Thèse préparée dans le cadre d'une cotutelle,
entre l'Université de Grenoble et l'Université de Montréal

Création interactive de mondes virtuels:
Combiner génération procédurale et contrôle utilisateur intuitif

par
Arnaud Emilien

Laboratoire Jean Kuntzmann (LJK)
École doctorale EDMSTII
Université de Grenoble

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences
Université de Montréal

Thèse présentée à la Faculté des arts et des sciences
en vue de l'obtention du grade de Philosophiæ Doctor (Ph.D.)
en informatique

Décembre, 2014

Résumé

La complexité des mondes virtuels ne cesse d'augmenter et les techniques de modélisation classiques peinent à satisfaire les contraintes de quantité nécessaires à la production de telles scènes. Les techniques de *génération procédurale* permettent la création automatisée de mondes virtuels complexes à l'aide d'algorithmes, mais sont souvent contre-intuitives et par conséquent réservées à des artistes expérimentés. En effet, ces méthodes offrent peu de contrôle à l'utilisateur et sont rarement interactives. De plus, il s'agit souvent pour l'utilisateur de trouver des valeurs pour leurs nombreux paramètres en effectuant des séries d'essais et d'erreurs jusqu'à l'obtention d'un résultat satisfaisant, ce qui est souvent long et fastidieux.

L'objectif de cette thèse est de combiner la puissance créatrice de la génération procédurale avec un contrôle utilisateur intuitif afin de proposer de nouvelles méthodes interactives de modélisation de mondes virtuels. Tout d'abord, nous présentons une méthode de génération procédurale de villages sur des terrains accidentés, dont les éléments sont soumis à de fortes contraintes de l'environnement. Ensuite, nous proposons une méthode interactive de modélisation de cascades, basée sur un contrôle utilisateur fin et la génération automatisée d'un contenu cohérent en regard de l'hydrologie et du terrain. Puis, nous présentons une méthode d'édition de terrains par croquis, où les éléments caractéristiques du terrain comme les lignes de crêtes sont analysés et déformés pour correspondre aux silhouettes complexes tracées par l'utilisateur. Enfin, nous proposons une métaphore de peinture pour la création et l'édition interactive des mondes virtuels, où des techniques de synthèse d'éléments vectoriels sont utilisées pour automatiser la déformation et l'édition de la scène tout en préservant sa cohérence.

Mots-clés: Procedural, Edition interactive, intuitif, mondes virtuels

Abstract

The complexity required for virtual worlds is always increasing. Conventional modeling techniques are struggling to meet the constraints and efficiency required for the production of such scenes. *Procedural generation* techniques use algorithms for the automated creation of virtual worlds, but are often non-intuitive and therefore reserved to experienced programmers. Indeed, these methods offer fewer controls to users and are rarely interactive. Moreover, the user often needs to find values for several parameters. The user only gets indirect control through a series of trials and errors, which makes modeling tasks long and tedious.

The objective of this thesis is to combine the power of procedural modeling techniques with intuitive user control towards interactive methods for designing virtual worlds. First, we present a technique for procedural modeling of villages over arbitrary terrains, where elements are subjected to strong environmental constraints. Second, we propose an interactive technique for the procedural modeling of waterfall sceneries, combining intuitive user control with the automated generation of consistent content, in regard of hydrology and terrain constraints. Then, we describe an interactive sketch-based technique for editing terrains, where terrain features are extracted and deformed to fit the user sketches. Finally, we present a painting metaphor for virtual world creation and editing, where methods for example-based synthesis of vectorial elements are used to automate deformation and editing of the scene while maintaining its consistency.

Keywords: Procedural, Interactive Editing, Intuitive, Virtual worlds

Contents

List of Tables

List of Figures

Remerciements

Je remercie tout d'abord mes encadrants Marie-Paule Cani et Pierre Poulin, pour leur enthousiasme, leur patience, leurs conseils et l'aide scientifique qu'ils ont apportée tout au long de cette thèse. Je les remercie aussi pour le temps qu'ils ont consacré à la relecture de ce manuscrit.

Je tiens aussi à remercier les nombreuses personnes avec qui j'ai eu le plaisir de collaborer, Flora Ponjou Tasse, Derek Nowrouzezahrai, Adrien Peytavie et Eric Galin. Je remercie tout particulièrement mon colocataire de bureau Adrien Bernhardt, pour toutes les discussions et l'aide apportée lors de ma première année de thèse. Un énorme merci à Ulysse Vimont, qui a été mon colocataire de bureau et collaborateur lors de ma dernière année de thèse, pour l'aide très précieuse qu'il a su apporter lors de mes derniers travaux de recherche, tout en m'ayant supporté quotidiennement. Merci aussi à Élie Michel que j'ai eu le plaisir d'encadrer lors de son stage.

Je remercie aussi tous les membres d'IMAGINE de Grenoble pour leur accueil et leur contribution à la bonne ambiance de l'équipe. Merci à mes amis canadiens du LIGUM avec qui j'ai passé une superbe année, dans un pays que j'ai eu le plaisir de découvrir à travers toutes ses saisons. Merci à Benjamin, Cédric, Rémi, Aude, Gilles-Philippe, Jonathan, Eric ... de m'avoir supporté pendant tout ce temps, et avec qui j'ai passé de bons moments.

Enfin, j'adresse un grand merci à ma famille pour son soutien infaillible et sans qui tout ceci n'aurait pas été possible, et finalement un énorme merci à Aurélie, pour avoir toujours été présente, et pour avoir été mon rayon de soleil durant ces trois années de thèse.

Chapter 1

Introduction

The modeling of virtual worlds is an important theme in computer graphics, as the associated techniques and results are being heavily used in video games, movies, and simulators. Because of the ever growing demand, virtual worlds are more and more expensive to produce, as their size and quantity of details are increasing. Indeed, with traditional modeling software all tasks are performed "by hand" by artists. If for instance an artist wants to create a city of several square kilometers, then the creation will be long and tedious because of the amount of details that must be produced. This is also true for landscapes, which should ideally combine arbitrary terrains with rivers, forests, plants, rocks, grass, etc., and adapted urban and non-urban settlements, with roads, houses, fences, plants, etc.

Procedural modeling relies on the use of algorithms to generate content automatically. Unsurprisingly, it has been the subject of much research in recent years. These methods are increasingly used in the field of virtual world modeling, for instance to generate a terrain, its vegetation, its river network, or even its cities. However, they are often unintuitive to use as they are rarely interactive, and offer few controls. Moreover, they are indirectly controlled by parameters set by artists, generally tuned through an often frustrating set of trials and errors until the desired result is obtained.

To overcome the problem of indirect controls, which is a major obstacle to the adoption of such methods, the field of *interactive procedural modeling* has received increasing attention over the last decade. It ranges from methods offering some soft controls of the procedures, for example by painting regions of interest on a map, to highly interactive ones, where the interactions resemble those of classical modeling tools, for example where the user sketches the shape of a tree while leaving to the software the generation of its branches and leaves.

The notion of *inverse procedural modeling* has been introduced to categorize methods that automatically determine the correct settings to generate a given type of results. The principle is similar to the work on image and texture synthesis, and aims to reduce or even eliminate the tedious trial-and-error process to find good settings, and even the algorithm rules. However, as the results grow in complexity in interconnected components, no generic method has been proposed yet for inverse procedural modeling of virtual worlds, and it is even questionable what kind of interface could be provided for it to answer all desires of an artist.

Open Problems in Virtual World Modeling

Designing interactive methods for modeling virtual worlds poses many challenges:

- the methods should *handle the virtual world complexity* and the associated environment constraints while *remaining easy to use* for non experts;
- the methods should provide *smart user controls*, enabling the user to focus on his coarse or fine editing desires, while the methods automatically take over the long and tedious tasks;
- the methods should also be *real time* and provide results that are as close as possible to final results, to be usable in interactive modeling sessions;
- the methods should *automatically find the settings* to use within their automatic algorithms.

Contributions

Throughout this thesis, we try to answer the following question:

"How can we improve the interactive design of virtual worlds to better match user needs and computer capabilities?"

We study this question through four systems combining different levels of intuitive user controls with a variety of strategies for procedural generation. The contributions of this work are as follows:

- First, we focus our research on the understudied problem of procedural village modeling. The main contribution of this work is the adaptation of the village to several environment constraints, while being easily configurable by the user. Our method is able to generate a variety of village types on arbitrary terrains, from a mountain hamlet to a fisherman village, and this, even though the number of settings can be at times overwhelming. However, user control remains indirect, being kept close of standard controls in procedural modeling systems.
- Second, we present a method for the interactive procedural modeling of waterfall sceneries. This work allows a user to interactively create a coarse waterfall network on an existing terrain, while the algorithm automatically generates a detailed scenery and deals with the environment constraints and with the hydraulic validity of the flow. This process leads to automatic adaptation of the terrain to the waterfalls, and of the waterfalls to the terrain.
- Third, we introduce an interactive first-person sketch-based method for terrain editing. Enabling users to draw mountain silhouettes from a first-person viewpoint, our algorithm analyzes and deforms an existing terrain to match the sketched curves. This process retains as much as possible the details and plausibility of the original terrain, but deforms it to satisfy the constraints issued from the sketched silhouettes.
- Finally, we present a more general interactive painting system for modeling virtual worlds, which relies on example-based synthesis. The user creates and edits complex vectorial scenes using similar interactions as in modern painting systems. Meanwhile, the algorithm automatically preserves the scene's consistency and accelerates many otherwise tedious tasks by analyzing different properties from examples in the current scene, and by seamlessly generating results that satisfy similarity constraints.

Structure of the Document

This document is divided into five main parts. First, we present in Chapter 2 a survey of procedural and interactive modeling techniques for the design of virtual worlds. We cover automatic procedural modeling of natural sceneries, interactive procedural modeling, sketch-based modeling, example-based synthesis, and inverse procedural modeling. In Chapter 3, we detail our method for the procedural generation of villages on arbitrary terrains. In Chapter 4, we present our method for the interactive modeling of coherent waterfall sceneries. In Chapter 5, we describe our method for the sketch-based editing of terrain. In Chapter 6, we present our method for interactive painting of virtual worlds. Lastly, we conclude this work and discuss some avenues for future work, extending our methods given our various experiences with all the systems described.

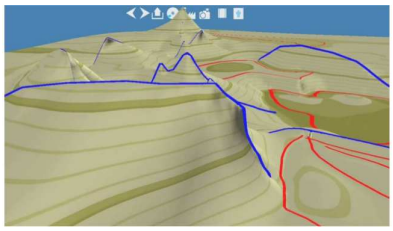
Related Publications

Three chapters of this thesis have previously appeared as publications. They have been slightly modified to be better integrated into this unified document. They are:

- Chapter 3: This chapter is an extended and updated version of an article presented at the *Computer Graphics International '12* conference and published as a special issue of journal *The Visual Computer* [EBP*12].
- Chapter 4: This chapter is an extended version of a paper first presented at the national conference *AFIG '13* [EPCV13] and then selected for an improved version in French in the associated journal *REFIG* [EPCV14a]. A further improved version has been published in the journal *Computer Graphics Forum* [EPCV14b].
- Chapter 5: This chapter details a collaboration with Flora Ponjou Tasse, Ph.D. student at Cambridge University, during her visit to Inria Grenoble. This work was presented at the conference *Graphics Interface* [TEC*14a]. It was selected and an extended version was published in the journal *Computer & Graphics* [TEC*14b].
- Chapter 6: This chapter presents our most recent results, and we expect to submit an associated paper soon. As such, the reader should be aware of less-polished results, and should look forward to the publication.

Chapter 2

Modeling Virtual Worlds: State of the art


Contents

This chapter gives a state of the art on virtual world modeling. While modeling in itself is a huge field of research, and virtual worlds can include almost anything, we cannot pretend to cover all work related to modeling. However, we tried to address most work related to the specifics of virtual worlds and their usual contents.

We first present a review of procedural generation techniques for virtual worlds. We enrich this review with the main concepts on example-based synthesis, and present the literature of inverse procedural modeling. We conclude this state of the art by presenting various approaches on interactive modeling, including interactive procedural modeling, sketch-based modeling, and a brief review of shape deformation approaches in a non-procedural context.

2.1 Procedural Modeling of Virtual Worlds

Procedural modeling involves the automated creation of content using an algorithm. This approach has been successfully used in many fields, such as creating textures, geometries, animations, or even sounds. In this state of the art we focus on procedural methods designed for virtual world generation.

Many types of procedural methods have been developped for specific types of virtual world objects, including terrains [GGG*13], plants [LRBP12], cities [CEW*08], road networks [GPGB11], buildings [LWW08], etc. We conduct a brief survey in this section, to illustrate the most common families of methods. For more details please consult a more complete state of the art [STBB14] dedicated to this topic.

Note that in this section we detail only automated methods for virtual world modeling. Techniques for inverse procedural modeling and interactive procedural modeling are detailed respectively in Sections 2.2.4 and 2.3.

2.1.1 Terrains

In this section, we review the main data structures and real-time rendering techniques for terrains, as they play a key role in the support and constraints brought by the interactive modeling of virtual worlds. We then review how to procedurally generate them.

2.1.1.1 Data Structures

Real-time rendering of terrains is a key issue for almost all interactive applications, as terrains form the major element of landscapes. We review briefly in this section the main approaches proposed to represent and render terrains in real time.

Heightfields. Terrains are mostly represented as heightfields, a 2D regular grid where each cell stores elevations at a given 2D location. The simplicity and efficiency of the representation have made this structure popular in interactive software. Moreover it is well adapted to GPUs, even for very large terrains (e.g., [LH04, BN08, DIP14]). The major limitation of heightfields is their inability to represent complex shapes with several height values at the same 2D location, such as for caves, arches, and overhangs.

Voxels. A voxel representation stores volumetric density values in a 3D grid. They can be used to model complex terrains with arbitrary topology [Gei07], but suffer from high memory costs and

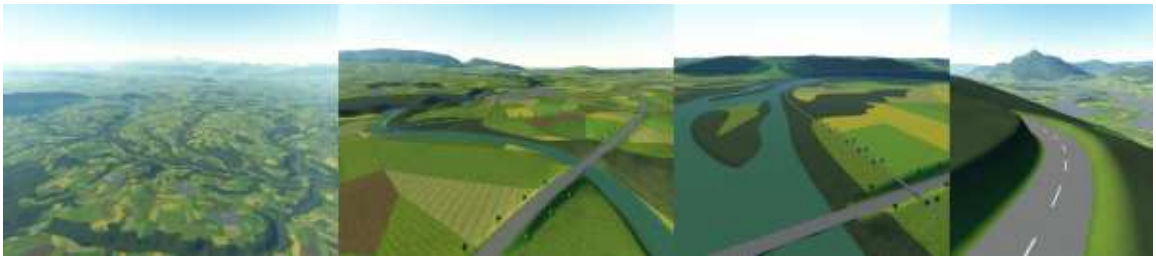


Figure 2.1: Real-time rendering of terrains using quadtrees [BN08], where the terrain's level of detail is automatically adapted to the camera position. The terrain is subdivided in square cells of various sizes, depending on their distance to the camera, and the color texture and heightfield are computed independently for each cell. When the camera moves, the structure is updated (i.e., cells are merged or subdivided) and their textures recomputed. This method enables real-time rendering of very large terrains, even planets.

expensive surface extraction processes. In addition, a terrain being mainly flat, most of the stored data is not relevant. To reduce this problem, hybrid models combining heightfields and voxels have been introduced [Cry09], but require tedious manual settings.

Material Stacks. Material stacks are a compact representation for 3D terrains. The data are stored in a 2D grid of stacks, representing several material heights at a given location. They have first been introduced in hydraulic erosion methods [BF01, BF02], and have been used in terrain modeling methods [PGGM09], whose model support arches, overhangs, and caves. The terrain geometry is implicitly computed from the voxel grid using convolution operators. The major limitation of this technique is the cost of the surface extraction algorithm, and despite recent work [LMS11, LS12], this technique remains expensive when computed and rendered in real time.

Overhang Maps. Another technique involves adding to a heightfield a map representing horizontal displacements to create overhangs [GM01]. However, this approach is not sufficient for more complex shapes, such as caves and multi-layered surfaces.

In our work, we rely on the heightfield representation of terrains, and use a *quadtree* rendering technique [BN08] (Figure 2.1). In Chapter 4, we enrich the heightfield with an overhang map [GM01] to add overhangs to our waterfall sceneries.

2.1.1.2 Terrain Generation

Many solutions are proposed for modeling terrains [SDKT*09, STBB14], from fully procedural methods, physical simulations, to those combining example or texture synthesis with some sketch-based interaction.

Noise. Many procedural terrain modeling methods are based on the fact that terrains are self-similar, i.e., statistically invariant under magnification [Man83]. The main generation approach consists of pseudo-randomly editing height values on a flat terrain using fractal noise [Mil86, MKM89] or Perlin noise [Per85]. Indeed, by combining several octaves (frequencies organized in levels) of

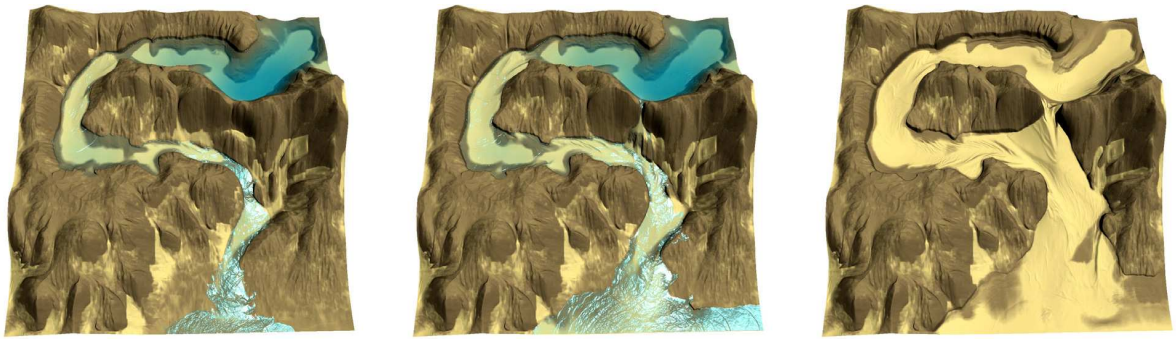


Figure 2.2: Erosion progressively digging a terrain during a modeling session [vBBK08].

noise, that are then interpolated, it is possible to generate detailed and plausible elevation maps, using the scaled noise value as terrain elevation at a given location. These methods are a popular choice for landscape modeling due to their easy implementation and efficient computation. For more information on fractal terrain generation methods, see the book from Ebert et al. [EMP*02].

Fractal-based approaches can generate a wide range of large terrains with unlimited level of details. However, they are limited by the lack of user control, non-intuitive parameters manipulation, and the absence of erosion effects such as drainage patterns.

Subdivision. Early work on terrain generation is also based on subdivision methods, like the method of *mid-point displacement* [FFC82, Mil86, Lew87]. In this subdivision algorithm, each new point is moved randomly in height with respect to the mean and relative altitude of its neighborhood. This process is iteratively applied with a decreasing offset range, until the desired resolution is reached. This technique has recently been adapted to GPU [BW06]. Similarly to noise-based methods, these methods are hard to control and are also unable to produce plausible terrains.

Erosion. A second class of algorithms for modeling terrains is based on simulating natural phenomena, including the simulation of erosion. These techniques can supplement noise-based or subdivision-based generators to enhance the realism of generated scenes. Musgrave et al. [MKM89] present the first method for thermal and hydraulic erosions based on geomorphology rules. Erosion simulation is performed using cellular automata, where cell material is progressively dissolved and displaced to neighboring cells. Most of the following algorithms rely on similar cellular automata, and focus on improving dissolution and displacement rules, while introducing several new materials.

Roudier et al. [RPP93] introduce a hydraulic erosion simulation that uses different materials at various locations resulting in different interactions with water. This method has been enriched by Beneš and Forsbach [BF01], considering the field as composed of several layers of materials. Nagashima [Nag97] combines thermal and hydraulic erosions in a river network pre-generated with a 2D fractal function. Chiba et al. [CMF98] generate a vector field of water flow that then controls how sediment moves during erosion. This process produces hierarchical ridge structures and thus enhances realism. Beneš and Arriaga [BA05] perform the erosion of soft material over hard material to model table mountains with realistic erosion patterns. Neidhold et al. [NWD05] present a physically correct simulation based on fluid dynamics and interactive methods that enable the input of global parameters, such as rainfall or local water sources. Beneš et al. [BTHB06] present full 3D volumetric hydraulic erosion simulations, allowing them to create waterfalls, while Šťava et

Figure 2.3: Real-time rendering of rivers using tiles of textures [vH11]. Left: The river surface is subdivided in "tiles". Each tile has its own wave direction and speed, that are smoothly interpolated to produce the river surface animation. Middle: River with directional waves. Right: Animated marsh surface.

al. [vBBK08] interactively sculpt complex landscapes (Figure 2.2). Kristof et al. [KBKŠ09] propose fast hydraulic erosion based on Smooth Particle Hydrodynamics(SPH), a general and popular method for simulating fluids. Pytel and Mann [PM13] present a hydraulic erosion system able to simulate avalanching, an important phenomenon to consider in order to model terrains with realistic shapes.

The main drawback of all these methods is that they only allow indirect user-control through trials and errors, requiring a good understanding of the underlying physics, time, and efforts to get the expected results.

2.1.2 Water Bodies

Procedural generation of water bodies, such as rivers, lakes, streams, oceans, and waterfalls, has been more scarcely studied than terrains. Since we are interested in user-designed water flows, shallow water simulation methods [SBC*11], used to compute the trajectory of a river on a terrain, are beyond the scope of this review.

Rivers. River rendering is usually done using planar geometries and animated textures. Yu et al. [YNBH09] derive an animated texture from the motion of particles simulated at the surface of a river, which provides an appearance of complex fluid. These textures can be augmented with their flow skirting around stones and river banks [YNS11]. Instead, Van Hoesel [vH11] tiles flow textures and modulates their application on water surface polygons according to the flow speed (Figure 2.3). His method proves to be quite efficient, compact, and effective. In Chapter 4, our flow textures are inspired by all these methods, including an interactive flow and diffusion editor with a sketching interface [ZIH*11].

River Networks. Several algorithms have been proposed to generate river networks, particularly to generate them within a terrain, and even to generate terrains. Indeed, instead of generating the hydraulic system after terrain generation, methods have been proposed to generate the terrain based on hydraulic network generation. Kelley et al. [KMN88] create a terrain by the generation of watersheds. Derzapf et al. [DGGK11] generate river networks at a planetary scale. Teoh [Teo09] presents *Riverland*, an algorithm that generates a complete terrain based on hydraulic network generation. Génevaux et al. [GGG*13] procedurally create terrains from a consistent river drainage network (Figure 2.4). A hierarchical drainage network grows from several locations around the

Figure 2.4: Terrain generation using a coherent hydraulic network [GGG*13].

borders of an island, following terrain and river slope constraints provided by the user. By estimating several hydraulic parameters, they determine river types and shapes, and generate the terrain geometry using combining operators.

The use of river networks in the generation process considerably enhances the realism of generated terrains. However, these methods are limited to terrains shaped by hydraulic erosion.

Waterfalls. Two main methods are specifically applied to create and render animated waterfalls: particle systems and textured polygons. Particle systems [SDZ*07], even when optimized with hierarchical methods or screen-space methods [BSW10], suffer from their inherent complexity to simulate efficiently networks of waterfalls in large environments. Sakaguchi et al. [SDZ*07] pre-configured particle systems for specific waterfalls, that are assembled to generate a network of waterfalls. The coherency between the terrain and the resulting waterfall must however be ensured by the user. Animated textures layered over polygons is a much more efficient and scalable approach [GCZ*06], and it could even be augmented with some lighter form of particle systems [HW04].

2.1.3 Plants and Ecosystems

Some of the most common procedural algorithms are based on formal grammars, like the *L-Systems* [Lin68] especially designed for the generation of trees and plants. An *L-System* is a formal grammar that uses a set of symbols and rewriting rules, applied in parallel.

Here is a simple example to illustrate the substitution principle of *L-Systems*:

$$\begin{array}{ll} \text{Symbols:} & A, B \\ \text{Rules:} & A \rightarrow B \\ & B \rightarrow \text{AB} \\ \text{Axiom:} & A \end{array}$$

The iterative application of the rules of this grammar give the following results:

$$A, B, AB, BAB, ABBAB, BABABBAB, ...$$

Considering trees and plants as recursive structures [PLH*90], it is possible to describe rules of growth as grammars to generate trunks, branches, and leaves that compose trees [dREF*88] (Figure 2.5). This method is also used to generate growing plants and flowers [PHM93]. *Open L-System*

Figure 2.5: Using *L-Systems* for generating trees. Left: Rewriting rules for the growth of a tree [dREF*88]. Right: Growth of trees under competition for resources with an *Open L-System* [MP96].

is an extension by Měch and Prusinkiewicz [MP96] that takes into account external constraints during generation. These constraints allow for example simulating competition for resources during growth, and generate more realistic results (Figure 2.5, right). Deussen et al. [DHL*98] generate complete ecosystems from these grammars, by defining rules to generate distributed plant seeds. Peyrat et al. [PTMG08] propose a method for generating leaves with an aging process encoded in an *L-System*, thus simulating color changes, holes, and cracks.

Procedural models have also been proposed to model other types of vegetation elements. For instance, Desbenoit et al. [DGA] generate lichen through simulation. Using a particle-based seeding process and aggregation tests, lichen grows over 3D objects in a plausible way, depending on environment constraints, such as humidity and shape.

2.1.4 Urban Environments

Generating villages (Chapter 3) requires the generation of road networks, street networks, and the creation of 3D buildings. This section reviews the literature in these domains.

Street Networks. Generating procedural cities is a very active research area since these environments are among the most complex and expensive to produce by hand. Existing methods for modeling cities start by generating a street network. Cycles formed by neighboring streets are tessellated into blocks serving as footprints for buildings. Inspired from *L-Systems* [MP96], Parish and Müller pioneer work for street network generation is fully automatic [PM01]. The street network iteratively grows over the terrain, following grammar rules. To control growth directions, this method takes into account various constraints such as the presence of water or population density, and business centers, residential areas, etc., in a large city environment. Figure 2.6 illustrates results from this method, and the power of such algorithms. In parallel, other approaches have been introduced such, as tensor-field-based or example-based city layout generation [CEW*08, AVB08]. Another approach simulates city growth based on urban simulation [WMWG09], resulting in highly complex and realistic cities. Note that these methods are generally dedicated to American-like cities, with semi-regular networks of mostly parallel and perpendicular streets.

Non-urban Settlements. The previous methods for generating cities define the street network and then create building parcels in a second step. Therefore, they cannot be applied to scattered settlements, where modeling the interaction between progressive settlement and road network extension is mandatory. To our knowledge, the only work addressing the generation of non-urban

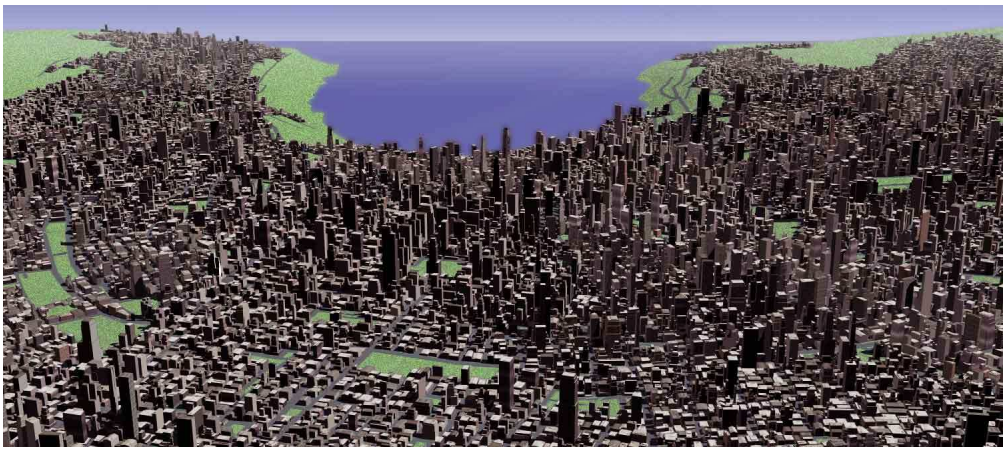


Figure 2.6: Using *L-Systems* to generate cities [PM01].

settlements was dedicated to South African informal settlements [GMB06]. After using a particle system to generate settlement seeds, different combinations of Voronoi diagrams are used to tessellate the terrain.

This inspired our work on land parcel generation (Chapter 3), although we had to develop a new, anisotropic land conquest method to account for alignments with roads and terrain features, observed in real village layouts. Lastly, progressive growth of non-urban settlements based on environmental constraints is somewhat similar to the spread of biological species. Inspired by a model for lichen spreading on a support [DGA], we rely on particles to progressively seed settlements based on interest maps. However, villages develop in more structured ways, forcing us to take the road network into account in the simulation loop.

Road Networks. Road networks are also important features on terrains outside of cities. Sun et al. [SYBG02] introduce a method to create road networks using growth rules and road templates based on patterns observed in real road networks. Galin et al. [GPMG10] present a technique for generating roads on arbitrary terrains, based on a pathfinding algorithm that optimizes the path of each road in function of various costs, such as terrain slopes, and road bends. It enables to take into account the environment, such as the presence of vegetation or water. This method allows the creation of realistic roads, including highways and mountain roads. The authors then extend their work to generate comprehensive road networks, connecting different cities and optimizing connections between them [GPGB11].

Our method in Chapter 3 extends the approach of Galin et al. [GPMG10] to road networks between hamlets or houses, leading us to a new, road re-use strategy.

Buildings. When a street layout defines footprints for buildings, a popular approach to procedural generation of buildings uses grammars. *L-Systems* have been used by Parish and Müller [PM01] to generate buildings, and their results have demonstrated the power of such applications. The algorithm progressively constructs the building geometry following the provided rules, such as building block extrusion from the building footprint and floor segmentation from the newly generated building block.

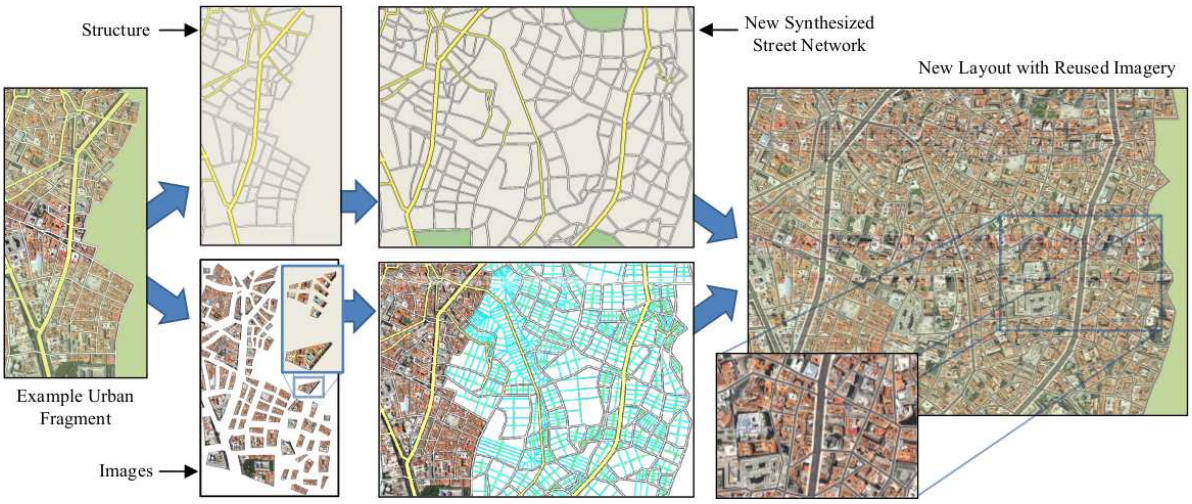


Figure 2.7: Interactive example-based urban layout synthesis [AVB08].

Wonka et al. [WWSR03] present *split grammars*, a grammar type especially efficient for building facade generation. *Shape grammars* have been then introduced by Müller et al. [MWH*06], where the rules' symbols are geometric shapes that are replaced and refined during the generation process. These grammars are so far the most common method for the generation of buildings exteriors.

Rather than using grammars, Leblanc et al. [LHP11] present an approach based on *components* for procedurally generating entire buildings, with consistent outer and inner shapes. A component is defined by a 2D or 3D shape and a set of attributes. Buildings are procedurally generated using a series of statements that progressively modify existing components or create new ones.

Our method for generating houses on mountanious terrains extends existing grammar-based approaches: we define in Chapter 3 an *open shape grammar* enabling facade elements to self-adapt to external constraints.

2.1.5 Conclusion on Procedural Modeling

The strength of procedural methods is their ability to generate large amount of content with only a few rules, whose parameters are defined by the user. These rules enable the emergence of complex shapes that artists may have difficulties to produce, for instance, erosion algorithms produce realistic erosion patterns on terrains that may be hard to do by hand. However, procedural approaches only provide indirect controls via rules editing. The modeling process is therefore usually a succession of trials and errors, until a satisfactory result is obtained. In addition, it is often necessary to fully understand the model and the roles of parameters to achieve the desired result, limiting the use of these tools to specialists. Finally, these methods have proved powerful in their respective contexts, but they are often tedious to control, even for experienced users.

