custom procedure implemented in SketchaWorld, we consider the landscape generation procedure. We devised this procedure on the basis of the classic fractal terrain generation methods, using Perlin noise at its core [Perlin 02]. This procedure has been detailed elsewhere [Smelik 10a], and is outlined in Algorithm 5. The landscape specification is derived from the coarse grid of ecotopes painted by the designer. The definition of each ecotope includes, among other things, a minimum and maximum elevation and a roughness percentage, describing how rough or smooth its terrain should be. From this definition, each cell in the ecotope grid is assigned a randomized, *local* variation of these ranges. These local values in the grid are smoothed using a Gaussian kernel, to obtain natural

Name	Generation procedure
Landscape	Procedure outlined in Algorithm 5.
River	Procedure outlined in Algorithm 1
Canal	Custom procedure.
Ditch	Custom procedure.
Lake	Custom procedure.
Levee	Custom procedure.
Forest	Extension of the simulation method in [Deussen 98].
Tree line	Custom procedure.
Hedge	Custom procedure.
Field	Custom procedure.
Road	Extension the path planning method in [Kelly 07].
Building lot	Implementation of CGA ([Müller 06]) and a custom script-based procedure.
City	A combination of the district layout procedure based on urban land use models, described in [Groenewegen 09], and the procedure in [Parish 01].

Table 7.4: Incorporated procedural methods for terrain features in the prototype SketchaWorld.

changes in elevation. For each point (x, y) in the landscape, the coarse grid g is interpolated with Catmull-Rom splines to obtain the ranges \bar{r} at the desired spatial resolution (e.g., 1 meter). Note that the coordinate (x, y) is first perturbed in 2D to decrease the regularity of this interpolation, resulting in (x', y') . A combination of several “flavours” of fractal noise, such as ridged multi-fractal noise [Musgrave 89], mixed according to the roughness factor, are used to determine the elevation value within the range \bar{r} . The distribution of soil material is based on the ecotope value at (x', y') , and by mapping the elevation value to a lookup table. The procedure results in $(result_{elevation}, result_{soil})$ being stored at position (x, y) in a height-map data-structure. In this way, a plausible landscape is generated.

As explained in Chapter 3, the definition of a terrain feature is kept separate from the procedure used to generate it. We have strived to minimize the amount of repetitive implementation work for the definition of terrain features. To implement the skeleton of a new terrain feature, one defines a feature specification outline and its semantic attributes by deriving an abstract class, and configures feature priorities and connections. For user interaction, we register the new terrain feature specification with SketchaWorld’s GUI, on the basis of which we can automatically create the required interface elements, such as buttons to set semantic attributes, etc. Although the implementation of a new feature’s generation procedure can take a considerable amount of effort, the feature itself can be integrated to function in the prototype with

Algorithm 5 Landscape layer generation method

```

// g - coarse grid of ecotopes (elevation ranges, roughness, etc.)
GenerateLandscape(ecotope grid g, random seed s):
  for y = 0 to height do
    for x = 0 to width do
      // perturb x and y locally
      p = perturb(x, y, s) // in range  $[-\pi \dots \pi]$ 
      x' = x + offsetperturb cos(p), y' = y + offsetperturb sin(p)
      // obtain 4x4 grid region for Catmull-Rom interpolation
      xg = (x' - gridcelldim(g)/2) / gridcelldim(g) // grid x
      yg = (y' - gridcelldim(g)/2) / gridcelldim(g) // grid y
      x1 = [xg], xf = xg - x1, x0 = x1 - 1, x2 = x1 + 1, x3 = x2 + 1
      y1 = [yg], yf = yg - y1, y0 = y1 - 1, y2 = y1 + 1, y3 = y2 + 1
      // interpolate 4x4 grid cells based on fractions (xf, yf)
      // interpolation result  $\bar{r}.xyz = (\min, \max, \text{roughness})$ 
       $\bar{r}$  = spline(g, x0, x1, x2, x3, y0, y1, y2, y3, xf, yf)
      // generate 3 noise values, mix based on roughness
       $\bar{n}$  = noiseValues(x', y', s) // all in range  $[-1 \dots 1]$ 
       $\bar{f}$  = mixFactors( $\bar{r}.z$ ) // all in range  $[0 \dots 1]$ ,  $|\bar{f}| = 1$ 
      v =  $\bar{f} \cdot \bar{n}$  // combined noise value
      // result for x, y: elevation and soil values
      resultelevation = r.x + (1/2 + v/2)(r.y - r.x)
      resultsoil = distribute(x', y', resultelevation, ecotope(g, xg, yg))
    end for
  end for

```

minimal effort.

7.1.3 Configuration and templates

As already noted in Chapter 4, procedural generation methods need much tuning and experimentation with their settings to make them usable and generate plausible results. The configurability of the prototype is therefore important to effectively support integration. Procedures can register parameters for configuration in an external XML-file. Obviously, the configuration is not limited to procedure parameters; other definitions such as feature priorities, ecotope definitions, etc. are also configurable from file. The complete configuration of the prototype can be updated at run-time, thus shortening the integration cycle.

Many of these settings influence generated results in a way that is typical for a specific area or country in the real world. For instance, the environment in The Netherlands is very much planned and controlled by man, while in other countries the environment is often more natural and pristine. To represent this, we can define *virtual world templates*. These templates can specialize any configuration setting to

a specific value more suitable for that area of the world. Examples include the species of vegetation that are present in a forest by default, the available ecotopes, and the roughness of a river profile. In this way, we can create geo-typical worlds that resemble to some extent a geo-specific location (see, e.g., our previous work [Smelik 09b]).

An interesting extension to shorten procedure development and integration would be to provide an interactive scripting interface. Using this scripting language, we could define feature procedures on the basis of a library of smaller procedures and operations. In this line, the Houdini tools are a very successful example of the power of interactively composing procedures out of small building blocks [Side Effects Software 11].

7.1.4 User interface design

For embedding our declarative modelling approach in the graphical user interface of the SketchaWorld prototype, we have taken two main design requirements into account:

1. all interaction tools are to be goal-driven, focussing on *what* a designer wants to create;
2. the interface is to be accessible to users with no prior 3D or procedural modelling experience.

Figure 6.6, in the previous chapter, presented a screenshot of the user interface of our SketchaWorld prototype. The accessibility requirement entailed that we had to abstract most of the 3D modelling or procedural generation operations typically required to create a 3D virtual world. We concluded that the concept of sketching out a coarse map of the world can be intuitively grasped by most potential users. For this reason, we have made our top-down design view resemble a map with a scale grid, isolines and other hints. These are also useful for offering a sense of scale and distance.

Furthermore, to avoid overwhelming new users, we kept our interface clean and uncluttered. Although modelling a virtual world is a much more complicated task than, for instance, drawing a bitmap image, we have tried to keep the complexity of the user interface comparable to simple image editing software.

7.2 Performance considerations

In this section, we discuss performance aspects for procedural operations and data management to allow for interactive modelling, and indicate some improvements that could be made to our current implementation. The efficient rendering of the generated results is treated in the Section 7.3.

7.2.1 Use of GPU computing

For each edit action by a designer, a number of procedural operations need to be performed. In case the edit action causes feature interactions to occur, that number quickly increases, as features are restructured and in some cases even completely regenerated. This makes it challenging to provide an interactive modelling experience, especially for complex operations. As explained above, using asynchronous execution, the designer is not hindered by unfinished operations and can continue to model. However, the delay cannot be longer than a couple of seconds. If operations take longer, without any feedback on their results, the designer is essentially forced to wait for the operations to complete.

Fortunately, several parts of the procedural generation process are very well-suited for parallel processing. An emerging trend in parallel programming is to use a Graphics Processing Unit (GPU) as a general purpose computation device, because it has a larger number of floating point processors available that can process small programs, called kernels, in parallel. A number of procedural operations in SketchaWorld are implemented on the GPU using the Open Computing Language (OpenCL [Khronos Group 11]), a C-like programming language for performing all sorts of computations on GPUs. The speedup for these operations is typically an order of magnitude compared to our original CPU implementation.

An example of an expensive procedural operation that maps really well to GPU processing is the landscape generator. The reason for this is that it is possible to determine the definitive elevation of each point without considering neighbouring points. As a result, the landscape generation algorithm described in the previous section can be performed completely on the GPU, using a sequence of OpenCL kernels. The GPU is best suited for executing many independent computations in parallel, each of these having relatively limited memory requirements and a high number of operations to perform. Variations on this, i.e., kernels that need to access memory more often or have certain steps at which the results of neighbouring threads need to be considered, can also be implemented on the GPU with some effort, but the speedup is less impressive. Nevertheless, for operations with a high number of identical tasks, it is often worth the effort.

In SketchaWorld, examples of procedures implemented in OpenCL include the elevation and soil map generation, and the creation of images, such as textures and the 2D view on the virtual world. In fact, the GPU is a good match for handling all operations on a height-map, such as a landscape modification, e.g., a road embankment. All these modifications have a clearly defined 2D footprint, limiting where the operation should be applied. GPU rasterization is, of course, the most efficient way to generate this geometric footprint as a 2D map, which can be consulted to determine where to apply a modification and to what extent. Although OpenCL does not support rasterizing geometry, we can use the interoperability with the Open Graphics Library (OpenGL) to obtain the footprint. In a pre-processing step, we use OpenGL to render the simple 2D footprint geometry to texture, and pass this

texture to the corresponding OpenCL modification kernel. This approach, which is very similar to the work of Bruneton et al. (see [Bruneton 08]), results in very efficient computation of landscape connections with features.

To improve even further the performance of our prototype using parallel processing, there are several avenues we could explore. An obvious step is to optimize the performance of existing kernel implementations, and to identify additional operations which could be ported to the GPU. Furthermore, for large operations that consist of executing several kernels in sequence, an improvement to our current implementation would be to make use of the concurrent kernel execution and data transfer that modern GPUs support. On a higher-level, a dependency analysis on edit actions and associated procedural operations could determine which operations can be executed in parallel on a multi-core CPU.

7.2.2 Efficient data management

Besides the computational efficiency of the procedural framework, we need to consider the memory requirements of the virtual world model. The landscape is often the most demanding in terms of memory use. We want to be able to model relatively large landscapes (i.e., larger than 250km^2) at a high resolution (e.g., 1 m), but if we fit the landscape's height-map of such dimensions (i.e., more than 1GB) in main memory, it will quickly claim a substantial part of the memory available for the application. Furthermore, we regularly need to upload this landscape to the GPU to perform all kinds of procedural operations on it, and GPU memory is obviously much more limited.

Fortunately, the solution to this problem is, in case of the landscape, well-known. We have split the landscape in manageable square tiles that we manage on disk. The tiles are loaded from disk on demand; to speed up access we maintain a fixed-size list of most recently used tiles in memory. As tiles are swapped out of this list, we check if they have local modifications, in which case we commit them to disk. Additionally, because operations often require to read data at a less detailed resolution, we maintain a resolution pyramid of the complete height-map. Changes to landscape are always made on the most detailed tile level, after which this pyramid is partially updated.

Procedural operations often have to perform a large number of spatial queries. For instance, a road generator could query all roads close to a specific location. To make these queries efficient, typically an acceleration structure is used to hierarchically organize all objects in sets according to proximity. A quad-tree is not so suitable here, as it is primarily used to organize sets of points. Instead, we use an R-tree for each of the virtual world layers [Guttman 84]. An R-tree, often used in geospatial databases, organizes objects according to their 2D bounding box. In practice, this structure works best for objects that have a reasonably tightly fitting bounding box, such as vegetation, building parcels, and streets in a dense road network. For rivers or major roads, the bounding boxes typically encompass a large part of the virtual world, and, as a result, the speedup is limited.

There are still quite some improvements that can be made to the management of data in SketchaWorld. These improvements will probably result in a measurable performance improvement for large virtual worlds, and raise the upper limit on the size of virtual worlds that can still be comfortably edited, which is, at the time of writing, around 900 km². For instance, we could apply an efficient and lossless compression algorithm to the landscape tiles, to decrease disk read times. Furthermore, as virtual worlds increase in size, it will become necessary to split up other virtual world layers in some form of tiles as well.

7.3 Creating the 3D virtual world

This section deals with the creation of the 3D virtual world model. Starting with a barren terrain, this 3D model is incrementally updated as features change throughout the modelling session. In this way, we can provide designers with a 3D preview of the world as they design it. We first describe the automatic creation of this 3D model on the basis of the semantic virtual world model. Then we explain the real-time rendering techniques we apply to visualise this world. And finally, we discuss how, in a similar fashion, other representations can be derived from the virtual world model.

7.3.1 Generation of the 3D geometric model

The semantic virtual world model contains all information required to automatically produce its 3D representation. All semantic objects have a geometric representation in this world. A terrain feature, such as a city, is represented through the semantic objects it is composed of, e.g., its streets, buildings, vegetation, etc.

There are a number of different ways of deriving 3D geometry from a semantic object. The process can be a simple conversion step, the placement of an external 3D model, or, in some cases, a (small) procedural generation process.

As an example of procedural generation of 3D geometry, we consider the road feature. The creation of geometry for a road in isolation can still be considered a simple conversion step, i.e., sweeping the road's lateral profile along its path. However, a more complex procedure is to be executed once a road connects with another feature, such as a river or another road. In case of connecting with another road, the connection will be represented as a junction object. Creating a plausible junction is a complex rule-based generation process, especially if multiple heterogeneous roads are involved, e.g., different types of roads, varying number of lanes or other differences in lateral profile. A river connection is typically represented as a bridge object. The geometry of this bridge can be created by a rule-based geometry creation script, or a similar procedural technique such as shape rewriting.

Another good example of a procedural generation step in this process is the creation of 3D buildings. We already saw in Section 4.2 that this might even involve a

number of procedural components working together according to a building plan. Depending on the *building type* attribute, a procedural generation plan is selected and applied to the parcel shape. Note that such a parcel object can have many more attributes that influence this generation process, such as slope, and minimum or maximum height requirements.

7.3.2 Scene-graph organization

For the 3D preview in SketchaWorld, we use OpenSceneGraph as the rendering framework, which is an open source OpenGL based scene-graph renderer with a large and active community of users [Osfield 11]. A scene-graph is an acyclic directed graph that is used to create a hierarchic organization of all geometric objects in the scene. In this hierarchy, nodes inherit relative transformations as well as material and other graphic state settings from their parent nodes. As a result of using a scene-graph based renderer, we have to organize the generated geometry of all semantic objects in this hierarchic graph.

The render efficiency of our 3D virtual world model very much depends on the actual organization of the scene-graph. Therefore, our geometry creation method should not only include newly generated geometry in the scene-graph, but also optimize this graph in each update step. We now briefly discuss the performance considerations and trade-offs that we have made in this process.

The most efficient way of rendering geometry is not to render it. The process of identifying which objects are outside the field of view, and thus do not need to be rendered, is called *culling*. One of the first passes over the complete scene-graph is called the cull-traversal. Based on the bounding volumes of the geometric objects, the renderer can determine which objects should be rendered and which can be culled. Because of this, for effective culling we need to organize the world geometry in small groups with a high coherence in world position, such that we obtain tightly fitting bounding volumes.

However, this measure has its downside. First of all, organizing geometry in small groups results in a scene-graph with a large number of nodes. For each of these, the cull decision process needs to be executed, every frame. This quickly results in an expensive cull-traversal. Furthermore, having smaller groups of geometry conflicts with the desired batching strategy used for modern GPUs, explained below.

Even with culling, the total polygon count often surpasses the capabilities of current GPUs. To further alleviate the amount of polygons that need to be rendered, the well-known concept of *level of detail* (LOD) comes into play. Objects in the far distance contribute little to the final frame, in some cases only a handful of pixels. As a result, dense geometry is often not required for these objects, as their details cannot be made out at that distance anyway. The most common LOD strategy is to have several geometric representations of a single object, with varying vertex count. Based on some distance or error metric, the renderer selects which representation to use.