2.2 Example-based Modeling

In this section we present different approaches that have in common the fact of using one or several examples provided by the user in order to generate visually similar results.

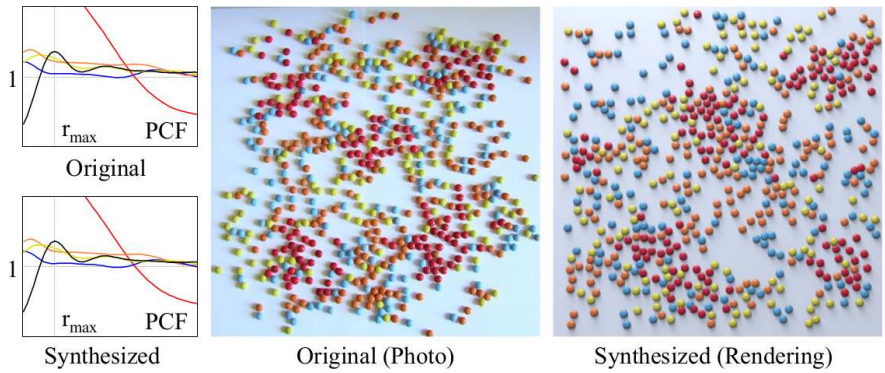


Figure 2.8: Analysis of the pair correlation functions (PCF) of point distributions, and generation of point distributions with pair correlation functions matching the original ones [OG12].

We first study methods for texture synthesis, then point processes and arrangements synthesis, structure synthesis, and finally *inverse procedural modeling*.

2.2.1 Texture Synthesis

Texture synthesis methods generate a texture or an image from an example provided by the user while preserving its visual aspect. There are two main categories of methods for texture synthesis: *pixel-based* and *patch-based* synthesis. For more details, Wei et al. [WLK*09] present a detailed state-of-the-art survey of these techniques.

The *pixel-based* approach considers a pixel as the basic texture unit, and generates an image pixel by pixel using algorithms to determine which pixel to copy and where. These algorithms use probabilistic algorithms [HB95] or neighborhood analysis to better capture local behaviors [Ash01].

To preserve image structures, *patch-based* approaches group pixels into *patches*. For example Dischler et al. [DMLG02] decompose the image into *texture particles*, and analyze their spatial arrangements. Then, they generate particles using a *seeding* process and the analyzed parameters, and create a new image by blending particle contributions.

Structured Image Synthesis. Image containing structured objects need specific synthesis algorithms to be generated correctly. Aliaga et al. [AVB08] propose an interactive method to edit complex urban layouts (Figure 2.7). To generate new roads, they first generate a point distribution of street intersections. Then, they connect the intersections using a marching algorithm, which tries to preserve the example properties, such as the tortuosity and neighborhood distances. The final urban layout appearance is generated using image-based synthesis of the original example combined to the newly generated street network.

In a similar way, Sibbing et al. [SPK10] analyze the branching structures of an image, such as an image of a river, and generate new images while matching the statistical properties of the captured structures. Image synthesis of the river is guided by a network structure generated from the analyzed properties.

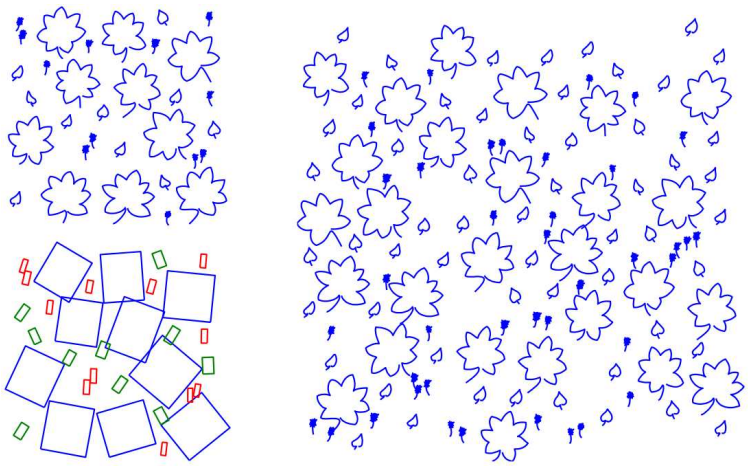


Figure 2.9: Synthesis of an arrangement consisting of vector elements (top left) that are analyzed and separated into groups of elements (bottom left), and synthesized to form a new image (right) [HLT*09].

2.2.2 Point Processes

Point processes are methods developed for the analysis and generation of point patterns respecting specific statistical properties. They include Poisson disk sampling [Wei08, LD08] and blue noise sampling [Fat11]. The idea is to analyze a point distribution to determine their probability distribution, often by matching several parameters of a given distribution, in order to generate new distributions following the analyzed distribution. Other approaches have been proposed to generate points with more complex arrangement patterns [LWSF10, ZHWW12, OG12], using pairwise point distances as the key characteristic of point processes. For instance, Li et al. [LWSF10] propose a method for anisotropic blue noise sampling. Öztireli and Gross [OG12] rely on the pair correlation function (or radial density function) to analyze and generate a point distribution respecting the example's properties (Figure 2.8). After having analyzed the pair correlation for each combination of point categories, they generate new point arrangements using a new generalized dart-throwing algorithm followed by a gradient descent, thus making the new sample statistics similar to the examplar. However, this synthesis process has long computational times, incompatible with interactive editing.

In Chapter 6, our synthesis method is inspired by the latter work for its use of the pair correlation function, but we generate points with a less precise but interactive process.

2.2.3 Arrangement Synthesis

Arrangement synthesis deals with the analysis and generation of a set of objects or shapes, while respecting rules learned from examples. In this research area, much effort is applied to vector images, as presented by Hurtut [Hur10] in a complete state of the art.

These methods are based on geometric solutions [BBT*06, IMIM08] or on distribution analysis [HLT*09]. Barla et al. [BBT*06] propose a method to synthesize 2D point and line arrangements, and ensure that the generated arrangement corresponds to the analyzed statistics, using Delaunay triangulation and several neighboring properties. They generate a new arrangement by constructing

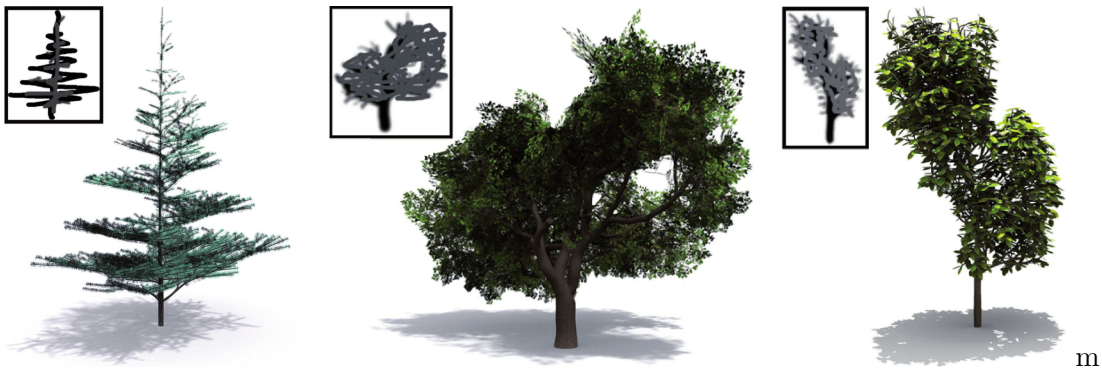


Figure 2.10: Procedural generation using inverse algorithm based on the Metropolis-Hastings algorithm. The algorithm enables the generation of vegetation constrained by the user sketch by automatically finding rules and parameters of a procedural tree generator [TLL*11].

a mesh with a neighborhood-matching algorithm. Hurtut et al. [HLT*09] generate a new arrangement using a statistical approach: they analyze the arrangement statistics, such as distances and orientations between elements, and generate a new arrangement by Metropolis-Hastings sampling (Figure 2.9). Jenny et al. [JHH10] present a method for synthesizing several point patterns on a map, enabling the sampling of objects with different sizes and shapes, such as different drawings of trees. They start from a regular grid and progressively perturb it to obtain valid distributions of anisotropic objects. While Jenny et al. [JHH10] rely on a dithered grid, Landes et al. [LGH13] propose a synthesis solution for the same problem, but rely on a shape-aware synthesis model, where the spatial relationship measurements between elements take into account their geometry, enabling to generate distributions of anisotropic elements.

A method for detecting symmetries and curvilinear arrangements in vector data has been proposed by Yeh and Měch [YM09]. After having identified arrangement rules, the technique can be used to interact with the result, such as changing the spacing between elements forming a curve. This work can be considered as a premise to inverse procedural modeling. Yeh et al. [YYW*12] propose a method based on Markov chain Monte Carlo to synthesize an arbitrary scene, where relationships between objects are encoded as constraints. However, they rely on the analysis of a large example basis, which often needs to be created by hand.

2.2.4 Inverse Procedural Modeling

In this section we present inverse procedural methods, whose aim is to find the appropriate parameters or rules of a procedural method.

Bokeloh et al. [BWS10] deform a 3D object according to its symmetries and deduce shape grammar construction rules. With the deduced grammar, they are able to generate new 3D objects with shapes assemblaged with parts cut from the original object.

Šťava et al. [ŠBM*10] apply the principle of inverse grammar on a 2D vector image, which is transformed into an *L-System*. The principle is to analyze the transformation matrices between groups of objects, which are first recognized and categorized as symbols, and to deduce their repetition and positioning rules. They enable user defines weightings to favor certain groupings or hierarchy criteria, for example, the distance between objects rather than similarities. This technique is especially adapted for structured scenes comprising a large number of repeated units.

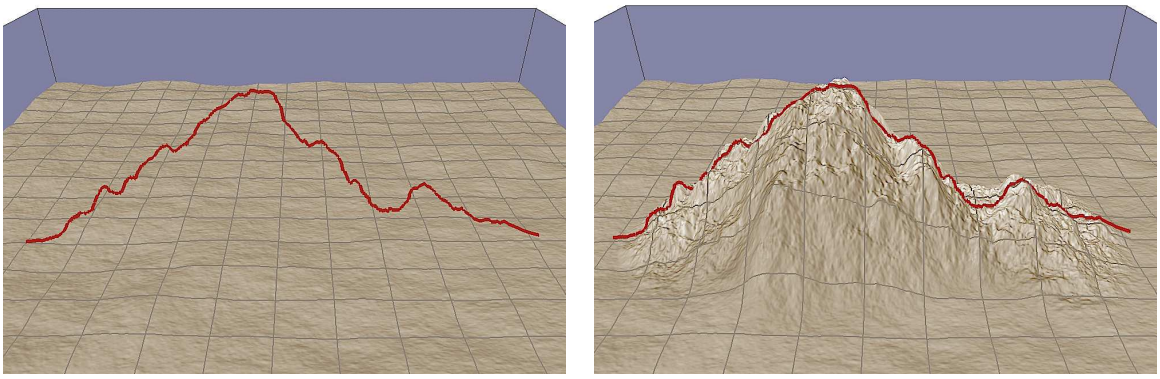


Figure 2.11: Terrain interactively generated from a user sketch. The user draws the mountain silhouette and the algorithm generates the associated mountain using multiresolution surface deformation. High frequency details in the silhouette are used to propagate noise. However, the silhouette can only be planar [GMS09].

Another way to use inverse procedural generation is to provide high-level constraints and let the algorithm find rules and parameters to create the desired result. Talton et al. [TLL*11] propose a method based on the Metropolis-Hastings algorithm, to browse all possible solutions and propose a result respecting various constraints, such as a 3D mesh defining the volume of a tree (Figure 2.10). The method is general and the authors apply it on procedural models of trees, cities, buildings, and Mondrian paintings. However, computational times are high, preventing the use of the method in interactive editing frameworks.

The principle of inverse generation has also been studied in the context of interactive procedural modeling of cities. Vanegas et al. [VGDA*12] provide to the user high-level controls localized in the editor scene, and generate a city meeting all these constraints. The algorithm is based on a *Markov* algorithm and requires a lot of learning on all cases of possible deformations, which makes difficult the addition of new interactions. Nevertheless, the results are very fast and allow user interaction in real time.

Lastly, a new method [ŠPK*14] has been proposed for inverse procedural modeling of trees. Taking arbitrary tree models as input, they use Monte Carlo Markov Chains to find botanic rules to generate similar trees. Contrary to other methods that tend to simplify the informations of a tree grammar, they rely on a large number of ecological properties and are able to model extremely detailed and varied trees.

2.2.5 Conclusion on Example-based Modeling

The area of inverse procedural modeling is very recent and growing. Being able to automatically find the rules of an algorithm is certainly a very good approach to reconcile intuitive control and procedural generation. The main drawback of these methods is their need to know what is the method for which they are searching the settings. Good inverse procedural methods have to be as general as possible to enable the capture of arbitrary scenes. Moreover, they must be fast enough to be used in an interactive tool. In Chapter 6, we present an interactive modeling tool that provides a general synthesis method to edit virtual worlds.

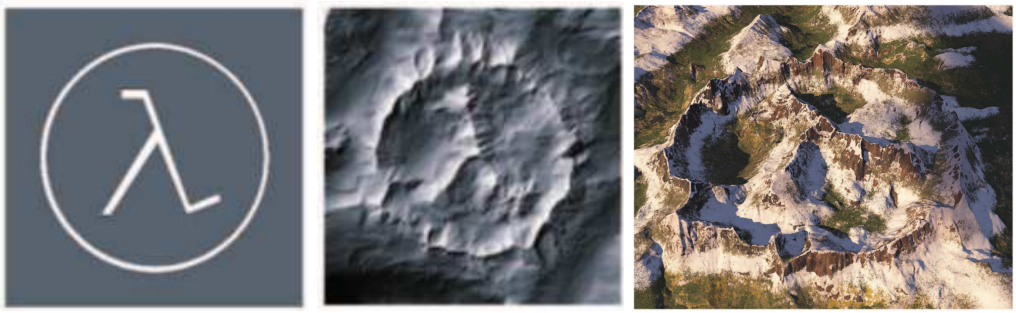


Figure 2.12: Terrain generation using example-based synthesis [ZSTR07]. Left: Input sketch. Middle: Generated heightfield using a texture synthesis method combining several images extracted from an example heighfield. Right: Same heightfield, textured, shaded, and viewed from a slightly lower viewpoint.

2.3 Interactive Modeling

To overcome some shortcomings of conventional procedural generation techniques, many studies have emerged, a number of them trying to make these techniques interactive, giving a more precise and more intuitive control to the user. In this section, we first review the field of sketch-based procedural modeling, and then present several other interactive methods for procedural modeling. Finally we present several approaches adressing interactive editing, where many rely on procedural rules to cleverly deform images or 3D objects.

2.3.1 Sketch-based Modeling

Traditional software interfaces consist of windows, menus, and buttons, and their use is often complex and non-intuitive. In contrast, sketch-based methods enable the artist to draw directly in a scene, as if using pencil and paper. The aim of these techniques is to minimize the rigid and uninspiring classical GUI and try to get closer to creative processes.

In the remaining of this section, we present sketch-based methods used to model virtual sceneries. For more information on sketch-based modeling, we invite the reader to consult the complete but less recent state of the art presented by Olsen et al. [OSSJ09].

Terrains. Sketching interfaces have been increasingly popular for terrain modeling. Cohen et al. [CHZ00] and Watanabe and Igarashi [WI04] present the first terrain modeling interfaces that take as input one 2D silhouette stroke directly drawn on a 3D terrain. The stroke is interpreted as a flat ridgeline of mountains, that are generated as rounder or sharper mountains depending on the stroke shape. Only a single silhouette stroke can be drawn and treated at a time.

Gain et al. [GMS09] take advantage of the power of procedural generation to generate heightfields using fractals, and provide a sketching interface to finely control the content. In their interface, the user draws mountain silhouettes and contours. The heightfield is then generated using fractals constrained by the curves (Figure 2.11).

Smelik et al. [STdKB10] present the first framework for the sketch-based modeling of complete virtual worlds. Their method combines iterative manual editing operations and several procedural generation algorithms. The world consistency is automatically preserved after each edition and

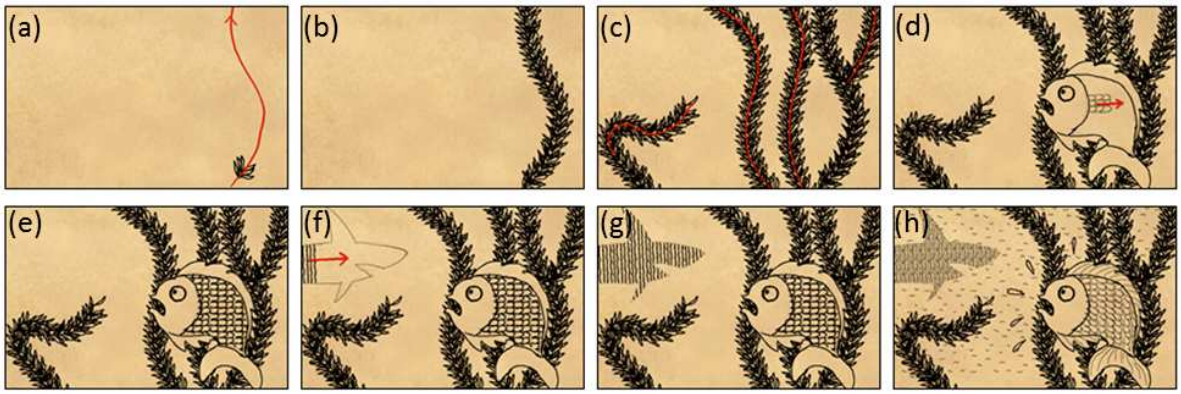


Figure 2.13: Interactive sketching of pen-and-ink illustrations based on texture synthesis. The user first draws a pattern (for instance few algae branches), and guides a synthesis process with a directional curve. The method can also fill closed spaces and align patterns with the curve direction [KIZD12].

regeneration, using a semantic model. However, the iteration capabilities between manual editing and procedural modeling is limited. Indeed, all user fine-grained manual editions on the procedural results are lost when a procedural regeneration occurs. The authors clearly identify problems to address in order to propose a completely iterative framework combining procedural modeling and manual editing.

Dos Passos and Igarashi [dPI13] propose a first-person viewpoint sketch-based modeling for terrains. The terrain is generated by extracting parts of an example elevation map, chosen for the similarity of their silhouettes and a sub-part of the user sketch, and combining them in order to create a terrain with a silhouette matching the user sketches. However, this method is limited to simple sketches, and only creates mountains with flat silhouettes.

In nature, mountain silhouettes from a viewpoint are usually not the result of one mountain with a flat ridgeline, but of several mountains at different distances with complex ridgelines even disconnected in 3D. In Chapter 5, we propose a method for modeling terrains from a first-person viewpoint, where silhouettes are composed by several mountains that we deform to match the user sketches.

Bernhardt et al. [BMV*11] propose similar user interaction to edit heightfields. They interactively create terrains from a first-person viewpoint using parameterized vectorial curves. Those curves have noise, and angle and roughness constraints, that enable the modeling of specific terrain features such as ridgelines, riverbeds, and cliffs. First introduced by Hnaidi et al. [HGA*10], curve constraints are propagated using diffusion curves [OBW*08]. The algorithm is solved in real time and provides immediate feedback to the user when editing the terrain. The authors stress the importance of immediate feedback, otherwise the creative process is broken and the interest of sketching is much reduced.

While the methods presented previously generate a new terrain from scratch, other methods use texture synthesis to generate terrains. Zhou et al. [ZSTR07] use 2D sketches to drive a patch-based terrain synthesis from real terrain data (Figure 2.12). Tasse et al. [TGM12] present an enhanced texture-based terrain generation method that re-uses the same sketching interface.

Vanek et al. [VBHŠ11] present a terrain editor based on an interactive physic-based simulation.

Figure 2.14: Interactive modeling of a building grammar. With an approach similar to classical 3D modeling software, the user can interactively set grammar rules during an editing process [LWW08].

For instance, the user can use a rain brush to erode a terrain, or merge patches of existing terrains into the edited one. The method provides a tile-based segmentation of the world enabling real-time simulation of large environments.

Trees. Sketch-based systems can provide a more intuitive way of generating tree models. For instance, Chen et al. [CNX*08] sketch a coarse trunk and leaves contours to generate a complex tree using a Markov random field.

Wither et al. [WBCG09] present a multi-scale editing technique to generate complex trees. From the tree scale to the leaf scale, the user can draw foliage silhouettes and sub-silhouettes, that are used to generate complex tree structures using botanical knowledge. Moreover, the method supports local refinement, that enables the user to specify local leaf properties that are locally spread to its neighborhood.

TreeSketch [LRBP12] is an application for creating trees using a touch pad, in which the user can control the shape of the trunk and branches, but also the distribution and type of leaves. The trees are constructed progressively by the user, whose gestures locally deform a procedural tree model. Botanical and physical properties are also ensured while the user sketches his intentions with different abstract tools. This application has been made freely available and was a huge success, proving that there is a public looking for intuitive procedural tools.

Clouds. Wither et al. [WBC08] propose the first sketch-based technique for modeling cumulus clouds. The user can sketch several cloud silhouettes on different layers, from different viewpoints. These silhouettes are used to generate the 3D mesh of the cloud volume, whose silhouettes match the user strokes.

Waterfall Videos. In a more peripheral work about video sequences of waterfalls, Bhat et al. [BSHK04] let a user sketch over a frame to indicate waterfall flows. The animated sequences of the flow are extracted from the video and waterfall videos are generated using new user strokes to guide their shape.

Artistic Brushing Techniques. Because our work seeks to make user interaction more intuitive, it is interesting to enrich here our literature review with artist-oriented non-procedural editing techniques.

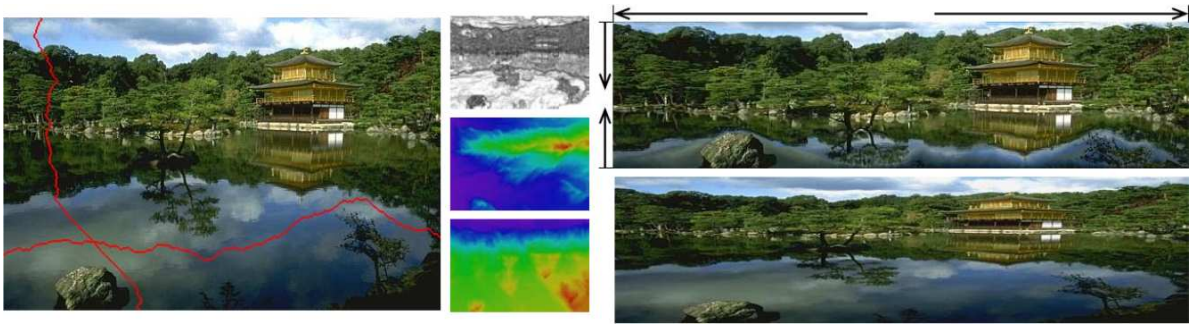


Figure 2.15: Image deformation using the seamcarving approach [AS07]. Left: Two cuts are computed on the original image, depending on its corresponding energy maps (middle). Right: The image is horizontally enlarged using several cuts and pixel duplication steps, and vertically shrunk. The result image (top) is less deformed than with a classical uniform scaling (bottom).

Vignette [KIZD12] is an interactive system for creating *pen-and-ink* illustrations. This sketching system is designed to offer artists an effective way to create 2D paintings, paying particular attention not to "break the creative flow". Indeed they offer three original and intuitive texture synthesis techniques. The *brush* duplicates an element along a curve. The *continuous hatching* tool fills a region with a particular pattern, driven by a vector field. Finally, *flood fill* fills an area by randomly placing items in the area and optimizing their positions. With these three methods, artists are able to achieve very intuitively fillings and repetitions (Figure 2.13).

Similarly, *DecoBrush* [LBW*14] enables the synthesis of an example image, chosen in an ornament library, along a stroke drawn by an artist. The method supports complex repetition and branching configurations and can be used to create structured decorative patterns.

Sun et al. [SZZ*13] present *Texture Brush*, an interactive method for painting textures on 3D objects. The user draws a stroke on a 3D model, which is used to parameterize smooth texture coordinates used to texture the object.

Milliez et al. [MNB*14] present hierarchical motion brushes. The user first draws a set of motion brushes (i.e., several frames of a painting animation), and defines their hierarchy. Then, the user simply draws a stroke to create complex animations, such as a rain drop first deformed during its fall and then transformed into several animated splashes.

Commercial softwares, such as Adobe Illustrator and Adobe Photoshop, provide several "pattern brushes" to create vectorial sceneries. The user draws a stroke along which a distribution of elements is generated, using user-defined settings. However, the proposed probability distributions are simple and offer few controls. In Chapter 6, we present a method relying on real time example-based synthesis to paint elements in virtual worlds.

2.3.2 Interactive Procedural Modeling

In this section we review diverse non-sketch-based interactive methods for more intuitively controlling procedural modeling of virtual worlds.

To better control the final shape of a proceduraly generated city, Kelly and McCabe [KM07] offer a tool in which the main roads and the outline of the city are directly edited, and the algorithm generates secondary roads and buildings with an *L-system*. Chen et al. [CEW*08] enable the

creation of a city by interactively defining a vector field, imposing the direction of the streets in the generation process. Aliaga et al. [ABVA08] present an interactive system for reconfiguring urban layouts, and a little later, Vanegas et al. [VABW09] present an interactive system to design urban spaces using geometrical and behavioral modeling. In their method, the artist paints several maps, such as residential density, to control the growth of a city. Lipp et al. [LSWW11] propose an interactive system to edit a city layout, where a user can copy, cut, paste, translate, or rotate roads and parcels, while the system automatically ensures scene consistency.

The generation of buildings using shape grammars [MWH*06] requires a strong knowledge of the process, which often limits its use to experienced programmers. Lipp et al. [LWW08] offer an interactive editing method with the goal of being as intuitive as possible. With this technique, the artist is able to directly edit the shape grammar similarly to conventional 3D modeling tools (Figure 2.14).

Beneš et al. [BAŠ09] propose an interface to control vegetation growth while relying on biological-based simulation. The user defines plant parameters, initial plant positions, and obstacles. Then, plant growth is simulated using a spatial colonization algorithm. Resulting plants are adapted to their neighborhood and external constraints.

Smith et al. [SWM11] present an iterative level editor for a 2D video game, which combines user control and procedural generation. After the user has placed constraints in the 2D level, the algorithm seeks to fill the rest of the level while respecting the fact that the level must be "playable", i.e., that there is a solution for a player to actually complete the level.

Beneš et al. [BŠMM11] generalize the concept of environment by spatially dividing a procedural model into several sub-models, called *guides*. The guides can communicate with each other by exchanging parameters, and can be interactively deformed. For instance, a procedural tree can be divided into smaller procedural branches that inter-communicate during the growth process.

2.3.3 Deformation

In Chapter 6, we present a technique enabling deformations of distributions and structures. Thus, we review in this section several approaches for deforming images, meshes, and structures.

Igarashi et al. [IMH05] apply the principle of *as-rigid-as-possible* to manipulate and distort 2D objects. After creating a mesh with a Delaunay triangulation, they apply the deformation and calculate the resulting mesh by performing two minimizations on two energy functions: vertex positions and triangle scales. Thus solving a linear system, they deform intuitively polygonal objects.

Avidan et al. [AS07] introduce *seamcarving*, a method for resizing images, that preserves important parts of the image during its deformation. This process uses an energy function and computes several paths traversing it, that define where the image could be cut while incurring less objectionable artifacts (Figure 2.15). This method has also been adapted to mesh deformation [DK14]. Dong et al. [DZPZ09] present an enhanced version of image seamcarving relying on grid deformation to preserve image structures. Kraevoy et al. [KSSCO08] present a similar technique applied to meshes, where they compute a volumetric grid of vulnerabilities and resize the mesh while minimizing their deformation.

Bokeloh et al. [BWKS11] present a pattern-based deformation of structured meshes, where they analyze an input mesh and find symmetries and repetitions. Then they can deform it while the system ensures the mesh structure's consistency. This structure analysis approach is then extended

with a new algebraic model [BWSK12] that enables the edition of complex structured meshes, such as chairs, stairs, or castles.

Milliez et al. [MWCS13] address structure deformation comprising multiple constrained interrelated objects, for instance a wall of a castle that would consist of walls and towers. Their technique can transform, compress, stretch, mix, or separate elements effectively. Their solution is based on energy minimization of a system with multiple equilibrium states, that are weighted when changing from one state to another. One state represents one geometric shape of an object, for example, a piece of wall can be a wall, a larger wall, a smaller wall, a round tower, and a square tower.

2.3.4 Conclusion on Interactive Modeling

Methods combining interactive controls and procedural generation are promising ways to model virtual worlds. However, finding good controls for complex procedural models is not a trivial task. All the presented methods make a trade-off between manual control and automatic generation, which depends of the nature of the task and of the target audience. Sketch-based interfaces are some of the easiest to use due to their paint-like nature, and they are still capable to provide fine controls. Intelligent deformation techniques are also an important part of interactive editing, where the user may want to deform a scene and adapt it to new constraints while preserving its visual properties.

2.4 Conclusion

The field of virtual world modeling comprises a large number of different approaches, particularly with procedural methods moving more and more towards user-centered approaches to provide powerful yet intuitive algorithms. No work has been done on the issues of generating villages, so we begin our contributions with this particular case in Chapter 3. Then, we move to the field of interactive procedural modeling in the specific case of waterfall sceneries in Chapter 4, an interesting example where the user would like to master design without having to take care of all constraints to be met for realism. We then study sketch-based interaction in the case of terrains (Chapter 5), where the user would like to control silhouettes from a given viewpoint without having to take care of all details making a terrain plausible. Finally, we go further in the interactive editing and delve in the domain of arrangement synthesis in Chapter 6, where we present a method for painting virtual worlds based on example-based synthesis.

Chapter 3

Procedural Generation of Villages on Arbitrary Terrains



Contents

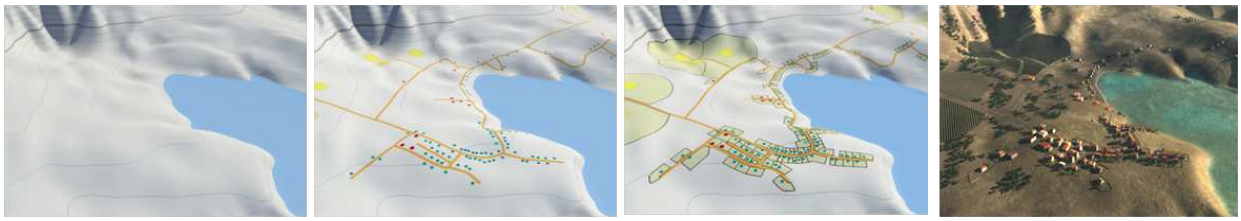
This chapter is an extended and updated version of our article presented at the conference *Computer Graphics International '12* and published as a special issue of journal *The Visual Computer* [EBP*12].

Villages are a vital part of the human habitat, especially in regions with ancient settlements. However, although procedural modeling of cities has attracted a lot of attention for the past decade, populating arbitrary landscapes with non-urban settlements such as villages remains an open problem. Indeed, as presented in Chapter 2, the modeling of human settlements has been restricted, up to now, to the generation of large cities, where building blocks are used to populate regular street networks [PM01, KM07, CEW*08, AVB08, VGDA*12]. Adaptation to rough terrains has been scarcely studied, and no previous work was conducted, to our knowledge, on the generation of villages.

Contrary to the large cities usually studied in computer graphics, most human settlements did not result from some pre-defined land-use plan, but from people progressively settling in safe, well-serviced, sunny or convenient locations for farming or fishing. Meanwhile, the road network progressively grew and in turn attracted new settlers [Bar99]. The result of such progressive settlement can still be observed in many regions over the world. For instance, it is the cause of the unique look of typical highland hamlets in the European Alps or of ancient villages on the banks of the Mediterranean Sea. Generating scattered settlements is a challenging problem that requires stepping away from the standard modeling paradigm used for cities such as such as the one from Parish and Müller [PM01]. In addition, modeling villages on mountanous terrains requires generating complex land parcels, driven by both winding roads and terrain slopes, and houses with irregular door and window positions.

Figure 3.1: Example of the kind of settlement that we would like to capture: a fortified village constructed on a cliff.

In this chapter, we present our method for generating villages on arbitrary terrains. More precisely, our goal is to propose a new method for generating small, European-like villages, where people took benefit of terrain features to progressively settle in safe, sunny or simply convenient places (Figure 3.1). Our method generates all the elements defining a village, from the road



(a) input (b) skeleton (c) parcels (d) geometry

Figure 3.2: Overview of our method: given an input terrain (a), we first generate the skeleton of the village (roads and settlement seeds) (b), then add parcels to the village layout (c), and finally we generate 3D geometry for houses (d).

network to the individual parcels of land, and to 3D houses adapted to the local slopes. Our main contributions are as follows:

- we propose a coupled settlement and road generation process that progressively creates a village layout on arbitrary terrain based on a growth scenario and on dynamic interest maps (Section 3.2);
- we introduce an anisotropic conquest process that creates plausible individual parcels of land (Section 3.3);
- we present an *open shape grammar* able to adapt the geometry of houses to the local terrain slopes (Section 3.4).