We use OSG's built-in distance-based LOD strategy, where one has to specify for each LOD the minimum and maximum range to select this representation. In some cases, where the polygon count is prohibitively large, the LOD method can be extended with a paging scheme, where high-detailed representations are loaded from hard disk, once needed. This is the case for rendering large landscapes or detailed city models, as the total amount of geometry required to render the complete height-map might not even fit in main memory.

Specifically for the landscape, the geometry is typically split in manageable square patches of terrain. It is most efficient to organize these individual patches of the landscape into a hierarchical acceleration structure, such as a quad-tree. Each group node at each level of this quad-tree has four child nodes, and each of these nodes contains a specific terrain patch at its current LOD and a file reference to a more detailed LOD. When traversing this landscape quad tree, depending on the computed distance for each node, either the current LOD geometry is rendered, or the higher LOD is loaded in from disk. To further reduce the amount of memory required for this part of the scene-graph, we instantiate a flat patch of geometry of the desired size, which we displace in a vertex shader based on the corresponding part of the height-map, provided as floating-point texture.

For all geometry in the scene-graph, we implemented two optimizations. Firstly, switching states between draw calls (e.g., because a different texture or shader is to be used) is still relatively expensive, therefore we group geometry during creation based on the material. Secondly, the total number of draw calls should be kept to a maximum per frame (depending on the target hardware). This can best be done by collecting geometry to be rendered with the same state into large batches. With each new generation of GPUs, batch sizes, i.e., the supported polygon count per draw call typically increases. Because of the parallel nature of GPUs, the difference in draw time for a very small or large batch is not significant. To achieve a high throughput, we need to group geometry in batches of the largest size that still can be comfortably handled by the GPU.

With the described measures, typical virtual worlds in SketchaWorld are rendered at the desired 60 Hz frame rate. As GPUs continue to evolve, render strategies and targets such as batch sizes will need to be re-evaluated to optimally match new hardware.

7.3.3 Rendering the virtual world

In our implementation, we moved from the classic forward rendering approach, as implemented in OpenSceneGraph, to a *deferred rendering* approach. Contrary to forward rendering, deferred rendering splits the process of rendering the virtual world in two phases:

1. all geometry is rendered to a number of intermediate buffers without any lighting effects. The combined buffers, better known as the G-buffer, contain all

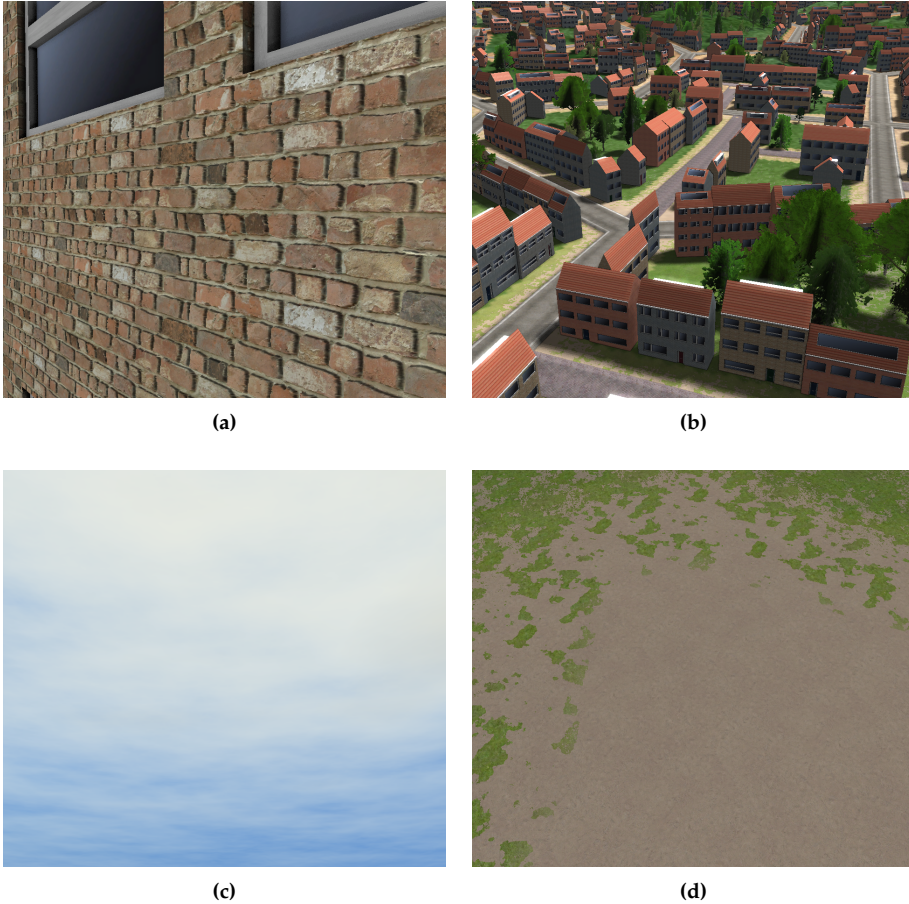


Figure 7.3: Examples of rendering techniques implemented in the prototype. (a) Parallax occlusion mapping gives the illusion of 3D geometry. (b) Cascaded shadow mapping allows for relatively detailed shadows nearby and at medium distance. (c) Layers of 2D clouds. (d) Thresholding technique applied to landscape detail textures.

information required for lighting computations, i.e., depth, normal, diffuse and specular material colour, and possibly additional attributes;

2. lighting and post-processing is performed based on these intermediate buffers, resulting in the complete frame.

The advantage of the deferred approach is that expensive lighting computations need only be performed for the visible geometry. Furthermore, regardless of the number of lights in the scene, the world geometry is rendered only once.

A downside of the deferred approach is the inability to render transparent geometry. Transparency, in effect, implies that the final colour value of one pixel depends on multiple layers of geometry, while the intermediate buffer support only one stored value per pixel. In our prototype, this is solved by rendering transparent objects by forward rendering and combining it with the result of the deferred renderer.

A number of advanced rendering techniques are implemented in our renderer to enhance the visual appearance of the virtual world. For some of the surfaces, such as building walls, we apply parallax mapping. This technique uses a normal and a displacement map to compute, dependent on the view direction, a local offset for sampling the diffuse texture. This offset stretches the texture along the 3D shape implied by the displacement map. Combined with the ambient occlusion, this gives a convincing 3D appearance to these 2D surfaces, as can be observed in Figure 7.3 (a).

Shadows are an important visual cue that improves the perceived realism of a scene. There is a large and continuously growing body of research on real-time shadows. The most basic approach is to render a shadow map from the light source's perspective, and use this shadow map to determine from the viewpoint whether each pixel is in the shadow or not. A major problem with this approach is the resulting hard shadow transitions. Furthermore, the technique does not scale for large outdoor scenes: to be able to represent shadows for objects in the distance, one would need a huge shadow map texture. These two problems are addressed in our implementation with two well-known techniques: Percentage Closer Soft Shadows (PCSS) and Cascaded Shadow Maps (CSM). PCSS takes multiple samples from the shadow map in a neighbourhood, to estimate the distance between the occluding geometry and the shadowed surface. Based on this distance, the penumbra of the shadow is approximated. This gives the shadows degrees of softness that vary according to the determined distance. CSM use a set of shadow maps that are consecutively laid out in view space, starting from the view point along the view direction. As a result, every next shadow map encompasses an increasingly large area of the virtual world. This approach gives detailed shadows up close and is still able to capture shadows in the distance (see Figure 7.3 (b)), without requiring an enormous shadow map. In our implementation, we use 6 shadow maps of 1024 x 1024 pixels.

The SketchaWorld prototype contains a time-of-day model that can be interactively set. The time of day influences the position of the sun and moon directional lights, and using simple pre-computed model, the sky is shaded accordingly. Clouds are represented as layers of scrolling 2D textures (Figure 7.3 (c)), and distance fog is applied to all geometry. Furthermore, we implemented a variety of post-processing effects that can be applied in our prototype, including screen-space ambient occlusion, a bloom effect and tone mapping.

To render the soil material of the landscape, we use a lower resolution texture (generated based on the soil material map) for far away views, combined with several layers of detail textures for close up. To combine these texture layers into one diffuse colour, we use a blend map per texture layer. These blend maps are generated per

patch of landscape, as part of the landscape quad tree creation process. Straightforward blending of the texture layers results in blurry transitions between soil materials, with one detail texture linearly fading out while the other fades in. However, in the real world, transitions between e.g., grass and sand are sharp, and therefore, the landscape texturing should reflect this. We use a thresholding technique, which uses an importance map of each layer combined with mentioned blend maps. This hand-drawn importance map is encoded in the alpha channel of all detail textures. Instead of using the blend value of a layer directly as an interpolation factor for the texture mixing, we interpret the blend value as a percentage of material that should be present at that location. This percentage is used as a threshold for the importance map defined for the material, to make a binary decision whether to apply the material or not. This results in organic but sharp transitions, as can be seen in Figure 7.3 (d).

The techniques described here are just some of the many possible options for enriching the visual appearance of the virtual world. For an excellent textbook on these rendering techniques and more, we refer to [Akenine-Möller 08].

7.3.4 Exporting the virtual world

Similarly to the model conversion needed to obtain the 2D visualisation and 3D geometry of the virtual world, we can create export facilities for using the virtual worlds in external applications. Making use of the semantics defined in the virtual world model, we can export more information than 3D geometry. This information can be used for intelligent agents (e.g., road networks, data for path planning and terrain reasoning) or for run-time interactions and services, e.g., as described in [Kessing 09]. Furthermore, although the virtual worlds created in SketchaWorld are geo-typical, we can export geographic data (such as elevation raster and features vector data) of this fictional world. This broadens the range of industry tools where the generated worlds can be used, as they are typically designed for geo-specific scenarios and, as such, require GIS data as input. Furthermore, we implemented an exporter of the virtual world as a 2D map in Scalable Vector Format (SVG). This map can be printed for use in specific training applications.

The 3D geometry of the virtual world is typically exported in OpenSceneGraph's native format or in the common interchange format COLLADA. For specific applications, where COLLADA is not supported as an import format, an application-specific export module needs to be implemented. An example of this can be found in Chapter 9, where we describe a project in which we implemented export facilities for use of the virtual worlds in the popular Unreal game engine. We are currently planning to create export facilities for the increasingly often used Unity game engine.

7.4 Discussion

Our framework for supporting declarative modelling of virtual worlds was implemented in the SketchaWorld prototype. SketchaWorld is not a restricted proof-of-concept implementation, but rather a fully functional prototype system that is currently being applied in real-world cases (see Chapter 9).

This chapter discussed relevant design and implementation aspects of the prototype. We presented its high-level design and flow of operations. The asynchronous workflow presents an implementation challenge regarding operational management, as the model on which operations are executed is often slightly out of date compared to the view the designer interacts with. However, it results in increased responsiveness of the prototype, which in turn leads to increased productivity for designers, as they can continue to interact the system while results are being generated.

We described the procedural methods and features currently integrated. As is typical for procedural methods, implementing a working proof of concept often takes far less effort than tweaking the algorithm and parameters to produce the desired results, in all circumstances. For this, the iteration time was somewhat alleviated because of the prototype's ability to reload configurations at run time. However, an integrated scripting language for prototyping procedural methods would have been even more helpful.

To interactively model relatively large virtual worlds, some performance optimization measures were taken, including the use of GPU computing, and data tiling and paging. As we strive to increase our virtual worlds to a size of up to 100km by 100km, we will need to implement additional optimizations, such as height-map compression. This is especially the case considering that the landscape needs a relatively high resolution to be able to correctly represent e.g., the embankment of a small road or the bed and bank of a ditch.

Finally, we described how we generate, optimize and render the 3D model of the virtual world. As GPU hardware continues to evolve, we will regularly need to re-evaluate our scene-graph optimizations and employed render techniques.

In the next chapter, we will examine the results of this prototype by means of example modelling sessions.

8

Example modelling sessions

The previous chapter discussed the fully functional SketchaWorld prototype, which implements the declarative modelling approach. This chapter presents its results through a number of example modelling sessions, each focusing on a specific type of user control. We start with an example of creating a virtual world with the main interaction method: procedural sketching. We then demonstrate how we can further enhance and fine-tune such a procedural world using the refinement facilities. And finally, we highlight how high-level constraints influence the virtual world throughout a modelling session.

8.1 Modelling session 1: procedural sketching

This session involves procedural sketching of a medium-sized world.

8.1.1 Motivation

The first modelling session aims at demonstrating how quickly one can create a virtual world, consisting of a landscape with a city along a river. This session solely uses our main interaction method, procedural sketching (see Section 6.3). The total modelling session took less than ten minutes from start to finish.

The example session also involves consistency maintenance, discussed in Chapter 5. In this session, while we sketch the feature specifications, several feature interactions occur, resulting in conflicts and connections. Throughout this session,

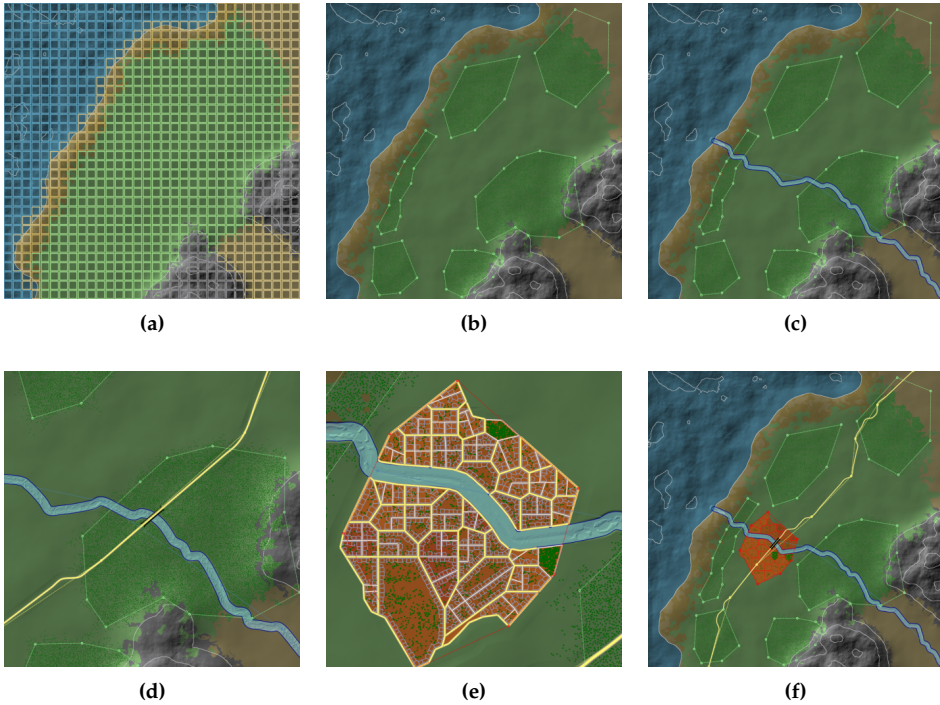


Figure 8.1: First procedural sketching session: (a) basic landscape, defined by brushing ecotopes, (b) several forest features outlined in the temperate flat lands, (c) river defined by a coarse path flowing towards the sea, (d) road feature crossing this river, (e) city created along the river banks, (f) road rerouted to also run through the city.

a 3D geometric model derived from the layered semantic virtual world model, is incrementally updated, allowing us to preview the results in 3D, as explained in Chapter 7.

In Section 7.1.3, we discussed the function of *virtual world templates*. In this session, we select the Central-Asia template. The choice of template has an effect on many of the procedure settings and the generated content, the type and style of buildings, the city structure, the species of vegetation, etc. In our previous work [Smelik 09b], we presented in detail how we implemented the Central-Asia template and what materials it was based on.

The desired output of this modelling session is a virtual world of 64km^2 (at 1m resolution) with interesting features. The world consists of a landscape made of a mix of arid and temperate ecotopes, some forest features, a river, a road and a city features, located between the coast and some mountains. For all features, we have used default values for semantic attributes and only sketched their coarse outlines.

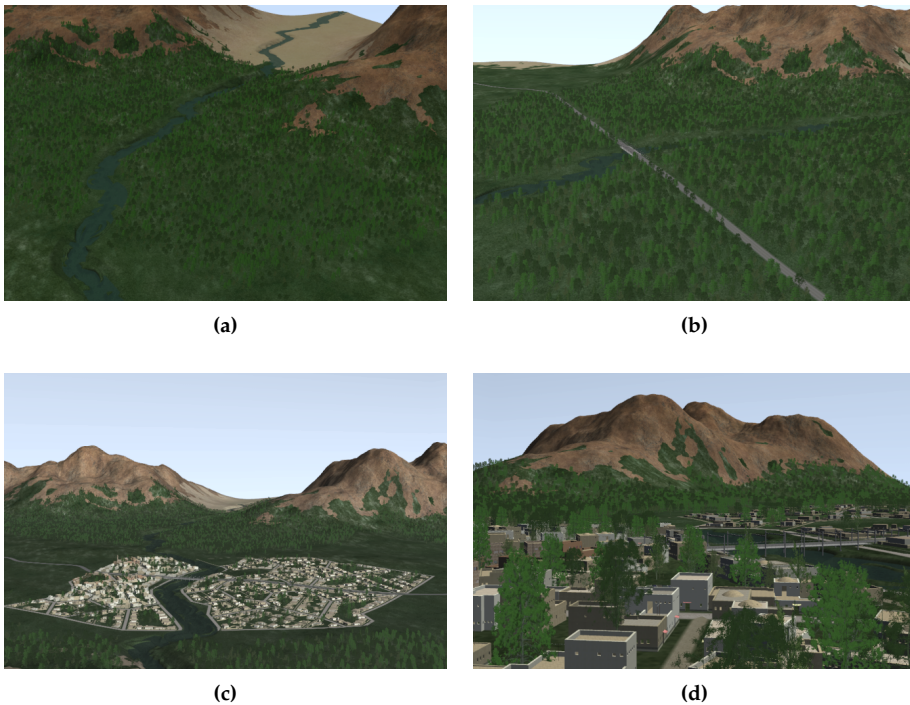


Figure 8.2: 3D virtual world, resulting from the first example session of Figure 8.1: (a) natural environment with a river (b) road crossing river, (c) complete virtual world with city, (d) close-up of the city, near the riverside.

8.1.2 Walkthrough of modelling session

Figure 8.1 (a) depicts the basic landscape, sketched in *landscape mode* by brushing the ecotope grid: a coastline with a green zone, some mountains and arid land in the south-east. On top of this generated landscape, using the tools provided in *feature mode*, virtual world features are added. We specify several forest features using polygon outlines, resulting in the vegetation distribution visualised in Figure 8.1 (b). Note that, despite the chosen outlines, trees do not grow on steep slopes, rocky terrain and barren land.

A river feature is declared to run from the mountain lands in the south-east towards the ocean in Figure 8.1 (c). After we have coarsely defined the river's path using a polyline, a suitable course is plotted across the landscape. Due to the river's default priorities, its requested extent is completely granted, which leads to restructuring of the affected forests. As a connection with the landscape, the river bed and banks are carved into the elevation profile.

In Figure 8.1 (d) we introduce a major road, again by very coarsely defining its path using a polyline. A connection is defined between the road and river features. As a result, a bridge is inserted at the river crossing connecting the road segments. Also here, the landscape is modified accordingly, in this case to form a road embankment. The forests lose again a part of their initial extent, in this case to this primary road. Figure 8.2 (a) depicts the resulting virtual world at this stage.

Subsequently, we specify a small city around the river by simply sketching the city outline using a polygon. As a result, its districts and secondary roads form around the river, as illustrated in Figure 8.1 (e). Finally, we decide to reroute the primary road, using the control points of its specification, to now run across this city in Figure 8.1 (f). This edit action causes the road to claim and obtain part of the city's extent. Therefore, this city is restructured to include this main road and its bridge. Figure 8.2 (b) depicts the final virtual world. A close up of the city is shown in Figure 8.2 (c); notice how generated building match the selected Central-Asia template.

8.2 Modelling session 2: refining intent

This session involves a combination of procedural sketching and feature refinements.

8.2.1 Motivation

In the previous session, we used procedural sketching to define a small number of large features, and kept all feature's semantic attributes on their default settings. Because of this, we only had to sketch a handful of feature's outlines to create the world. Depending on the application and the type of training or game scenario, that level of user control might be enough to create a suitable virtual world.

In most other application scenarios, however, we will want to edit on a finer-grained scale, to further customize the world according to our intent. In this session, we show that editing on a smaller scale is also possible. Besides sketching feature outlines, we use semantic attributes and feature refinements to modify the results.

In this session, we create a small landscape of 1km^2 . However, as we perform more edit actions and refinements, the modelling session time increases slightly, in total about 15 minutes. The session produces a geo-typical world set in The Netherlands, using our Dutch virtual world template. As a result, the ecotopes and tree species available are most from the temperate climate. Furthermore, the style of man-made objects, such as buildings, is very different.

The goal of this modelling session is to create a natural environment, rich in vegetation, where a small agricultural settlement is situated. We employ a number of features different from the previous session: a field, a country road, a ditch and a farm house.

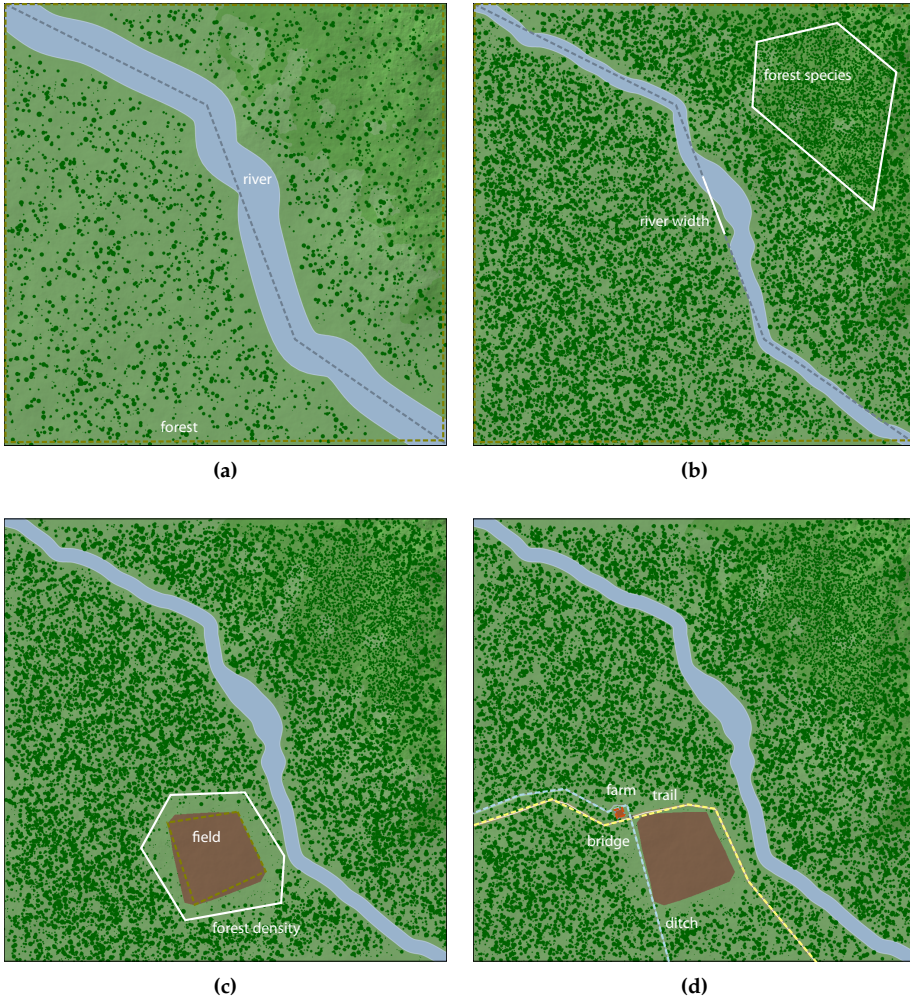


Figure 8.3: Second modelling session: (a) temperate landscape consisting of a forest, through which a river flows (b) after setting semantic attributes and introducing refinements to the river and the forest, (c) a clearance in the forest, created using a density refinement and an agricultural field, (d) introduced a ditch, trail with bridge and farm.

8.2.2 Walkthrough of modelling session

Starting this modelling session, our first step is to quickly create a natural environment in a temperate climate, see Figure 8.3 (a). For this, we use the temperate ecotopes flat lands and hills. The next step is to outline a river's path through the landscape.

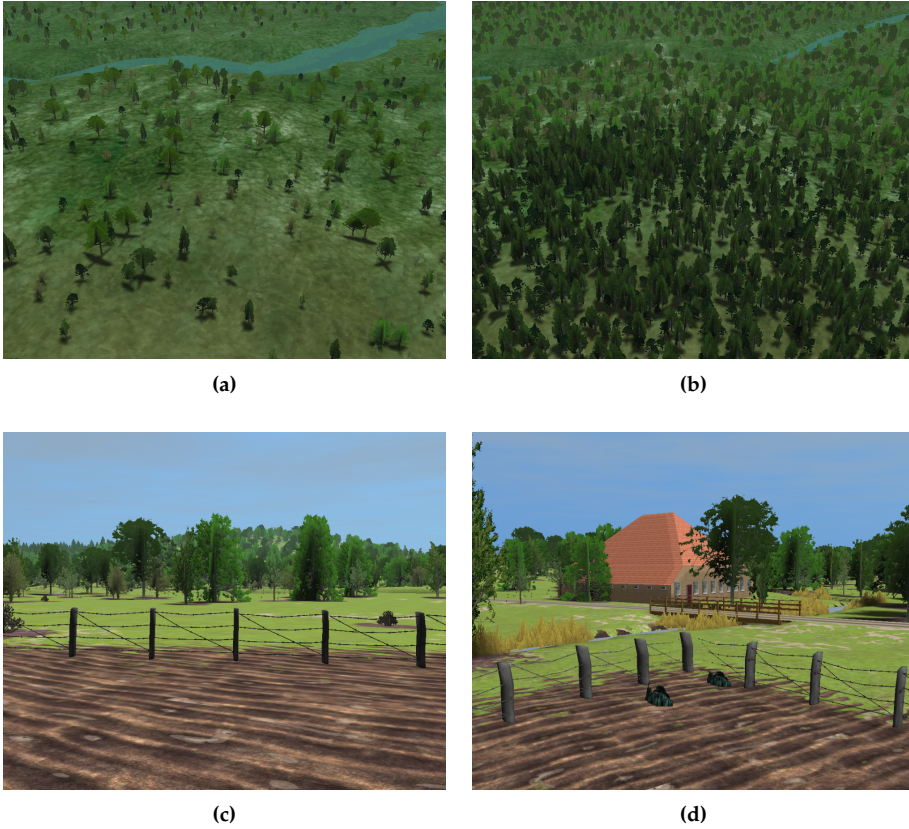


Figure 8.4: 3D virtual world, resulting from the second example session of Figure 8.3: (a) natural environment generated with default attribute values, (b) refined natural environment, (c) clearing with farming field, (d) farm and wooden bridge over the ditch.

Finally, we define a forest that covers the entire virtual world. Figure 8.4 (a) presents a view from the hill looking down to the river.

Examining this, we realize that the results generated using the default feature specification attributes do not quite match our intent: we would like to have a more dense forest, and a somewhat narrower river. In Figure 8.3 (b), we see the result of our refinements, for which we performed four edit actions. We increased the *density* of the forest. To create more variation in the forest, we defined a *species* refinement at the hill and choose a different set of vegetation to be distributed within that area. Next, we modified the uniform river *width* attribute from its default to 25m, and introduce a small local variation to this using a *width* refinement. Figure 8.4 (b) depicts the refined natural environment.

Now that we are satisfied with the natural landscape, we want to introduce some small, man-made elements to it. We start by specifying an agricultural field in the forest. Through consistency maintenance, the forest loses part of its extent to this new field. As a result, the field is cleared of trees. However, for the settlement, we would like to have a larger area around this field where the forest is very lightly populated. For this, we create a *density* refinement shown in Figure 8.3 (c). We can see the resulting farming field and forest clearance in Figure 8.4 (c).

To finish this session, we place a number of small scale features, shown in Figure 8.3 (d). First, we specify a ditch as a water supply along the field. Secondly, we create a trail road around the field; where it crosses the ditch, a small wooden bridge is created. Finally, we place a farm house. The now complete environment is depicted in Figure 8.4 (d).

8.3 Modelling session 3: semantic constraints

This final modelling session makes use of semantic constraints, combined with procedural sketching.

8.3.1 Motivation

The previous session demonstrated the ability to customize the world on a relatively fine-grained scale, using semantic attributes and feature refinements. In the final modelling session, we focus on declaring high-level intent using semantic constraints.

Semantic constraints, presented in Section 6.2, illustrated with some results of the *line-of-sight*, *choke point* and *route* constraints. One of the convenient aspects of semantic constraint is the fact that they are automatically maintained during the iterative modelling process.

In this session we create a small island in an ocean, in which we will apply a *line-of-sight* constraint. As we only use high-level operations in this session, the modelling session time is short, in total about 10 minutes. The session is set in Europe, using the corresponding virtual world template.

The plan for this modelling session is to create an attractive island with forests, a small town and sandy beaches, as of yet unspoiled by tourism. On this island, we will build the first holiday villa, employing the following set of features: an undeveloped field, a building lot, a rural road, a small city, forests, and tree lines.

8.3.2 Walkthrough of modelling session

Starting this modelling session, our first step is to create the island in its rustic state, see Figure 8.5 (a). For defining the island in the ocean, we use coastal and temperate ecotopes. On this island, we outline several forests, and a road that runs along the coast, and some tree lines next to the road. In the middle of the island, we outline a

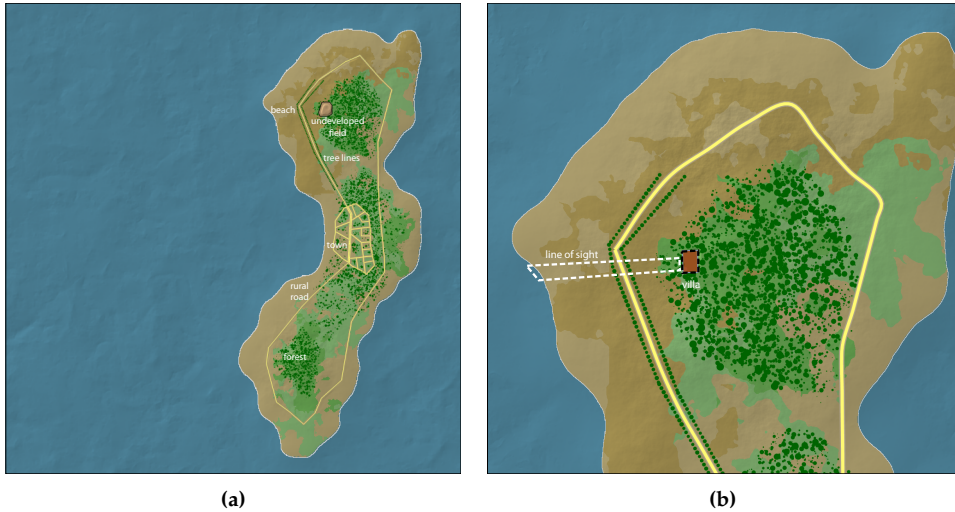


Figure 8.5: Third modelling session: (a) island with a town, much vegetation and beaches, a rural road along the coast, and an undeveloped site where a villa is planned to be built. (b) Definition of *line-of-sight* constraint from the holiday villa to the beach and ocean.

small town across the round-trip road. Finally, we define an undeveloped field at the building site of our villa to be. Figure 8.4 (a) shows the view on the island from the villa's planned location.

Next, on the undeveloped field we build the holiday villa (depicted in Figure 4.7). Examining the generated results in 3D, see Figure 8.4 (b), we find that the villa does not have a proper, unobstructed view on the beach and ocean, not even from the balcony, which is an important requirement for such a holiday rental building.