3.1 Overview

Definitions. In this work, we call *village* any non-urban, sparse settlement, for instance a group of terraced houses around a church, some remote hamlets, and a few isolated farms between them. We define the *region of interest* on which the village is to be created as a compact region $\Omega \in \mathbb{R}^2$. Ω is supposed to be connected to the outside world through a set of *connection points* Ψ, located at the edges of Ω, and that will serve as extremities for the future road network. The nature of the environment is pre-defined using functions over Ω. The terrain is represented as a heightfield, with vegetation and water maps. The functions $h(\mathbf{p})$, $w(\mathbf{p})$, and $v(\mathbf{p})$ respectively denote the elevation, the water height, and the vegetation density at a given point $\mathbf{p}$.

To enable the creation of various villages on the same terrain, we use a village type $\mathcal{V}$ (e.g., high-land settlement, defensive village, fisherman village). Each village has specific parameters that change the way they grow. The village growth follows a *growth scenario*, an ordered list of village types $\mathcal{V}$ and building types $\mathcal{B}$ to create over time. For instance, a village initially created as a farming village can become a fortified village, and then come back to a farming village. The generation process will alternatively generate *settlement seeds* B_i – marking the future locations of buildings, and of the roads R_j that serve them. We call *village skeleton* $\mathcal{S} = (\{B_i\}, \{R_j\})$ the result of this step. A settlement seed is defined as $B_i = (\mathcal{B}, \mathbf{p})$, where $\mathcal{B}$ is the building type (e.g., castle, church, terraced house, farm), and $\mathbf{p} \in \Omega$ is a position. A *road* R_j is defined by a set of node positions $\{\mathbf{p_k}\}$ controlling its central curve. During the generation process, we call *building*

encyclopedia a function that returns, for each pair $(\mathcal{V}, \mathcal{B})$, the set of parameters used to seed a building.

The village layout we need to compute is not only composed of the village skeleton, but also includes a tessellation of Ω into individual land parcels $\mathcal{P}_i$ and the associated building footprints. We call F_i the footprint of building B_i, defined as a subpart of $\mathcal{P}_i$. This footprint will serve as foundation for the geometry of the building.

Our algorithm for generating villages on arbitrary terrains is summarized in Figure 3.2. Given Ω, a few environment maps, and a user-defined growth scenario, we first grow the village skeleton, then generate land parcels and building footprints to get the village layout, and finally create 3D geometry. These three steps are detailed in Sections 3.2 to 3.4.

3.2 Growth of a Village Skeleton

The main idea that drives the entire generation process came from an attempt to answer the question : "Is there a building because a road existed, or is there a road because of the building presence ?". We believe that the answer to that question is : both cases do occur.

Indeed, when a new building is created in a place far from a road, a road needs to be created so the new building can be reached. Because of the traffic generated by the new road, or because a new place is now accessible, more buildings may be constructed around the first building or along the road.

As a consequence, we cannot generate a road network and then populate it with buildings, as it is done for the procedural generation of classical cities, and we cannot generate buildings and then try to connect them with a set of roads, because the building creation needs to be influenced by the road presence.

Coupled generation of buildings and roads. Our generation algorithm consequently alternates between building and road generation steps (Figure 3.3). The buildings are generated using a particle-based method for distributing buildings seeds on the terrain while considering environmental constraints and interest functions depending on the building type. This approach allows for dynamic update of interest regions, for instance when new roads are created.

In the remaining of this section, we detail the growth scenario (Section 3.2.1) and the interest maps (Section 3.2.2), and how we use them to generate building seeds (Section 3.2.3) and the road network (Section 3.2.4) that form the village skeleton.

3.2.1 Growth Scenario

The growth scenario allows the user to control the evolution of a village by defining a list of temporal events that can either be a change of village type or the seeding of several buildings of a given type.

Generation of building lots. A vast majority of events in the growth scenario concern the creation of n building seeds of a given type $\mathcal{B}$. During the execution of the scenario, the creation of a new building seed is immediately followed by its connection to the road network.

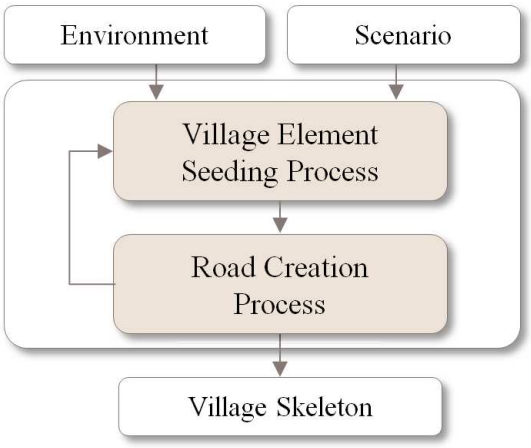


Figure 3.3: Algorithm for the village skeleton growth.

Change of village type. To enable the creation of various villages on the same terrain, we use a village type $\mathcal{V}$ (e.g., high-land settlement, defensive village, fisherman village). Indeed, the village type $\mathcal{V}$ sets some of the parameters used for seeding buildings, and thus affects the way the village will grow.

Moreover, it is also possible to model a change of environment or a change of population needs over time by changing the village type. For instance, we can start the village generation during a peaceful period, and generate a few sparse farms. Then, we can simulate a war by choosing a defensive village type, and the next houses will be generated in clusters in a safe place, such as at the top of a hill. Then we can change again the village type back to a prosperous farming period and continue the village growth (Figure 3.4).

Change village type	farming village
Generate building	1 church
Generate building	10 farms
Change village type	defensive village
Generate building	5 houses
Generate building	1 farm
Generate building	1 castle
Generate building	10 houses
Generate building	2 farms
Generate building	5 houses
Generate building	1 farm
Change village type	farming village
Generate building	10 houses
Generate building	20 farms

Figure 3.4: Growth scenario example for the fortified village of Figure 3.19.

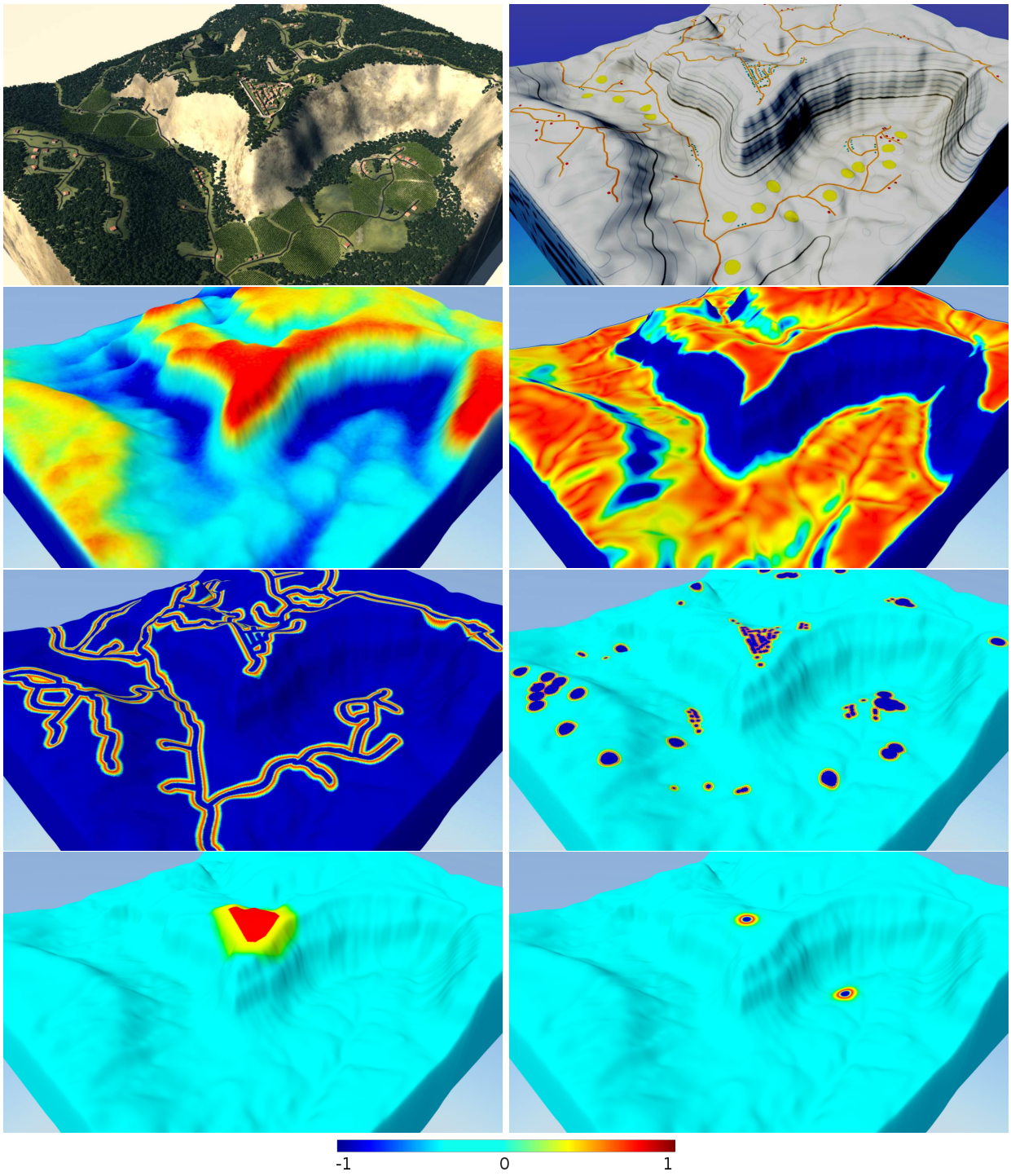


Figure 3.5: Visualization of interest maps for village type ($\mathcal{V}$ = fortified) and building type ($\mathcal{B}$ = house). Current village, current village skeleton, geographical domination, slopes, accessibility, sociability, fortification, and worship.

3.2.2 Dynamic Interest Maps

In our approach, we make the assumption that humans create buildings in interesting locations that respect the buildings constraints. Our seeding algorithm (Section 3.2.3) aims to find the best location $\mathbf{p}$ for a building of type $\mathcal{B}$ and a village type $\mathcal{V}$.

The location interest is evaluated by combining, with coefficients depending on the village and building types, n independent functions $f_i(\mathbf{p}, \mathcal{V}, \mathcal{B}) \in [-1, 1]$ representing different interest criteria. A negative value is given if the location is undesirable (-1 if impossible), while a positive value indicates a favorable evaluation of the criteria at $\mathbf{p}$. The combination is controlled by the building encyclopedia, where a set of n weighing factors $\{w_i(\mathcal{V}, \mathcal{B})\}$ is pre-defined for each couple $(\mathcal{V}, \mathcal{B})$. This method is general: it can combine a variety of criteria according to the desired results.

The interest of a location $\mathbf{p}$ for a building type $\mathcal{B}$ of a village type $\mathcal{V}$ is then given by:

$$\mathcal{I}(\mathbf{p}, \mathcal{B}, \mathcal{V}) \begin{cases} 0 & \text{if } \exists i \text{ such that } f_i(\mathbf{p}, \mathcal{V}, \mathcal{B}) = -1 \\ \max(0, \sum w_i(\mathcal{V}, \mathcal{B}) f_i(\mathbf{p}, \mathcal{V}, \mathcal{B})) & \text{otherwise.} \end{cases}$$

We present in Figure 3.6 four different templates that are used to evaluate the interest functions described later in this section. Those functions, given the shape parameters λ_{min}, λ_0, and λ_{max}, return a value $\in [-1, 1]$ depending on the distance to an object type. Parameters λ_{min}, λ_0, and λ_{max} are provided by the encyclopedia depending on the building and village types. It is their value variations, as well as the interest weights, that allow for the creation of various village types.

Some interest functions, stored as static 2D maps, are pre-computed before the village generation starts. Others, such as sociability and accessibility, are dynamically updated when a new building (resp. road) is created. The values of these interest functions can be displayed on the ground using a colormap-based visualization, as shown in Figure 3.5.

Sociability. This criterion measures the interest of buildings to be clustered. Since being too close to a neighbor is generally less attractive than being at a short distance, we thus use a sum of attraction-repulsion functions $f_{att}(d)$, where d is the distance between B and the surrounding buildings. Parameters λ_{min}, λ_0, and λ_{max} are pre-defined for each couple of building types, and stored in the building encyclopedia. This enables us to set distinct preferred distance values between terraced houses and between farms.

Worship. This function models the attraction of houses to religious elements, such as temples, churches, statues, or monasteries. These elements, often at the very center of villages, are those around which houses were initially constructed. We use the same kind of attraction function than for sociability, but computed only for the surrounding buildings of religious type.

Accessibility. This function expresses both that a building close to an existing road is easier to reach, and that settlers usually prefer to create their houses in a well-serviced place. We use an asymmetric bell-shaped function (Figure 3.6) of the distance d to the closest road. Parameters λ_0 (preferred distance to a road) and λ_{min}, λ_{max} (defining the interval outside which construction is prohibited), depend on the building type.

Slope. We use a bell-shaped function depending on building type to express preference to a given slope value, and to prohibit construction outside of a given slope range. This can be used to attract corn farms to flat areas and vineyards to hills.

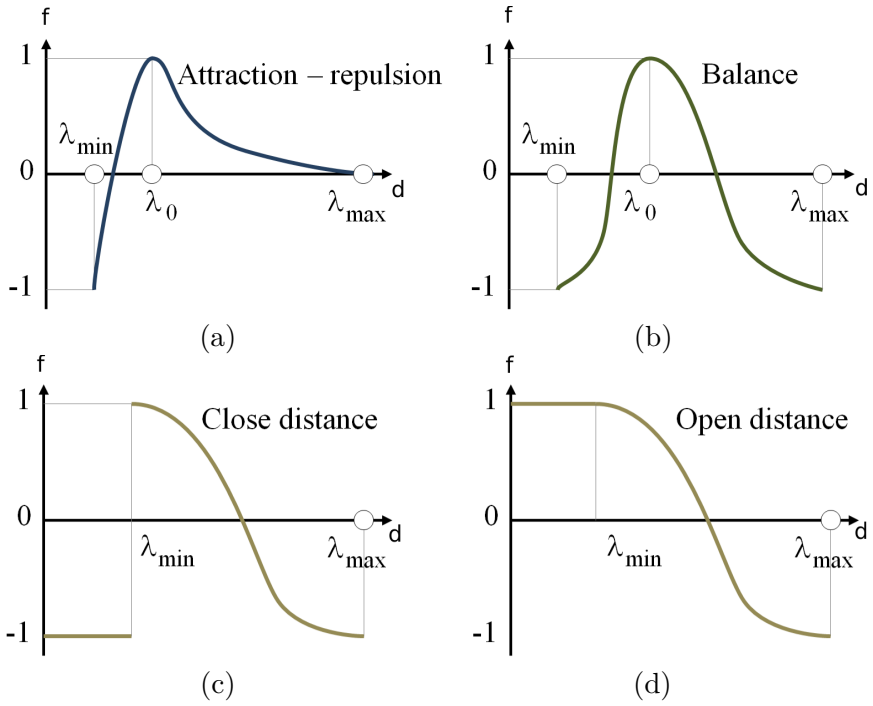


Figure 3.6: Functions used to compute interests. Attraction-repulsion function (sociability and worship) (a), balance function (roads and slope) (b), close distance function (water) (c), open distance function (fortifications) (d).

Water. Being able to attract houses to the seashore, a lake, or a river is important. We use a *close-distance* function (Figure 3.6), a decreasing function of distance to the nearest water body. Minimum and maximum distances, λ_{min} and λ_{max}, depend on the building type.

Fortification. During wartime, constructing within a fortification or close enough to a castle, is important for houses. This interest is computed using an *open-distance* function (Figure 3.6) (with $\lambda_{min} = 0$), a decreasing function of the shortest distance to the nearest fortified enclosure. f_{open} is equal to 1 inside fortifications.

Geographical domination. Either an indicator of social superiority or as necessity for defense, being at a higher spot than surrounding buildings is an important factor. Churches and monasteries are often built in overlooking places, so that they can be seen from afar. The importance of height decreases with the distance to the point of interest. We use the following function:

$$f(\mathbf{x}) = \sum_{\mathbf{p}, \|\mathbf{x}-\mathbf{p}\|<r} \frac{h(\mathbf{x}) - h(\mathbf{p})}{1 + \|\mathbf{x} - \mathbf{p}\|^2}.$$

$\mathbf{p}$ denotes sample points on the terrain, $\|\mathbf{x} - \mathbf{p}\|$ the Euclidean distance between $\mathbf{p}$ and $\mathbf{x}$, r the influence radius, and $h(\mathbf{p})$ the height of the terrain at $\mathbf{p}$.

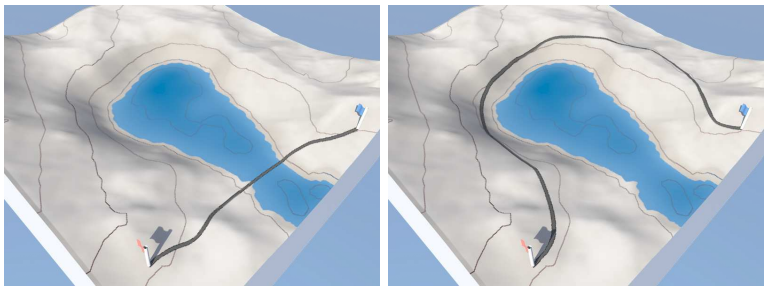


Figure 3.7: Procedural road generation with different weight parameters [GPMG10]. Left: distance and slope costs only. Right: taking water into account.

3.2.3 Aggregation-based Building Seeding

The way human settlements spread over a terrain is somewhat similar to the growth of natural plant species. Our approach resembles the "Open Diffuse Limited Aggregation" model presented by Desbenoit et al. [DGA] and adapts it in several original ways. Instead of moving particles over the land randomly to set the location of a new building, we rely on a stochastic positioning process followed by a local interest-based aggregation.

Seeding algorithm. The seeding algorithm (Algorithm 1) works as follows. A position $\mathbf{p}$ for building B to be positioned is randomly selected, and the conditions for constructing at $\mathbf{p}$ are checked (for instance, a farm cannot be built in the middle of a lake). If construction is possible, we compute a local interest value $\mathcal{I}(\mathbf{p}, \mathcal{B}, \mathcal{V})$ that measures the advantage for the building to be at its current location. The parameters of the interest function, extracted from the building encyclopedia, depend on the village and building types. Then, we perform a random choice, called the *aggregation test*, for deciding whether position $\mathbf{p}$ should be kept or not for B, with a probability of success depending on $\mathcal{I}(\mathbf{p}, \mathcal{B}, \mathcal{V})$. In case of failure, we randomly select a new position and iterate the process until a good position is found. When the building is constructed, we immediately create a road to connect it to the road network. Consequently, the next building seeding step will take this new building and this new road into account.

Algorithm 1 Aggregation-based building seeding algorithm.

repeat
  $\mathbf{p} \leftarrow$ random location *//stochastic sampling on the terrain*
  $p \leftarrow \mathcal{I}(\mathbf{p}, \mathcal{B}, \mathcal{V})$ *//evaluate the position interest*
  $r \leftarrow$ uniform probability in $[0, 1]$ *//sampling value for the aggregation test*
until $r < p$
$\mathcal{S} \leftarrow B(\mathbf{p}, \mathcal{B}, \mathcal{V})$ *//add the building to the skeleton*
$\mathcal{S} \leftarrow connectRoadToBuilding(B)$ *//generate a road to connect the building*

3.2.4 Connection to the Road Network

As stated in the seeding algorithm (Algorithm 1), we connect each new building to the road network just after its construction. To do so, we propose an extended version of the road generation method

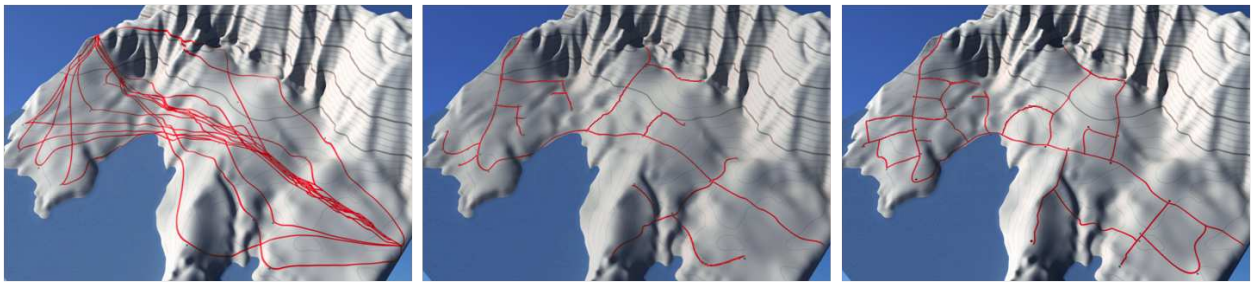


Figure 3.8: Creating new roads. From left to right: without and with road re-use, with road cycle generation.

introduced by Galin et al. [GPMG10]. We present in this section the original algorithm, and then describe our contributions related to road network generation.

Original algorithm. The road generation algorithm [GPMG10] relies on a shortest path algorithm, of type A^*. Given a start and an end point, the algorithm finds a path on the terrain that minimizes a given cost function. The construction cost of a road R is a weighted sum of different cost functions g_i:

$$\mathcal{C}(R) = \sum_{j=0}^{m} w_j g_j(R).$$

The algorithm uses several cost functions, such as a distance cost to favor shorter paths, but also a slope cost to prevent the road to climb a too steep area, a water or vegetation cost to prevent the road to cross difficult areas (Figure 3.7).

We use this algorithm to connect every new building B to Ψ, the set of pre-defined connections to the outside world. In addition to slope, curvature, and water costs, we use another function g expressing the cost for a road segment of crossing an existing building. This cost is set to a large constant to prevent collisions.

First we improve the previous algorithm to enable road re-use, then we force the construction of road cycles to prevent the apparition of a tree-like road network, which would not be plausible.

Road re-use. To prevent the method from generating fully ramified road networks, we note that, as in real life, the lower-cost connection to the existing network should be looked for. This is modeled by introducing a new *re-use weight* $w_{ex} \ll 1$, used to reduce the cost of the path when using existing road segments:

$$\mathcal{C}'(R) = \begin{cases} w_{ex}\mathcal{C}(R) & \text{if } R \text{ belongs to a road} \\ \mathcal{C}(R) & \text{otherwise.} \end{cases}$$

This way, a new segment is correctly connected to the network, as illustrated in Figure 3.8. Because our road construction method has a re-use strategy, connecting B to Ψ will often result in the creation of a road trying to connect to the nearest existing road.

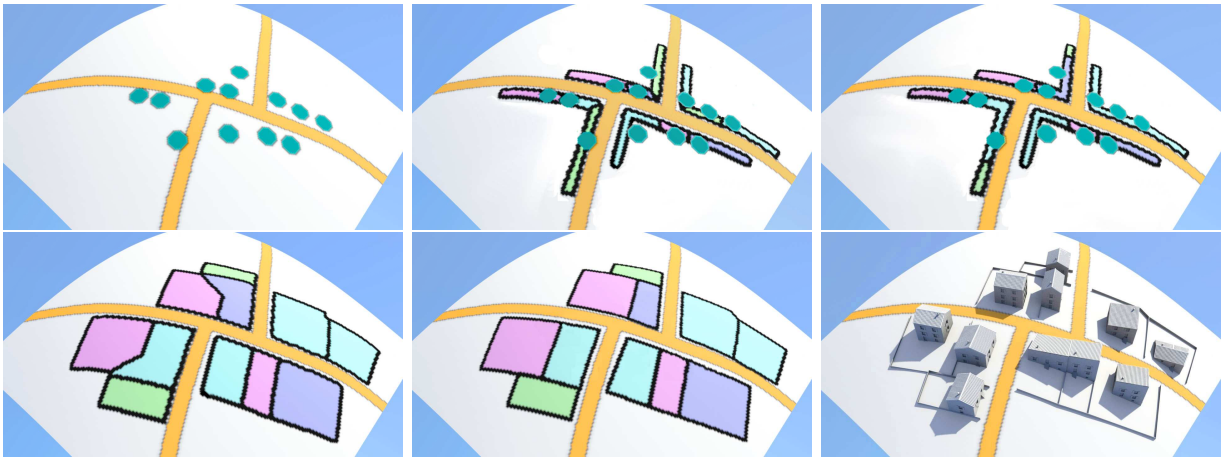


Figure 3.9: Land parcel generation algorithm. From left to right, top to bottom: village skeleton, road conquest, corner conquest, region conquest, parcel simplification, and building generation.

Road cycles. Real road networks often include cycles, providing shortcuts. We thus add a cycle construction step (Figure 3.8). Once a first road to a new building is computed, we try to extend it by looking for the closest road node p in a cone of angle θ from B, centered on the current road direction. We then generate a road between B and p. This road is created, leading to a new cycle, whenever it is not too close to the other road serving B (the latter may occur due to slope constraints forcing roads to go around obstacles).

3.3 Land Parcel Generation

Computing a village skeleton (road trajectories and building seeds) is not sufficient for generating the layout of a village: we also need to tessellate the terrain into individual parcels of land, where houses, gardens, or fields will be defined.

In classical procedural city modeling techniques, parcels are computed by subdividing street cycles into building lots. Recently, Vanegas et al. [VKW*12] proposed a new approach for generating building lots, which solves the problem of complex corner situations. However, because we generate a sparse settlement, we often do not have a cycle to subdivide, but a set of building seeds near village roads.

A first approach, investigated by Glass et al. [GMB06], is to rely on Voronoi diagrams to define a parcel of land around each building seed. This approach leads to mostly isotropic parcels that lack structure and are not aligned with roads.

After carefully analyzing the layout of parcels in real villages, we observed that most parcels have one side neighboring a road and two sides perpendicular to it, the shape of the last side being driven by other constraints such as the presence of neighbors or large changes in terrain slopes. Therefore, we rely on a three-step, anisotropic land conquest method to define adapted land parcels (Figure 3.9). First seeds conquer their road territory (Section 3.3.1). Then, they expand from the road using anisotropic conquest (Section 3.3.2). Last, the resulting parcel is simplified to avoid sharp angles (Section 3.3.3).

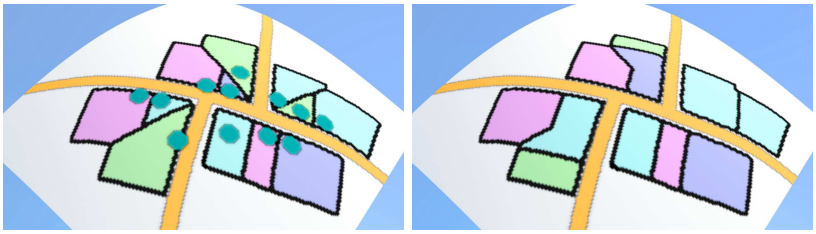


Figure 3.10: Without (left) and with (right) a corner conquest pass.

3.3.1 Road Conquest

Since each building seed is served by roads, we first define the part of the roads belonging to each parcel (Figure 3.9). Let source point S_i of parcel P_i be the projection of B_i on the closest road. We perform the road conquest by propagating P_i on both sides of S_i along the road, until collision with a neighboring parcel or until a maximum distance from S_i is reached.

3.3.2 Corner Conquest

When two buildings are at the same distance from a junction, road conquest leads to a collision at the angle, resulting into unplausible land parcels with sharp angles. Figure 3.10 illustrates the difference between our parcel algorithm with and without corner conquest.

Observing from real layouts that land at a corner between two roads generally belongs to a single lot, we use a corner conquest pass to resolve these conflicts: we allow parcel P_i arrived first at a junction to annex its neighborhood.

Note that building seeds that lose access to roads are suppressed.

3.3.3 Anisotropic Land Conquest

Once a building seed owns its parts of the roads, its land parcel is grown using grid-based propagation. Each road cell belonging to P_i is marked in the grid as a source S_i, and each of these sources is associated a fund $c_{\max}$. Then the sources are iteratively spread out. The process stops when the total cost of conquest from S_i reaches $c_{\max}$.

In general, land parcels are orthogonal to the road, with a shape depending on other constraints such as slopes. We thus use an anisotropic function to model the spreading cost. The cost dc_s for conquering an unoccupied cell d_s is set to:

$$dc_s(d_s, \mathbf{n}) = \sum_{i=0}^{n-1} \omega_i \, c_i(d_s, \mathbf{n})$$

where c_i are independent cost functions modelling external constraints, all depending on the conquest direction $\mathbf{n}$, defined as the normal to the road at the source. The weights ω_i, set through the building encyclopedia, depend on the village and building types $(\mathcal{V}, \mathcal{B})$. They enable us to ensure, for instance, that a castle or a farm will get more land than a terraced house. We present below several useful cost functions.

Conquest cost. This cost is set to the distance from the current cell to source S. We use the infinite distance $d(\mathbf{p}, \mathbf{n}, \mathbf{t}) = |\max(\mathbf{p} \cdot \mathbf{n}, \mathbf{p} \cdot \mathbf{t})|$ to get almost quadrilateral parcels for them.

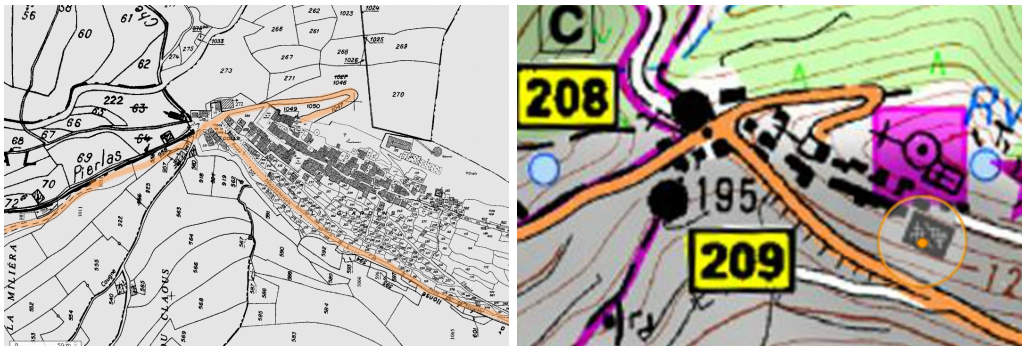


Figure 3.11: Left: layout of a real village with land parcels. Right: terrain isolines from an IGN map. We can observe parcel shapes (left figure) that depend on slope directions (right figure).

Water, wall, and road costs. The conquest cost for water, wall, or road cells is $+\infty$. This allows us to constrain the parcel shape with the road curvature, and to prevent the parcel to cross a wall or water bodies.

Slope cost. As observed in real village layouts (Figure 3.11), the shape of land parcels (especially those with fields) is sensible to local slopes, almost enabling to guess the main terrain features from them. We model this using an anisotropic cost function, for which spreading in the main slope direction is difficult. Slope cost is computed as a quadratic function of the directional gradient of the terrain height, to reduce the influence of small slope variations and increase those of larger ones.

3.3.4 Parcel Simplification

Once the grid cells belonging to P_i are computed, we extract a polyline representing its contour. Meanwhile, the shape of P_i is simplified to account for the fact that even in small villages, parcel boundaries mainly consist of straight lines. This simplification is done in two steps, inspired from mesh simplification methods.

First, we remove vertices that have little influence on the contour shape. Let $e_0 = (\mathbf{p}_0, \mathbf{p}_1)$ and $e_1 = (\mathbf{p}_1, \mathbf{p}_2)$ denote two edges. If angle $\angle(\mathbf{n}_0, \mathbf{n}_1)$, where $\mathbf{n}_0 = \mathbf{p}_1 - \mathbf{p}_0$ and $\mathbf{n}_1 = \mathbf{p}_2 - \mathbf{p}_1$, is smaller than a constant threshold ϵ we replace the two edges by $e_2 = (\mathbf{p}_0, \mathbf{p}_2)$. In the second step, we remove unplausible acute angles that appear at some T-vertices of the parcel boundary mesh (Figure 3.9).

3.3.5 Building Footprint Computation

Because it is the most frequently observed shape for buildings (Figure 3.16), we decided to illustrate our method with only quadrilateral footprints. Note that the building footprint algorithm should depend on the building type; however we did not implement it in our prototype.

We initialize a quad in each parcel, at the closest position to the road, and oriented according to the closest normal to the road. The quad grows until it either reaches the maximal size for its building type, or collides with the contour of the parcel. If one of the segments is close to the contour, its vertices are projected onto it, enabling the generation of terraced houses when several neighboring houses use this strategy.

3.4 Geometry Generation

The final step of our method is the creation of the 3D geometry of the village, including roads, buildings, and vegetation. While existing methods, such as from Galin et al. [GPMG10], can be used to generate accurate road geometry, existing methods for generating houses from their footprints [KW11] need to be extended to allow the generation of plausible houses on hilly terrains.