To ensure that this high-level requirement is preserved, we declare a *line-of-sight* constraint from the villa to the beach and waterline (Figure 8.4 (a)). The direct application of this constraint results in an improved view on the beach and ocean, as can be observed in Figure 8.4 (c).

At this point, we realize that our original specification of the island should have been different. Instead of a coastline consisting of only flat beaches, we would like to have a combination of beaches and tall sand dunes. Because of the consistency maintenance mechanisms, such a drastic late change to the landscape presents no problems and requires no additional effort. The forest, road and tree lines automatically restructure to the new landscape profile. Furthermore, because the changes to the landscape are within the extent of the *line-of-sight* constraint, the constraint is automatically re-evaluated. This results in a modification of the regenerated forest and minor changes to the landscape profile, see Figure 8.4 (d).

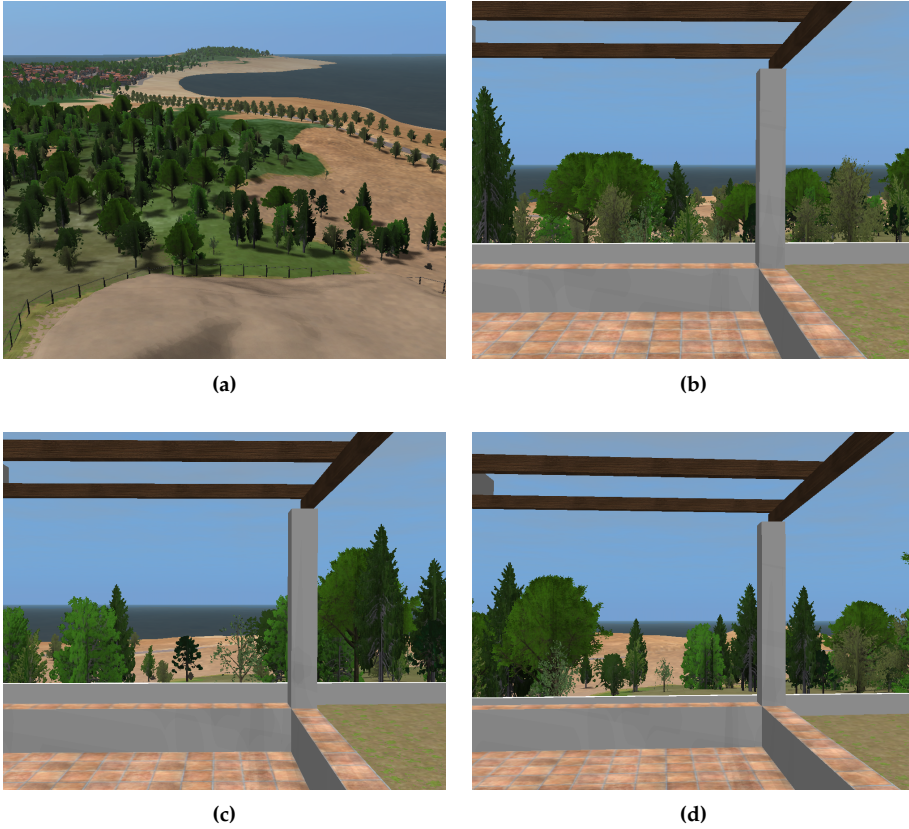


Figure 8.6: Results of the third modelling session of Figure 8.5: (a) the island created using procedural sketching, (b) blocked view from the villa's balcony, (c) clear view after application of *line-of-sight* constraint, (d) *line-of-sight* is maintained while creating high dunes near the beach.

8.4 Discussion

In this chapter, we presented three example modelling sessions performed with our SketchaWorld prototype. The three examples were of varying scale and including different features and edit actions. Together, the three sessions confirmed several important aspects that highlight the advantages of our declarative modelling approach:

1. using procedural sketching, we can create a complete virtual world in minutes (session 1);
2. we can declare and refine our intent to steer procedures in an intuitive and controllable manner (session 2);

3. our high-level requirements can be automatically maintained throughout an interactive modelling session (session 3).

The levels of user control identified in Chapter 6 are featured in the different examples. It is clear that each of these levels has its own added value and that their operations complement each other. All these operations use procedural techniques in some way. As a result, they have increased productivity compared to manual modelling operations.

Another factor that has a large positive effect on the modelling productivity in these sessions is our automatic consistency maintenance mechanism. A simple example of this are the edit actions performed to go from Figure 8.3 (a) to (b); as we set the river to be narrower, this river is automatically resized, the new river bedding is created and the forest reclaims part of its extent lost to the river, placing additional vegetation.

Regarding the *accessibility* of our prototype, we have experienced that procedural sketching and semantic attributes are intuitive enough to use without much hints or explanation. Feature refinements and especially semantic constraints require some hints for new designers to grasp them, but once their function is clear, they are also easy and fast to use.

The example sessions also illustrated the usefulness of *virtual world templates*. Even though geo-typical worlds have no direct real-world correspondence, to obtain a virtual world in a somewhat consistent style, one has to specialize the procedures and content to match specific regions of the world.

Content variety forms a practical limitation of our prototype system. To be able to generate a wide variety of interesting 3D virtual worlds, a large variety of content is required, such as soil material detail textures, building generation scripts or shape grammars, and plenty of other 3D models, from light poles to windmills. For each new virtual world template, (part of) this content needs to be modified, specialized or replaced. This customization typically takes considerable time, and involves both technical and creative work.

The example modelling sessions in this chapter demonstrate how the declarative modelling approach and the SketchaWorld prototype can be applied in practice. In fact, this prototype has already been used in a number of real-world projects, which we will examine in the next chapter.

9

Real-world application of the prototype

The previous chapter showed some illustrative results of the prototype for declarative modelling of virtual worlds. In order to get feedback on the usability of this approach and its interaction methods, we organized informal feedback sessions with Dutch game design professionals. The feedback obtained in these sessions led us to further refine procedural sketching with new levels of user control, as discussed in Chapter 6. However, in order to validate the quality of the approach, it is important to apply it in *real-world* scenarios.

This chapter discusses a number of real-world cases where SketchaWorld has been applied. These projects ran simultaneously with this PhD project (in 2010 and 2011), and involved external partners. The first application of SketchaWorld is for in-house developed simulators for training military personnel. The second project developed the incorporation of GIS data within SketchaWorld in order to combine geo-typical and geo-specific modelling within one framework. The results were exported to Levee Patroller, a serious game for training levee inspection personnel. The last project is currently realizing the SketchaWorld prototype as a plugin for a commercial virtual world modelling system. The resulting system is to be used by military training instructors.

9.1 Case 1: Military training simulators

This case involved exporting virtual worlds to two military training simulators that were developed at TNO: Tactical Air Defence and FACSIM. In a tactical air defence

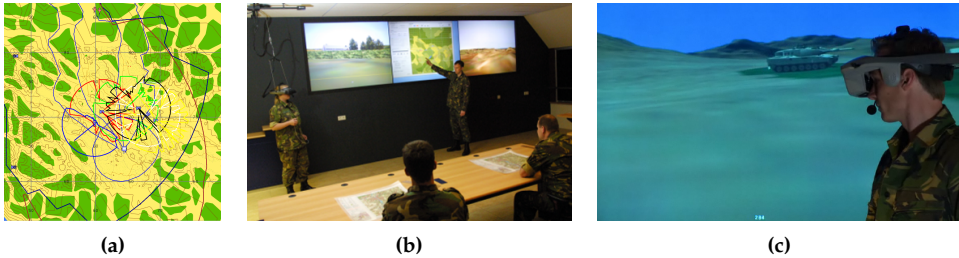


Figure 9.1: Military training simulators. (a) Creating an air defence solution, where the view area of each team is shown, (b) FACSIM training classroom setup, (c) Exported virtual world from SketchaWorld in FACSIM.

scenario, trainees have the challenging task of setting up a ground-based defence system against threats from the air, see Figure 9.1 (a). The goal in such a scenario is to protect a zone of terrain featuring some high-value objects, for instance a city, an airport, or an oil refinery. At their disposal are a number of mobile anti-air teams. The trainees plan the deployment of each team in their zone. For this, they have to consider many variables, factors, and uncertainties, but in all considerations the landscape and its features play a major role. The plan is evaluated by running the threat scenario to see whether their solution successfully defends the zone from air attack. During this evaluation, the trainees have a 3D view on the situation as it evolves.

Instead of air defence, FACSIM focuses on close air support. Close air support missions involve aircraft assisting ground units by engaging nearby hostile ground targets. In such missions it is often very difficult for a pilot to determine the exact position of a (moving) ground target from the air and to engage the target safely, avoiding collateral damage. This is especially the case when friendly units or civilians are close by. Units on the ground often have a much clearer view of the situation and hence have important tactical information the pilot needs in order to perform his mission successfully. The role of a Forward Air Controller (FAC) is to guide pilots to their target, primarily by voice communication. To successfully guide an aircraft to a ground target, the FAC must be able to conceive how the pilot sees the battlefield; he has to identify features and landmarks that are clearly distinguishable from the sky for the pilot to orientate and navigate with, e.g., a village's church, or a nearby forest. Figure 9.1 (b) shows a classroom training using FACSIM.

In our previous work [Smelik 10a], we provide more information on this case and the simulators involved.

9.1.1 Motivation and objectives

One of our motivations for this case is to make automatic creation of virtual worlds accessible to people with no specific expertise in 3D modelling. In particular, one group we want to target are instructors that use serious gaming or training simulators to teach their trainees particular skills or tactics.

Especially in military training, the precise layout of the virtual world plays a very important role. It has a direct effect on the training session, in the sense that the terrain configuration determines for a large part the tactical situation. As a result of this, the choice of terrain in military training simulators turns out to be an important part of creating a training scenario. Scenario creation usually starts with obtaining a suitable virtual world model. However, the instructor often had to choose from a fixed set of geo-specific models that were provided with the simulator. Although these models can cover many possible settings, they do significantly limit the number of potential training scenarios. We believe that it is very beneficial for game-based training, if instructors are able to create their own curriculum, including the virtual world models in which the scenarios are to take place [Kuijper 11].

For training scenarios, geo-typical virtual worlds can be a more suitable alternative than the geo-specific worlds of, for instance, mission rehearsal scenarios. Therefore, it is very convenient for instructors to be able to use SketchaWorld to design new geo-typical worlds or modify existing ones. This allows them to have direct control on the complexity of the tactical situation, e.g., by introducing blocking features to limit lines of sight, or, the other way around, to use a line-of-sight constraint to guarantee visibility. A more complex virtual world typically entails a more difficult scenario, as, for instance, a good air defence solution will be less obvious, or it might be more difficult to get a clear situation overview for a pilot. Increasing the variety in scenarios also prevents the trainees from becoming too familiar with the specifics of a particular environment.

9.1.2 Technical realization

Both Tactical Air Defence and FACSIM use an in-house simulation engine, named Enhanced Virtual Environments (EVE). The visual component of this engine is based on OpenSceneGraph, which means that exporting the 3D virtual world model was fairly straightforward. Only minor modifications had to be made to the exported model, to match the hardware restrictions of the operational training systems.

However, for an instructor to be able to use virtual worlds generated in SketchaWorld, the scenario editors required additional information to be derived from our semantic virtual world model. For both simulators, these were digital maps at different zoom levels and resolution, and a height-map in a specific format. To provide this, we created a custom export module. This module automatically generates the complete set of files in the required formats and folder structure. Furthermore, it generates all configuration files necessary to integrate the new virtual world in the scenario

editor. This way, the pipeline from a generated virtual world in SketchaWorld to a scenario in e.g., FACSIM requires no significant manual effort.

9.1.3 Results

Using the developed export module, we automatically export any new virtual world created in SketchaWorld to both training simulators. Figure 9.1 (c) shows a virtual world exported to EVE. Initial experiences with this setup are encouraging, and training instructors are enthusiastic about the possibilities of our approach. We would like to use this setup to further evaluate how training instructors can use SketchaWorld to customize their training curriculum.

9.2 Case 2: Levee patroller

This section describes a Knowledge Transfer Project, funded by the GATE research program, and in cooperation with Deltares (<http://www.deltares.nl>), a research institute specializing in water and soil management. It aims at using SketchaWorld to procedurally create geo-specific game worlds for their game Levee Patroller.

9.2.1 Motivation and objectives

Levee Patroller is a serious game developed by Deltares [Harteveld 10], for use in their training curriculum for professional patrollers from Dutch water management boards, who inspect the many levees that protect The Netherlands from the North Sea and inland rivers. The objective of the game is to learn identify incipient failures of levees, classify them, their causes and the urgency of the situation, and report an accurate assessment back to a control room. Trainees navigate the virtual world in first-person perspective, armed with (virtual) measuring and communication equipment, looking for cues of a potential levee failure, for instance a minor crack in the levee surface or small-scale water breaches.

So far, Levee Patroller uses a number of hand-modelled geo-typical worlds resembling Dutch rural landscapes. However, water management boards have shown interest in training patrollers in a virtual replica of the actual environments they inspect. Hand modelling these new virtual worlds on the basis of photographs and maps was deemed too laborious and expensive. Hence the need for automatic generation of such geo-specific environments.

A common method for generating geo-specific 3D virtual world models is to start from GIS source data. This data typically includes grids of elevation measurements, photographs obtained from satellites or airplanes, and polygonal vector data describing, for instance, the position of houses, roads, trees, etc. Sophisticated commercial modelling tools can combine these different data sources and automatically generate a more or less corresponding 3D virtual world.

A number of challenges are inherent to that process, for example:

- the required source data might be unavailable, restricted or too expensive to obtain;
- the source data may contain errors or there can be inconsistencies between different sources;
- the available source data might lack the level of detail required to obtain a usable virtual world.

In most cases, some source data is available for reasonable prices, but in itself is not enough to generate a complete and usable environment. This means that hand modelling is often required, either by augmenting the source data by hand (for instance, manually interpreting and extracting building footprints from a satellite image) or by enriching the 3D virtual world model using standard 3D modelling techniques.

For training games such as Levee Patroller, an additional challenge exists. Its goal is not to reproduce a model that corresponds *exactly* to the real world, but a recognizable world that is suitable for the training purposes.

To adapt a 3D virtual world to a playable Levee Patroller game level, some sacrifices to realism and real-world correspondence have to be made in order to better support the gameplay and provide the desired training value. An example of this is the size and scale of the game world: to fit within a time slot of training session, the game world will have to be more compact and concentrated than the actual patrol territories of water boards. Furthermore, there are a few technical constraints imposed by Levee Patroller, such as predetermined size and shape for levee features in order to support previously scripted levee failure mechanisms.

For the common scenario in which available source data lacks the required resolution, procedural modelling is an ideal method to fill in details. However, current commercial GIS packages can add procedural detail only to a very limited extent by providing, for instance, a method to randomly scatter objects in a polygonal area. Especially for man-made environments, this kind of method will not result in suitable models.

Considering this, the declarative modelling approach as implemented in Sketcha-World was found very suitable for this problem. With a number of extensions implemented in this project, the prototype provides a fast and efficient way to create geo-specific game worlds, by importing available GIS data, procedurally generating missing details, and allowing for editing of the world to match it with gameplay requirements. In this way, Levee Patroller can provide game levels that closely resemble real-world locations, allowing users to train their skills in areas they are familiar with.

9.2.2 Technical realization

The technical challenges of this project can be split into two categories:

1. Import and process GIS data in the SketchaWorld prototype;
2. Export facilities to the game engine and level editor used by Levee Patroller.

Importing GIS data into SketchaWorld

The two main reasons for introducing geo-specific worlds in SketchaWorld are that it allows us to enrich coarse GIS data with procedural details, and that we can interactively edit the virtual world at a high level of abstraction. It was, therefore, a natural choice to map imported GIS elements to feature *specifications* instead of directly to semantic objects or geometry. In this way, we can apply our procedural methods to the input data. Furthermore, we can interactively edit and experiment with the geo-specific world, benefiting from our consistency maintenance and short feedback loop.

As mentioned above, GIS data typically comes in two forms, raster data and vector data. Elevation data is normally stored in raster format, while all features and objects are provided as layers of 2D polygonal shapes (point, polyline or polygon) with arbitrary attributes.

Importing elevation data into our framework's landscape is straightforward, as we can import the data directly to our height-map tiles. Depending on the source, the elevation resolution of GIS data is typically fairly coarse (e.g., 30m). In that case, we linearly interpolate the elevation values to obtain a resolution suitable for landscape modifications, such as road embankments. However, for The Netherlands, high-resolution elevation data is available (1m). Furthermore, we use filtered elevation data, i.e., measured elevation data that has been post-processed to remove e.g., buildings from the profile, and fill any area for which no valid data is available with a flat elevation profile.

There are a few issues with the geo-specific version of our landscape though. First of all, we cannot derive a landscape specification (i.e., a grid of ecotopes) from the elevation raster data alone. Instead, we have to use the source data directly as the resulting landscape. This is somewhat inconvenient, as we cannot use procedural sketching to edit the landscape. Secondly, the distribution of soil material is not defined in the GIS raster data. This means that we initially have to set a uniform soil material for the landscape. However, we can refine this uniform soil material distribution later, using vector data describing land use. Finally, the elevation data can include measurement inaccuracies or, more significantly, mismatches between the elevation profile of a feature and its vector representation. An example of this is a ditch, of which the vector representation and the elevation profile can be meters apart due to shifts in position. For this reason, we apply the landscape modifications of all features to ensure at least that e.g., a ditch always has a corresponding profile in the elevation map.

The specifications of the features of the virtual world are derived from the imported layers of vector data. As explained, these vector data elements consist of

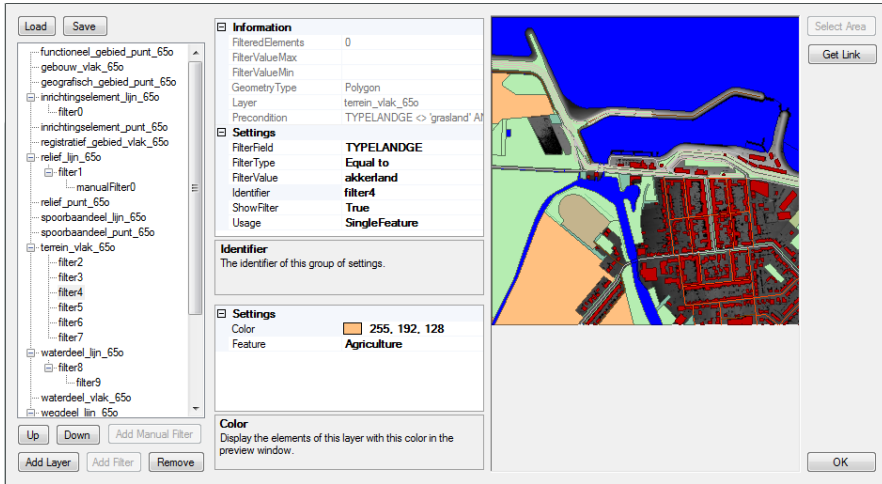


Figure 9.2: GIS import interface of SketchaWorld, showing included vector layers, mapping definition tools and a map visualisation of the data.

primitive polygonal shapes, which we can map to the outlines of feature specifications.

However, determining which element in vector data should be mapped to what type of feature in SketchaWorld is a process that cannot be performed automatically. This is because the attributes that provide additional information on the vector elements are not standardized. Although several classification schemes have been proposed in the past, typically there is no semantic model used in vector element classification. Instead, all data annotations are hand-written in natural language and, as a result, are often incomplete and inconsistent. Importing vector data thus requires one to manually define a mapping scheme from vector elements to feature specifications. We devised a number of ways to create this mapping:

- A vector layer can be uniformly mapped to one particular type of feature;
- A *filter* can be defined for a vector layer, performing a selection query that can match on a specific attribute value (e.g., an element description string, such as “tree line”), or defined value ranges (e.g., all elements with an area smaller than 400m²). Queries can be combined with the standard Boolean operators (e.g., and, not, or). If the selection condition is satisfied for a certain vector element in the layer, the mapping defined for the filter is applied to this element. Multiple filters can be applied to a vector layer being evaluated in sequence.
- The above methods are applied to all vector elements in a layer. Using *manual selection*, one can make an exceptional mapping for a particular vector element.

For instance, in a parcel layer that is uniformly mapped to row houses, we can manually map one specific vector element to a particular sort of building, e.g., a shopping mall.

For all these mapping modes, we can also set specific values for the semantic attributes of the selected feature specification, e.g., to set the type of road or the species of trees in a tree line. Combined with the filters and manual selection, this allows one to define an accurate and refined mapping scheme.

Our GIS import interface is shown in Figure 9.3 (a). Here, we can examine the imported layers and define the feature mappings. Furthermore, the GIS data is visualised in a map view, showing which vector elements a filter applies to, etc. This map interface is used to select the area of interest, i.e., to define within the data set the area to import into our virtual world, and also to manually select an element for a custom mapping. A mapping scheme can be saved in order to apply it to another area of interest within the data set. Note that such a scheme is often specific to a particular data set, as other data sets may define vector attributes in a different language or terminology.

After a batch import of the selected area as a set of feature specifications, designers can inspect the results in the 3D virtual world preview, and modify the world using our standard methods such as procedural sketching and feature refinements (see Chapter 6).

Exporting results to game engine

After importing GIS data and editing the resulting world in SketchaWorld, the next step is to export the results to the Levee Patroller serious game. Levee Patroller was created using the Unreal engine, version 2 (UE2 [Epic Games 02]). The specifics of this engine introduce several constraints that required us to make a custom export module. First of all, Unreal is a commercial engine, for which no source or external API is publicly available. This excludes any direct communication of generated results over an interface.

For this reason, SketchaWorld has to export the generated world as a set of files to be imported in the Unreal game level editor. Importing all these individual files by hand would significantly slow down the modelling pipeline. Fortunately, the Unreal level editor supports a custom scripting language. To automate the import process, we generate a number of scripts that create the basic level configuration, and, while importing, correctly place all geometric objects in the level. These scripts are executed from within the Unreal level editor.

Being a somewhat older game engine, UE2 only supports the import of very specific file formats, such as the Autodesk Scene Export (.ase) for geometric models. For this reason, we created conversion modules to export our height-map and geometric objects to these file formats.

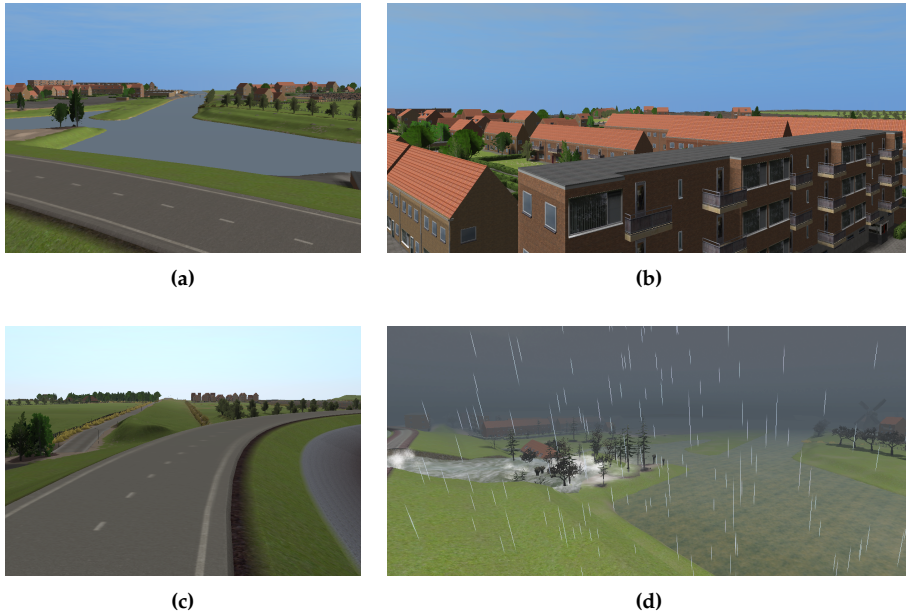


Figure 9.3: From GIS data to Levee Patroller game level. GIS data imported via the interface as features in SketchaWorld: (a) view on the river, (b) inside the town, (c) rural area. (d) Final game level in Levee Patroller, showing a levee failure.

The engine has some additional performance constraints that have to be taken into account, for example the maximum supported size of a virtual world is around 4 km². To gain optimal rendering performance, individual geometric objects, such as trees, doors and windows, should whenever possible be *instanced* (a technique that enables rendering several instances of one mesh simultaneously, using one draw call). We incorporated support for this into our export module.

Using this export module, transferring results from SketchaWorld to Levee Patroller can be performed in a matter of minutes, without significant manual effort. This very much helps to accelerate the overall pipeline from GIS data to a playable Levee Patroller game level.

9.2.3 Results

The project has resulted in a fast and relatively smooth pipeline from GIS data to a playable game level in Levee Patroller. As described above, at three stages in the process, designers can influence the generated results:

1. by defining the mapping from vector element to feature, in the GIS import

process;

2. by editing the world using procedural sketching to better suit gameplay requirements;
3. by manually refining small details in the imported game level, to improve gameplay or aesthetics.

Figure 9.3 (b) shows part of a Levee Patroller game level generated using Sketcha-World. In the final phase of this project, we are now focussing on enhancing the level of detail of the virtual world, by procedurally generating smaller details, such as urban clutter. This will reduce the amount of manual modelling required in the last stage of the process.

9.3 Case 3: Landscape

This section describes a National Technology Project named Landscape, funded by the Dutch Ministry of Defence, and in cooperation with re-lion (see <http://www.re-lion.com>), a Dutch simulation company specialized in driving simulators and military training systems. Much of the motivation discussed in Section 9.1 applies to this project as well. The special focus of this project is to provide military training instructors with accessible tools to create new virtual worlds for two commercial military training games:

1. Virtual Battlespace 2 (VBS2), an infantry training game, created on the basis of the entertainment game Armed Assault [Bohemia Interactive Australia 11]. VBS2 is a very popular military training simulator in many countries, including the USA, UK and Australia.
2. SteelBeasts, a training game for armoured vehicles, also based on an entertainment game by the same name [eSim Games 11].

The project results in a prototype modelling system that can export both geo-specific and geo-typical virtual worlds to both simulators. The prototype will be used by a large group of military training instructors to customize their training scenarios.

The prototype in development is based on re-lion's in-house developed Builder modelling system [re-lion 11]. Its main interaction method will be manual modelling in 3D. As modelling a virtual world in 3D can be very complex and time intensive, part of the challenge in this project is to provide instructors with smart 3D modelling tools that reduce this complexity and increase their productivity. An example is their road creation tool, which, on the basis of user-defined Bézier curves and a selected lateral profile, automatically creates plausible 3D roads.

Still, even with improved modelling tools, manually creating a large virtual world can be very time consuming. For geo-specific virtual worlds, instructors can import

GIS data sets as the basis for the virtual world. For geo-typical worlds, SketchaWorld comes into play. Instructors can use our interaction methods, such as procedural sketching, to quickly create a useful basis of the virtual world. Next, using current Builder tools, they can perform small-scale refinements on this world, before exporting the results to the mentioned training games.

In the architecture of the Landscape prototype, SketchaWorld is integrated as a plugin to Builder. A common interface has been defined to support communication between the systems and to exchange procedurally generated results. In this setup, re-lion's Builder provides instructors with the 3D virtual world preview, in which they can directly edit the generated results. This preview is updated through notification events of changes to the virtual world semantic model, in the same way as the SketchaWorld views are kept in sync (see Figure 7.2).

At the time of writing, this project is still ongoing. The basic interface between SketchaWorld and Builder has been established, and generated features and objects are interactively processed by Builder to create the 3D geometric virtual world model. In the current implementation, there are two separate phases in the modelling process: the creation of the virtual world using procedural sketching, and the manual refinements in Builder. To *seamlessly mix* these modes of editing, will require to overcome many of the problems discussed in Chapter 6, for which the ability to lock elements of the virtual world is an excellent candidate.

9.4 Discussion

In this chapter, we discussed three projects in which SketchaWorld is being applied to a real-world case. These projects confirm that our approach is mature. Furthermore, the projects provided useful feedback, tests and validation for our approach and the current implementation of the prototype.

From our experience in these projects, we can conclude that export facilities and modules of the virtual world semantic model are straightforward to create. We will continue to broaden the range of game engines and file formats to which we can export results.

Handling *geo-specific* data can quickly become a very complex process, because of the data inconsistencies and correlation errors. These issues are already being addressed by ongoing other research and by dedicated, complex GIS processing tools. Considering this, it makes sense to restrict the use of geo-specific data in our approach to applications where regional resemblance is more important than direct one-to-one correspondence. This can be the case if available GIS data is very coarse and procedural interpretation is required to obtain a lifelike world, or if gameplay constraints and editability have a higher priority than real-world correlation.

The importance and urgency of the research on the integration of manual and procedural modelling is again underlined in the discussed Landscape project. Seamlessly mixing manual, fine-grained 3D modelling with procedural operations and

consistency maintenance remains a challenging research topic. We need more fine-grained editing facilities to preserve both virtual world consistency and designer intent simultaneously.

At the time of writing, the mentioned projects are all still in progress, and will continue after this PhD project. Besides these, we will continue to look for opportunities and collaborations to apply the results of this research in practice. The current projects are all in the domain of serious gaming for defence or safety. It would be very useful and interesting to get more feedback and validation by applying this approach to cases in the entertainment gaming domain.

10

Conclusions

This chapter concludes the thesis with a discussion on the merits of our declarative modelling approach and on some of the opportunities we see for extending this research. We start with a summary of the main research contributions of our approach. Next, we discuss the main advantages and current limitations of our research and prototype. Finally, we present recommendations for future work in this area.

10.1 Research contributions

We identified the challenges currently faced by designers of virtual worlds, arising from limitations of both manual and procedural modelling approaches. From recent developments in procedural generation research, we concluded that improvements in user control, interactivity and integration of results are now not only feasible but also essential to increase the acceptance of procedural generation in mainstream virtual world development. Considering this, we can revisit our main research question as posed in the introduction of this thesis:

How can we improve the process of virtual world generation?

The framework for *declarative modelling of virtual worlds* presented in this thesis improves the process of virtual world generation in a number of ways:

1. by integrating procedural methods, it provides a considerable productivity gain with respect to manual modelling;

2. by providing designers with intuitive and goal-oriented editing facilities, it makes procedural content generation accessible to non-specialist designers;
3. by offering interactive procedural modelling operations made possible by performance optimizations and GPU computing, it realizes a short feedback loop between edit action and effect;
4. by providing interaction methods that operate at different levels of granularity, it offers adequate user control to generate virtual worlds that better match designer intent;
5. by using its automatic consistency maintenance mechanism, designers are encouraged to freely manipulate and experiment with any terrain features in the virtual world, which in turn is always kept internally consistent.

We can therefore conclude that the combination of these contributions, as successfully implemented in our prototype SketchaWorld, provides substantial help for designers to generate virtual worlds.

We now briefly revisit the main chapters in this thesis to summarize their individual contribution in answering the research question.

A semantic model for virtual worlds

We presented a semantic model of the virtual world, which structures terrain features in several levels of abstraction, from a coarse user specification to concrete 3D geometry. This model separates the semantic definition of a terrain feature from the actual procedural technique used to generate it, making it flexible to incorporate new procedural techniques. Moreover, using our semantic library, objects in each terrain feature are enriched with relevant information on their functionalities, services and roles in the virtual world. Using the library, we support additional applications of the generated objects besides visualisation, as the required representations can be automatically derived from our model. Our semantic model for virtual worlds provides a solid foundation upon which to build our framework.