Indeed, we can observe in Figure 3.12 that often in small mountain villages, windows and doors have unusual shapes, and façades have complex layouts so as to conform to architectural constraints such as non-collision with the ground, alignment with floors whenever possible, and guaranteeing at least a door and a window per room.

Figure 3.12: Some examples where the architecture of a house has been adapted to the slopes of the underlying terrain.

In this section, we introduce the concept of *Open Shape Grammar* to adapt geometry generation to such constraints (Section 3.4.1). Then we detail the rules used in our prototype to model doors and windows (Section 3.4.2).

3.4.1 Open Shape Grammar

Shape grammars [MWH*06] are already capable of taking into account internal constraints such as the collision between a window and a wall, to prevent its construction. We extend the construction constraints by enabling on-the-flight adaptation of newly created façade elements so that various plausibility constraints are met. In our work, this process is implemented for doors, windows, and stairs.

We define an *Open Shape Grammar* as a shape grammar where the application of a selected rule depends on external constraints. For instance, a rule can be canceled if some external constraints are not met, such as non-collision with the ground or with other buildings. Open Shape Grammar rules also incorporate adaptation mechanisms: each rule selection yields a series of attempts to create the output shape according to constraints due to the environment. The element is created (and then the rule outputs success) as soon as a valid configuration is found. The application of the rule is canceled (and it outputs failure) if a maximum number of attempts is reached, and all have failed.

Algorithm 2 details a simplified example of such rules. λ contains an ordered list of parameters for the creation of object B(λ), and C is the fail case.

Algorithm 2 Example of Open Shape Grammar rule.

```
A → while(∃λ ∈ Λ) try B(λ) else C
B(λ) → if(external constraints of λ are respected)
            create B(λ)
            throw success
       else
            throw failure
C → ...
```

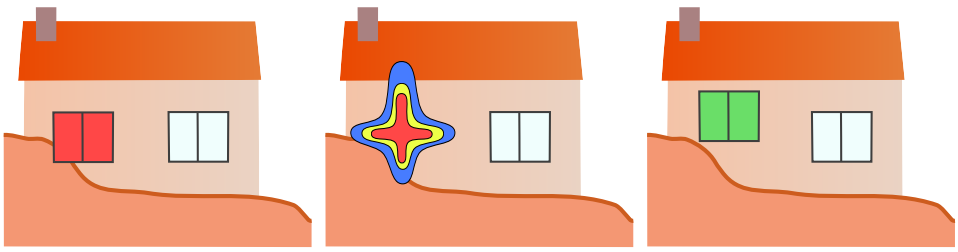
In practice, we use that principle to generate windows and doors that are adapted to the terrain constraints. To do so, we introduce position and shape adaptation rules, described below.

3.4.2 Geometry Generation Algorithm

The geometry of a building is generated from the building footprint using a standard method [MWH*06]: first we generate the floors and the roof, then we add façade elements, such as doors and windows. The latter is done using an open shape grammar, with the two kinds of adaptations detailed below.

Position adaptation. Each element is first positioned at the most plausible location: aligned with a floor and centered on the wall for a window; centered horizontally on the first floor for a door. Next, we move the element on the wall surface with a *displacement cost kernel* $\mathcal{K}$ until the element has a valid position (Figure 3.13). The kernel enables us to set preferences on the correction direction, such as favoring horizontal displacements over vertical ones.

Let $\mathbf{p}$ be the position of the element on the surface and $c(\mathbf{p})$ its cost. If the position is not valid, the unexplored neighborhood of the current position $\mathbf{p}$ is added to a priority queue with a cost equal to $\mathcal{K}(dx, dy) + c(\mathbf{p})$. The position of lower cost is evaluated next. The creation of the element is canceled after a user-defined number of failed tests. Figure 3.13 depicts the priority map for a window. In our example, we use a cost to favor horizontal displacement over vertical and diagonal ones, to match our observations (Figure 3.12).



$$\mathcal{K} = \begin{pmatrix} 2.0 & 0.1 & 2.0 \\ 0.5 & \infty & 0.5 \\ 2.0 & 0.1 & 2.0 \end{pmatrix}$$

Figure 3.13: Adaptation of window location: collision, priority map, result, displacement cost kernel.

Shape adaptation. The second level of adaptation is to change the geometry of the façade element that we try to create (Figure 3.14). The candidate shapes are stored in a pre-defined priority queue, depending on the building type. To position an object, we initialize its shape to

the top of the priority queue. If all positioning attempts fail, the shape is changed and the process starts again. If no shape is appropriate, the object is not built.

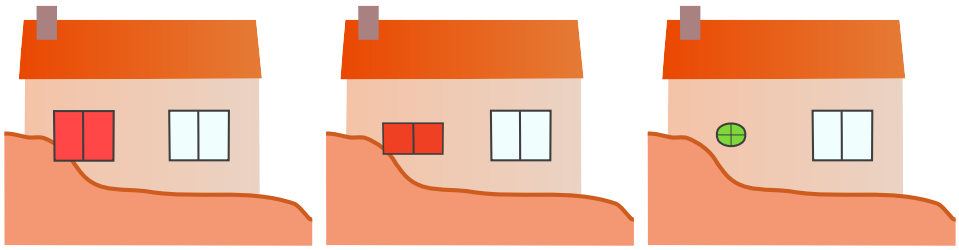


Figure 3.14: Shape adaptation using an Open Shape Grammar. For each window type we test for position (here, with only horizontal moves and a minimum distance between windows), and change shape if construction is impossible.

In our prototype, we generate windows following Algorithm 3. The position adaptation rule was also used for the door generation, as the Figure 3.17 shows.

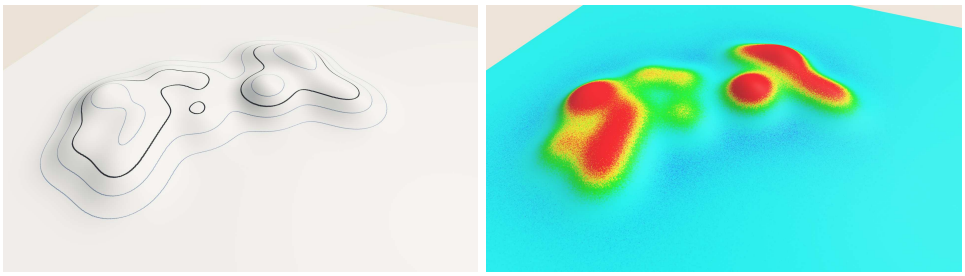
Algorithm 3 Open Shape Grammar rules for windows.

```
WindowFacade → while (∃ s in S)
                   while (∃ p in P)
                       try (Window(p, s))
               Wall
Window (p, s) → if(doNotCollideWithGround (p, s))
                   ...
                   throw success
               else
                   throw failure
Wall → ...
```

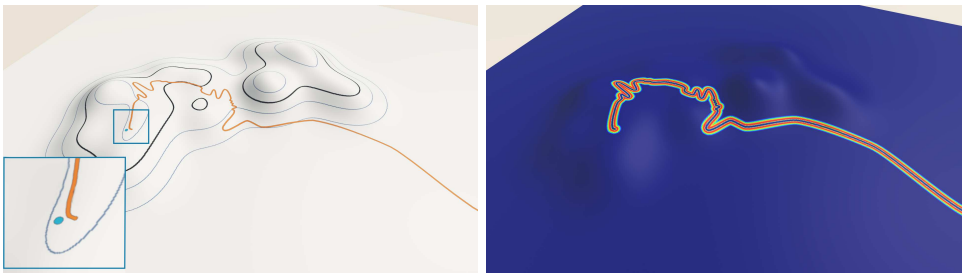
3.5 Results

Our village modeling system is coded in C++. Renderings were performed off-line. In this section, we present the results of our procedural method for generating villages.

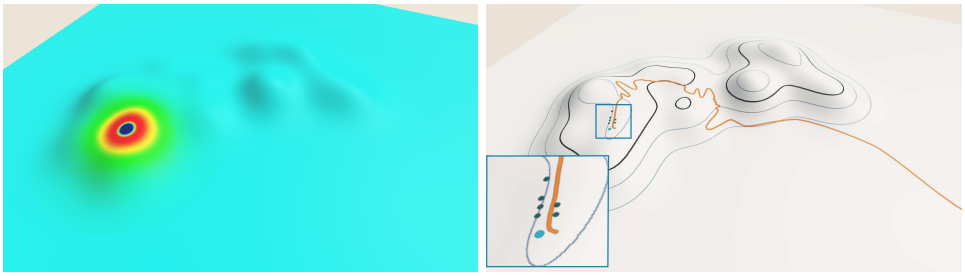
Example of village growth. The main contribution of our algorithm is the coupled generation of buildings and roads to produce a village skeleton. We detail in Figure 3.15 the evolution of a village and of some of its interest maps to illustrate our method.



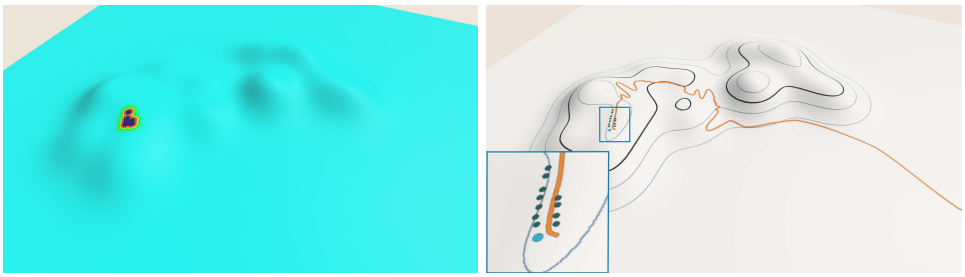
Left: initial terrain. Right: geological domination interest.



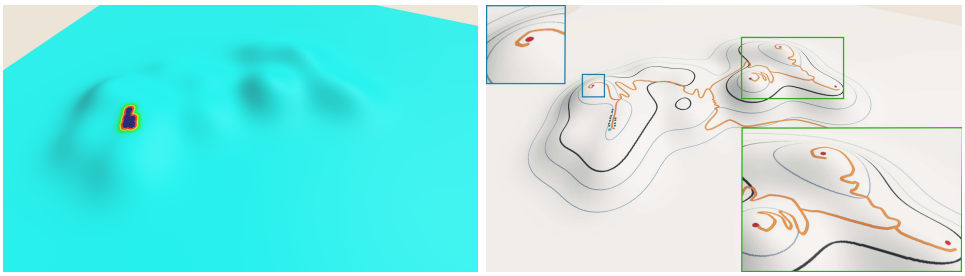
Left: church creation (looking for geological domination) and first road creation. Right: updated accessibility interest.



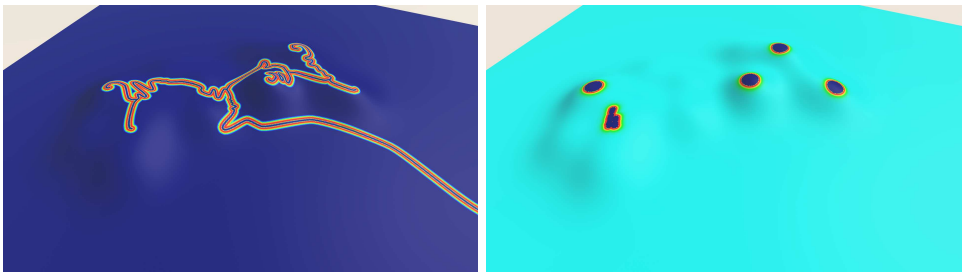
Left: worship interest. Right: creation of several houses along the existing road (accessibility interest), and near to the church (worship interest).



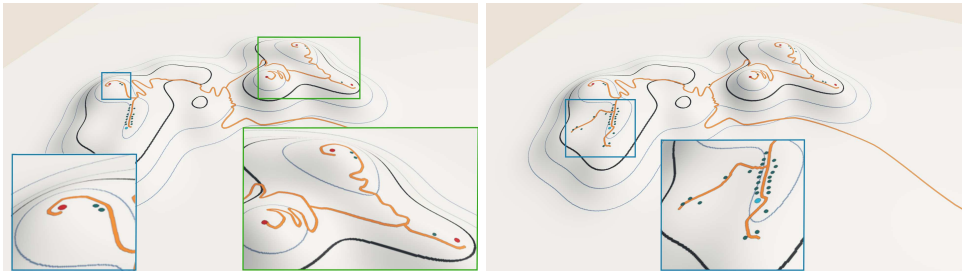
Left: sociability interest. Right: creation of several houses, attracted by the existing houses (sociability interest), the road, and the church.



Left: updated sociability interest, after the generation of new houses. Right: generation of villas, looking for isolated places (sociability interest with a high repulsion).



Left: updated accessibility interest. Right: updated sociability interest.



Left: generating more houses, attracted by the new roads and villas. Right: generating more houses, some of them creating new roads.

Generating several farms, looking for flat areas.

Figure 3.15: Example of village growth.

Parcel generation. To validate our parcel generation method, we compared the shapes of parcels we created with those of real village layouts, with similar building distributions and road networks. One of our results is depicted in Figure 3.16. Parcels have similar shapes and the mean number of neighbors (2.873 with our model, 2.812 in the real data) and of contour edges (4.29 with our model, 4.068 in the real data) are similar.

Figure 3.16: Comparison of real versus generated land parcels on terrains with similar roads and building distributions. From left to Right: real village parcels, Voronoi cells, parcels generated by our method.

Geometry generation. Our method for building generation with an Open Shape Grammar allows us to create houses on steep slopes while avoiding collisions of doors and windows with the ground. Note in Figure 3.17 the changes of position and shape of these elements.

Figure 3.17: Example of houses created by our system using Open Shape Grammars.

Influence of parameters. Figure 3.18 shows the influence of parameters during seeding. Distance to sea was the main criterion for the creation of the first village, whereas the second was seeking for domination.

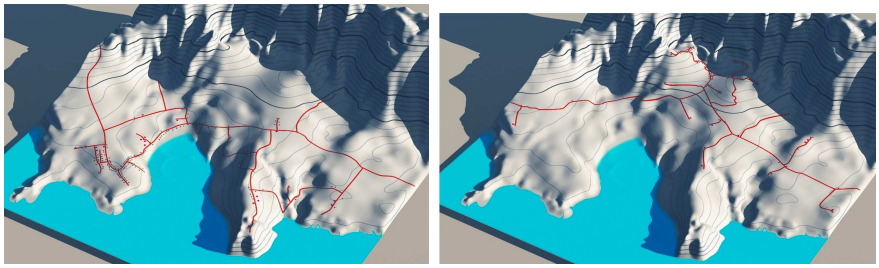


Figure 3.18: The left image shows a fisherman village, favoring distance to the sea; the right image shows a defensive village, favoring geographical domination.

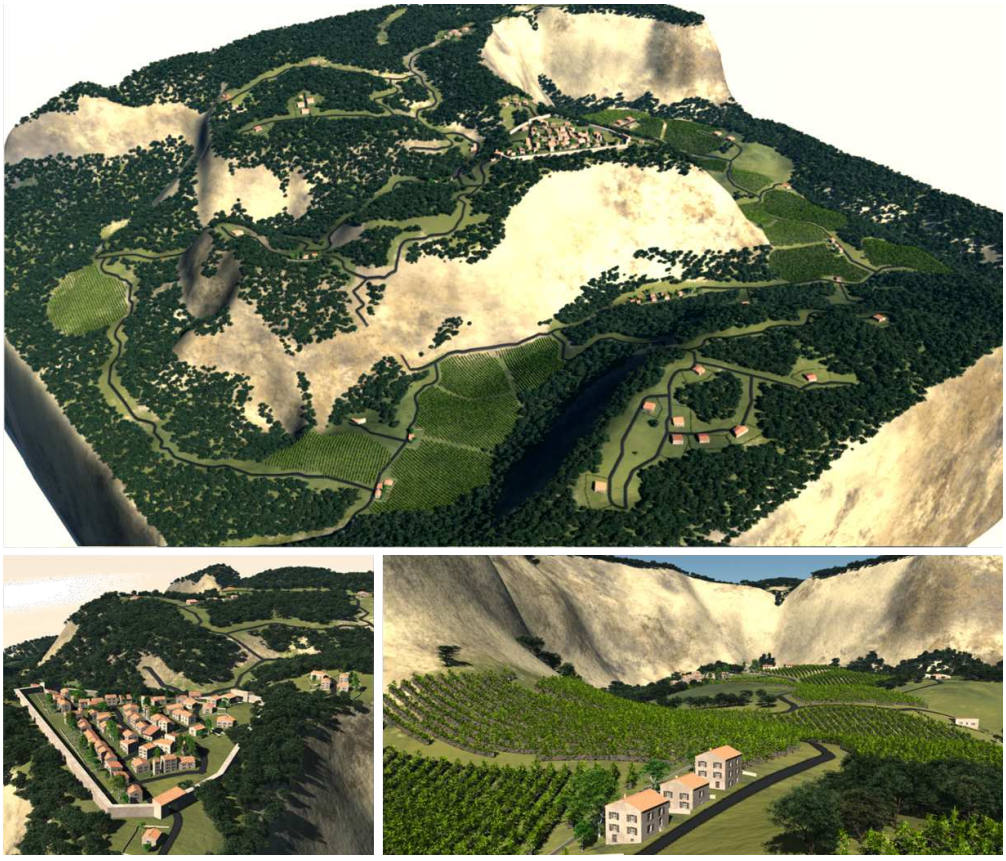


Figure 3.19: Fortified village at the top of a cliff, using a wartime growth scenario followed by farming style settlement.

Performance. Table 3.1 gives the time spent in each phase of the generation process. The skeleton generation timings are due to the high rejection rate of the aggregation test for positioning each building. We accelerated slightly this step by generating several buildings at once near a good location, when found. Moreover, our naive stochastic sampling could be improved by a better seeding algorithm, for instance, by moving a building seed using the interest gradient to reach a local maximum, and generating a new location only if the local maximum fails the aggregation test.

Land parcel generation is the most compute-intensive part of our algorithm, due to the size of the grid used to perform the spreading (we used a 4096×4096 grid). Note that depending on the desired output (with or without individual gardens around houses), this step can be skipped.

Fig.	Skeleton	Parcels	Geometry
3.18	4 : 00	7 : 00	0 : 20
3.20	5 : 00	11 : 00	0 : 30
3.19	7 : 00	13 : 00	0 : 30

Table 3.1: Computation time (in minutes) for generating the villages shown in Figures 3.19, 3.20, and 3.18.

Figure 3.20: A real (top left) and a procedurally generated highland hamlet (top right, and bottom).

Village diversity. Figures 3.19, 3.20, and 3.21 show different kinds of villages generated by our method. Figure 3.19 shows a fortified village at the top of a cliff where the main criteria were geographical domination and protection by fortifications. Figure 3.20 shows mountain villages, for which geographical domination was the main factor influencing seeding. Figure 3.21 shows a fisherman village, where houses look for flat regions near to the sea. These results demonstrate the effectiveness of our approach for generating settlements that conform to European layout styles. We believe that non-European village types could be created as well by modifying and tuning the growth scenario and cost functions.

Limitations. The main limitation of our method is the number of user-defined parameters, currently 150 per village type. Fortunately, these parameters, stored in the building encyclopedia, can be re-used to create a large variety of villages, depending on the terrain and on easily specified growth scenarios typically created in 2 minutes. Displaying the interest values on the terrain helps users understand and parameterize the method, although their design goals may still be reached after a long series of trials and errors, as in every procedural generation method.

Figure 3.21: Fisherman village.

3.6 Conclusion

In this chapter, we presented an original method for generating scattered settlements on arbitrary terrains, enabling villages and hamlets, with the associated roads, forests, and fields to be built on arbitrary landscapes. We demonstrated that our method can generate different types of villages with coherent and adapted geometry. We validated our results through comparisons with real layouts and photos.

In the future, a more efficient seeding algorithm would greatly contribute to accelerate the skeleton generation step. Also, while our parcel generation deals correctly with street corners and terrain adaptation, its computation time is too long. A real-time method, inspired by recent progress in city parcel generation [VKW*12], still needs to be found.

This first chapter of contributions presented a method combining procedural generation with intuitive parameters. The strength of the method is to enable the emergence of plausible villages using a set of parameters that are easy to understand. However, despite good results, this work made us aware of the difficulties of generating a particular result via indirect controls only. The rest of this thesis therefore investigates interactive methods more focused on interactive editing, providing direct controls over the generated scenes. In Chapter 4, we will study the interactive procedural modeling of waterfall sceneries, where the user directly controls a coarse waterfall network and the algorithm generates the associated detailed waterfall scene.

Chapter 4

Interactive Procedural Modeling of Coherent Waterfall Scenes

Contents

This chapter is an extended version of a paper first presented at the national conference *AFIG '13* [EPCV13] and then selected for an improved version in the associated journal *REFIG* [EPCV14a]. A further improved version has been published in journal *Computer Graphics Forum* [EPCV14b].

Waterfalls offer some of the most beautiful settings in nature. Despite this, no easy-to-use method for designing waterfall scenes has been developed so far in computer graphics. One solution consists of using physically based simulation of fluids on top of a modified terrain [BTHB06]. Unfortunately, modeling a terrain in order to produce specific waterfalls is an extremely daunting task. Fluid simulation depends on slopes, collisions, riverbeds, source flux, water properties, flow speed, etc. All the work of deforming a terrain is not warrant of the final appearance of a waterfall, as everything follows an indirect, highly nonlinear behavior. Another potential solution consists of manually creating a few waterfalls with standard modeling tools, using meshes and/or particles, and then positioning them along the terrain and connecting them by streams [SDZ*07]. In this long and tedious process, the artist also needs to manually maintain the consistency between the terrain and waterfalls, as well as the self-consistency of the waterfall network.

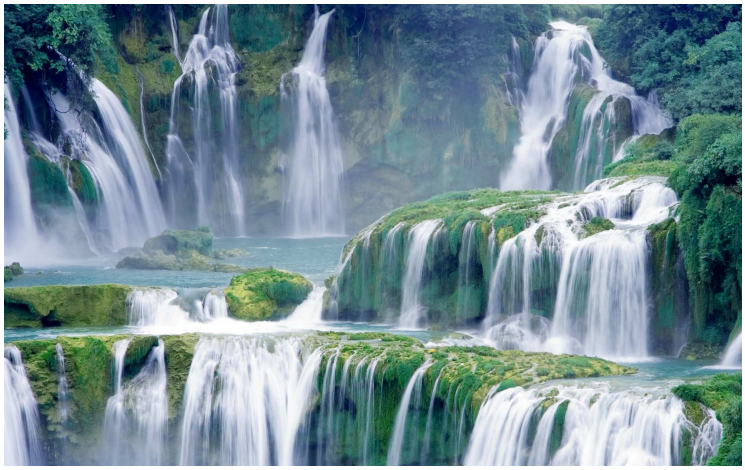


Figure 4.1: Example of the kind of waterfall sceneries that we would like to model.

In this chapter, we combine interactive editing and procedural methods to enable fast and easy design of plausible waterfall scenes. Our solution, based on a new interactive procedural model for flowing water networks, allows users to easily design complex waterfall scenes while automatically enforcing the physical consistency of the results, both in terms of hydraulic flow and of plausible embedding into the terrain. Our main contributions are as follows:

- we propose a slope-flow diagram-based classification of waterfalls (Section 4.1.1);
- we present three parametric models for designing waterfall elements (Section 4.2.1);
- we introduce a procedural method for ensuring waterfall network consistency (Section 4.2);
- we propose automatic methods for locally adapting water trajectories to the terrain and/or the terrain to the flow (Sections 4.3 and 4.4).

All these contributions are combined in a framework allowing an artistic approach to river and waterfall design. The resulting scene could either be used as a synthetic environment for games or films, or as an initial setup for further refinement through physically based simulation.

4.1 Overview

The key goal of our method is to leave coarse design of waterfall scenes in the hands of the user, while providing automatic ways to generate plausible and detailed results.

In the real world, flow networks formed by complex waterfalls, such as the one in Figure 4.1, include free-falls, segments where running water remains in contact with the terrain, as well as pools. Moreover, each of these segments can be of many different types, from rivers to rapids for the ones in contact with the terrain, and from plunges to cataracts for free-falls. Having to manually select plausible types for each segment of a full network would be both tedious and require specialized knowledge from the user.

Below, we propose a two-level classification for segments of waterfall networks, enabling us to leave the choice of low-level classes to the user, while automatically computing the most appropriate running-water types from quantitative information, such as the slope of the underlying terrain and the intensity of the flow. The processing pipeline that we use for modeling a waterfall scene, based on this analysis, is presented next.

4.1.1 Two-level Classification for Running Water

Waterfall scenes are comprised of three easily identified types of elements: running-water segments that remain in contact with the terrain, free-fall segments where water is in the air, and pools that receive water from free-falls. Our goal is to provide some coarse, intuitive control to the user; we leave the choice of these three classes, *contact*, *free-fall*, and *pool*, to the user during interactive design.

In contrast, we would like to free the user from the manual and explicit determination of the precise aspects that each running-water segment should take, because this can be determined in a more plausible way using an automatic procedure. After studying the existing classifications of streams and falls, we designed a new slope-flow classification that matches our goals, as explained next.

Running water can take on many forms, from rivers to rapids, and from plunges to cataracts. See Figure 4.2 for an illustration of these classes. Several classifications of waterfalls are proposed by hydrologists, geologists, and artists, in order to capture this variety:

- volume-based classifications of waterfalls [Bei06] sort waterfalls into classes by using a logarithmic scale over the volume of water in the air at a given time. Although easy to compute from quantitative information, this classification provides little clue on the visual aspect of the fall. Moreover, it is restricted to free-falls, and therefore does not fully meet our needs;

- geometric classifications, such as the one found in the Waterfall Lover's Guides [Plu05, DD06] or the one depicted in Figure 4.2, analyze the different types of geometries that can be observed in nature. However, they only provide visual information. No quantitative measurement is proposed to automatically compute the class that a running-water segment belongs to.

In this work, we would like to classify waterfall segments from quantitative information, while getting visual clues enabling us to generate plausible 3D representations for each segment. We therefore decided to augment the geometric classification of Figure 4.2 with quantitative evaluation of the classes, as in volume-based approaches. However, we need measures applicable both to free-falls and to running water in contact with the terrain, and therefore our solution is different.

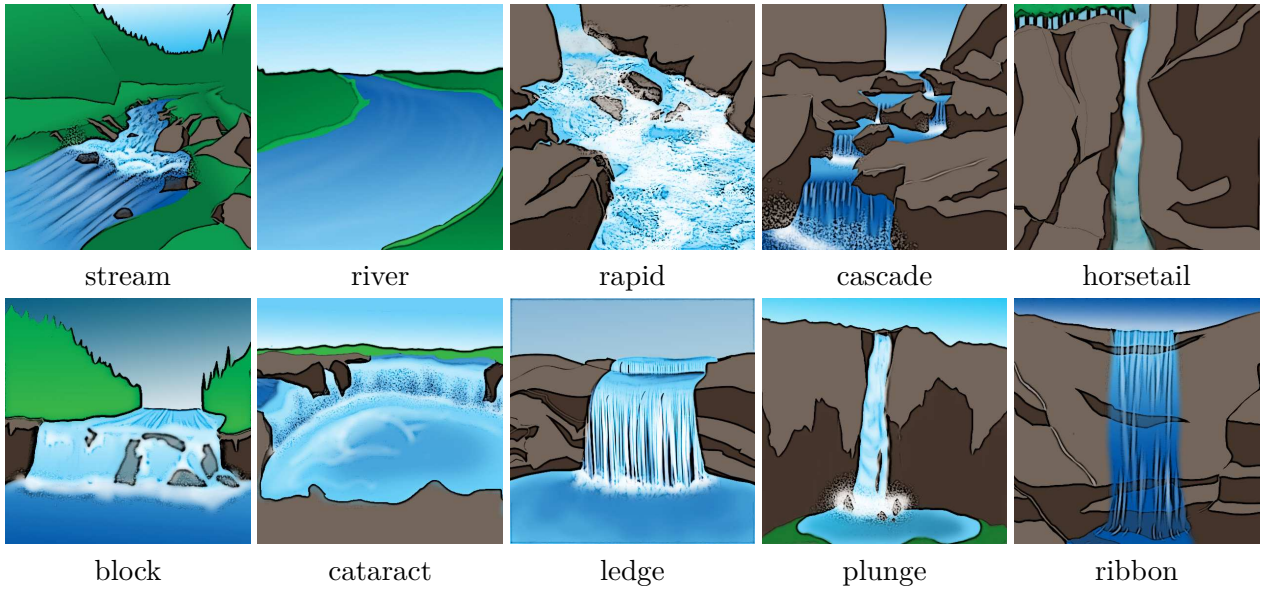


Figure 4.2: Artistic drawings illustrating the different types of running water and free-falls that can be found in nature.

By studying existing geometric classifications and looking at waterfall informations in many real cases, we noticed that the geometric type of running water mostly depends on two important quantitative parameters: the flow value (defined as the volume of water per second traveling through a cross-section of the segment), and the local slope of the terrain. Intuitively, when the flow decreases, a river becomes a stream, a cascade becomes a horsetail, and a free-fall cataract becomes a ledge. Meanwhile, if a given water flow is running on terrains of increasing slope, a river tends to become a rapid, and then eventually a block.

To provide a quantitative classification, we define the different classes of running-water segments as regions in a unified slope-flow diagram. This is done as follows: we first used real examples to find a consistent set of seed values for typical elements of each of the visual types depicted in Figure 4.2. These representative slope and flow values are listed in the left columns of Table 4.1. The regions associated with each class are then defined using a qualitative Voronoi segmentation in slope-flow space. The resulting classification is depicted in Figure 4.3. Note that the types of water flows on the left of the division in red belong to contact waterfall segments, while those on the right belong to free-fall waterfall segments.

We also used the visual classifications serving as reference to associate visual parameters with each type of elements in our classes, given in the columns to the right in Table 4.1. These values will be used for generating the appropriate visual representation for each element of the waterfall scene.

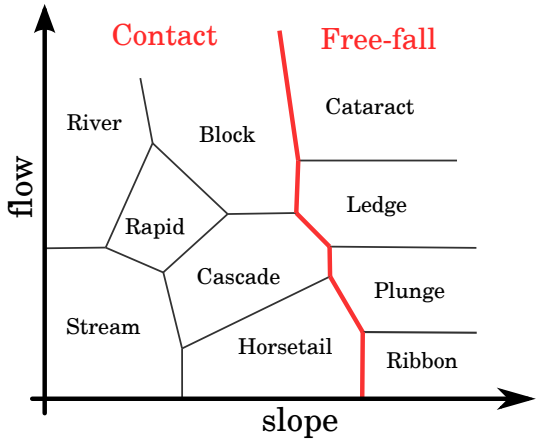


Figure 4.3: Our classification of waterfall types.

Type	Slope	Flow	Foam	Rocks	Dist.	Part.
Ribbon	$\pi/2$	1	0	0.3	0	0
Plunge	$\pi/2$	2	1	0.4	0	0.2
Ledge	$\pi/2$	5	0	0.5	0	0.5
Cataract	$\pi/2$	8	1	1	1	1
Stream	$\pi/16$	2	0	0.2	0	0
River	$\pi/32$	5	0	0.2	0.1	0
Rapid	$\pi/16$	5	0.2	0.5	0.5	0
Cascade	$\pi/8$	3	0.5	1	1	0
Horsetail	$\pi/4$	2	1	0	0	0
Block	$\pi/4$	6	1	0	0.2	0

Table 4.1: Classification of running-water segments that may appear in a waterfall network. The coordinates (slope, flow) give the position of the class seed in the slope-flow diagram of Figure 4.3. Slope is an average inclination in radians, while flow is expressed in "*flow units*", which gives an informal notion of relative proportions between waterfall types. Foam, rocks, disturbance, and particle are parameters ($\in [0, 1]$) used in our procedural generation of geometry and for rendering.

4.1.2 Processing Pipeline

In the remainder of this chapter, we define a *waterfall network* as a network of *waterfall segments* and of *pools*. *Waterfall segments* can either be of type *contact* (in contact with the terrain) or of type *free-fall* (in the air).