Integration of procedural methods

We presented the structured integration of procedural methods at two levels of abstraction: terrain features and semantic objects. At the level of terrain features, we conceived a communication *interface* for the interaction between the framework and integrated procedural methods, aimed at generating terrain features. At the level of semantic objects, we introduced the concept of a *semantic moderator* to coordinate different procedural techniques to collaborate in the generation of a consistent complex object, such as a building, according to a global plan. With these two integration

methods embedded in our modelling framework, we are able to harmonically apply the existing body of procedural techniques in combination to produce complete virtual worlds with highly detailed objects.

Virtual world consistency maintenance

We introduced automatic virtual world consistency maintenance, which uses generic methods to handle interactions among features. This removes a huge burden from the designer, who is now freed from the task of continuously fitting all features together and keeping the world consistent. Because of the generality of these methods, introducing a new feature requires no dedicated interaction handling methods with other incorporated features.

User control in procedural modelling

We combined intuitive interaction methods with several levels of user control: semantic constraints, procedural sketching and feature refinements. All these interaction methods can be freely mixed and are interactively evaluated, providing designers with a short feedback loop. Each of these methods has its own added value and they complement each other. Also, we introduced the ability to lock an area of the virtual world, helping designers to limit possible inconveniences of automatic consistency maintenance. For this, we identified the different types of zones required to create plausible transitions from locked to regenerated content.

10.2 Discussion of results

As described in Chapter 7, our framework for declarative modelling of virtual worlds was implemented in our fully functional prototype *SketchaWorld*, which has already been applied in several real-world cases (see Chapter 9). It demonstrates that the above mentioned contributions perform in an interactive modelling environment, and are mature enough to be applied in practice.

Using our *SketchaWorld* prototype, designers can concentrate on *what* they want to create instead of *how* they should model it. The prototype is also quite easy to use, and, as such, makes procedural generation accessible to novice designers. In informal user test sessions, we have observed that people are able to quickly grasp the interaction methods and create a basic 3D virtual world. From our own experiences, and based on the feedback we have received during the project, we can safely conclude that, with minimal experience with procedural sketching, our declarative modelling prototype allows one to iteratively model a relatively detailed and useful 3D virtual world in less than an hour.

10.2.1 Current limitations

Naturally, the *usefulness* of a generated virtual world model very much depends on the context in which it is to be applied. In this research, we defined the scope of our virtual worlds as *geo-typical*, relatively large outdoor environments. This puts some restrictions on the potential applications of our current research results. In the entertainment gaming industry, for example, such an approach is not directly suitable for abstract game worlds. For small-scale multi-player maps in classic first person shooters, a basis for the virtual world can be generated, but as gameplay specific balancing requirements cannot currently be incorporated in the procedural generation process, much manual modelling effort would be still required. In the training and simulation domain, even though we elaborated a proof of concept for incorporating *geo-specific* input in the declarative modelling process, our approach is not intended for applications requiring one-to-one correspondence, such as mission preparation, for which many suitable alternatives exist.

An important limitation of our current research is that we have not yet seamlessly integrated fine-grained user control (e.g., manipulating individual objects) in our framework. Obviously, one can switch to manual editing of the generated content, in SketchaWorld or any external 3D modelling tool. However, this is inflexible, excludes any return to procedural regeneration or consistency maintenance, and thus limits iterative modelling, which is exactly what is currently plaguing the creative process of virtual world design. As identified in Chapter 6, there are quite a few research challenges that need to be addressed before the integration of fine-grained user control can be successful. As this level of user control is very much desired by experienced virtual world designers, we strongly recommend it as future work.

Regarding our prototype implementation, as mentioned in Chapter 7, currently SketchaWorld can comfortably model virtual worlds of medium sizes, e.g., 400 km². In order to alleviate these size limitations, we would need to implement substantial performance optimizations. Another convenient improvement would be to support alternative input devices in addition to the current mouse-controlled procedural sketching interface. As game designers often prefer to sketch out concept drawings using a tablet, it is advantageous to also support input on the basis of sketch strokes.

10.3 Recommendations for future work

We believe that the framework described in this thesis and SketchaWorld, its prototype implementation, provide a stable and appropriate platform for continuing research in the context of declarative modelling of virtual worlds. To conclude this thesis, we present a set of recommendations for future work in this field:

- *Bringing research together.* We think that the integration framework presented in this thesis creates ample opportunities for further research, including collaborations. The current SketchaWorld prototype provides a flexible platform

for integrating new research results. It would be valuable to cooperate with other researchers in integrating novel, state of the art procedural methods. This would allow for validation and application of their new and specialized feature generation methods in the larger context of complete virtual worlds, and would enrich the prototype with new or improved results. Furthermore, it would provide additional test cases for many aspects of the framework, such as the integration of procedures, feature interactions and consistency maintenance.

- *Applying results in practice.* Throughout the research project, we have already received quite some informal feedback from a diverse group of people. However, we consider it important to have more user feedback on the effectiveness and intuitiveness of our approach. We have no plans for a large formal user study; instead, we plan to obtain this feedback by deploying the prototype SketchaWorld in different projects and contexts, to reach a wide and diverse range of potential users. In Chapter 9, we discussed several ongoing projects that are good examples of this. The coming years, we will continue to look for opportunities to apply these research results and let more designers experiment with our tool.
- *User control in procedural modelling.* One of our main goals was to make procedural modelling of virtual worlds more controllable and accessible, and we think we have successfully contributed to this goal. Still, we realize that a lot more remains to be done in this field to provide designers with finer user control. We would definitely recommend that research in the field of procedural modelling continues to focus on this goal. We believe it to be the key factor for the acceptance of procedural techniques in mainstream content creation. In particular, as discussed in Chapter 6, we see much value in providing designers with effective means of *preserving* content, both by means of locking facilities for protecting elements of the virtual world, and by offering adequate manual edit operations on generated content. The main challenges here are defining *generic* and *reusable* approaches to these problems, and providing designers with very fine-grained ways to declare their *intent*.

The last five years have seen increasing attention being paid to procedural generation of virtual worlds, both in academia and in industry. With the recent advances in processing power and GPU computing, combined with the increasing popularity of huge game worlds and the widespread application of game technology outside the entertainment domain, the time seems ripe for procedural generation of virtual worlds. We therefore expect that, in the coming years, we will see a rise in practical applications of procedural techniques and an increased acceptance of the procedural generation as a worthy and effective modelling method for mainstream content creation.

Bibliography

- [Akenine-Möller 08] Tomas Akenine-Möller, Eric Haines & Natty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.
- [Aliaga 08] Daniel G. Aliaga, Carlos A. Vanegas & Bedřich Beneš. *Interactive Example-based Urban Layout Synthesis*. ACM Transactions on Graphics: Proceedings of ACM SIGGRAPH Asia 2008, vol. 27, pages 1–10, December 2008.
- [Amburn 86] Phil Amburn, Eric Grant & Turner Whitted. *Managing Geometric Complexity with Enhanced Procedural Models*. In SIGGRAPH '86: Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques, pages 189–195, New York, NY, USA, 1986. ACM.
- [Anh 07] Nguyen Hoang Anh, Alexei Sourin & Parimal Aswani. *Physically based Hydraulic Erosion Simulation on Graphics Processing Unit*. In GRAPHITE '07: Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia, pages 257–264, New York, NY, USA, 2007. ACM.
- [Belhadj 05] Farès Belhadj & Pierre Audibert. *Modeling Landscapes with Ridges and Rivers: Bottom Up Approach*. In GRAPHITE '05: Proceedings of the 3rd International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia, pages 447–450, New York, NY, USA, 2005. ACM.
- [Belhadj 07] Farès Belhadj. *Terrain Modeling: a Constrained Fractal Model*. In AFRI-GRAPH '07: Proceedings of the 5th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa, pages 197–204, New York, NY, USA, 2007. ACM.
- [Beneš 01] Bedřich Beneš & Rafael Forsbach. *Layered Data Representation for Visual Simulation of Terrain Erosion*. In SCCG '01: Proceedings of the 17th Spring Conference on Computer Graphics, pages 80–86, Washington, DC, USA, April 2001. IEEE Computer Society.
- [Beneš 11a] Bedřich Beneš, Michel Abdul Massih, Philip Jarvis, Daniel G. Aliaga & Carlos A. Vanegas. *Urban Ecosystem Design*. In I3D '11: Symposium on Interactive 3D Graphics and Games, pages 167–174, New York, NY, USA, 2011. ACM.
- [Beneš 11b] Bedřich Beneš, Ondřej Šťava, Radomír Měch & Gavin Miller. *Guided Procedural Modeling*. In Computer Graphics Forum: Proceedings of Eurographics 2011, pages 325–334, Llandudno, UK, 2011. Eurographics Association.
- [Bernhardt 11] Adrien Bernhardt, André Maximo, Luiz Velho, Houssam Hnaidi & Marie-Paule Cani. *Real-time Terrain Modeling using CPU-GPU Coupled Computation*. In SIBGRAPI '11: Proceedings of the 24th Conference on Graphics, Patterns and Images, 2011.

- [Bethesda Game Studios 11] Bethesda Game Studios. *The Elder Scrolls V: Skyrim*. Available from <http://www.elderscrolls.com>, 2011.
- [Bidarra 00] Rafael Bidarra & Wim F. Bronsvort. *Semantic Feature Modelling*. Computer-Aided Design, vol. 32, no. 3, pages 201–225, 2000.
- [Bidarra 10] Rafael Bidarra, Klaas Jan de Kraker, Ruben M. Smelik & Tim Tutenel. *Integrating Semantics and Procedural Generation: Key Enabling Factors for Declarative Modeling of Virtual Worlds*. In Proceedings of the FOCUS K3D Conference on Semantic 3D Media and Content, Sophia Antipolis - Méditerranée, France, February 2010.
- [Bohemia Interactive Australia 11] Bohemia Interactive Australia. *Virtual Battlespace 2*. Available from www.vbs2.com, 2011.
- [Bruneton 08] Éric Bruneton & Fabrice Neyret. *Real-time Rendering and Editing of Vector-based Terrains*. In Computer Graphics Forum: Eurographics 2008 Proceedings, vol. 27, pages 311–320, Crete, Greece, 2008.
- [Bundysoft 11] Bundysoft. *L3DT*. Available from <http://www.bundysoft.com/L3DT/>, 2011.
- [Cagdas 96] Gulen Cagdas. *A Shape Grammar Model for Designing Row-houses*. Design Studies, vol. 17, no. 1, pages 35 – 51, 1996.
- [Charman 93] Philippe Charman. *Solving Space Planning Problems Using Constraint Technology*. In NATO ASI Constraint Programming: Students' Presentations, TR CS 57/93, Institute of Cybernetics, Estonian Academy of Sciences, Tallinn, Estonia, pages 80–96, 1993.
- [Chen 08] Guoning Chen, Gregory Esch, Peter Wonka, Pascal Müller & Eugene Zhang. *Interactive Procedural Street Modeling*. In SIGGRAPH '08: Proceedings of the 35th Annual Conference on Computer Graphics and Interactive Techniques, vol. 27, pages 1–10, New York, NY, USA, 2008. ACM.
- [Coelho 05] António Fernando Coelho, António Augusto de Sousa & Fernando Nunes Ferreira. *Modelling Urban Scenes for LBMS*. In Web3D '05: Proceedings of the 10th International Conference on 3D Web Technology, pages 37–46, New York, NY, USA, 2005. ACM.
- [de Carpentier 09] Giliam J.P. de Carpentier & Rafael Bidarra. *Interactive GPU-based Procedural Heightfield Brushes*. In FDG '09: Proceedings of the 4th International Conference on the Foundations of Digital Games, Florida, USA, April 2009.
- [de Villiers 06] Matthew de Villiers & Neilan Naicker. *A Sketching Interface for Procedural City Generation*. Technical report, Department of Computer Science, University of Cape Town, November 2006.
- [Deussen 98] Oliver Deussen, Pat Hanrahan, Bernd Lintermann, Radomír Měch, Matt Pharr & Przemyslaw Prusinkiewicz. *Realistic Modeling and Rendering of Plant Ecosystems*. In SIGGRAPH '98: Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques, pages 275–286, New York, NY, USA, 1998. ACM.
- [Doran 10] Jon Doran & Ian Parberry. *Controlled Procedural Terrain Generation Using Software Agents*. IEEE Transactions on Computational Intelligence and AI in Games, vol. 2, no. 2, pages 111–119, June 2010.
- [Ebert 03] David S. Ebert, Steven Worley, Forest K. Musgrave, Darwyn Peachey & Ken Perlin. *Texturing & Modeling, a Procedural Approach*. Elsevier, 3rd edition, 2003.

- [Epic Games 02] Epic Games. *Unreal Engine 2*. Available from <http://www.unrealengine.com>, 2002.
- [Esch 07] Greg Esch, Peter Wonka, Pascal Müller & Eugene Zhang. *Interactive Procedural Street Modeling*. In SIGGRAPH '07: ACM SIGGRAPH 2007 sketches, New York, NY, USA, 2007. ACM.
- [eSim Games 11] eSim Games. *SteelBeast Professional*. Available from <http://www.esimgames.com>, 2011.
- [Finkenzeller 08a] Dieter Finkenzeller. *Detailed Building Façades*. IEEE Computer Graphics and Applications, vol. 28, no. 3, pages 58–66, 2008.
- [Finkenzeller 08b] Dieter Finkenzeller & Jan Bender. *Semantic Representation of Complex Building Structures*. In CGV '08: Computer Graphics and Visualization, pages 259–264, Amsterdam, The Netherlands, July 2008.
- [Fournier 82] Alain Fournier, Don Fussell & Loren Carpenter. *Computer Rendering of Stochastic Models*. Communications of the ACM, vol. 25, no. 6, pages 371–384, 1982.
- [Gain 09] James Gain, Patrick Marais & Wolfgang Strasser. *Terrain Sketching*. In I3D '09: Proceedings of the Symposium on Interactive 3D Graphics and Games, pages 31–38, New York, NY, USA, 2009. ACM.
- [Galin 10] Eric Galin, Adrien Peytavie, Nicolas Marchal & Eric Guérin. *Procedural Generation of Roads*. In Computer Graphics Forum: Proceedings of Eurographics 2010, vol. 29, pages 429–438, Norrköping, Sweden, May 2010. Eurographics Association.
- [Gamito 01] Manuel N. Gamito & F. Kenton Musgrave. *Procedural Landscapes with Overhangs*. In 10th Portuguese Computer Graphics Meeting, pages 33–42, 2001.
- [Gamr7 11] Gamr7. *Urban PAD*. Available from <http://www.gamr7.com>, 2011.
- [Glass 06] Kevin R. Glass, Chantelle Morkel & Shaun D. Bangay. *Duplicating Road Patterns in South African Informal Settlements Using Procedural Techniques*. In AFRIGRAPH '06: Proceedings of the 4th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa, pages 161–169, New York, NY, USA, 2006. ACM.
- [Greenworks 11] Greenworks. *XFrog*. Available from <http://www.xfrog.com>, 2011.
- [Greuter 03] Stefan Greuter, Jeremy Parker, Nigel Stewart & Geoff Leach. *Real-time Procedural Generation of 'Pseudo Infinite' Cities*. In GRAPHITE '03: Proceedings of the 1st International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia, pages 87–94, New York, NY, USA, 2003. ACM.
- [Groenewegen 09] Saskia A. Groenewegen, Ruben M. Smelik, Klaas Jan de Kraker & Rafael Bidarra. *Procedural City Layout Generation Based On Urban Land Use Models*. In Eurographics 2009: Short Papers, pages 45–48, Munich, Germany, 2009. Eurographics Association.
- [Guttman 84] Antonin Guttman. *R-trees: a Dynamic Index Structure for Spatial Searching*. In Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, pages 47–57, New York, NY, USA, 1984. ACM.
- [Hahn 06] Evan Hahn, Prosenjit Bose & Anthony Whitehead. *Persistent Realtime Building Interior Generation*. In Sandbox 2006: Proceedings of the ACM SIGGRAPH Symposium on Videogames, pages 179–186, New York, NY, USA, 2006. ACM.