Our processing pipeline, based on the two-level classification we just defined, develops as follows (see Figure 4.4):

1. The user starts by creating a waterfall scene, building an *oriented vectorial controller network* $\mathcal{U}$ over an existing terrain. This is done using three vectorial controllers—*contact*, *free-fall*, and *pool*—all based on Cardinal splines. *Contact* and *pool* are respectively open and close

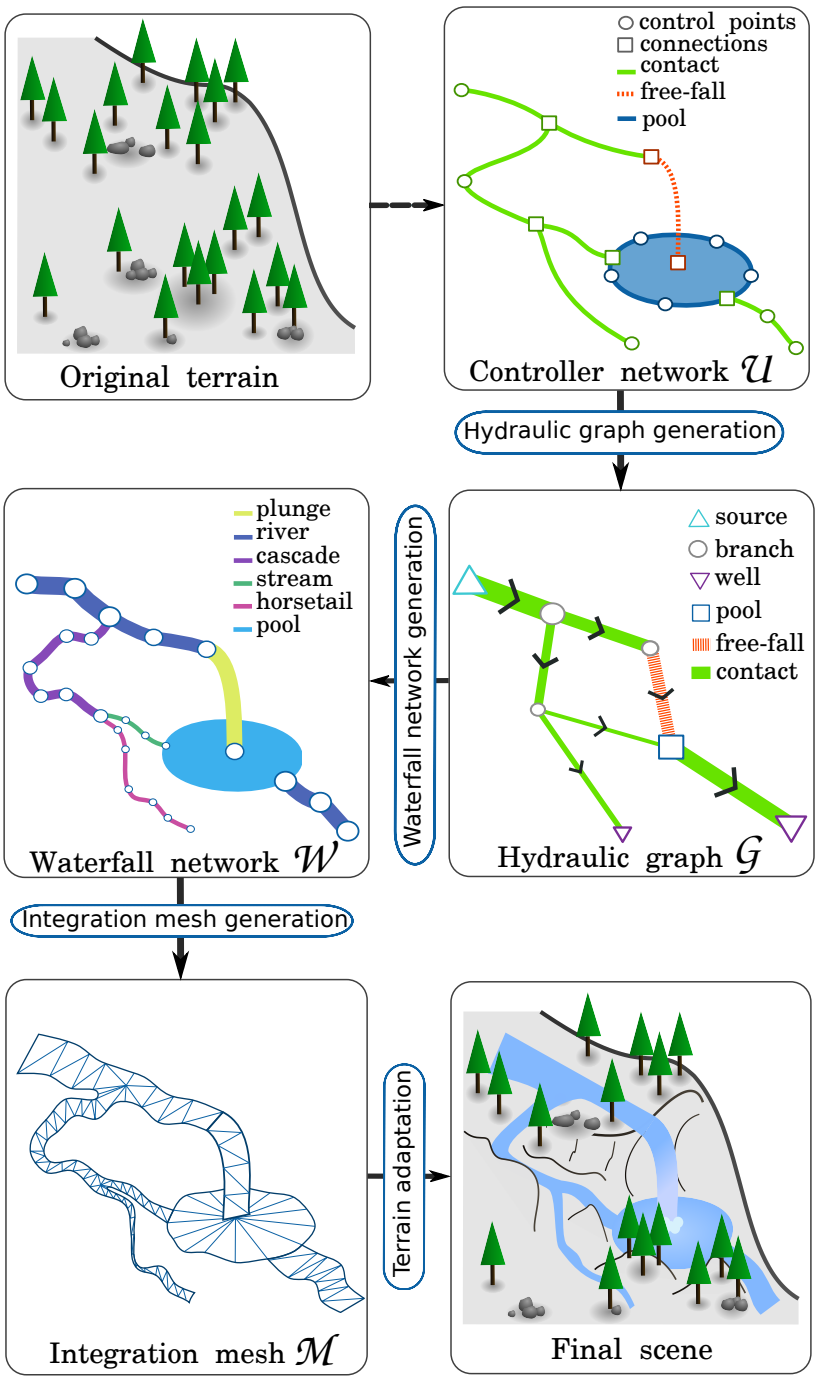


Figure 4.4: Processing pipeline used for creating waterfall scenes. The first step is the creation by the artist of the controller network $\mathcal{U}$. Then, a hydraulic graph $\mathcal{G}$ is generated; the width of an arc encodes flow quantity. The waterfall network $\mathcal{W}$ is then generated with a subdivision algorithm, and the waterfall types are determined. Finally, the integration mesh $\mathcal{M}$ is generated. It is used to deform the terrain and to generate procedural details.

curves interactively created using control points while free-falls are parabolas, automatically parameterized by their start and end points. During interaction, controllers are not constrained to be placed in contact with the terrain, since the terrain will be adapted later according to the user design, but the flow is constrained to go downhill along each segment of the controller network. While the default flow intensities on each segment of the network can be interactively tuned by the user, a consistent hydraulic graph ($\mathcal{G}$ with fully consistent flow values) is automatically computed at the end of the interactive modeling process. These first steps are detailed in Section 4.2.

2. The next step is the generation of the *waterfall network* $\mathcal{W}$, which uses the coarse water trajectories from the controller network and the flow information from the hydraulic graph to define a more precise representation of the waterfall. In addition to directly using the control curves that they defined, the user has the option of further refining the geometry of the network through an automatic procedure that locally adapts running-water trajectories to the underlying terrain. Each curve of the network is then divided into a number of water segments, whose sub-class is determined by the slope-flow classification from Section 4.1.1. Finally, the last geometrical parameters, such as flow width and depth, are computed for each segment of the waterfall network (Section 4.3).

3. Lastly, the 3D representation for the waterfall network, called the *integration mesh*, is generated and embedded into the scene through appropriate local deformations of the terrain. Although they can include large changes, such as digging a canyon to allow a stream to find its way downhill (in the extreme case where the user designed water segments go through a mountain), constraints applied to the terrain are mainly aimed at automatically adding all the details that make the scene plausible: this includes borders along streams, riverbeds, and the creation of overhangs behind free-falls. Appropriate decorative elements such as trees and rocks are generated at this stage, using the distance to the closest riverbed and the class of the waterfall segment it belongs to. Rendering attributes are also set from the class of each segment (Section 4.4).

Note that we chose to compute a coherent hydraulic graph $\mathcal{G}$ (Step 1 of the pipeline above) based on the controller network $\mathcal{U}$, i.e., from the coarse trajectories defined by the user, before refining these trajectories into a waterfall network $\mathcal{W}$. This enables us to use consistent flow values in Step 2, while refining water trajectories. This helps us, for instance, to prevent large rivers from being refined into a series of short twists when adapted to the terrain, although a smaller stream would be allowed to be more winding. The fact that our algorithm interleaves geometric computation with consistency checks, leads to more plausible results. The remainder of this chapter details the three steps of the pipeline.

4.2 From User Input to a Coherent Hydraulic Graph

We will discuss how the user creates the controller network $\mathcal{U}$ (Section 4.2.1), how we generate the associated hydraulic graph $\mathcal{G}$, and how we compute its flow (Section 4.2.2).

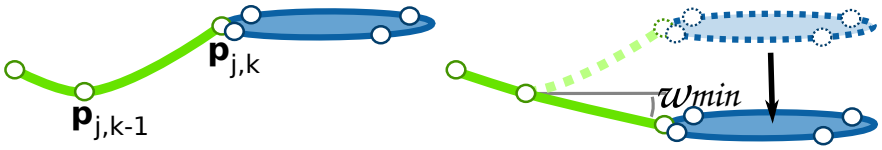


Figure 4.5: Minimum slope constraints are imposed on each curve of the controller network, to make it consistent with the user-defined flow direction. Left: A controller network. Right: The algorithm verifies that the minimum slope ω_{min} is respected between *contact* control points $\mathbf{p}_{j,k-1}$ and $\mathbf{p}_{j,k}$. Since this is not the case here, $\mathbf{p}_{j,k}$ is lowered. All subsequent controller points will also respect this constraint. In this figure, the pool control points are therefore lowered.

4.2.1 Controller Network Creation

The user builds a controller network $\mathcal{U}$ (Figure 4.4) by creating and manipulating different vectorial elements $\mathcal{V}_i$ and by interconnecting them.

We define *controllers* using a Cardinal spline with a controller type $\alpha_i \in \{$*free-fall*, *contact*, *pool*$\}$. The splines are composed of a series of control points $\Delta_i = \{\mathbf{p}_{i,k}\}$, which can be connected to other controller control points to create a controller network (Figure 4.4, Step 1).

Depending on α_i, controllers are set differently: a controller *contact* is created by positioning a series of control points $\mathbf{p}$ over the terrain. This controller is used to create all the elements that remain in contact with the ground, such as rivers. The *pool* controller is a closed cardinal spline, and is used to create pool contours. Because a controller *pool* is flat, the user first positions a horizontal plane in space, and then traces a closed curve over the plane using control points. Finally, the *free-fall* controller is used to create an element that loses contact with the ground. Consequently, only a start point and an end point are positioned on the terrain, while the lower point being typically placed inside a *pool*. The user associates a flow direction to *free-fall* and *contact* curves.

During editing, point and curve magnets are used to facilitate the interactive creation of the connections between elements, like in most classical vectorial editing tools. Moreover, the user can insert, delete, and move control points on each curve, and cut or merge controller curves.

Projection. To facilitate the adaptation of the controller network to the underlying terrain, users are provided with a projection tool, which can be used to project control points of the Cardinal splines onto the terrain. If desired, they can also define an offset from the terrain's local height for each control point. When the projection tool is used for pools, which are constrained to be flat, the pool level is automatically set to the average height of all projected contour points.

Minimum Slope. For the scene to remain consistent, the system must ensure that all slopes set to the controller network segments allow the water to flow downstream in the user-defined direction. This is accomplished using automatic correction of the user-defined positions, during a traversal of the controller network (see Figure 4.5): starting with the flow *sources* of the network, and traversing it in a topologically sorted order, we check if each control point $\mathbf{p}_{j,k}$ of curve Δ_j of each controller maintains the minimum slope ω_{min} with respect to its predecessor $\mathbf{p}_{j,k-1}$. If not, point $\mathbf{p}_{j,k}$ is lowered to match the constraint. For each connection, we check that all the outgoing nodes are lower than the incoming nodes, and update their positions if needed. Pool contour points are lowered, if necessary, according to the full set of free-fall segments coming into the pool.

4.2.2 Hydraulic Graph Generation

The hydraulic graph $\mathcal{G}$ (Figure 4.4, Step 2) is an oriented graph, generated from the controller network $\mathcal{U}$, in which the flow originates from the *sources* and exits by the *wells*. It is composed by a set of nodes $\mathcal{N}_j = (\beta_j, \gamma_j)$ and a set of arcs $\mathcal{A}_k = ((\mathcal{N}_i, \mathcal{N}_j), \eta_k, \gamma_k)$, where $\beta_j \in \{$*source*, *well*, *branch*, *pool*$\}$ is its type, γ_j is the flow going through the arc or the node, and $\eta_k \in \{$*contact*, *free-fall*$\}$ is the arc type.

For each controller $\mathcal{V}_i$ we create an arc if its type α_i is *free-fall* or *contact*, and create a node if α_i is *pool*. Nodes for *sources* and *wells* are introduced at the extremities of the graph, *branches* are created at the intersections between controllers.

It would be time-consuming and non-intuitive in cases of failure to perform a complex physical simulation on the flow propagation in order to deduce the entire flow properties (local intensity, speed, etc.) everywhere along the network. We were inspired by the interactive system of Zhu et al. [ZIH*11], and therefore simplified our hydraulic model by considering it as a "pipe-like" graph. This enables us to deal only, at this stage, with flow exchanges at the nodes of the hydraulic graph. In the remainder of this section we detail how to compute these flow exchanges using simple and intuitive functions, while providing a coherent visual aspect.

Flow Consistency. A hydraulic node should be at equilibrium, i.e., its incoming flow should equal its outgoing flow. Expressing this at each node of the hydraulic graph leads to a system of interconnected equations. There is generally an infinite number of solutions to this system. Therefore, instead of using a global solver to find consistent flow values, we solve them in succession for each node of the graph, exploring it in dependency order from sources to wells. This enables us to take into account the user specifications for the relative strength of the input flows, and to generate a solution that also best matches the coarse geometric trajectories, in terms of branching angles at each node. Our method for doing so is explained next.

Separating Incoming Flow into Outgoing Branches. The flow of an outgoing arc at a branch node $\mathcal{N}_i$ should depend on the angles between the inflow and outflow arcs, in order to capture the natural course of water. For instance, we expect that most of the flow should follow its own original direction.

Let $\mathcal{X}_j$ be an input arc of $\mathcal{N}_i$, and $\mathbf{u}_j$ its incoming direction (Figure 4.6). We distribute its flow γ_j to each output arc $\mathcal{Y}_k$ with a direction $\mathbf{v}_k$ according to a normalized weight:

$$w_{jk} = 1 + (\mathbf{u}_j \cdot \mathbf{v}_k) \qquad \bar{w}_{jk} = \frac{w_{jk}}{\sum_k w_{jk}}.$$

Thus, the outgoing flow γ_k for arc $\mathcal{Y}_k$ is computed as

$$\gamma_k = \sum_j \gamma_j \bar{w}_{jk}.$$

Separating the Flow out of Pools. Because the shape of a pool could be complex, distributing the flow according only to the angles between inflows and outflows would not be realistic, while computing a full simulation would be computationally expensive and not suitable for an interactive system [ZIH*11]. Instead, we simply distribute inflows equally to all the outgoing arcs, except when we need to take into account relative flow strength pre-set by the user for these branches.

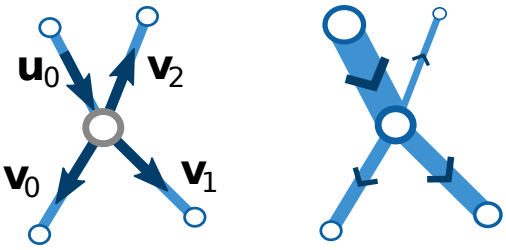


Figure 4.6: Flow in a branching. Left: Input and output directions. Right: Resulting flow exchange, drawn as segment thicknesses.

4.3 Waterfall Network Generation

In this section, we discuss the generation of a waterfall network $\mathcal{W}$ from the controller network $\mathcal{U}$, which contains the waterfalls coarse trajectories, and the hydraulic graph $\mathcal{G}$, which contains the flow information. The waterfall network is composed of waterfall segments $\mathcal{S}_i$, interconnected by waterfall nodes $\mathcal{B}_i$.

We start the construction process by subdividing the controller trajectories and locally adapting them to the terrain (Section 4.3.1). Then, we extract from them the waterfall segments and nodes that form the waterfall network (Section 4.3.2). We compute the type of each segment using the classification from Section 4.1.1, and set its other parameters while taking flow and slope into account.

4.3.1 Subdivision and Adaptation to Terrain

In order to procedurally improve the contact curves created by the user, we provide a fractal-like subdivision based on midpoint displacement, which takes the underlying terrain model into account (see Figure 4.7). Except for very intricate terrains, our subdivision scheme approximates the natural trajectory of a flow running down a slope while preserving the global shape of the user-defined curve.

Let $\mathcal{A}_j$ be an arc of the graph $\mathcal{G}$, and $\mathcal{V}_i$ its corresponding controller, with Δ the points of the curve manually created by the user, and γ the flow going through the arc. Consider the segment formed by two consecutive points $\mathbf{p}_i$ and $\mathbf{p}_{i+1}$. We subdivide this segment at its middle point $\mathbf{m}$ and move it along direction $\mathbf{u}$, perpendicular to this segment. This creates two new segments, V and W, represented by normalized vectors $\mathbf{v}$ and $\mathbf{w}$. While the original midpoint displacement method applies a random displacement to $\mathbf{m}$ with a maximum amplitude τ, we instead search for $\mathbf{x} = \mathbf{m} + \lambda\mathbf{u}$, $\lambda \in [-\tau, \tau]$, minimizing the cost function:

$$C(\mathbf{x}) = w_g C_g(\mathbf{x}) + w_a C_a(\mathbf{x}) + w_r C_r(\mathbf{x})$$

where $C_g(\mathbf{x})$ is the gradient cost, $C_a(\mathbf{x})$ the angle cost, and $C_r(\mathbf{x})$ the random cost. The values w_g, w_a, and w_r are weight coefficients associated to the costs.

Gradient Cost. The gradient cost favors paths that follow the slope of the terrain, and is defined as:

$$C_g(\mathbf{x}) = \frac{\int_V \|\mathbf{g}(\mathbf{t}))\| f_p(\mathbf{v}, \mathbf{g}(\mathbf{t}))d\mathbf{t} + \int_W \|\mathbf{g}(\mathbf{t}))\| f_p(\mathbf{w}, \mathbf{g}(\mathbf{t}))d\mathbf{t}}{\|\mathbf{p}_{i+1} - \mathbf{p}_i\|},$$

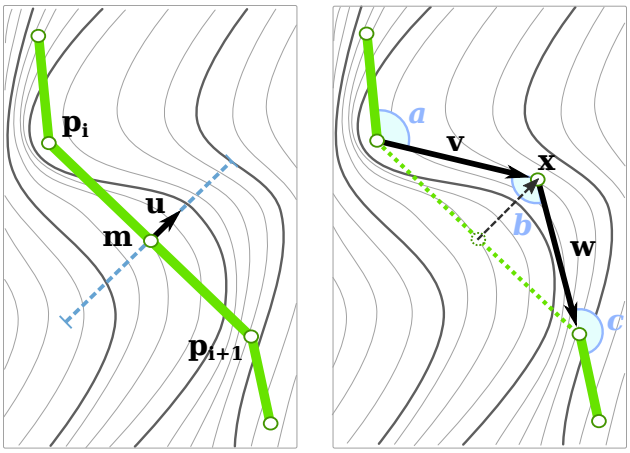


Figure 4.7: One step of our recursive subdivision process for waterfall network segments: the segment formed by $\mathbf{p}_i$ and $\mathbf{p}_{i+1}$ is subdivided at $\mathbf{m}$, which is moved along perpendicular direction $\mathbf{u}$. The final position $\mathbf{x}$ is the point that minimizes our cost function $C(\mathbf{x})$.

$$f_p(\mathbf{v}, \mathbf{g}(\mathbf{t})) = \left(-2\mathbf{v} \cdot \frac{\mathbf{g}(\mathbf{t})}{\|\mathbf{g}(\mathbf{t})\|}\right) + 1,$$

where vector $\mathbf{g}(\mathbf{t})$ is the gradient of the terrain elevation at position $\mathbf{t}$. By integrating the elevation gradient along segments V and W, and by using the scalar product between the gradient and the segment vectors ($\mathbf{v}$ or $\mathbf{w}$) as a penalty coefficient, the path will follow the slope and avoid obstacles (Figure 4.7). Moreover, instead of simply using the scalar product value as a penalty coefficient, we use function f_p to penalize paths that follow the isolines (i.e., when the scalar product with the gradient is zero); climbing a slope is penalized even more severely. Consequently, this cost is low when $\mathbf{v}$ and $\mathbf{w}$ are aligned in the same direction as the slope of the terrain, and high otherwise.

Angle Cost. The angle cost prevents undesirable sharp angles that could appear between two consecutive segments, because of their independent subdivisions. We take into account the angles a, b, and c (in radians), created at the introduction of new point $\mathbf{x}$ (see Figure 4.7), as:

$$C_a(\mathbf{x}) = \left(\frac{a}{\pi}\right)^2 + \left(\frac{b}{\pi}\right)^2 + \left(\frac{c}{\pi}\right)^2.$$

Random Cost. Finally, to add fractal-like details on flat terrains, we use the random cost $C_r(\mathbf{x})$, which is negligible when the other costs are high.

The segments are recursively subdivided until their length is inferior to l. Displacement amplitude τ and detail size l are set as:

$$\tau = \|\mathbf{u}\|/2 \qquad\qquad l = \gamma/\sigma.$$

The minimum subdivision length l is proportional to the segment flow, where σ is a user parameter. This enables us to get a more detailed trajectory for small flow values, enabling streams to become more winding than rivers. In our prototype we use the following values: $\sigma = 1/2$, $w_g = 1$, $w_a = 0.1$, and $w_r = 0.2$ for all our examples.

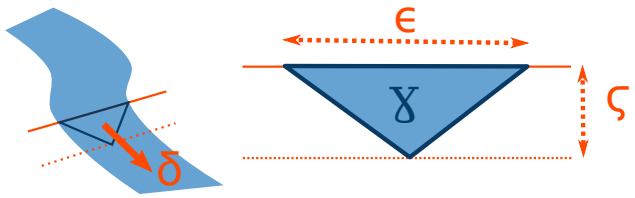


Figure 4.8: Triangular cross section of a waterfall segment, with a constant flow.

4.3.2 Waterfall Network Construction

The waterfall network is composed of waterfall segments $\mathcal{S}_i = (\mathbf{u}_i, \gamma_i, \kappa_i, \delta_i, \epsilon_i, \zeta_i)$ where $\mathbf{u}_i$ is the segment vector, γ_i is the flow going through the segment, $\kappa_i \in \{$*stream*, *horsetail*, *cascade*, *rapid*, *block*, *river*, *ribbon*, *plunge*, *ledge*, *cataract*$\}$ is the waterfall type according to our classification (Figure 4.3), δ_i is the speed of the flow, and ϵ_i and ζ_i are respectively the width and depth of the riverbed. These segments are interconnected by waterfall nodes $\mathcal{B}_j = (\mu_j, \gamma_j, \zeta_j)$ where $\mu_j \in \{$*source*, *well*, *branch*, *pool*$\}$ is the node type, γ_j the total incoming flow, and ζ_j the depth.

Each segment vector $\mathbf{u}_i$ is directly extracted from the subdivided trajectories (Figure 4.4), and its flow γ_i is equal to the flow of its corresponding graph arc. All consecutive segments are connected by a branch node with only one input and one output. The other segments are connected by the waterfall nodes $\mathcal{B}_j$ constructed from the hydraulic graph nodes $\mathcal{N}_k$ and their associated controller $\mathcal{V}_l$. The remaining parts of this section describe how the other segment properties are computed.

Waterfall Segment Type. For each waterfall segment $\mathcal{S}_i$, we know its slope s_i and its flow γ_i. These values automatically determine the waterfall segment's type κ_i by casting its coordinates (s_i, γ_i) in the slope-flow graph of Figure 4.3. We use Voronoi cells, whose coordinates are detailed in Table 4.1, to determine to which type the coordinates belong.

Note that if the user did not use a plausible waterfall controller, e.g., if he created a free-fall on a flat terrain, or a contact on a very steep terrain, the waterfall segment may be outside of the valid range of values in the slope-flow graph. In this case, the parametric model is still set using the closest type, but the user is notified (i.e., the related segment is drawn in red). He can then either validate the current design (even if it is not fully realistic), or select more realistic controllers.

Waterfall Segment Properties. The final step in the generation of the waterfall network is to compute the last properties of each segment $\mathcal{S}_i$, i.e., its speed δ, its riverbed width ϵ and depth ζ, from the slope and flow information we have. We propose a resolution that leads to satisfying results while being intuitive and fast to compute. Our hypothesis is to consider the waterfall segment as a closed pipe, with a constant flow and a triangular cross section (Figure 4.8).

In this case, the flow is given by:

$$\gamma = \frac{\delta \ \epsilon \ \zeta}{2}. \tag{4.1}$$

Since we have one equation with three unknowns (speed δ, width ϵ, and depth ζ), and complex inter-dependencies, a physically accurate solution should rely on strong assumptions, which cannot be justified in our context. Instead, we decided to solve the system by providing simple and intuitive functions to set plausible values for these variables.

We first propose to solve for the speed as a simple linear function of slope s: $\delta = ks$, $k \in \mathbb{R}^+$. Then, we set the depth as a function f_d of the width and of the segment type as: $\zeta = f_d(\epsilon, \gamma)$. Note

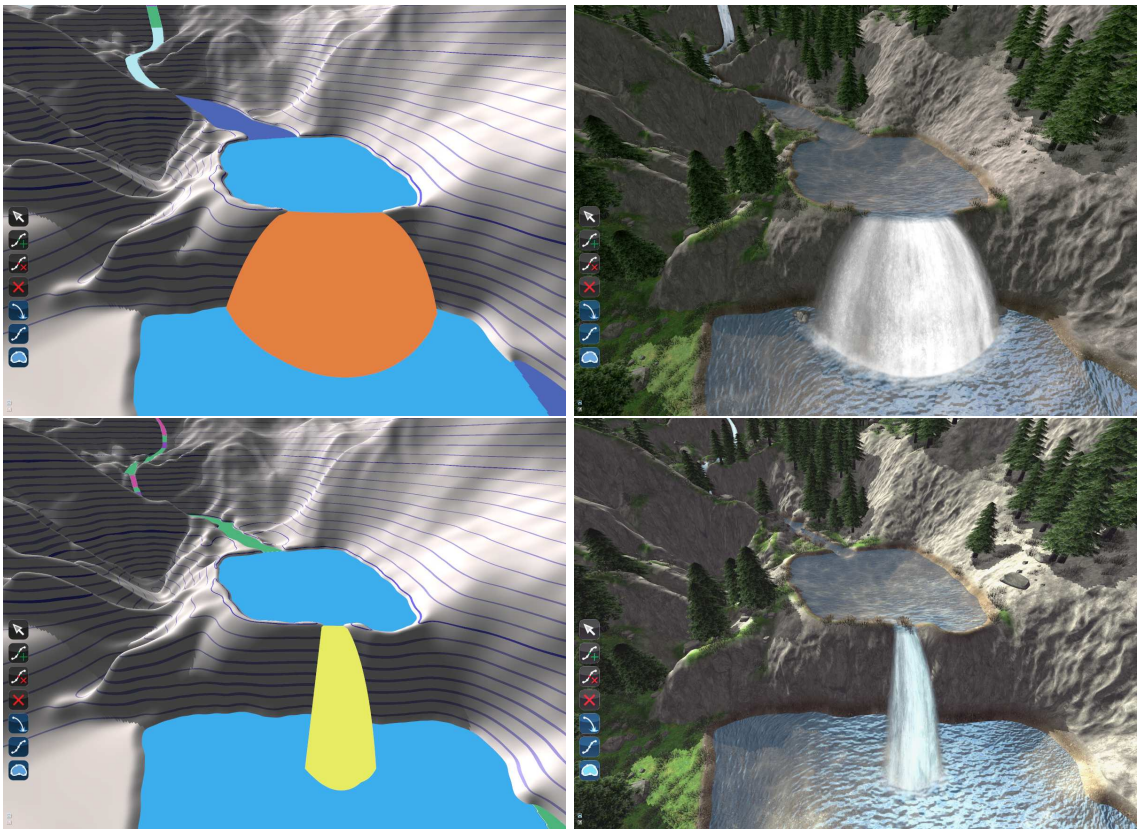


Figure 4.9: Varying the flow within a graph. The waterfall elements are changed automatically, in conformity with the classification, and the visual aspects are immediately adapted to the changes.

that our slope constraint enforces that $s > 0$ everywhere a waterfall segment. We used $k = 10$ in all our scenes. In our prototype we use only one profile function, $f_d(\epsilon, \gamma) = \frac{1}{2}\gamma$. Applying Equation 4.1, it is now possible to compute the width and to deduce the depth value.

While simple, our model efficiently provides plausible results using intuitive parameters. In Figure 4.9, the procedural component of our modeling system correctly handles a reduction of the input flow: when the upstream flow is manually reduced, the following free-fall flow reduces accordingly. The subsequent nodes in the graph are then affected. Note how the type of the outgoing element automatically changes, i.e., from ledge to plunge. Moreover, nothing prevents our method from being extended to account for more complex river profiles, e.g., as those introduced by Génevaux et al. [GGG*13].

4.4 Terrains Adapting to Waterfalls

In this section, we discuss how to adapt the terrain to the waterfalls. First, we generate the waterfall integration mesh (Section 4.4.1), then we use it to generate vectorial constraints to deform the terrain (Section 4.4.2) and to generate additional maps (Section 4.4.3). The latter are used for the visual integration of the waterfall (e.g., texture changes on the terrain and on the waterfall) and for the generation of procedural decorations.

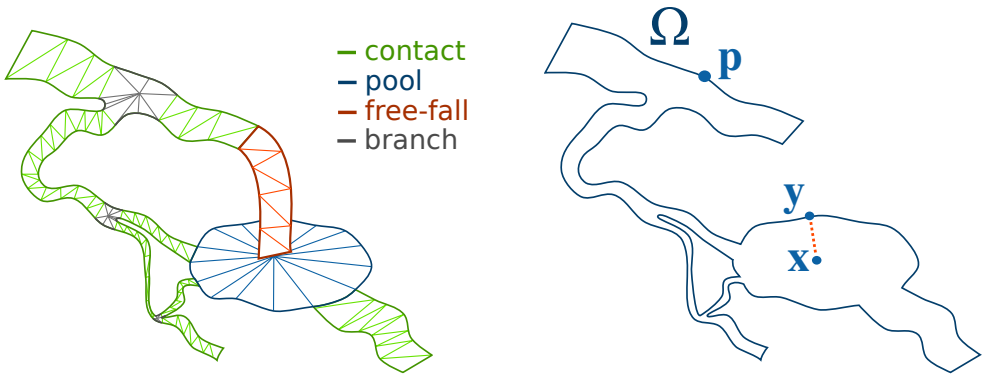


Figure 4.10: Left: Integration mesh composed by the individual waterfall element meshes. Right: Integration mesh borders.

4.4.1 Integration Mesh Generation

The integration mesh is a 3D mesh defining the waterfall surface. It is composed of the surface meshes for all waterfall elements (Figure 4.10 left), which are generated using the waterfall network trajectories and the widths of the elements.

The meshes for the contact elements are computed by extruding the width ϵ along the segment trajectories. We carefully handle the meshing connections at branch intersections, so meshes do not overlap each other. Free-fall meshes are extruded from the borders of their incoming segments (e.g., pool borders) and follow the free-fall element curve. Pool meshes are simply triangulated from their contours.

4.4.2 Constraint-based Terrain Deformation

The terrain is stored as a heightfield, on which a deformation map (displacement map) is applied to adapt the terrain to waterfalls. This deformation map is computed using diffusion curves [OBW*08], as inspired by the terrain modeling method of Hnaidi et al. [HGA*10]. However, our solution differs from theirs in that it propagates deformations (i.e., height differences) instead of heights, and it generates three different types of constraints (border, riverbed, and overhang) using our procedural model. As a result of our approach, small details on the original terrain will remain on the terrain after deformation.

Border Constraints. These constraints force the water to naturally follow waterfall trajectories, by defining height and gradient constraints on the mesh contour (Figure 4.11 top left).

Let Ω be the contour of the integration mesh (Figure 4.10 right). For each point $\mathbf{p} = (i, j, z) \in \Omega$, we define $d(x, y)$ as the difference between the terrain height $H(x, y)$ and the height at the point, i.e., $d(x, y) = z - h(x, y)$ (Figure 4.11 top center). The difference $d(x, y)$ is the value that will be "diffused" within our constraints solver. To perform this diffusion, we use the multigrid solver introduced by Hnaidi et al. [HGA*10].

First, the terrain constraints $d(x, y)$ are rasterized into a 2D grid. Then, we perform several diffusion iterations to compute a deformation map. Let $d^k_{i,j}$ be the deformation value at the location

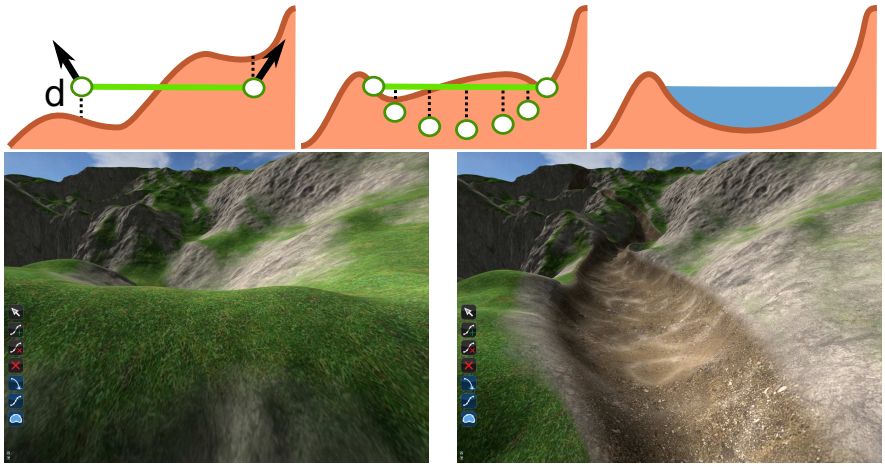


Figure 4.11: Border and riverbed constraints. Top left: Border constraints. Center: Riverbed constraints. Right: Final terrain. Bottom left: Original terrain. Right: Deformed terrain with apparent riverbed.

(i, j) at iteration k. The next iteration is computed as follows:

$$d_{i,j}^{k+1} = \begin{cases} d_{i,j}^{k} & \text{if } d_{i,j}^{0} \text{ is a constraint} \\ \frac{1}{4}(d_{i,j}^{k} + d_{i+1,j}^{k} + d_{i,j+1}^{k} + d_{i+1,j+1}^{k}) & \text{otherwise.} \end{cases}$$

The diffusion is performed in a multigrid solver, and the result is the 2.5D deformation map D. The final terrain heightmap H' is simply the addition of the original terrain height H and of the deformation map D:

$$H'(i, j) = H(i, j) + D(i, j).$$

Riverbed Constraints. The border constraints do not provide any control on the river profile, which should be a function of the river type (see Génevaux et al. [GGG*13]). For this reason, riverbed constraints are added.

For a given point $\mathbf{x}$ located inside the waterfall border (see Figure 4.10 right), we create a height constraint based on its distance to $\mathbf{y} \in \Omega$, the nearest point of the border, to which a profile function f_d is applied. The operation is repeated on a dense sampling basis. This defines a set of additional elevation constraints processed using our solver (see Figure 4.11 bottom).