- [Hammes 01] Johan Hammes. *Modeling of Ecosystems as a Data Source for Real-Time Terrain Rendering*. In DEM '01: Proceedings of the First International Symposium on Digital Earth Moving, pages 98–111, London, UK, 2001. Springer-Verlag.
- [Harteveld 10] Casper Harteveld, Rui Guimarães, Igor Mayer & Rafael Bidarra. *Balancing Play, Meaning and Reality: The Design Philosophy of LEVEE PATROLLER*. Simulation and Gaming, vol. 41, no. 3, pages 316–340, 2010.
- [Hnaidi 10] Houssam Hnaidi, Eric Guérin, Samir Akkouche, Adrien Peytavie & Eric Galin. *Feature based Terrain Generation using Diffusion Equation*. In Computer Graphics Forum: Proceedings of Pacific Graphics 2010, vol. 29, pages 2179–2186, 2010.
- [Huijser 10] Remco Huijser, Jeroen Dobbe, Wim F. Bronsvort & Rafael Bidarra. *Procedural Natural Systems for Game Level Design*. In Proceedings of SBGames 2010, pages 177–186, Florianopolis, SC, Brazil, 2010.
- [id Software 92] id Software. *Wolfenstein 3D*. Available from <http://www.idsoftware.com>, 1992.
- [IDV 11] IDV. *SpeedTree*. Available from <http://www.speedtree.com>, 2011.
- [Kallmann 98] Marcelo Kallmann & Daniel Thalmann. *Modeling Objects for Interaction Tasks*. In Proceedings of the Eurographics Workshop on Animation and Simulation, pages 73–86, 1998.
- [Kamal 07] K. Raiyan Kamal & Yusuf Sarwar Uddin. *Parametrically Controlled Terrain Generation*. In GRAPHITE '07: Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia, pages 17–23, New York, NY, USA, 2007. ACM.
- [Kelley 88] Alex D. Kelley, Michael C. Malin & Gregory M. Nielson. *Terrain Simulation Using a Model of Stream Erosion*. In SIGGRAPH '88: Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques, pages 263–268, New York, NY, USA, 1988. ACM.
- [Kelly 06] George Kelly & Hugh McCabe. *A Survey of Procedural Techniques for City Generation*. Institute of Technology Blanchardstown Journal, vol. 14, pages 87–130, 2006.
- [Kelly 07] George Kelly & Hugh McCabe. *Citygen: An Interactive System for Procedural City Generation*. In Proceedings of GDTW 2007: The Fifth Annual International Conference in Computer Game Design and Technology, pages 8–16, Liverpool, UK, November 2007.
- [Kessing 09] Jassin Kessing, Tim Tutenel & Rafael Bidarra. *Services in Game Worlds: a Semantic Approach to Improve Object Interaction*. In Proceedings of the International Conference on Entertainment Computing, pages 276–281, 2009.
- [Khronos Group 11] Khronos Group. The OpenCL Specification 1.1. 2011.
- [Koning 81] H. Koning & J. Eizenberg. *The Language of the Prairie: Frank Lloyd Wright's Prairie Houses*. Environment and Planning B: Planning and Design, vol. 8, no. 3, pages 295–323, 1981.
- [Kuijper 11] Frido Kuijper, Ruben M. Smelik & Rob van Son. *A Declarative Instructor-centered Approach to Modeling Synthetic Environments*. In Proceedings of the I/ITSEC 2011 Conference, Orlando, Florida, USA, 2011.

- [Kwon 03] Doo Young Kwon. ArchiDNA: A Generative System for Shape Configurations. Master's thesis, University of Washington, 2003.
- [Larive 06] Mathieu Larive & Veronique Gaildrat. *Wall Grammar for Building Generation*. In GRAPHITE '06: Proceedings of the 4th International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia, pages 429–437, New York, NY, USA, 2006. ACM.
- [Lechner 03] Thomas Lechner, Ben Watson & Uri Wilensky. *Procedural City Modeling*. In 1st Midwestern Graphics Conference, St. Louis, MO, USA, 2003.
- [Lechner 06] Thomas Lechner, Pin Ren, Ben Watson, Craig Brozefski & Uri Wilenski. *Procedural Modeling of Urban Land Use*. In SIGGRAPH '06: ACM SIGGRAPH 2006 Research posters, page 135, New York, NY, USA, 2006. ACM.
- [Lintermann 99] Bernd Lintermann & Oliver Deussen. *Interactive Modeling of Plants*. IEEE Computer Graphics and Applications, vol. 19, no. 1, pages 56–65, 1999.
- [Lipp 08] Markus Lipp, Peter Wonka & Michael Wimmer. *Interactive Visual Editing of Grammars for Procedural Architecture*. In SIGGRAPH '08: Proceedings of the 35th Annual Conference on Computer Graphics and Interactive Techniques, pages 1–10, New York, NY, USA, 2008. ACM.
- [Lipp 11] Markus Lipp, Daniel Scherzer, Peter Wonka & Michael Wimmer. *Interactive Modeling of City Layouts using Layers of Procedural Content*. In Computer Graphics Forum: Eurographics 2011, vol. 30, pages 345 – 354, Llandudno, UK, April 2011. Eurographics Association.
- [Lopes 10] Ricardo Lopes, Tim Tutenel, Ruben M. Smelik, Klaas Jan de Kraker & Rafael Bidarra. *A Constrained Growth Method for Procedural Floor Plan Generation*. In Proceedings of GAME-ON 2010, the 11th International Conference on Intelligent Games and Simulation. EUROSIS, 2010.
- [Lopes 11] Ricardo Lopes & Rafael Bidarra. *A Semantic Generation Framework for Enabling Adaptive Game Worlds*. In ACE '11: 8th International Conference on Advances in Computer Entertainment Technology, New York, 2011. ACM.
- [Mandelbrot 82] Benoît B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman, 1982.
- [Marson 10] Fernando Marson & Soraia R. Musse. *Automatic Generation of Floor Plans Based on Squarified Treemaps Algorithm*. IJCGT International Journal on Computers Games Technology, vol. 2010, pages 1–10, January 2010.
- [Martin 06] Jess Martin. *Procedural House Generation: a Method for Dynamically Generating Floor Plans*. I3D '06: Poster Proceedings of the 2006 SIGGRAPH Symposium on Interactive 3D Graphics and Games, 2006.
- [McCrae 09] James McCrae & Karan Singh. *Sketch-based Path Design*. In GI '09: Proceedings of Graphics Interface 2009, pages 95–102, Toronto, Ontario, Canada, 2009. Canadian Information Processing Society.
- [Merrell 10] Paul Merrell, Eric Schkufza & Vladlen Koltun. *Computer-Generated Residential Building Layouts*. ACM Transactions on Graphics, vol. 29, no. 5, pages 181:1–181:12, 2010.

- [Merrell 11] Paul Merrell, Eric Schkufza, Zeyang Li, Maneesh Agrawala & Vladlen Koltun. *Interactive Furniture Layout using Interior Design Guidelines*. In SIGGRAPH '11: Proceedings of the 38th Annual Conference on Computer Graphics and Interactive Techniques, pages 87:1–87:10, New York, NY, USA, 2011. ACM.
- [Miller 86] Gavin S. P. Miller. *The Definition and Rendering of Terrain Maps*. In SIGGRAPH '86: Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques, vol. 20, pages 39–48, New York, NY, USA, 1986. ACM.
- [Miller 95] George A. Miller. *WordNet: A Lexical Database for English*. Communications of the ACM, vol. 38, pages 39–41, 1995.
- [Müller 06] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer & Luc Van Gool. *Procedural Modeling of Buildings*. In SIGGRAPH '06: Proceedings of the 33rd Annual Conference on Computer Graphics and Interactive Techniques, pages 614–623, New York, NY, USA, 2006. ACM.
- [Müller 07] Pascal Müller, Gang Zeng, Peter Wonka & Luc Van Gool. *Image-based Procedural Modeling of Facades*. In SIGGRAPH '07: Proceedings of the 34th Annual Conference on Computer Graphics and Interactive Techniques, vol. 26, pages 85:1–85:10, New York, NY, USA, 2007. ACM.
- [Musgrave 89] F. Kenton Musgrave, Craig E. Kolb & Robert S. Mace. *The Synthesis and Rendering of Eroded Fractal Terrains*. In SIGGRAPH '89: Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques, pages 41–50, New York, NY, USA, 1989. ACM.
- [Musgrave 93] F. Kenton Musgrave. *Methods for Realistic Landscape Imaging*. PhD thesis, Yale University, New Haven, CT, USA, 1993.
- [Olsen 04] Jacob Olsen. *Realtime Procedural Terrain Generation*. Technical Report, University of Southern Denmark, October 2004.
- [Osfield 11] Robert Osfield. *OpenSceneGraph*. Available from <http://www.openscenegraph.org>, 2011.
- [Parish 01] Yoav I. H. Parish & Pascal Müller. *Procedural Modeling of Cities*. In SIGGRAPH '01: Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques, pages 301–308, New York, NY, USA, 2001. ACM.
- [Perlin 85] Ken Perlin. *An Image Synthesizer*. In SIGGRAPH '85: Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques, vol. 19, pages 287–296, New York, NY, USA, 1985. ACM.
- [Perlin 02] Ken Perlin. *Improving Noise*. In SIGGRAPH '02: Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques, pages 681–682, New York, NY, USA, 2002. ACM.
- [Peters 03] Christopher Peters, Simon Dobbyn, Brian MacNamee & Carol O'Sullivan. *Smart Objects for Attentive Agents*. In Proceedings of the International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision, 2003.
- [Peytavie 09] Adrien Peytavie, Eric Galin, Jérôme Grosjean & Stéphane Merillou. *Arches: a Framework for Modeling Complex Terrains*. In Computer Graphics Forum: Proceedings of Eurographics 2009, pages 457–467. Eurographics Association, 2009.
- [PixelActive 11] PixelActive. *CityScape*. Available from <http://pixelactive3d.com>, 2011.

- [Planetside 11] Planetside. *TerraGen 2*. Available from <http://www.planetside.co.uk>, 2011.
- [Procedural 11] Procedural. *CityEngine*. Available from <http://www.procedural.com>, 2011.
- [Prusinkiewicz 90] Przemyslaw Prusinkiewicz & Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, New York, NY, USA, 1990.
- [Prusinkiewicz 93] Przemyslaw Prusinkiewicz & Mark Hammel. *A Fractal Model of Mountains with Rivers*. In *Proceedings of Graphics Interface '93*, pages 174–180, May 1993.
- [Prusinkiewicz 01] Przemyslaw Prusinkiewicz, Lars Mündermann, Radoslaw Karwowski & Brendan Lane. *The Use of Positional Information in the Modeling of Plants*. In *SIGGRAPH '01: Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, pages 289–300, New York, NY, USA, 2001. ACM.
- [Rau-Chaplin 96] Andrew Rau-Chaplin, Brian Mackay-Lyons & Peter F. Spierenburg. *The LaHave House Project: Towards an Automated Architectural Design Service*. In *CADEX '96: Proceedings of the International Conference on Computer-Aided Design*, Hagenberg, Austria, September 1996.
- [re-lion 11] re-lion. *Builder*. Available from <http://www.re-lion.com/products/builder>, 2011.
- [Roden 04] T. Roden & I. Parberry. *From Artistry to Automation: A Structured Methodology for Procedural Content Creation*. In *Proceedings of the 3rd International Conference on Entertainment Computing*, pages 151–156, Eindhoven, The Netherlands, September 2004.
- [Rosenberg 11] Johannes Rosenberg. *GeoControl 2*. Available from <http://www.geocontrol2.com>, 2011.
- [Saunders 06] Ryan L. Saunders. *Terrainosaurus: Realistic Terrain Synthesis Using Genetic Algorithms*. Master's thesis, Texas A&M University, December 2006.
- [Schneider 06] Jens Schneider, Tobias Boldte & Ruediger Westermann. *Real-Time Editing, Synthesis, and Rendering of Infinite Landscapes on GPUs*. In *Vision, Modeling and Visualization 2006*, November 2006.
- [Side Effects Software 11] Side Effects Software. *Houdini*. Available from www.houdini.com, 2011.
- [Smelik 08] Ruben M. Smelik, Tim Tutenel, Klaas Jan de Kraker & Rafael Bidarra. *A Proposal for a Procedural Terrain Modelling Framework*. In *EGVE '08: Poster Proceedings of the 14th Eurographics Symposium on Virtual Environments*, pages 39–42, Eindhoven, The Netherlands, May 2008.
- [Smelik 09a] Ruben M. Smelik, Klaas Jan de Kraker, Tim Tutenel, Rafael Bidarra & Saskia A. Groenewegen. *A Survey of Procedural Methods for Terrain Modelling*. In *3AMIGAS: Proceedings of the CASA 2009 Workshop on 3D Advanced Media in Gaming and Simulation*, pages 25–34, Amsterdam, The Netherlands, June 2009.
- [Smelik 09b] Ruben M. Smelik, Tim Tutenel, Klaas Jan de Kraker & Rafael Bidarra. *A Case Study on Procedural Modeling of Geo-Typical Southern Afghanistan Terrain*. In *Proceedings of the IMAGE 2009 Conference*, pages 329–337, St. Louis, MO, USA., July 2009. IMAGE Society.

- [Smelik 10a] Ruben M. Smelik, Tim Tutenel, Klaas Jan de Kraker & Rafael Bidarra. *Declarative Terrain Modeling for Military Training Games*. International Journal of Computer Game Technology (IJCGT), vol. 2010, pages 1–11, 2010.
- [Smelik 10b] Ruben M. Smelik, Tim Tutenel, Klaas Jan de Kraker & Rafael Bidarra. *Integrating Procedural Generation and Manual Editing of Virtual Worlds*. In PCGames '10: Proceedings of the 2010 Workshop on Procedural Content Generation in Games, pages 1–8, New York, NY, USA, 2010. ACM.
- [Smelik 10c] Ruben M. Smelik, Tim Tutenel, Klaas Jan de Kraker & Rafael Bidarra. *Interactive Creation of Virtual Worlds Using Procedural Sketching*. In Proceedings of Eurographics 2010: Short Papers. Eurographics Association, May 2010.
- [Smelik 11a] Ruben M. Smelik, Krzysztof A. Galka, Klaas Jan de Kraker, Frido Kuijper & Rafael Bidarra. *Semantic Constraints for Procedural Modelling of Virtual Worlds*. In PCGames '11: Proceedings of the 2011 Workshop on Procedural Content Generation in Games, pages 1–4, New York, NY, USA, June 2011. ACM.
- [Smelik 11b] Ruben M. Smelik, Tim Tutenel, Klaas Jan de Kraker & Rafael Bidarra. *A Declarative Approach to Procedural Modeling of Virtual Worlds*. Computers & Graphics, vol. 35, no. 2, pages 352–363, April 2011.
- [Smith 01] Graham Smith & Wolfgang Stürzlinger. *On the Utility of Semantic Constraints*. In Immersive Projection Technology and Virtual Environments, pages 41–50, May 2001.
- [Smith 10] Gillian Smith, Jim Whitehead & Michael Mateas. *Tanagra: a Mixed-Initiative Level Design Tool*. In FDG '10: Proceedings of the 5th International Conference on the Foundations of Digital Games, page 209–216, New York, NY, USA, 2010. ACM.
- [Stachniak 05] Szymon Stachniak & Wolfgang Stürzlinger. *An Algorithm for Automated Fractal Terrain Deformation*. Computer Graphics and Artificial Intelligence, vol. 1, pages 64–76, May 2005.
- [Stiny 71] George Stiny & James Gips. *Shape Grammars and the Generative Specification of Painting and Sculpture*. In Proceedings of the Workshop on Generalisation and Multiple Representation, 1971.
- [Sun 02] Jing Sun, Xiaobo Yu, George Baciuc & Mark Green. *Template-based Generation of Road Networks for Virtual City Modeling*. In VRST '02: Proceedings of the ACM Symposium on Virtual Reality Software and Technology, pages 33–40, New York, NY, USA, 2002. ACM.
- [Šťava 08] Ondřej Šťava, Bedřich Beneš, Matthew Brisbin & Jaroslav Krivánek. *Interactive Terrain Modeling Using Hydraulic Erosion*. In Eurographics / SIGGRAPH Symposium on Computer Animation, pages 201–210, Dublin, Ireland, 2008. Eurographics Association.
- [Šťava 10] Ondřej Šťava, Bedřich Beneš, R. Měch, D. G. Aliaga & P. Křištof. *Inverse Procedural Modeling by Automatic Generation of L-systems*. In Computer Graphics Forum: Proceedings of Eurographics 2010, vol. 29, pages 665–674. Eurographics Association, 2010.
- [Teoh 08] Soon Tee Teoh. *River and Coastal Action in Automatic Terrain Generation*. In CGVR '08: Proceedings of the 2008 International Conference on Computer Graphics & Virtual Reality, pages 3–9, Las Vegas, Nevada, USA, July 2008. CSREA Press.