Overhang Constraints. In order to create overhangs, we generate a horizontal displacement map based on the same diffusion technique. This map is then used to deform the terrain, as presented by Gamito and Musgrave [GM01].

Along the border at the top of a free-fall, we set a displacement constraint $\lambda\mathbf{u}$, where $\mathbf{u}$ is the free-fall direction and λ a constant defined by the user. In addition, we set a displacement constraint $-\lambda\mathbf{u}$ along the border of the receiving pool, under the free-fall.

As shown in Figure 4.12, these two constraints generate a flipped "S" curve, with the top of the overhang extending out of the terrain in the direction of the flow, and the bottom of the receiving pool extending into the cliff, due to erosion and falling rocks.

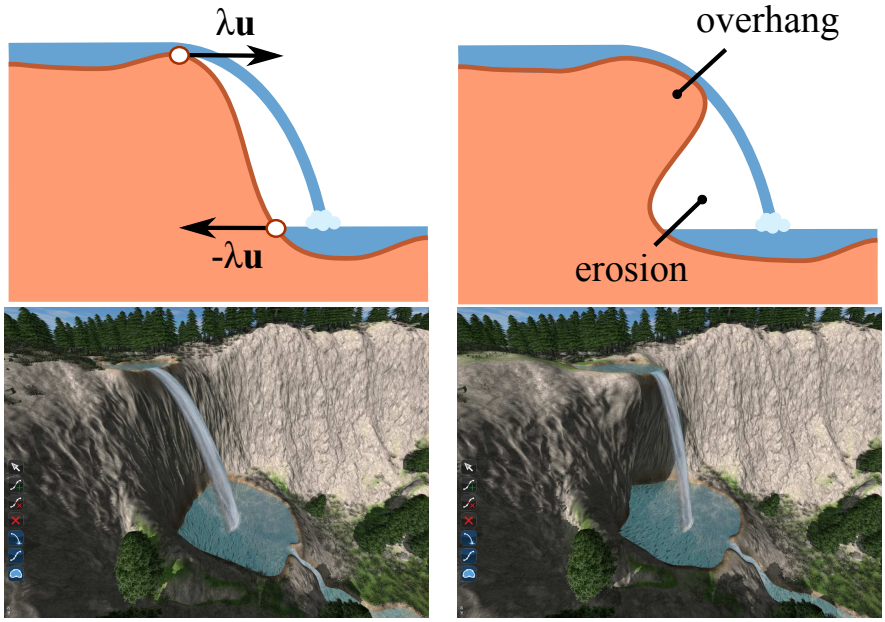


Figure 4.12: Top: Horizontal constraints modeling overhangs. Bottom: Free-fall without and with an overhang.

4.4.3 Procedural Decoration

To improve the integration of the waterfall into the terrain, we use several procedural decoration maps, generated using the footprint of the integration mesh and the waterfall elements type (Figures 4.13, 4.15). A decoration map is computed by rendering the integration mesh viewed from above into a texture, similar to the road footprints of Bruneton and Neyret [BN08]. During this process, we render each sub-mesh in a greyscale map, depending on its type and on the map that is being computed.

Terrain Decorations. Terrain decorations are generated using the water map, which corresponds to the integration mesh footprint. This map is used to mask the procedural seeding of trees and plants to prevent a generation within the water surface, and also to change the terrain texture to, for instance, a bedrock one.

Water Decorations. Table 4.1 lists a set of parameter values depending on the waterfall type. By using these values as greyscale values during the map computations, we generate a foam map, a rock map, and a disturbance map. The foam map identifies the presence of foam on the water surface and is used to select the water diffuse texture; the rock map indicates the density of rocks to generate; and the disturbance map, the amplitude of the waves on the water surface (Figure 4.13). The variation of values depending on the waterfall type allows increases of the visual difference between them, and improves the appearance of the scene. Note that some filtering is applied to these maps to reduce visible transitions between different types.

Speed Map. The speed map is a texture that represents the 2D speed of all water meshes in contact with the terrain in the scene. It is used for animating the textures of the water surface.

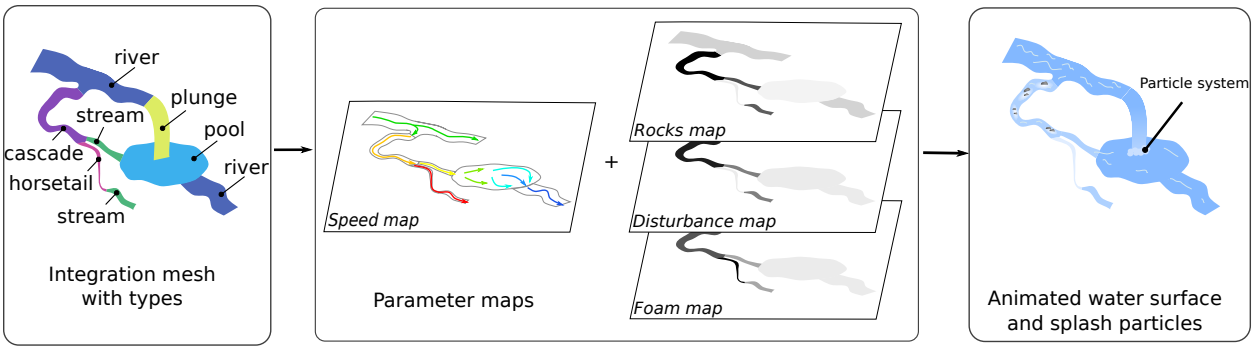


Figure 4.13: Using the integration mesh and the waterfall types, we generate various maps used to render the waterfalls.

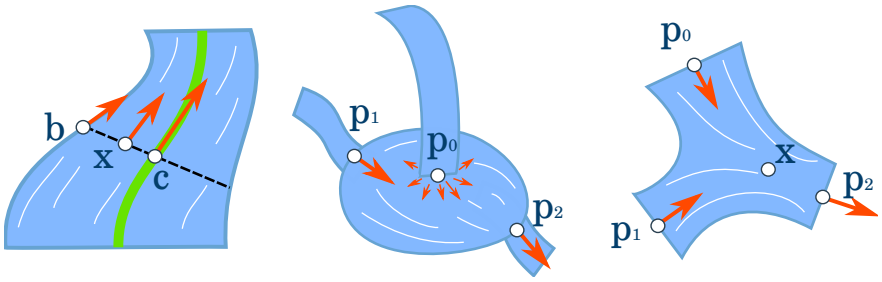


Figure 4.14: Internal speed computation for a contact (left), a pool (center), and a branch (right).

Figure 4.14 shows how the speed of the water is computed depending on the type of the segment. We use three different approaches to compute surface speed of *contacts*, *pools*, and *branches*. For a *contact*, the speed at point $\mathbf{x}$ is given by $\delta_{\mathbf{x}} = \delta_{\mathbf{c}} \left(1 - \frac{\|\mathbf{x}-\mathbf{c}\|}{\|\mathbf{b}-\mathbf{c}\|}\right)$ where $\mathbf{c}$ and $\mathbf{b}$ are the projections of $\mathbf{x}$ along the cross section on the main axis and on the shore respectively. For a *pool*, a fixed number of 2D fluid solver simulation steps [Sta99] are evaluated. For *branches*, we use 2D interpolation based on a standard technique of weighing by the inverse distance, where each point $\mathbf{p}_i$ is considered as a velocity constraint δ_i. The speed within a branch is given by $\delta_{\mathbf{x}} = \sum_i \omega_i \, \delta_i$. Interpolation weights ω_i are computed using the equation:

$$\omega_i = \frac{\tilde{\omega}_i}{\sum_i \tilde{\omega}_i} \qquad \tilde{\omega}_i = \prod_j \left(1 - \frac{\|\mathbf{x} - \mathbf{p}_i\|}{\|\mathbf{p}_j - \mathbf{p}_i\|}\right).$$

4.5 Results

The system is implemented in C++, using OpenGL and GLSL Compute Shaders. The computations are performed on an NVidia 660GTX GPU and an Intel® Xeon® E5-1650 CPU, running at 3.20 GHz with 16 GB of memory. The system uses two threads: one CPU thread for the interface and computation control, and one CPU/GPU thread for the GPU computations and rendering.

Rendering. The waterfalls in our editor are rendered in real time using the integration mesh and the parameter maps computed earlier (Figure 4.13). We use the technique of "tiled directional

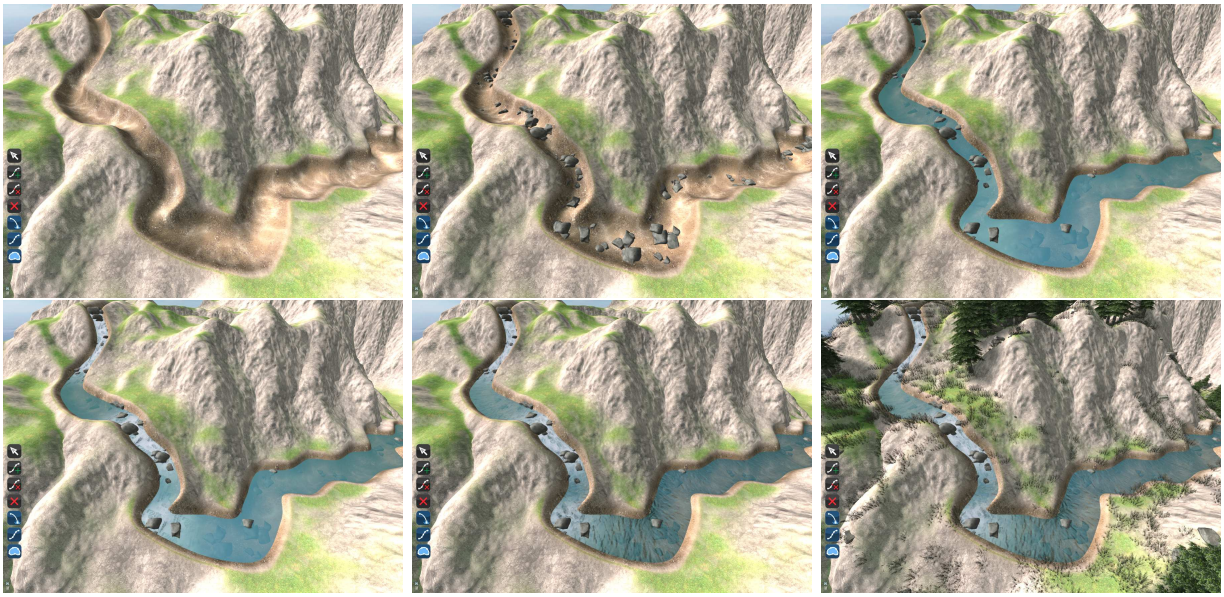


Figure 4.15: Incremental representation of procedural decorations. In usual order: Terrain only, adding rocks, water, foam, speed map, final result with vegetation.

Figure 4.16: An example of a waterfall scene, the *Iron hole*, on the island of La Réunion. Left: Photo of the real site © Serge Gélabert. Center: Our result after 10 minutes of interactive procedural modeling, starting from a similar terrain model. The scene contains 36 elements interconnected by pools and rivers, deforming the terrain and controlling the flow. Right: Visualization of the control elements that we used to create the waterfall network.

flow" [vH11] for the animation of both the normal texture of the waves and the diffuse texture of the foam. The splashes at the bottom of the falls are rendered using particles emitted from the free-fall ends.

Evaluation. Figures 4.19 and 4.18 show our system in action at different editing stages, with different stages described in the caption of each figure.

In Figure 4.16, we show a photo of a real waterfall network, and the result of a 10-minute session with our modeling system; we started with a terrain resembling the original real terrain, but without riverbeds. The figure shows that coherent waterfalls similar to those on the photo can be easily modeled, while guaranteeing their physical plausibility.

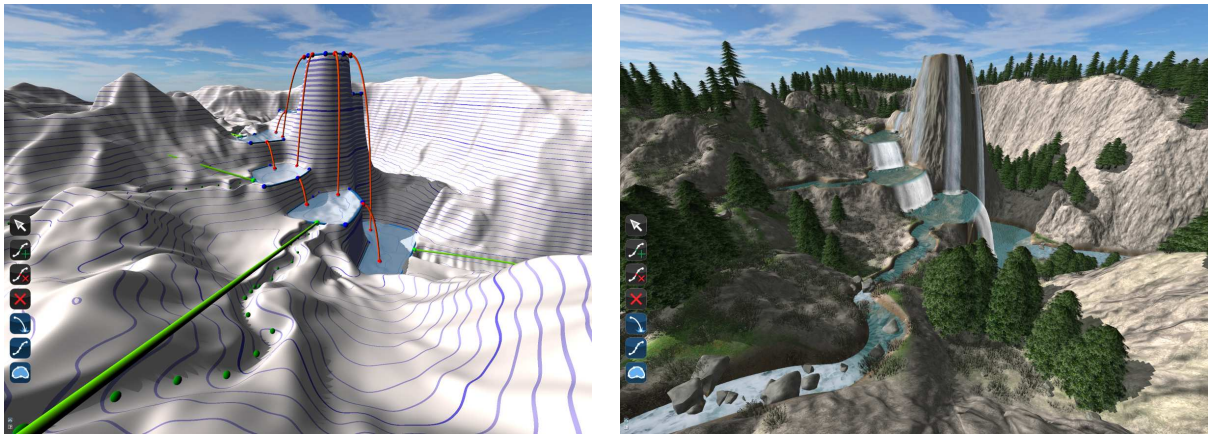


Figure 4.17: Waterfalls modeled by one of our digital artists.

We organized a user modeling session with two experienced digital artists. After a 20-minute training period, they were both able to create waterfall scenes such as the waterfall presented in Figure 4.17, all in under 30 minutes. The artists were pleased by the ease of use of our system and its efficiency. However, they expressed a desire for finer control over the result.

Performance. Our system generates a complex waterfall network over a terrain in a few seconds (see Table 5.1). The number of elements does not have a huge impact on computation time. Indeed, our algorithm uses mostly a fixed-size grid, so its complexity is independent from the number of waterfall elements. When increasing the number of elements, only the time for mesh generation, the riverbed constraints dense sampling (sub-part of the terrain deformation algorithm), and the internal speed computation of *pool* vary noticeably.

In order to ensure the consistency of our results we re-execute all computations each time an element is modified by the user. Many simple optimizations could detect what needs to be recomputed, and thus greatly improve the efficiency of our system; however, we felt that any such optimizations were not necessary in the current version of our prototype.

#		Computation times in ms						
Fig.	n	$\mathcal{G}$	$\mathcal{W}$	$\mathcal{M}$	Terrain	Speed	Maps	Proc.
4.16	36	2	2	112	758	258	428	284
4.19	17	1	1	177	677	384	404	297
4.18	29	1	1	77	871	264	494	324
4.17	26	1	2	30	1115	284	482	302

Table 4.2: From left to right, the columns of the table list the figure number and the number of waterfall controllers, followed by computation times for the hydraulic graph generation ($\mathcal{G}$), waterfall network generation ($\mathcal{W}$), mesh generation ($\mathcal{M}$), terrain adaptation using a 2048×2048 resolution, speed map generation, details map (foam, disturbance, and rocks) generation, and procedural detail generation.

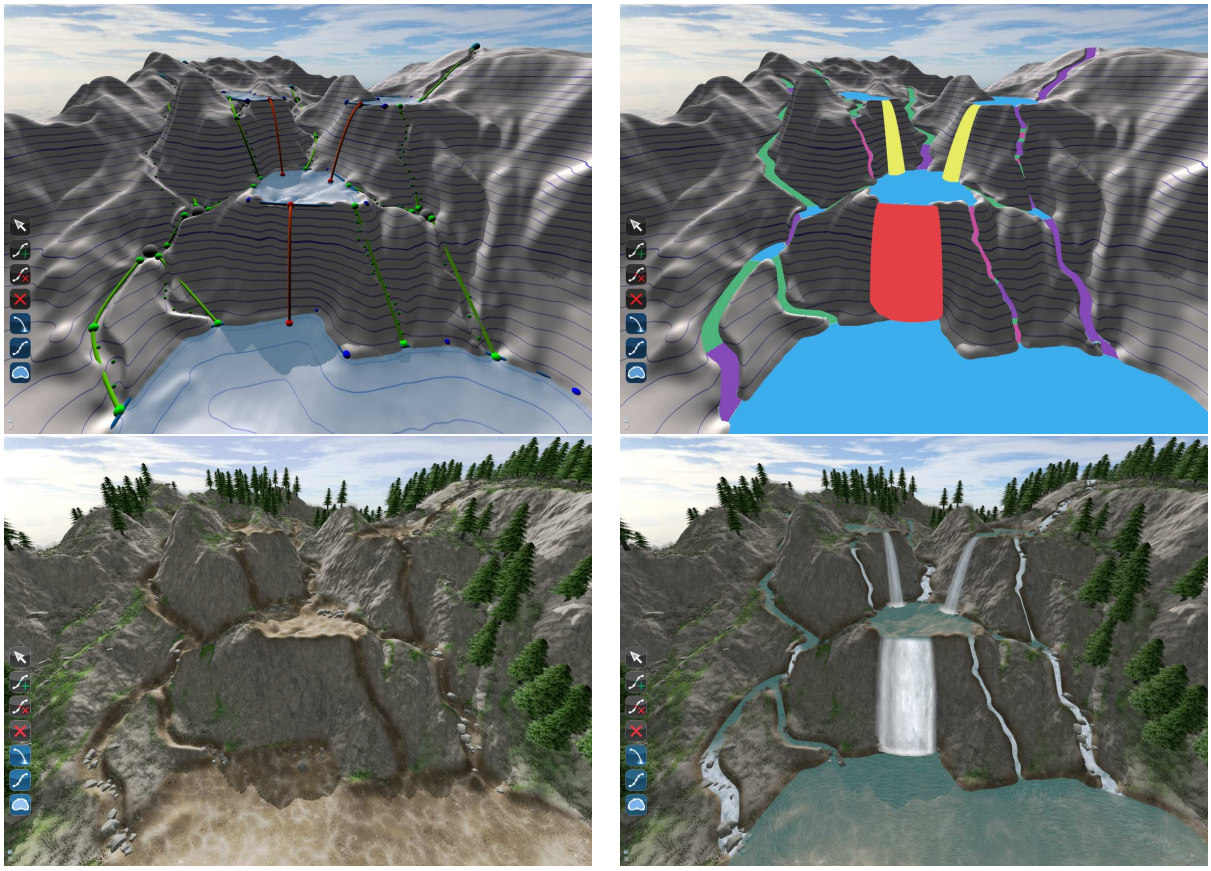


Figure 4.18: Another example of a waterfall scene. Top left: Controller network. Top right: Integration mesh with types. Bottom left: Deformed terrain. Bottom right: Final scene.

Limitations. The first limitation of our method is that the terrain is only adapted locally, and therefore does not preserve any global hydraulic properties. Indeed, a waterfall network can be created at an unplausible location on the terrain, failing to respect natural river paths shaped by the terrain slope, such as in the example of Figure 4.17. While this may lead to unplausible terrains, it also gives more artistic freedom to the user, which we feel is an important property of our system.

The heightfield representation of the terrain is another limitation, as it prevents the creation of caves and underground waterfalls, although horizontal displacement maps [GM01] enable us to create overhangs. With support for stack-based terrains [PGGM09], our system could handle more complex terrain elements.

In addition, our adaptation method relies on a grid-based algorithm, which limits its application to relatively small terrains. In our examples, we used a 2048×2048 grid with a resolution of 10 pixels per meter. We could adapt our method to multi-scale editing and rendering [YNBH09], by dividing the terrain into several tiles, each calculated independently [vBBK08], but we have not yet done so.

Moreover, the recursive nature of our algorithm limits the adaptation of paths in complex cases. Consequently, a good position selection at a given step does not imply a good position for the final

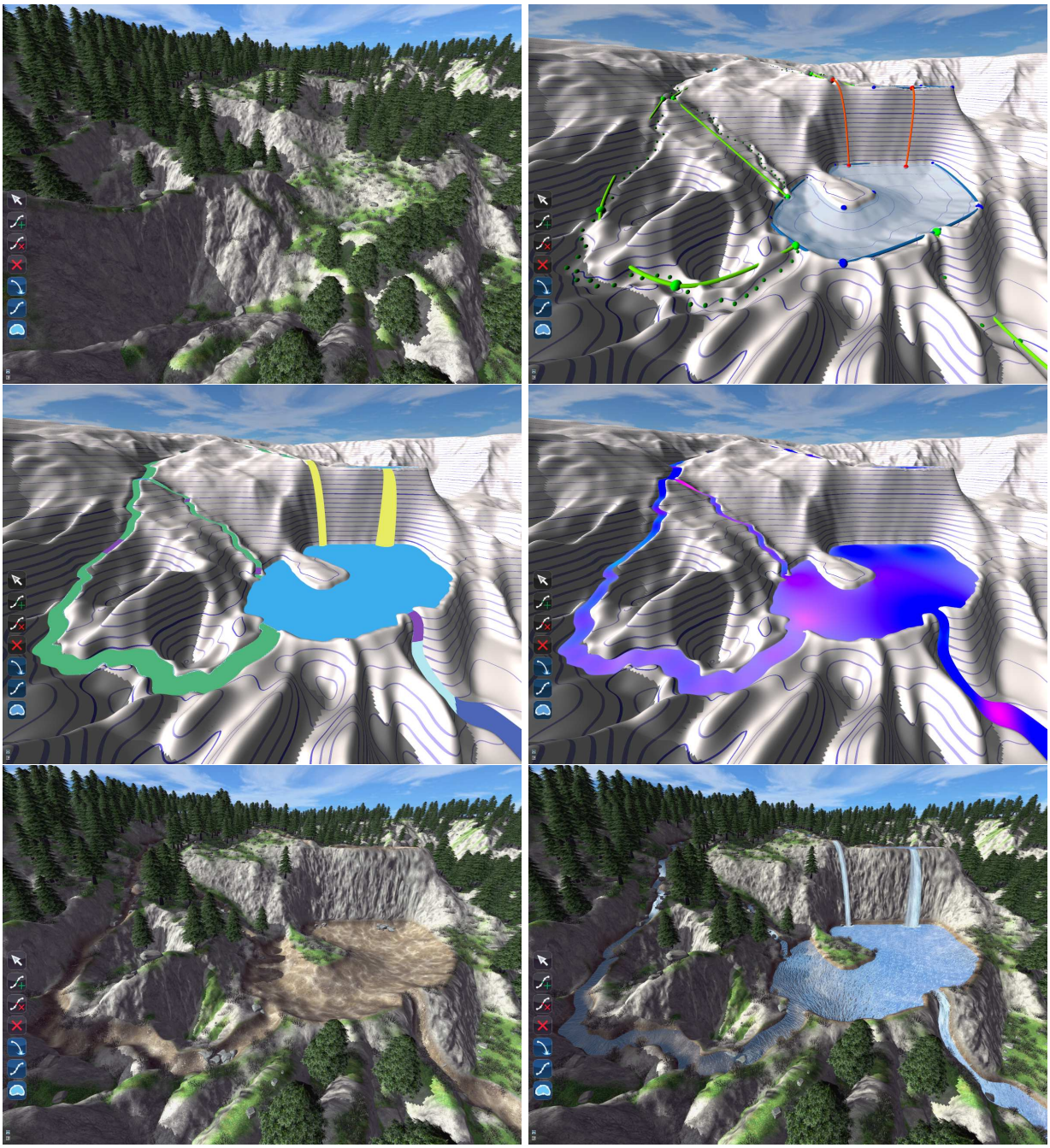


Figure 4.19: Our system in action. Top: Original terrain (left), creating the waterfall network (right). Middle: Generating the control mesh and deducing the types (left), resolving the speeds (right). Bottom: Adapted terrain (left), final scenery (right).

shape. For instance, if there are too many obstacles on the path, the heuristic will select a point to avoid them globally, but this selection may prevent further steps to avoid them. A solution inspired by algorithms for procedural roads [GPMG10], should be applicable to procedural river trajectories.

Finally, even if our algorithm supports interactive flow variations that change waterfall geometry and its adaptation to the terrain, we do not support a flow variation without changing the terrain adaptation. For example a drying-out waterfall or a river flood cannot be modeled with the current version of our system.

4.6 Conclusion

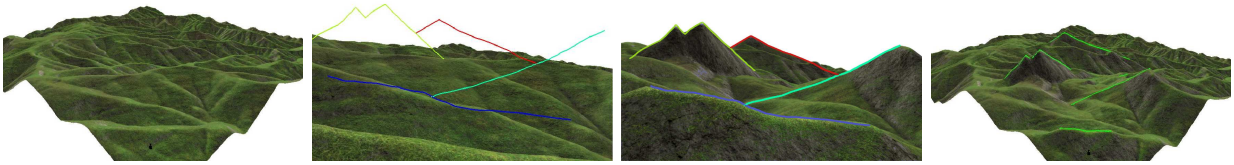
We presented the first interactive procedural modeling system for the design of waterfalls. Our system relies on a tight coupling of automatic generation and user interaction, where complex constraints and tedious tasks are handled by the procedural components of the system while enabling coarse to fine user control. This leads to an improved user experience and to the possibility for more creative modeling.

A number of aspects of our method are dedicated to the special case of waterfalls, including their categorization based on the slope-flow graph, and adapted tools for their interactive modeling. In fact, we are the first to present a system for the interactive design of coherent waterfalls. However, the methodology for the design of our system, with notably the way we interleave high-level user control with automatic processes to check consistency and add details, could be generalized to the modeling of many other natural sceneries, as well as to even less natural complex models. Future work could focus on the preservation of the hydraulic properties of the terrain during its deformation, or tackle the development of volumetric algorithms allowing the creation of truly 3D waterfall networks.

In this chapter, we presented a waterfall modeling method combining procedural modeling and coarse interactive editing. More precisely, we proposed direct intuitive user controls to edit the coarse waterfall network, while automatic methods generate fine-scale details and ensure scene consistency. However, during modeling sessions, we often felt the need of editing the terrain to better match our desired scenery. Moreover, while the method provides direct controls of the coarse network, the user cannot edit the detailed scene, making the design of a particular result difficult. While many methods have been introduced to generate terrains, few of them focus on the editing of existing terrains. In Chapter 5, we investigate the interactive editing of terrains from a first-person viewpoint, relying on automatic methods to deform it in order to match user constraints while preserving its consistency.

Chapter 5

Sketch-based Terrain Editing



Contents

This chapter details a collaboration with Flora Ponjou Tasse, Ph.D. student at Cambridge University during her visit to INRIA Grenoble. It was presented at the conference *Graphics Interface* [TEC*14a]. It was selected and an extended version was published in journal *Computer & Graphics* [TEC*14b].

Terrain is a key element in any outdoor environment, and terrain modelling has been the subject of a large amount of studies over the years. As presented in Chapter 2, the two most popular terrain modelling methods are procedural methods [FFC82, Mil86, Lew87, MKM89] and physics-based methods [MKM89, RPP93, CMF98, Nag97, NWD05, KBKŠ09]. The former are easy to implement and fast to compute, while the latter produce more realistic terrains with erosion effects and geologically sound features. However, the lack of controllability in these methods is a limitation for artists.

Sketch-based or example-based terrains have been very popular recently in addressing these issues [CHZ00, WI04, ZSTR07, GMS09, HGA*10, TGM12, GGG*13]. However, many of these methods assume that the user sketch is drawn from a top view, which makes shape control from a given viewpoint very difficult. Others only handle a restricted category of mountains, with flat silhouettes. Lastly, terrains fully generated from sketches typically lack the complex details that are typical in natural mountains. Dos Passos et al. [dPI13] recently presented a promising approach where example-based terrain modelling and first-person viewpoint sketching are combined.

Figure 5.1: Left: A photography of a real mountain scenery. Right: A sketch of mountain silhouettes.

First-person viewpoint sketching is a convenient way to create terrains, since the artist can edit the terrain from his viewpoint of interest, by sketching a small set of curves representing the main silhouettes of mountains. Figure 5.1 shows a real terrain and a corresponding silhouette sketch, drawn by an artist. The goal of our method is to propose a new method for editing terrain models as detailed as in the photo on the left, through the sketching of desired silhouettes, as shown in the image on the right.

In this work, we address the problem of intuitive shape control of a terrain from a first-person viewpoint, while generating a detailed output, plausible from anywhere (see Figure 5.7). Contrary to most existing techniques, we will not generate a new terrain from scratch, but instead deform an existing terrain to match user sketches. This approach can retain coherent small details from the existing terrain, while avoiding patch blending and repetition problems that are typical of example-based methods. The use of an existing terrain also enables matching sketched silhouettes with plausible, non-planar curves on the terrain. Our main contributions are as follows:

- we propose an algorithm for ordering sketch strokes with respect to their distance from the camera (Section 5.2);
- we present a method for matching terrain features with user-specified silhouettes, drawn from a given first-person viewpoint (Section 5.3);
- we introduce a deformation method for matching silhouette constraints while preventing them from being hidden by other parts of the terrain (Section 5.4).

Note that this work has been a collaboration with Flora Ponjou Tasse, a Ph.D. student from Cambridge University who visited our group from September to November 2013. My personal contributions to this chapter are mainly described in Sections 5.3.2 and 5.3.3. The rest of the chapter mostly describes Flora's contribution, although I contributed by participating to brainstorming meetings. These contributions are included in this document to allow for the understanding of the method as a whole.

5.1 Overview

Using our terrain editing method, users are able to navigate on the existing terrain with a first-person camera, and sketch one or multiple strokes, from the same camera position, that represent silhouettes that would be visible from that position. Our main goal is to deform the terrain such that these sketched user constraints are respected. To do so, the following requirements should be satisfied:

- Every sketched stroke should be a terrain silhouette, in the current perspective view from the first-person camera viewpoint.
- Each of these terrain silhouettes should be visible, i.e., not hidden by any other part of the terrain.
- The deformed terrain should not have artifacts nor contain unrealistic deformations, from any other viewpoint.

Our solution consists of five main steps, illustrated in Figure 5.2:

1. We order strokes according to their depth, from front to back with respect to the camera position. This order is used when we generate constraints for terrain deformation, so that a curve constraint is not occluded by another, when viewed from the first-person viewpoint.
2. Terrain features such as silhouettes and ridges are detected. Deforming existing terrain features to match the desired silhouettes results in a more realistic terrain since no extra feature is added and thus, the nature of the existing terrain is best preserved.
3. For each stroke, we select a terrain feature that will be deformed to fit the stroke, when viewed from the camera position. These deformed features represent the positional constraints that we use with a diffusion-based terrain deformation. A key idea of our framework is the expression of this feature-selection step as an energy minimization problem, in which we

penalize features with large altitude differences compared to their corresponding strokes, as well as features that would result in too large deformations.

4. We use a multi-grid Poisson solver for diffusion-based terrain deformation. It solves for altitude differences instead of absolute terrain positions, thus preserving the small-scale features of the input terrain.

5. After terrain deformation, other parts of the terrain may hide the user-specified silhouettes. To address this issue, we run the following iterative process: we detect terrain silhouettes that do not fit any user stroke and yet hide one of the sketched silhouettes. Extra deformation constraints are constructed to enforce lowering these protruding silhouettes until the user-sketched silhouettes are no longer occluded. The terrain is deformed with a combination of previous constraints and the newly constructed constraints. We repeat this process until all the unwanted protruding silhouettes have been eliminated.

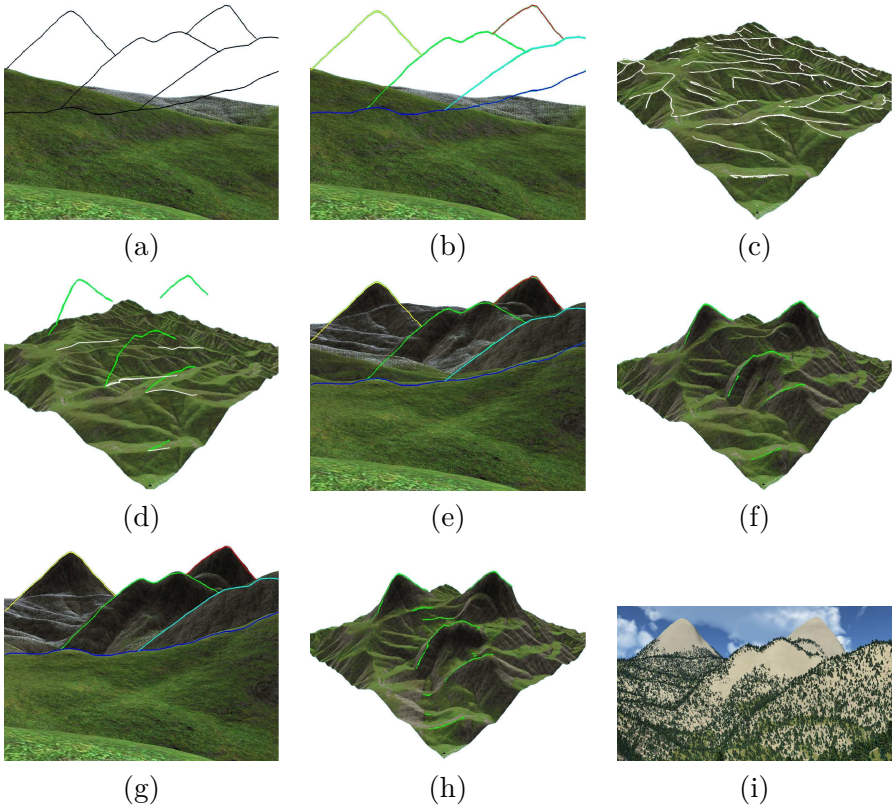


Figure 5.2: Overview of our terrain editing framework. (a) User 2D sketch, in a 3D interface. (b) 1. The stroke color indicates the automatically computed stroke ordering: blue indicates a stroke closer to the camera position and red, farther. (c) 2. The white curves indicate all terrain features that have been detected. (d) 3. These white curves indicate the detected terrain features that have been matched with user strokes. (e, f) 4. The terrain features are deformed so that they match the strokes from the user's viewpoint. (g, h) 5. The protruding silhouettes are lowered. (i) Deformed terrain.