- [Tutenel 08] Tim Tutenel, Rafael Bidarra, Ruben M. Smelik & Klaas Jan de Kraker. *The Role of Semantics in Games and Simulations*. ACM Computers in Entertainment, vol. 6, pages 1–35, 2008.
- [Tutenel 09a] Tim Tutenel, Rafael Bidarra, Ruben M. Smelik & Klaas Jan de Kraker. *Rule-based Layout Solving and its Application to Procedural Interior Generation*. In 3AMIGAS: Proceedings of the CASA 2009 Workshop on 3D Advanced Media in Gaming and Simulation, pages 15–24, Amsterdam, The Netherlands, June 2009.
- [Tutenel 09b] Tim Tutenel, Ruben M. Smelik, Klaas Jan de Kraker & Rafael Bidarra. *Using Semantics to Improve the Design of Game Worlds*. In AIIDE '09: Proceedings of the 5th Conference on Artificial Intelligence and Interactive Digital Entertainment, Stanford, CA, USA, October 2009.
- [Tutenel 10] Tim Tutenel, Rafael Bidarra, Ruben M. Smelik & Klaas Jan de Kraker. *A Semantic Scene Description Language for Procedural Layout Solving Problems*. In AIIDE '10: Proceedings of the 6th Conference on Artificial Intelligence and Interactive Digital Entertainment, Stanford, CA, USA, October 2010.
- [Tutenel 11] Tim Tutenel, Ruben M. Smelik, Ricardo Lopes, Klaas Jan de Kraker & Rafael Bidarra. *Generating Consistent Buildings: a Semantic Approach for Integrating Procedural Techniques*. IEEE Transactions on Computational Intelligence and AI in Games, vol. 3, no. 3, pages 274–288, 2011.
- [Tutenel 12] Tim Tutenel. *Semantic Game Worlds*. PhD thesis, Delft University of Technology, 2012. In preparation.
- [Vanegas 09] Carlos A. Vanegas, Daniel G. Aliaga, Bedřich Beneš & Paul A. Waddell. *Interactive Design of Urban Spaces using Geometrical and Behavioral Modeling*. ACM Transactions on Graphics: Proceedings of ACM SIGGRAPH Asia 2009, vol. 28, no. 5, pages 1–10, December 2009.
- [Vanek 11] Juraj Vanek, Bedřich Beneš, Adam Herout & Ondřej Šťáva. *Large-Scale Physics-Based Terrain Editing Using Adaptive Tiles on the GPU*. IEEE Computer Graphics and Applications, 2011.
- [Voronoi 08] Georgy F. Voronoi. *Nouvelles Applications des Paramètres Continus à la Théorie des Formes Quadratiques*. Journal für die Reine und Angewandte Mathematik, vol. 134, pages 198 – 287, 1908.
- [Voss 85] Richard F. Voss. *Fundamental Algorithms for Computer Graphics*, chapitre Random Fractal Forgeries, pages 805–835. Springer-Verlag, Berlin, 1985.
- [Watson 08] Benjamin Watson, Pascal Müller, Oleg Veryovka, Andy Fuller, Peter Wonka & Chris Sexton. *Procedural Urban Modeling in Practice*. IEEE Computer Graphics and Applications, vol. 28, pages 18–26, 2008.
- [Weber 09] Basil Weber, Pascal Müller, Peter Wonka & Markus Gross. *Interactive Geometric Simulation of 4D Cities*. Computer Graphics Forum: Proceedings of Eurographics 2009, vol. 28, pages 481–492, April 2009.
- [Wonka 03] Peter Wonka, Michael Wimmer, François Sillion & William Ribarsky. *Instant Architecture*. In SIGGRAPH '03: Proceedings of the 30th Annual Conference on Computer Graphics and Interactive Techniques, pages 669–677, New York, NY, USA, 2003. ACM.
- [Xu 02] Ken Xu, James Stewart & Eugene Fiume. *Constraint-Based Automatic Placement for Scene Composition*. In Graphics Interface, pages 25–34, 2002.

- [Yong 04] Liu Yong, Xu Congfu, Pan Zhigeng & Pan Yunhe. *Semantic Modeling Project: Building Vernacular House of Southeast China*. In VRCAI '04: Proceedings of the 2004 ACM SIGGRAPH International Conference on Virtual Reality Continuum and its Applications in Industry, pages 412–418, New York, NY, USA, 2004. ACM.
- [Yu 11] Lap-Fai Yu, Sai Kit Yeung, Chi-Keung Tang, Demetri Terzopoulos, Tony F. Chan & Stanley Osher. *Make it Home: Automatic Optimization of Furniture Arrangement*. In SIGGRAPH '11: Proceedings of the 38th Annual Conference on Computer Graphics and Interactive Techniques, pages 86:1–86:12, New York, NY, USA, 2011. ACM.
- [Zhou 07] Howard Zhou, Jie Sun, Greg Turk & James M. Rehg. *Terrain Synthesis from Digital Elevation Models*. IEEE Transactions on Visualization and Computer Graphics, vol. 13, no. 4, pages 834–848, July-Aug. 2007.

Summary

A Declarative Approach to Procedural Generation of Virtual Worlds

Ruben M. Smelik

With the ever increasing costs of manual content creation for 3D virtual worlds, the potential of generating content automatically becomes too attractive to ignore. However, for most designers, procedural generation methods are complex and unintuitive to use, and offer little user control. Furthermore, due to their specialized nature, separately generated results are not easily integrated into a complete and consistent virtual world.

In this thesis, we propose *declarative modelling of virtual worlds*, an approach that enables designers to concentrate on *what* they want to create instead of on *how* they should model it. To realize this approach, we have devised a framework, building upon proven results on procedural generation, constraint solving and semantic modelling.

The foundation of this framework is provided by a semantic model for virtual worlds, which structures terrain features in several levels of abstraction, from a coarse user specification to concrete 3D geometry, and enriches objects with relevant information on their functionalities, services and roles. The framework supports structured integration of procedural methods at different levels of abstraction. With these integration methods embedded in our framework, we are able to harmonically apply existing procedural methods in combination to generate complete virtual worlds with detailed objects.

We allow for intuitive interaction with the framework, providing user control at several levels of granularity. Our interaction methods, such as procedural sketching, can be freely mixed and are interactively evaluated, enabling a short feedback loop. Each of these methods has its own added value and they complement each other. In order to form a consistent and plausible environment, generated features also have to be properly embedded in the virtual world. To this end, we introduced automatic consistency maintenance, which uses generic methods to handle any interactions that occur between features. This removes a huge burden from the designer, who is now

freed from the task of continuously fitting all content together and keeping the world consistent.

We believe that the combination of these contributions, as successfully implemented in our prototype SketchaWorld, significantly helps designers in the process of generating virtual worlds.

Samenvatting

Een Declaratieve Aanpak voor het Procedureel Genereren van Virtuele Werelden

Ruben M. Smelik

Door de immer stijgende kosten van de handmatige contentproductie voor 3D virtuele werelden wordt de mogelijkheid om automatisch content te genereren bijzonder aantrekkelijk. Echter, voor de meeste ontwerpers zijn zulke procedurele generatiemethodes complex en niet intuïtief in gebruik. Daarnaast ondersteunen ze weinig mogelijkheden om het gegenereerde resultaat te beïnvloeden. Bovendien, omdat de methodes zeer gespecialiseerd zijn, is het niet eenvoudig om alle afzonderlijk gegenereerde resultaten te integreren in een volledige en consistente virtuele wereld.

In dit proefschrift presenteren we *declaratief modelleren van virtuele werelden*, een aanpak die ontwerpers in staat stelt om zich te concentreren op *wat* ze willen creëren in plaats van op *hoe* ze dit zouden moeten modelleren. Om deze aanpak tot stand te kunnen brengen hebben we een raamwerk opgesteld, voortbouwend op bewezen resultaten op het gebied van procedurele generatie-, constraintoplossings- en semantische modelleringstechnieken.

De basis van het raamwerk is een semantisch model voor virtuele werelden, dat de elementen van de wereld opdeelt in verschillende niveaus van abstractie, van grove gebruikersspecificatie tot concrete 3D geometrie, en dat alle objecten verder verrijkt met relevante kennis over hun functie, rol en te leveren diensten. Het raamwerk ondersteunt het planmatig integreren van procedurele generatiemethodes in deze verschillende niveaus van abstractie. Met behulp van de integratiemogelijkheden van het raamwerk kunnen we bestaande procedurele generatiemethodes harmonieus combineren om volledige virtuele werelden te genereren, gevuld met gedetailleerde objecten.

We ondersteunen intuïtieve interactie door ontwerpers invloed uit te laten oefenen op verschillende niveaus van fijnmazigheid. Onze interactiemethodes, zoals procedureel schetsen, kunnen in elke combinatie gebruikt worden. Gebruikersacties worden interactief doorgerekend, hetgeen leidt tot snelle terugkoppeling. Elke van deze interactiemethodes heeft zijn toegevoegde waarde en ze vullen elkaar aan. Om

een consistente en geloofwaardige omgeving te creëren is het noodzakelijk dat de afzonderlijke elementen van de wereld op een correcte wijze gecombineerd worden. Hiervoor hebben we automatische consistentiebeheer geïntroduceerd, hetgeen door middel van generieke methodes alle interacties tussen de afzonderlijke elementen afhandelt. Dit verlost de ontwerper van de arbeidsintensieve taak om continu zelf alle elementen samen te voegen en zo de virtuele wereld consistent te houden.

We denken dat de in dit proefschrift beschreven bijdragen tezamen, als zodanig geïmplementeerd in ons prototype SketchaWorld, ontwerpers belangrijke ondersteuning bieden in het proces van het creëren van virtuele werelden.

Curriculum Vitae

Ruben Michaël Smelik was born on June 16, 1982, in Haarlem, the Netherlands. He graduated in the year 2000 from the Stedelijk Gymnasium Haarlem, a grammar school founded in 1389. In 2006, he received his Master's degree in Computer Science from the University of Twente. His thesis, called *Specification and Construction of Control Flow Semantics*, introduced a visual language for specifying the semantics of control flow in programming languages and an automatic translation from such specifications to graph production systems for constructing flow graphs. In addition, he followed several courses of the Master Game and Media Technology at Utrecht University.

In the summer of 2007, he started working as a PhD student at the Modelling, Simulation and Gaming department at TNO in The Hague, in close cooperation with the Computer Graphics and CAD/CAM Group at Delft University of Technology. His research project was entitled *Automatic Creation of Virtual Worlds*. In this project, his focus was on methods and techniques for creating geo-typical virtual worlds for serious games and simulations. As part of this project, he supervised several Bachelor's and Master's student projects.

After his PhD project, he continues to work as a research scientist at TNO, focussing on 3D environment modelling and visualisation. In his work, he will also carry on the research on procedural generation and the development of the Sketcha-World prototype in future projects.

Acknowledgements

A PhD project can at times be quite challenging, demanding and stressful. Fortunately I've had the support of many people during the past four years, who have contributed to my research and prototype in different ways, and motivated me to see it through. One of the advantages of my project was that I could work at two locations: at the Waalsdorpervlakte in The Hague and at the Mekelweg in Delft. This kept my workweek interesting and varied, and provided extra opportunities to eat birthday cake.

I'd foremost like to thank Rafael Bidarra, my copromotor, for all the enthusiasm and dedication he has put into my project. I've learned much from his critical view on my writing, and could still learn a lot from his networking skills and his ability to motivate and inspire people. I hope we can continue to cooperate in this way in the coming years. I'd also like to thank my promotor, Erik Jansen, for his quick and thorough review of my thesis, and his help in dealing with all the bureaucracy associated with the PhD defence.

In Delft, I've worked closely together with my roommate Tim Tutenel. I'd like to thank him for his humour, for introducing me to a variety of excellent Belgian comedy shows, and for putting up with my complaining about daily annoyances. Ricardo Lopes joined our "gaming alley" later on, and together we had a lot of fun on our conference trips to San Francisco and our most recent trip to Bordeaux. I wish them all the luck on completing their projects.

Furthermore, I would like to thank all my colleagues at the Computer Graphics group for their tips, advice and lunch-time jokes during the past four years, in particular Jorik Blaas, Charl Botha, Wim Bronsvort, Gerwin de Haan, Ruud de Jong, Jassin Kessing, Fernando Marson, Rick van der Meiden, Peter van Nieuwenhuizen, Frits Post, Matthijs Sypkens Smit, Bart Vastenhouw, Xin Zhang, and, of course, Joël van Neerbos, who made a great contribution to SketchaWorld in our Knowledge Transfer Project.

Working at two locations, I've also had the luxury of two supervisors. Klaas Jan de Kraker has been a great mentor during the past four years. Even though, due to a reorganisation, he unfortunately could no longer officially be my advisor in the last

part of the project, he kept in touch and is now a member of my committee. Thank you for all your help and enthusiasm. Frido Kuijper was also involved throughout the project, and helped me in keeping up the pace in the final months. His extensive knowledge on environment modelling, combined with his critical attitude and a talent for discovering weak spots in my implementation, continues to improve the quality of the SketchaWorld prototype.

I'd like to thank Daan Kloet, my former manager, for his encouragement and support for SketchaWorld, and Martijn Stamm and Jeroen Dezaire for giving me the opportunity to finish my thesis in the summer months. And of course I'd like to thank all my current and former colleagues at TNO for their input and ideas, in particular: Wim Huiskamp, Philip Kerbusch, Yntze Meijer, Frank van Meurs, Rob van Son, Peter Wit, and especially Rick Appleton, Kurt Donkers and Rein Kadijk for the work they did in our National Technology Project.

I had the pleasure of being directly and indirectly involved in many Bachelor and Master student projects. I'd like to thank them all for their valuable contributions to the research and the prototype: Asmar Arsala, Johannes Bertens, Joachim Boers, Mattijs Driel, Krzysztof Galka, Koriijn van Golen, Saskia Groenewegen, Matthijs Oomkens, Zhi Kang Shao, and, in particular, Quintijn Hendrickx, who did an excellent job with his Challengent honours programme as well as his BSc project.

From Utrecht University, I'd like to thank Mark Overmars, Remco Veltkamp, Piet Buitendijk and Rita Jansen for setting up and managing the GATE project. It was stimulating to be part of such a large national research program on gaming. During the research project, we cooperated with Deltares and re-lion to test the prototype in practice. At Deltares, I'd especially like to thank Matthijs Schaap and Rens van den Bergh for the effort they put into our project and all the suggestions they gave me to improve SketchaWorld. At re-lion, I'd like to thank Daan Nusman and Alex Poelman for their work, as we are sprinting to finish the SketchaWorld plugin in Builder.

My friends provided very much welcome distractions in the weekends, and I'd like to thank them all for that. I'd especially like to mention Wim, who, besides being a very faithful friend for many years, also directly contributed by applying his creativity and editing skills to spice up my SketchaWorld demo movies. I also owe my parents, grandparents, sister Esther and brother Daniël many thanks for motivating me and patiently trying to understand what exactly I was doing.

Finally, I'd like to thank my love Tjarda, for always being there, putting up with the stressful periods and my reluctance to take a holiday break during these years, and for proofreading my entire (!) draft thesis. You are by far the best thing that has ever happened to me.