5.2 Sketch Analysis

In this section, we explain how depth ordering of silhouette strokes is extracted from user sketches.

Original Observations. The different silhouette strokes in the input sketch first need to be ordered, in terms of relative depth from the camera viewpoint. This will enable us to ensure, when they are matched with features, that they will not be hidden by other parts of the terrain. Our approach to do so is based on two observations:

- If in the XY viewing plane, a silhouette lies above another in Y, it obviously corresponds to a mountain A farther away from the viewpoint than the other mountain B. Otherwise A would hide B. Using 2D height coverage for ordering them in depth is however not sufficient, since some strokes may overlap in height, as with the green and blue strokes in Figure 5.3.
- Furthermore, the terrain being a heightfield, the projection of each stroke onto the horizon (x-axis of the viewing plane) is injective (no more than one height value per point).

These two observations allow us to solve the relative stroke ordering problem thanks to a new sweeping algorithm (Figure 5.3) presented below.

Sweeping Algorithm. We consider the projections of all the strokes onto the horizontal x axis (depicted in the bottom part of Figure 5.3) and sweep from left to right, examining the extremities (starting and ending points in the sweeping direction) and junction points of the silhouette strokes. While doing so, we label the strokes' extremities and the junction points in the following way: an extremity q_s of stroke s is a T-junction if its closest distance to another stroke r is smaller than a given threshold. An endpoint q_s is labeled "occluded-by" r if the oriented angle, measured counterclockwise, is between the tangent t_s[1] of s at q_s and the tangent t_r of r at q_s, $\angle(t_s, t_r) < \pi$. This indicates that s is occluded by, and thus is behind, r. Otherwise, s is in front of r and we label q_s as "in-front-of" r.

If a stroke s has no T-junction, then it is behind a stroke r either if the projection of s completely contains the projection of r or if the smallest height value of s's endpoints is larger than the smallest height value of r's endpoints.

While scanning the sketch from left to right, we insert each stroke in a sorting structure, at a relative depth position determined by the cues above. This results in a relative ordering of the user strokes.

5.3 Feature-based Terrain Deformation

The key idea of our approach is to create a 3D terrain that matches the user sketch, by deforming an existing terrain model. More precisely, we deform the features of the existing terrain, such as its ridgelines, to match the user silhouette strokes. Because a terrain could exhibit many features, we first have to compute to which one of them it is the most appropriate to apply a deformation. In this section, we detail how we compute the set of terrain features (Section 5.3.1), how we assign one of them to each user stroke (Section 5.3.2), and how we use them to deform the terrain (Section 5.3.3).

[1]Strokes are always oriented clockwise. Hence, stroke tangents are independent of the direction in which the stroke was sketched. When labeling a starting point q_s as T-junction, we flip its tangent.

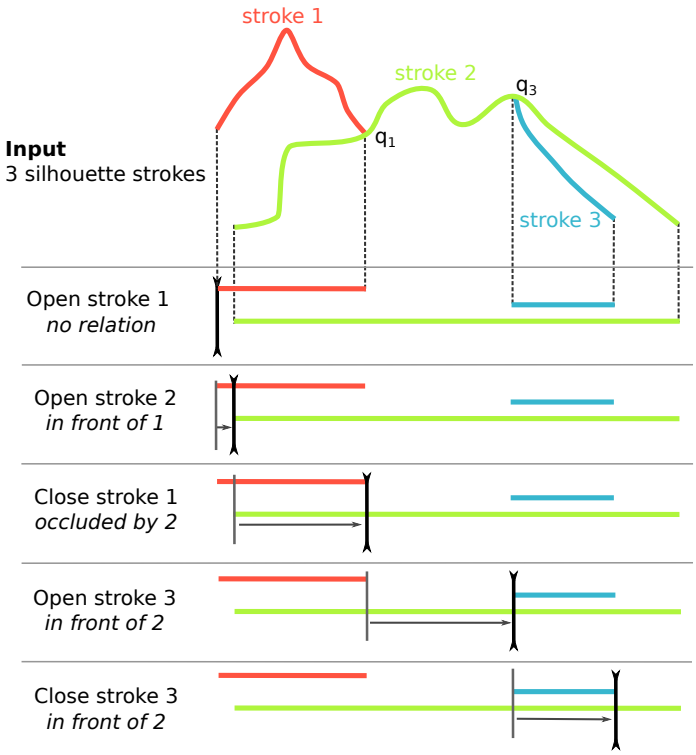


Figure 5.3: An input sketch (top) and the different steps of the sweeping algorithm used for scanning the sketch, labelling T-junctions and ordering strokes (bottom). As a result, stroke 3 is detected to be in front of stroke 2, which is itself in front of stroke 1.

5.3.1 Feature Detection

When we deform a terrain, we consider two types of features: terrain silhouettes (viewed from the first-person viewpoint) and terrain ridges.

Silhouette Detection. Silhouette detection on an existing terrain is based on a common and naive algorithm for computing the exact silhouettes of a 3D mesh. Silhouette edges are detected by finding all visible edges shared by a front face and a back face in the current perspective view. Neighboring silhouette edges are then linked to form long silhouette curves.

Ridge Detection. Ridge detection is based on the profile-recognition and polygon-breaking algorithm (PPA) of Chang et al. [CS07]. The PPA algorithm marks each terrain point that is likely to be on a ridge-line, based on the point height profile. Segments forming a cyclic graph, connect adjacent candidate points. Polygon-breaking repeatedly deletes the lowest segment in a cycle until the graph is acyclic. Finally, the branches on the produced tree structure are reduced and smoothed. The result is a graph where nodes are endpoints or branching points connected by curvilinear ridgelines. An improvement of the PPA algorithm connects all terrain points into a graph using a height-based or curvature-based weighting and computes the minimum spanning tree of that graph [BdBG10]. Because we are mainly concerned with performance and detection of

large-scale ridges, we simply connect candidate points as in the original PPA algorithm and replace the polygon-breaking with a minimum spanning forest algorithm.

5.3.2 Stroke-Feature Matching

In this section, we discuss a method for determining for each stroke, its associated terrain feature, which can be used to construct deformation constraints. Viewed from the first-person camera, these constraints should match the user-sketched strokes. To achieve this, we first construct a priority list of features for each stroke and then select features for each priority list such that the sum of their associated cost is minimized.

Feature Priority List Per Stroke. For a stroke s, we project all terrain features on the sketching plane (i.e., we use the 2D projection of the feature from the first-person viewpoint), and select feature curves that satisfy the following condition: the x interval that they cover matches the one of stroke s. We then deform the selected feature curves, and if necessary extend their endpoints, such that viewed from the camera position, they cover the length of s. This deformation f' is simply achieved by displacing the feature curve points according to their projection on the 2D stroke in the sketching plane, and their distance to the camera position. Let f be a terrain feature and f_p its projection on the stroke plane. Let f' be the terrain feature deformed so that its projection f'_p on the stroke plane matches stroke s. For each point $q' \in f'$, its altitude is computed as follows:

$$q'.z = q.z + k\|q_p - q_p^s\| \frac{\|q - eye\|}{\|q_p - eye\|}$$

where q is the original point in f, eye is the camera position, $k = -1$ if f_p is below s and $k = 1$ otherwise, q_p is the projection of q on the stroke plane, and q_p^s is the intersection of s with the vertical line passing at q_p.

We associate a cost $E(f, s)$ to each feature f with respect to stroke s:

$$E(f,s) = w_{dis}E_{dis}(f,s) + w_{def}E_{def}(f,s) + w_eE_e(f,s)$$

$$\begin{cases} E_{dis}(f,s) &= \frac{1}{l(f_p)} \int_{f_p} (f_p(t).z - s(t).z)dt \\ E_{def}(f,s) &= \frac{1}{l(f')} \int_{f'} (f'(t).z - h(t))dt \\ E_e(f,s) &= \frac{l(f')}{l(f)} \end{cases}$$

where w_{dis}, w_{def}, and w_e are weights, l is the length of its specified curve, e the length of its longest edge, and h the terrain altitude:

- E_{dis} represents the dissimilarity between f and s from the first-person viewpoint;
- E_{def} expresses the amount of terrain deformation that the feature deformation will cause;
- E_e penalizes features that were extended to fully cover s when viewed from the camera position.

All the results shown in this chapter were generated with $w_{dis} = w_{def} = w_e = 1$. All features are sorted in a priority list according to their cost. Figure 5.4 illustrates this process for a single stroke (in this simple case, the feature of minimal cost is selected).

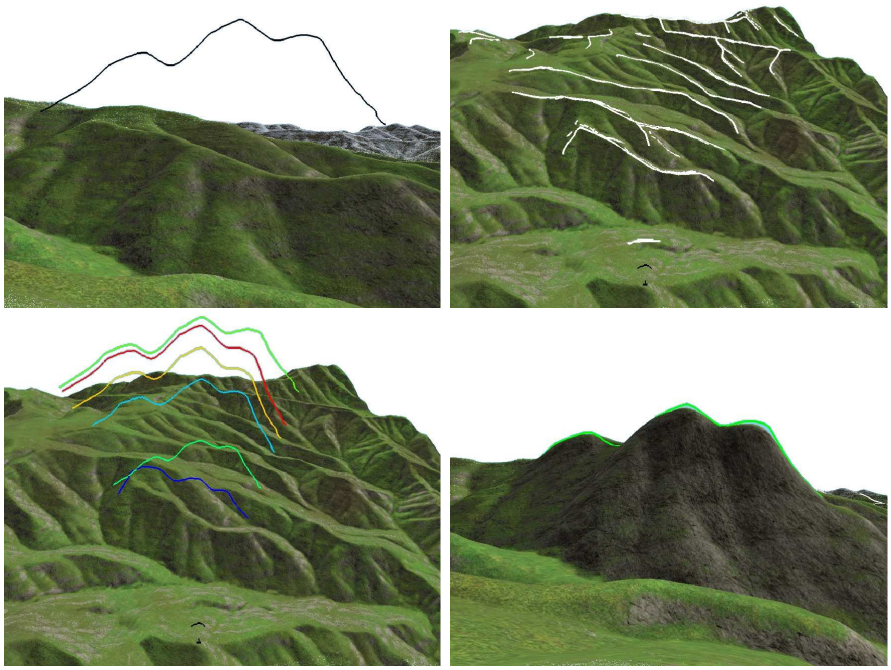


Figure 5.4: Computing possible features to match with a user stroke. Top: User sketch from a first-person viewpoint (left), feature detection from a higher viewpoint (right). Bottom: Possible candidate matches (left), terrain deformation using the best match (right). Feature color indicates cost: blue for the lowest cost and red for the largest.

Energy Minimization. To deform the terrain to match the user sketch, we want to assign a terrain feature to each stroke, according to its priority list, and create the associated deformation constraint. In addition to the feature order within the different stroke priority list, we need to take into account depth ordering for silhouette strokes computed in Section 5.2. Therefore, this selection process can be seen as a minimization problem. We want to find a set of stroke-feature matches such that the total cost of the assignments is minimized, and that the assigned features respect the pre-computed stroke ordering.

Let $S = \{s_i : i = 1, ..., n\}$ be the stroke list (ordered by depth) and f^i denote a feature in the priority list $L(s_i) = \{f^i_k : k = 1, ..., m_i\}$ for a stroke s_i. We are looking for $\{f^i : i \in 1, ..., n\}$ such that $f^i < f^j$ if $i < j$ and $\sum E(f^i, s_i)$ is minimized. Here, $f^i < f^j$ means that f^i should not be occluded by f^j, so that all deformation curve constraints are visible from the first-person viewpoint. We process the ordered stroke list from front to back, and after each stroke, we remove from the priority list of the next strokes, features that would be occluded if selected. We chose to process strokes from front to back for two main reasons. Firstly, strokes that are closest to the camera are processed first and due to E_{def}, the algorithm attempts to select constraints that will minimize terrain deformations. Thus, features closer to the camera are more likely to be selected. Secondly, if all the features of interest for a given stroke s_i were already selected, and therefore its priority list was empty, an arbitrary curve on the terrain would be used instead. If this ever occurs, we prefer it to be for background silhouettes.

In practice, feature selections that cause any stroke to have an empty priority list are penalized

with a very high cost. Thus, a configuration that guarantees at least one valid feature match for each stroke, is always selected, if it exists. If no such configuration exists and s_i has an empty priority list, we automatically compute a 3D embedding of the 2D stroke s_i and use the resulting curve as a deformation constraint. To easily compute this 3D embedding, we take the two strokes lying just in front and just behind s_i. Then we place s_i halfway between the terrain features assigned to these two strokes. If there is no stroke restricted to lie behind s_i, we place it behind the farthest stroke from the viewpoint. If there is no stroke restricted to lie in front of s_i, we place it in front of the closest stroke to the viewpoint. With this approach, each stroke is represented by a deformation constraint even if it was not matched to a terrain feature during energy minimization.

The energy minimization problem we have described so far is a NP-hard combinational optimization problem. Branch-and-bound approaches are often used to overcome such computationally expensive exhaustive searches [Cla97], since they are designed to discard non-optimal solutions early on. Here, we use the branch-and-bound scheme to efficiently discard all partial solutions that have a cost higher than the current best cost, without having to explore the whole solution tree. The scheme consists of two steps: a *branching* step and a *bounding* step. The branching step consists of exploring possible choices for s_{i+1} once we have made a feature selection for s_i. In other words, we split node (s_i, f^i) into multiple nodes (s_{i+1}, f_k^{i+1}), where f_k^{i+1} are features in the priority list of s_{i+1}. The bounding step allows the algorithm to stop exploring a partial solution if the total cost of features in the solution is higher than the cost of the best solution found so far.

It is possible for a feature to be the first choice in the priority list for two or more strokes. To handle this, when exploring a possible solution, a feature curve assigned to a stroke is no longer considered for subsequent strokes. Our branch-and-bound algorithm will explore other solutions with the feature curve assigned to different strokes as long these solutions are guaranteed to have a smaller cost than the current best solution.

Stroke in World Space. The previous minimization gives us, for each stroke s, an associated terrain feature f. However, stroke s has its points in screen space, whereas the points of f are in world space. We use f', the deformation of f whose projection matches s, as the nonplanar deformation curve that will be used to deform the terrain.

The possible undetermined point depths, at the stroke extremities, are set to follow the stroke tangent in world space.

Completing Selected 3D Features. Using user-specified endpoints of an occluded stroke during the generation of deformation constraints would create silhouettes that appear to start exactly at these endpoints. This can look quite unnatural when viewed from a different position than the first-person camera position used for sketching: indeed, the endpoint of the occluded stroke (a junction) is typically above the terrain and thus, a sharp deformation will be created at that point.

We address this problem by simply extending 3D features assigned to strokes at both endpoints along their tangents, until they reach the surface of the terrain. An example of this feature completion process is presented in Figure 5.5. More sophisticated contour completion methods, such as the one presented in *SmoothSketch* [KH06] could alternatively be used, but this simple method was sufficient in our case, and is proposed as an optional step in the editing process.

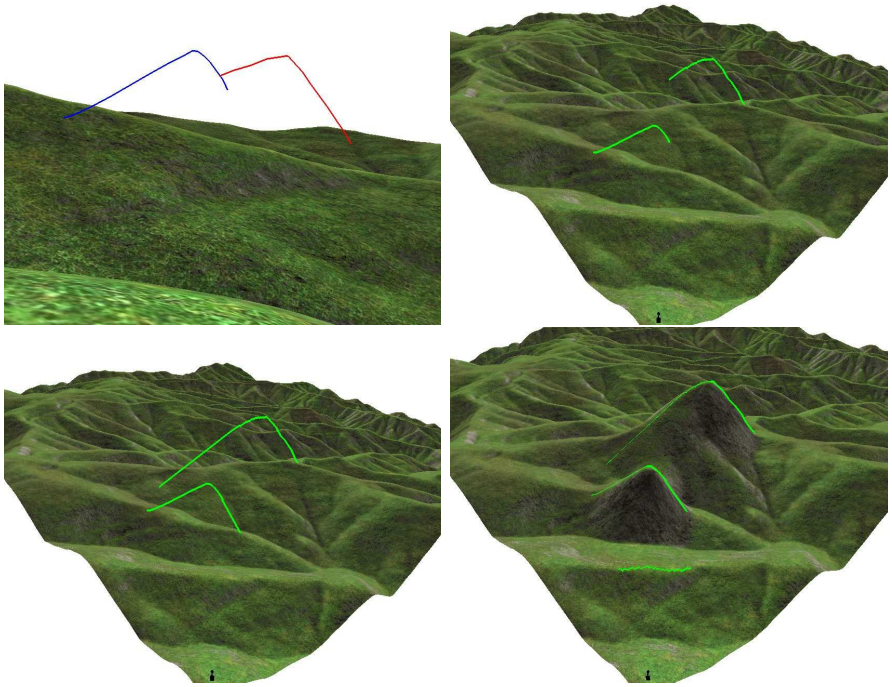


Figure 5.5: Completing selected features: After matching 2D strokes to terrain features, we extend these features until they reach the surface of the terrain, to ensure a smooth transition from specified silhouettes to the terrain. Top: User input (left), matched features (right). Bottom: Extension of the matched features (left), deformed terrain (right).

5.3.3 Terrain Deformation

Our deformation algorithm relies on iterative diffusion of displacement constraints, which are computed from 3D strokes positioned in world space. The diffusion method, first introduced in Section 4.4.2, consists in computing the difference of the curve height and terrain height $\mathcal{H}$, and to diffuse these differences (instead of absolute height values) using a multi-grid Poisson solver similar to the one developed by Hnaidi et al. [HGA*10].

More precisely, for each point $p = (x, y, z)$ of the stroke in world space, we compute $\delta = z - \mathcal{H}(x, y)$, and set it as a displacement constraint. The constraints are rasterized on a grid, whose resolution is equal to the terrain resolution. After having set the constraints of all strokes, we perform the diffusion, which gives the displacement map $\mathcal{M}$.

The displacement is finally applied to the terrain heightfield $\mathcal{H}$, whose feature line silhouettes are now matching the user strokes, when seen from the first-person viewpoint used for sketching. The deformation only consists of adding the two heights, $\mathcal{H}'(x, y) = \mathcal{H}(x, y) + \mathcal{M}(x, y)$, where $\mathcal{H}'$ is the resulting terrain. Because height differences are propagated, instead of absolute heights, the terrain preserves fine-scale details during deformation.

5.4 Lowering Protruding Silhouettes

After deformation, the user-defined silhouettes may be hidden by other parts of the terrain. To address this issue, we detect the unwanted protruding silhouettes and constrain them to a lower position so that the user-defined silhouettes become visible.

Detecting Protruding Silhouette Edges. First, all visible silhouettes are detected with the algorithm discussed in Section 5.3.1. These silhouettes are projected onto the sketching plane. Let s be a silhouette of the deformed landscape, inherited from the original terrain. Mountain silhouette s hides a user-specified silhouette g if s is closer to the camera than g and the projection s_p of s on the sketching plane has a higher altitude than g_p, the projection of g. In this case, s is an unwanted protruding silhouette. Determining how much s should be lowered is done as follows: let h be the maximum height difference between s and silhouette g hidden by s. h is the minimum altitude by which s should be lowered to ensure that the silhouettes it hides become visible. Our solution is simply to uniformly lower s by an offset h. This method is applied to all unwanted protruding silhouettes and we use the set of lowered silhouettes to form new deformation constraints.

Updating Deformation Constraints. The new deformation constraints from the lowered protruding silhouettes are added to the set of constraints associated to the sketched silhouettes, and the terrain is deformed once again using the method described in Section 5.3.3. This operation maintains the user-specified silhouettes while lowering areas around the unwanted protruding silhouettes, so that user specifications are satisfied.

The process of detecting protruding silhouettes and using this information to further constrain the terrain is repeated until no protruding silhouettes are detected. In practice, a single iteration is usually sufficient to make visible all user-specified silhouette strokes.

5.5 Results

Validation Examples. The examples in this section illustrate results of our method in a variety of cases. In particular, Figure 5.8 shows editing a terrain with a sketch containing five T-junctions. Our proposed approach differs from other sketch-based methods in that nonplanar silhouettes are generated from planar user-sketched strokes. This is illustrated in Figure 5.6.

Performance. The system is implemented in C++, and the computations are performed on an Intel® Xeon® E5-1650 CPU, running at 3.20 GHz with 16 GB of memory. We give computation times for the presented results in Table 5.1. The feature extraction and terrain deformation computation times only depend on terrain resolution, which is 512×512 in the examples. The feature matching depends on the number of strokes and on the number of extracted features. The most expensive part of the algorithm is the lowering silhouette one, because of our occlusion detection method. However, our naive algorithm could be optimized and computation times greatly reduced. The stroke-ordering algorithm has a negligible computation time. Considering manual input, sketching does not generally take more than a few seconds.

User-study. We performed an informal user-study with two experienced computer artists. The system was briefly introduced to the users, who had no prior knowledge about it. They were then

Fig.	Features	Matching	Deformation	Silhouettes
5.2	0.14	1.5	0.11	2.6
5.8	0.15	0.21	0.10	2.1
5.6	0.12	0.04	0.09	3.4
5.7	0.14	0.24	0.09	4.9

Table 5.1: Computation times (in seconds) of several examples for the presented algorithms: feature extraction, stroke-feature matching, terrain deformation, and lowering protruding silhouettes.

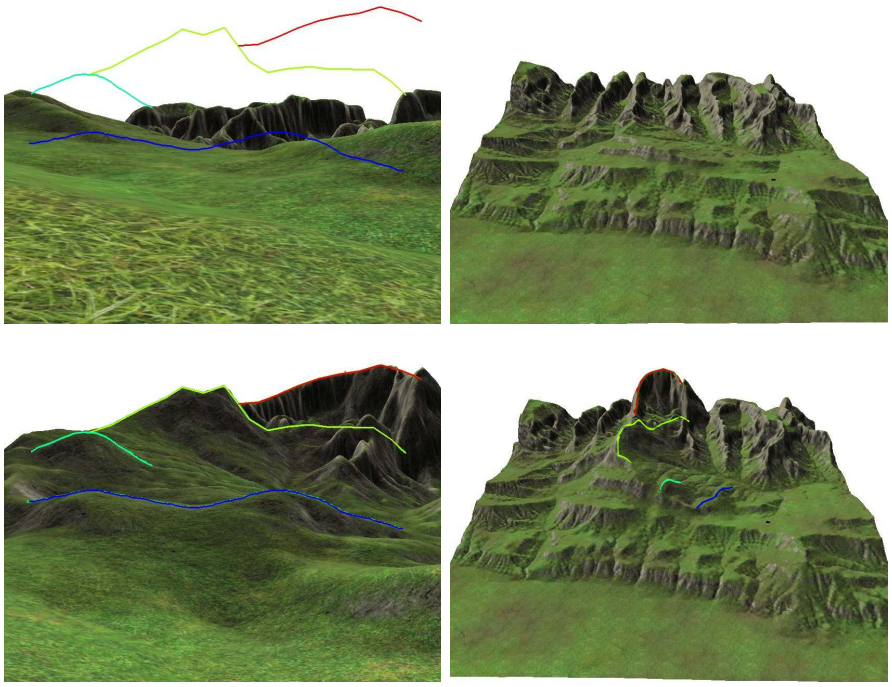


Figure 5.6: Terrain editing produces nonplanar silhouettes in the output, from 2D planar strokes. Top: User input (left), existing terrain (right). Bottom: Deformed terrain (left), result viewed from a different viewpoint (right).

asked to draw sketches to deform existing terrains. Both of them reported that our system is very intuitive to learn and use, and they were able to quickly create new sceneries. Their feedback indicates that the approach is original, and seems a promising way to create a scene that matches their artistic intent. However, they had some difficulties to predict results of their sketches. This could have been improved by manually tuning the feature matching weights, which was not among their skills.

Limitations. Although our system succeeds in matching user-sketches through a natural deformation of the terrain, using its existing features, the lack of predictability of the stroke-feature solver may be a problem. A more complete user-study would be useful to understand user intents when sketching over the existing terrain, and identify ranges of weight values allowing us to better match these intents. We could also improve our matching method using extra cost functions.

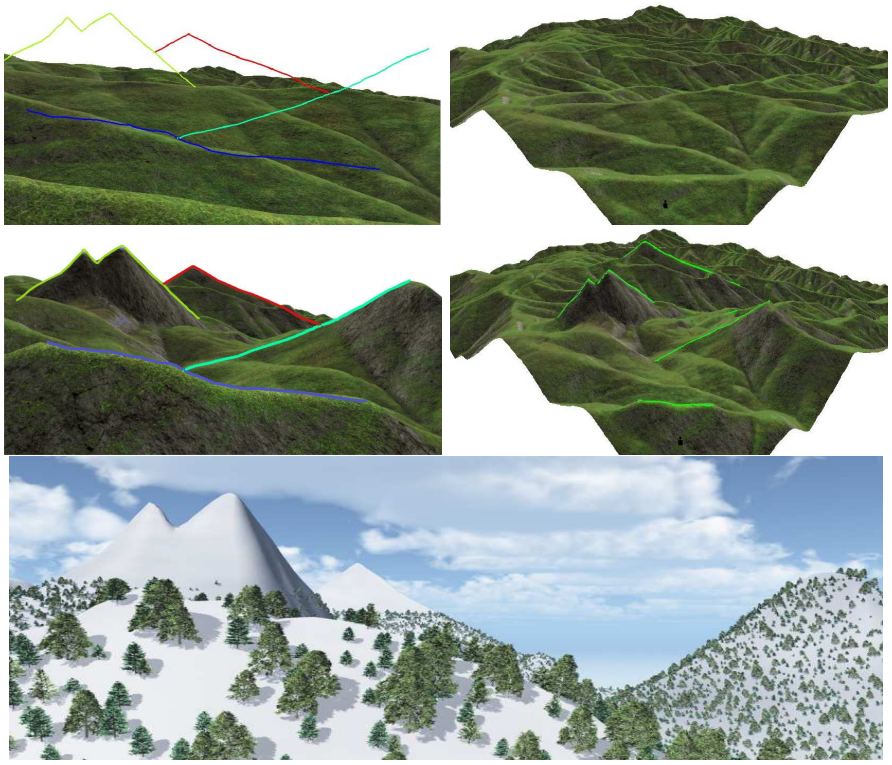


Figure 5.7: A typical artist sketch (top left) is used to edit an existing terrain (right). Results are shown in the second row from the same two viewpoints. Note the complex silhouettes with T-junctions, matched to features of the input terrain. The bottom image shows a rendering of the resulting terrain, from a closer viewpoint.

Another limitation comes from our deformation solver. The diffusion-based deformation method sometimes creates small declivities around the extremity of a constraint curve, when the slope of the curve is high and the extremity is located on the terrain: in this case, the terrain locally inflates, except at this endpoint, where the deformation is zero, which causes the problem. Using an inverse distance to deform the terrain [JJCH11] does not work either, because of our use of curves as constraints. Further work still needs to be done on terrain deformation, especially for curve-based deformations.

5.6 Conclusion

In this chapter, we presented the first sketch-based modeling method enabling the deformation of a terrain from a first-person viewpoint. The user sketches a few silhouette strokes forming a graph with T-junctions, similar to silhouette representations used in artistic terrain sketchings. The key feature of our method is that sketched silhouettes are matched with existing terrain features: this enables our technique to both match silhouette strokes with non-planar curves, and produce a deformation that does not spoil plausibility, since the structure of ridges and valleys typically remains unchanged.

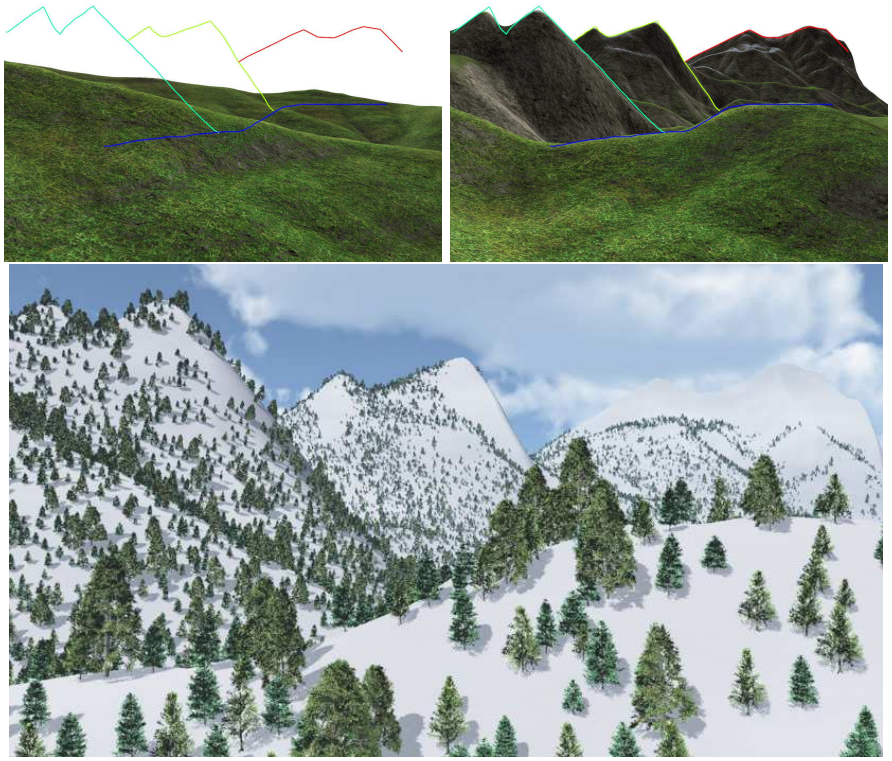


Figure 5.8: Terrain editing with user sketches. Top: User input (left), deformed terrain (right). Bottom: Final scene.

Being able to edit complex sceneries is a key issue in interactive modeling. We explored in this chapter an interactive method relying on a procedural algorithm to edit complex terrains while remaining easy to use. More precisely, the method combines fine intuitive editing of the terrain silhouette with procedural algorithms preserving the terrain consistency. In the next chapter, we extend the approach of procedural deformation of complex sceneries to the case of vectorial virtual world editing. Indeed, we propose an interactive painting method to analyze and edit object distributions and structured objects over an existing terrain, while using a procedural algorithm to automatically generate similar distributions, preserving scene consistency, and following analogies with traditional painting systems.

Chapter 6

Worldbrush: Painting and Deforming Virtual Worlds using Example-based Synthesis

Contents

In this chapter we present our last contributions, which have not been submitted for publication yet.

Virtual worlds are more and more expensive to produce, as their size and expected details keep increasing. Using conventional interactive world design software, they are usually created by manually placing each object individually. This process is long and tedious, especially when the user has to manipulate a large quantity of objects, such as trees composing a forest, or complex structured objects, such as houses and street segments to compose villages or cities. Moreover after each editing operation, the user must ensure scene consistency, breaking artistic flow.

Procedural methods have been heavily used in the last decade to create large quantities of objects in a small amount of time. They have been used successfully for individual plants, plant ecosystems, roads, cities, buildings, villages, etc. However, despite research focused on interactive procedural modeling, they remain reserved to experienced "programmer artists". Moreover, even if several methods exist for each individual type of object, only little research has investigated so far combining independently developed techniques into a simple framework [STdKB11]. Therefore, even with the help of procedural methods, the design of large scenes remains long and tedious, especially when creating complete virtual worlds involving different object types.

Our goal is to propose a new method for efficiently modeling complete and complex virtual worlds. The method should be highly interactive, thus enabling a good level of control, and rely on procedural generation techniques to offload parts of the long and tedious tasks. We build our system on the interaction metaphor of editing vectorial maps, such as the one in Figure 6.1. A 3D virtual world is then seamlessly generated from the map.

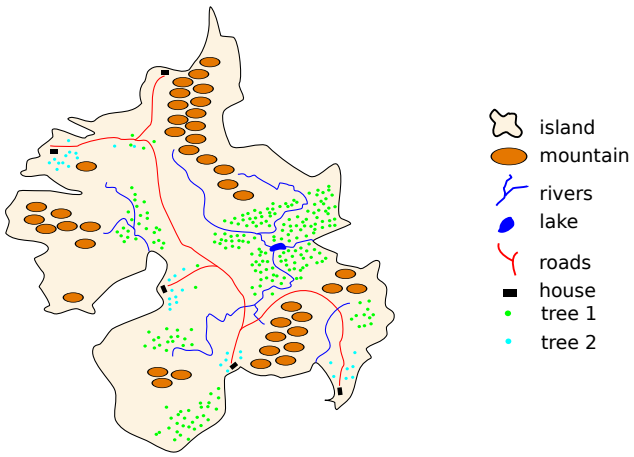


Figure 6.1: Example of a vectorial map representation of a virtual world. In this chapter, we focus on the interactive creation and editing of this kind of map, leaving the generation of the full 3D content to some automatic post-process.

In this chapter, we present our new approach for modeling virtual worlds, combining the controllability and ease of use of traditional 2D image editors with the power of example-based synthesis. More precisely, we present several tools inspired by those in 2D painting softwares in order to manipulate virtual scenes represented by point distributions and graphs. This choice is motivated by the fact that 2D painting is one of the most intuitive content creation approach. With our method, an artist can paint distributions of objects and graphs using special brushes whose new painted

content is extracted from example sceneries, and can edit scenes using adapted tools such as copy, paste, move, or scale. We also propose innovative tools based on interpolations and gradients. All our tools rely on example-based synthesis algorithms, which extract scene properties and enable the generation of new sceneries respecting these properties. For instance, a user first places manually some trees or houses in a small region, and then uses our tools to create a forest (resp. village) respecting the same tree (resp. house) distribution. Moving a forest or a village over a terrain will automatically deform it to adapt it to its new environment, thanks to the maintainability of inter-object relationships. Our brushes enable the painting of scenery elements on existing terrains.

Our work is mainly inspired by synthesis-based 2D painting tools [KIZD12], by advanced brushing techniques [LBW*14, MNB*14] where the modeling algorithms are based on traditional painting interfaces, and by the interactive editing of urban layouts using example-based synthesis [AVB08]. In a similar way, we propose advanced tools to paint and edit virtual worlds.

Our framework works as follows: the scene is represented by point distributions and graphs, which have a type, such as trees or roads, that the user can create in a traditional way by placing or sketching them by hand. When the user selects a region, the properties of elements in this region are analyzed. For instance, we analyze the radial distribution density of point distributions, or the number of neighbors of graph nodes for each type of element, as well as inter-relationships between several types. Then, when the user performs an action, we rely on an example-based synthesis method to generate new content, or deform the existing content while ensuring its consistency. Our main contributions are as follows:

- we propose a method for synthesizing vectorial maps of virtual worlds represented by point distributions and graphs based on the Metropolis-Hastings algorithm (Sections 6.2 and 6.3);
- we present synthesis-based tools for editing virtual scenes while preserving their properties, such as copy, paste, move, scale, and gradient (Section 6.4);
- we introduce synthesis-based brushes to paint virtual worlds (Section 6.5).

All these contributions are implemented in an interactive framework enabling the interactive creation and editing of virtual worlds.

6.1 Overview

In this section, we first define notions used in the remaining of the chapter, and then present an overview of our technique.

Scene Objects. The scene is composed of a set of vectorial objects, each belonging to a specific category $\mathcal{C} \in \{$*ground, island, mountain, river, road, castle, house, farm, pine tree, red tree*$\}$. We represent all individual scene objects of the same category as point distributions, and all structured objects as graphs. External constraints (i.e., arbitrary values over the terrain) are handled with a special *ground* category that contains a set of maps, for instance, elevation, slope, or interest maps. Thus, objects are of data type $\mathcal{T} \in \{$*distribution, graph, external*$\}$, and are grouped in layers $\mathcal{L} \in \{$*terrain, water, settlement, vegetation*$\}$. Table 6.1 shows a few examples of object categories and their corresponding layers and types.

$\mathcal{L}$	*terrain*			*water*	*settlement*				*vegetation*	
$\mathcal{C}$	*ground*	*island*	*mountain*	*river*	*road*	*castle*	*house*	*farm*	*pine tree*	*red tree*
$\mathcal{T}$	*external*	*graph*	*dist.*	*graph*	*graph*	*dist.*	*dist.*	*dist.*	*dist.*	*dist.*

Table 6.1: Example of object types associated to their layer and data types.

Interaction Matrix. Our algorithm relies on an interaction matrix $\mathcal{M}$, provided by the user, that describes the pairwise interactions between object categories. The interaction type $\mathcal{I}$ depends on the two category types, and can be $\mathcal{I} \in \{map,\ rdf,\ poly,\ arc\}$, where *rdf* stands for a radial distribution function. To increase synthesis speed, we generate each category of objects as independent layers, in a priority order provided by the user: each new layer is influenced by the layers with a higher priority that have been already placed over the terrain, and influences the layers that come after it. Thus, the interaction matrix is triangular; an object of category C_a can only interact with objects of category C_b with $C_b \geq C_a$. Table 6.2 shows the interaction matrix used in our prototype. Note that a category can interact with itself. For instance, we analyze point distributions as an interaction with themselves. The different types of interactions are defined in Section 6.2.1.

	ground	*island*	*mountain*	*river*	*road*	*castle*	*house*	*farm*	*pine tree*	*red tree*
ground										
island	*map*	*rdf*								
mountain	*map*	*poly*	*rdf*							
river	*map*	*poly*	*rdf*	*rdf*						
road	*map*	*poly*	*rdf*	*arc*	*rdf*					
castle	*map*	*poly*	*rdf*	*arc*	*arc*	*rdf*				
house	*map*	*poly*	*rdf*	*arc*	*arc*	*rdf*	*rdf*			
farm	*map*	*poly*	*rdf*	*arc*	*arc*	*rdf*	*rdf*	*rdf*		
pine tree	*map*	*poly*	*rdf*	*arc*	*arc*	*rdf*	*rdf*	*rdf*	*rdf*	
red tree	*map*	*poly*	*rdf*	*arc*	*arc*	*rdf*	*rdf*	*rdf*	*rdf*	*rdf*

Table 6.2: Object types associated to their layer and data types.

Color and Palette. To draw an analogy with painting software, we define a *color* as the statistics analyzed from a scene example following a given interaction matrix $\mathcal{M}$, and a *palette* as a collection of colors. Those statistics are stored as *interaction histograms*, which will be used to compute probability functions during synthesis. In our application, a palette is displayed at the right of the screen, and its default colors are precomputed from various scene exemplars. The user can select one of these colors as the *active color* and use it to paint or deform parts of the scene, or can create a new color by selecting a new exemplar (a region populated with manually placed objects) to be analyzed.

Active and Influence Regions. Let Ω be the world space containing all the scene objects. We define $\varphi \subset \Omega$ as the *active region*, and $\mathcal{X}$ as the *influence region* (Figure 6.2). The influence region is computed automatically from the active region as an offset of distance r_i around the active region φ, with $\mathcal{X} \cap \varphi = \emptyset$. Radius r_i can be changed in the editor, and the influence can be disabled if desired. In our prototype, we work with three shapes of active regions: the default selection uses an

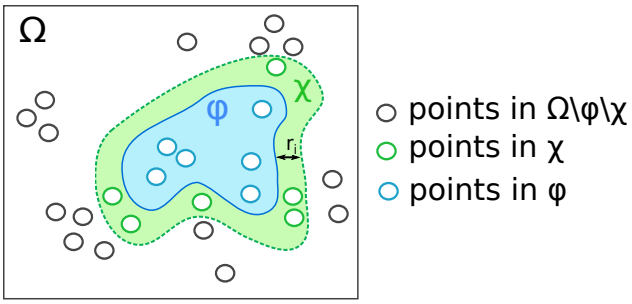


Figure 6.2: Example of active region φ and its influence region $\mathcal{X}$, with an influence radius r_i.

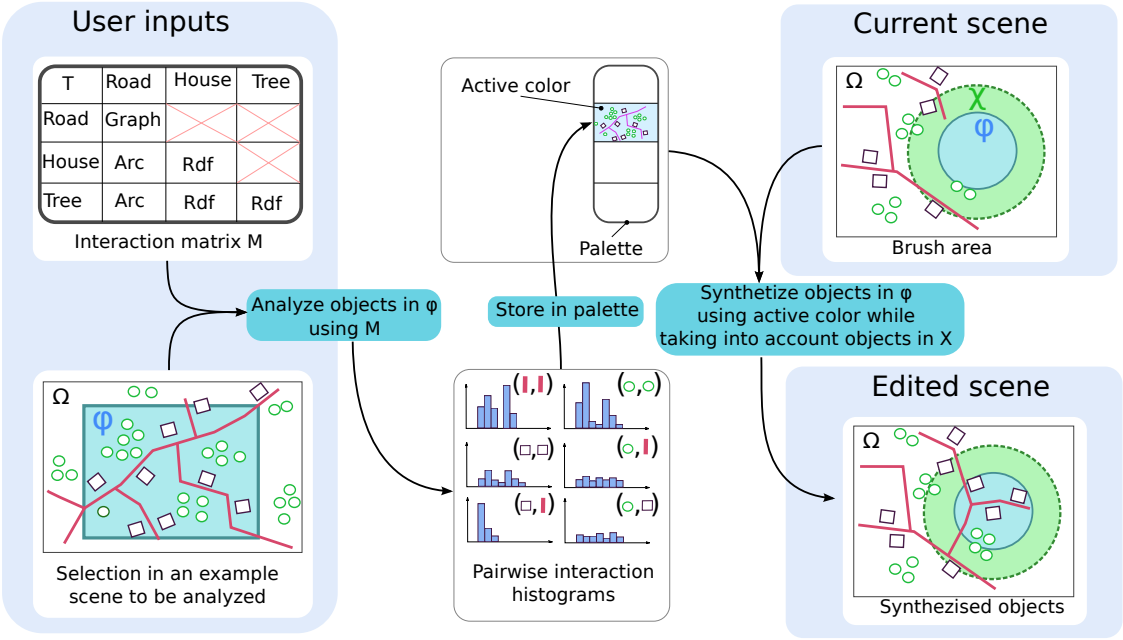


Figure 6.3: Method overview. The system takes as input a dependency matrix and an example scene created by the user. The user selects a region and the algorithm analyzes its properties as pairwise histograms, following the matrix dependencies. These properties are stored as a color in the palette. Then, when the user takes the brush to generate a portion of the scene, the algorithm synthesizes new objects underneath the brush, while taking into account the already existing surroundings.

axis-oriented bounding box, the brush shape has a circular region, and the polygon-based selection enables the use of any nonintersecting polygon as a region whose contour is directly sketched by the user.

Our technique relies on an example-based synthesis algorithm, which is used in several ways within our interactive tools. Our algorithm is able to synthesize both point distributions and graphs. Figure 6.3 shows an example of use of the system, where a user first creates a scene by hand, then selects and analyzes it to create a new color that is stored in the palette, and finally uses this color with a brush to create a new larger scene.

We present in Section 6.2 the point distribution synthesis algorithm and in Section 6.3 the graph

synthesis algorithm. The latter is decomposed in two steps: first we generate a node distribution using the point distribution synthesis algorithm, and then we connect the generated nodes with arcs using a new arc synthesis algorithm. In Section 6.4 we present new tools for editing vectorial sceneries based on these synthesis algorithms. We present in Section 6.5 a new interface enabling the painting of vectorial worlds.

6.2 Statistical Point Synthesis

Let us consider a point distribution of category $\mathcal{C}_a$. This distribution can only interact with objects of categories $\mathcal{C}_b \geq \mathcal{C}_a$. The priority order and interaction types are given by the interaction matrix, defined by the user. The interactions between point distributions are analyzed, and we synthesize new point distributions respecting the same properties. There are several interaction types, each described in Section 6.2.1. The synthesis algorithm uses a probability function described in Section 6.2.2, and is detailed in Section 6.2.3.

6.2.1 Analysis of Point Distributions

The interaction $\mathcal{I}$ of a point distribution with itself is analyzed using a *rdf* interaction type, and its interaction with other object types can be $\mathcal{I} \in \{$*rdf, arc, poly, map*$\}$. *rdf* is the radial distribution function, *arc* the distance to arcs, *poly* the signed distance to polygons, and *map* the value of external terrain constraints, such as its elevation or slope, at a given position. The type of interaction between objects is provided by the interaction matrix created by the user. Consequently, the user can specify which interactions will be analyzed. For each interaction, we store the analyzed statistics into histograms that will be used later during synthesis.

We describe next the different interaction types that are analyzed in our prototype. Note that these interactions were chosen during the algorithm development, but could be easily extended to other kinds of interaction, depending on the user needs.

Radial Distribution Function. The radial distribution function $g(r)$ characterizes how density varies in a point distribution as a function of distance r from a reference point. It is a powerful descriptor for point distributions, capable of capturing many behaviors, in particular point clusters, that are especially useful when synthesizing tree or house distributions.

We compute the radial distribution function histogram h_{rdf} as follows: consider X and Y, two point distributions in the scene of respectively n_X and n_Y points. Let φ be the region analyzed, A its area, and dA the area of a annular shell of radius $\delta_r k$ and thickness δ_r. For each object in X, we count the number of objects from Y that are in dA, and normalize it (Figure 6.4 left):

$$h_{rdf}(k) = \sum_{x_i \in X} \sum_{\substack{y_j \in Y \\ k\delta_r \leq d(x_i,y_j) < (k+1)\delta_r}} \frac{A}{dA n_X n_Y}.$$

The window-edge effect is important when manipulating small regions with few objects, which occurs often in our application. Indeed, when the *rdf* is evaluated for a point near the window border, it will count fewer objects than a point at the center, because it can only consider objects in $dA \cap \varphi$.

One approach could be to create an influence region around φ and use it to analyze the scene. However, it implies that all analyzed regions should be surrounded by an influence region whose

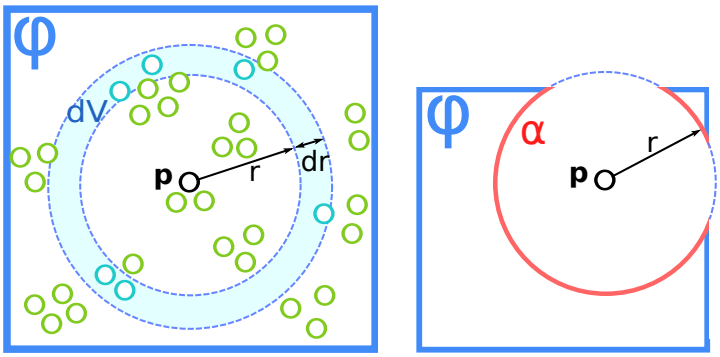


Figure 6.4: Left: Illustration of the radial density function. For a given point $\mathbf{p}$ and a radius $r = k\delta_r$, we count the number of points at distance $d \in [k\delta_r, (k+1)\delta_r[$ of $\mathbf{p}$. Right: Window effect correction. We evaluate α, the circumference of circle $C(\mathbf{p}, r)$ intersecting the active region φ. This value is used to normalize the integration region dA.

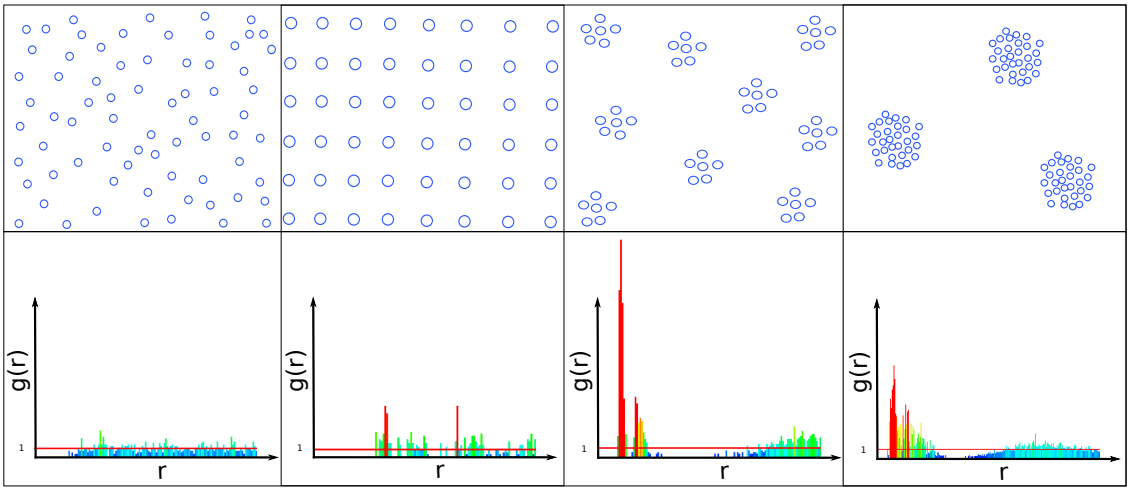


Figure 6.5: Examples of point distributions and their radial distribution functions. The distribution on the left is a random distribution, giving the *rdf* noisy values around 1. The two distributions on the right are cluster distributions with different parameters. An *rdf* of cluster distributions has a high peak near to the minimal distance between the points, and low values around the mean cluster size. Regular point distributions have an *rdf* with several peaks corresponding to multiples of the distance between points.

properties are equivalent to the analyzed region, thus forcing the user to create larger example scenes.

Our solution is to approximate the area $dA' = dA \cap \varphi$ by computing for each point the proportion $\alpha_{x_i,r}$ (Figure 6.4 right) such that:

$$dA' = r\delta_r 2\pi\alpha_{x_i,r}.$$

Distance to Graphs. With this interaction, we analyze the distribution of distances of point distributions to a graph. For each point, we compute the closest distance d to the graph and add it in the histogram. The histogram is then normalized by the number of elements.

$$h_g(k) = \sum_{\substack{x_i \in X \\ k\delta_r \leq \min\{d(x_i,y_j), y_j \in Y\} < (k+1)\delta_r}} \frac{1}{n_X}$$

This is used for synthesizing the interaction of point distributions with structured elements such as roads or rivers. For instance, we can analyze the distribution of distances of houses to roads to generate new houses along roads.

Signed Distance to Polygon. With this interaction, we analyzes the distribution of distances of points of a given type to a polygon. For each point of the distribution, we compute the closest distance d_s to a polygon and count it in the histogram. The histogram is then normalized by the number of elements:

$$h_p(k) = \sum_{\substack{x_i \in X \\ k\delta_r \leq \min\{d_s(x_i,y_j), y_j \in Y\} < (k+1)\delta_r}} \frac{1}{n_X}.$$

The distance d_s is signed, i.e., positive if the point is outside the polygon, and negative elsewhere. We also normalize the distance inside the polygon by dividing it by the maximum distance to borders inside the polygon d_{max}, to have a proportion independent of the polygon size:

$$d_s(x_i, y_j) = \begin{cases} d(x_i, y_j) & \text{if } x_i \text{ is outside } y_j, \\ -d(x_i, y_j)/d_{max}(y_j) & \text{otherwise.} \end{cases}$$

This distance is used for the interaction of point distributions and polygons, such as an island contour, and makes possible to characterize if the point is on the island or in the sea, and its distribution within the island. We can for instance generate boats near the coast and trees on the entire island.

Maps. It is often useful to consider terrain constraints during the analysis, such as elevation or slope. The object *ground* contains several maps, for instance for slope, height, and interest. Using this property, it is possible to copy trees on a rough terrain and generate new ones on another terrain, while respecting their slope distribution. The map histogram analyzes the distributions of map values $m(x_i)$ from points x_i:

$$h_m(k) = \sum_{\substack{x_i \in X \\ k\delta_v \leq m(x_i) < (k+1)\delta_v}} \frac{1}{n_X}.$$

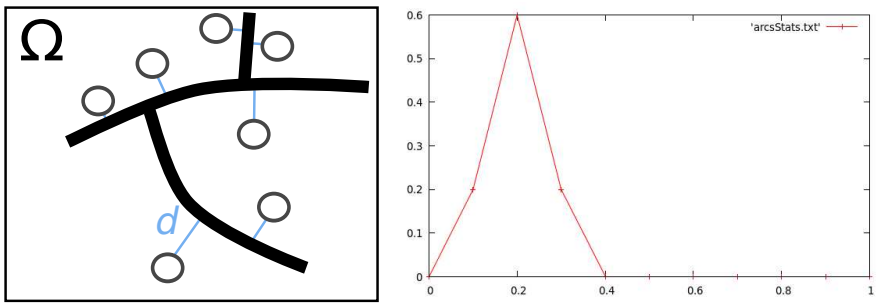


Figure 6.6: Left: A point distribution and a road network. For each point we compute d_{min} the closest distance to the graph. Right: Corresponding histogram of the closest distance to the graph.

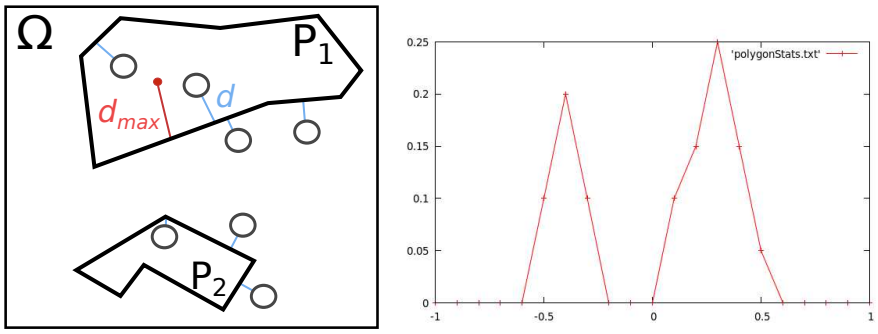


Figure 6.7: Left: Point distribution and two polygons. Right: Corresponding histogram of the signed closest distance to polygons.

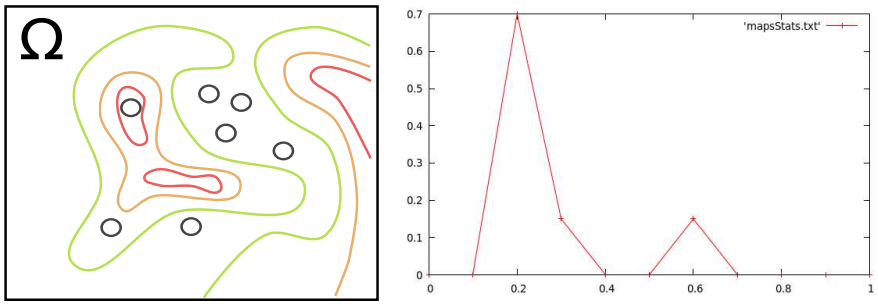


Figure 6.8: Example of histogram extracted corresponding to the elevation at the point locations.

6.2.2 Probability Density Function

In order to synthesize new objects respecting the interaction histograms analyzed, we use a classical point process based on Monte Carlo Markov Chain algorithms [HLT*09, LGH13].

We successively generate scene object categories with a hierarchical order. Let $\mathcal{C}_X$ be the category to be generated, influenced by categories $\mathcal{C}_{Y_k} \leq \mathcal{C}_X$. Let $\mathcal{I}_{X,Y_k}$ be the interaction type between the two categories and h_{X,Y_k} the probability function computed from the interaction histogram analyzed from the scene example.

The probability of a given point distribution $X = \{x_1, ..., x_{n_X}\}$ of category $\mathcal{C}_X$ is expressed with a probability density function, noted $f(X)$, that we define as follows:

$$f(X) \propto \prod_{\mathcal{C}_{Y_k} \leq \mathcal{C}_X} \prod_{x_i \in X} \prod_{y_i \in Y_k} h_{X,Y_k}(x_i, y_j).$$

6.2.3 Synthesis using Monte Carlo Markov Chain

The goal is to generate a new scene that matches statistics of exemplars. To do so we use a Metropolis-Hastings sampling method based on iterative birth-and-death perturbations [HLT*09].

Algorithm 4 Metropolis-Hastings sampling.

- randomly initialize arrangement $X \leftarrow X_0$, with $f(X_0) > 0$
for iteration t from 1 to T **do**
equiprobable perturbations:
Birth
- generate new element p at a random location in φ
- create candidate arrangement $X' = X \cup \{p\}$
- compute acceptance ratio $R_b = \frac{f(X')}{f(X)} \frac{n_X}{\mathcal{A}}$
- accept new arrangement $(X \leftarrow X')$ with probability R_b
Death
- select a random element $p \in X$
- create candidate arrangement $X' = X \setminus \{p\}$
- compute acceptance ratio $R_d = \frac{f(X')}{f(X)} \frac{\mathcal{A}}{n_X}$
- accept new arrangement $(X \leftarrow X')$ with probability R_d
end for

It consists of performing a fixed number T of steps, either a birth or a death. In our prototype, we use $T = 10^3$. The acceptance ratio R enables the evaluation of the new arrangement X' relatively to the previous one X. Because arrangements X and X' have different numbers of elements, the ratio must be multiplied by a ratio depending on $\mathcal{A}$, the area of φ, and n_X, the number of elements of X.

Note that it is essential that the initial arrangement X_0 has a nonzero probability $f(X_0) > 0$, because it would cause a undefined acceptance value and prevent any convergence.

Acceptation Ratio Simplification. The acceptation ratio of a birth is defined by $R_b = \frac{f(X)}{f(X')} \frac{n_X}{\mathcal{A}}$, which can be expensive to compute. Fortunately, since f is defined by a product of probabilities, and X and $X' = X \cup \{p\}$ are almost identical, the acceptation ratio can be simplified as follows :

$$
\begin{aligned}
R_b &= \frac{n_X}{\mathcal{A}} \frac{f(X')}{f(X)} \\
&= \frac{n_X}{\mathcal{A}} \frac{\prod_{\mathcal{C}_{Y_k} \leq \mathcal{C}_X} \prod_{x_i \in X'} \prod_{y_i \in Y_k} h_{X,Y_k}(x_i, y_j)}{\prod_{\mathcal{C}_{Y_k} \leq \mathcal{C}_X} \prod_{x_i \in X} \prod_{y_i \in Y_k} h_{X,Y_k}(x_i, y_j)} \\
&= \frac{n_X}{\mathcal{A}} \frac{\left(\prod_{\mathcal{C}_{Y_k} \leq \mathcal{C}_X} \prod_{x_i \in X} \prod_{y_i \in Y_k} h_{X,Y_k}(x_i, y_j)\right) \left(\prod_{\mathcal{C}_{Y_k} \leq \mathcal{C}_X} \prod_{y_i \in Y_k} h_{X,Y_k}(p, y_j)\right)}{\prod_{\mathcal{C}_{Y_k} \leq \mathcal{C}_X} \prod_{x_i \in X} \prod_{y_i \in Y_k} h_{X,Y_k}(x_i, y_j)} \\
&= \frac{n_X}{\mathcal{A}} \prod_{\mathcal{C}_{Y_k} \leq \mathcal{C}_X} \prod_{y_i \in Y_k} h_{X,Y_k}(p, y_j).
\end{aligned}
$$

A similar simplification can be used for a death ratio. Consequently, when performing a small deformation of an arrangement (i.e., adding or removing one element), only the probability associated to this element has to be computed, highly improving computation time.

6.3 Statistical Graph Synthesis

Graph synthesis is used to edit structured objects such as roads or rivers. Graphs are composed of nodes and arcs. In our method the two are analyzed and synthesized independently. Indeed, we decompose our graph algorithm in two steps: first we generate nodes using the point distribution synthesis method described previously; then we generate arcs between the generated nodes using a new algorithm based on a similar approach to the point distribution synthesis algorithm (Figure 6.9).

6.3.1 Arc Analysis

Similarly to point distribution analysis, we analyze several properties of graph structures, all stored in individual histograms to be used later as probability functions by the synthesis algorithm.

Arc lengths. The distribution of arc lengths within the graph is analyzed and stored in an histogram. This is used to favor the connection of nodes at similar distances, respecting the distance distribution of the example.

Arc angles. The distribution of angles formed by arcs is analyzed and stored in an histogram. This is used to favor arcs at similar angles relatively to existing arcs. This can be used to synthesis villages with random street directions and villages with perpendicular streets.

6.3.2 Arc Synthesis

In order to synthesize new arcs respecting the graph properties analyzed, we use an approach based on Monte Carlo Markov Chain algorithms.

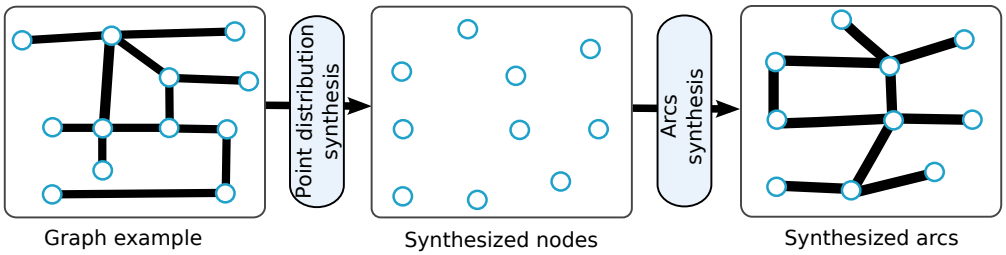


Figure 6.9: Graph synthesis steps. Left: A graph example to analyze. Center: Generation of a node distribution. Right: Connecting nodes with an arc synthesis algorithm.

Arc Probability Function. The probability of a given arc set $A = \{a_1, ..., a_{n_A}\}$ is expressed with an arc probability function, noted $f_a(X)$, that we define as follows:

$$f_a(A) \propto \left[\prod_{a_i \in A} h_l(a_i) \right] \left[\prod_{a_i \in A} \prod_{a_j \in N(a_i)} h_a(a_i, a_j) \right],$$

where h_l is the arc length probability, h_a is the arc angle probability, and $N(a_i)$ is the set of arcs in A sharing a node with a_i.

Metropolis-Hastings Synthesis. The arcs are generated using the same algorithm that the point distribution synthesis, i.e., we generate the set of arcs by using birth-and-death steps, and by using the arc probability function in the acceptance ratio computation.

Improvements. Our goal is to provide a generic algorithm to synthesize streets, roads, rivers, and island contours. The graph synthesis method presented in this section is an early attempt to achieve this goal. The current statistics (i.e., arc lengths and angles) are only geometrical and the results still need to be improved.

We are currently investigating topological statistics criteria, such as connectivity, graph robustness, number of cycles, and distribution of shortest paths between pairs of nodes. While we believe our approach is going in the right direction, more work is required for achieving satisfactory results.

6.4 Synthesis-based Deformation

In the previous section, we detailed how we analyze and synthesize vectorial scenes to generate new virtual worlds. However, the key idea is to apply synthesis methods to the interactive modeling of virtual worlds. In this section, we present new operators to edit virtual worlds using example-based synthesis, all inspired from classical editing operators in painting systems such as copy-paste, move, scale, and gradient.

6.4.1 Copy-paste

In classical softwares, copying-and-pasting a part of a scene results in duplicating elements with the same arrangement. This is less suitable for virtual worlds, which have multiple self-similar details, but not exactly the same.

Our approach for copying-and-pasting is not to copy the arrangement, but the *color*, in the sense of interaction statistics, computed from the scene analysis. When pasting, a new arrangement will be generated, respecting the statistical properties of the exemplar, while all the copied regions are unique.

To copy-and-paste, the user first selects an active region φ_c in the scene, by defining its contour. The elements in φ_c are analyzed and a new color is created. Then, the user sets a new active region φ_p, to define where the objects are going to be pasted, and the synthesis method (Algorithm 5) is used to generate them.

Note that φ_c and φ_p can be any type of region, with different shapes and sizes; our idea is of copying not individual elements, but their visual style. Moreover, φ_p can be set with an influence region $\mathcal{X}_p$, which will be taken into account while synthesizing new objects.

Algorithm 5 Copy-paste.

Copy
input : interaction matrix $\mathcal{M}$
- set φ_c region to copy
- active color $\leftarrow analysis(\varphi_c, \mathcal{M})$

Paste
- set φ_p region where to paste, and $\mathcal{X}_p$ its influence region
- remove objects in φ_p
- synthesize objects in φ_p using active color while taking into account objects in $\mathcal{X}_p$

Choice of the Selection Region. When copying, the shape of the selection region φ_c has an important impact with respect to the analyzed color. Figure 6.10 shows two results of copy-paste. One of them is a large surrounding selection, creating clusters, whereas the other one is a tighter selection, creating a denser distribution.

Choice of the Influence Region. When pasting, it is important to take into account the surrounding objects. Figure 6.11 shows two results of pasting, with and without an influence region. When pasting without the influence region, the new clusters are not coherent near region boundaries, whereas when the region is enabled, the new clusters extend more nicely clusters of the influence region.

6.4.2 Move

If a user moves a region to a new location, the displaced objects will be constrained by the new surrounding objects and by the external constraints of the new ground. For instance, when moving a forest off a terrain, slopes change, and consequently the tree arrangement must be adapted to these new constraints.

Using the copy-paste method described previously will result in a whole new arrangement, which might not be what the artist expects. Indeed, the artist could prefer a deformation preserving the original arrangement as much as possible, but adapted to the new location.

Improvements. To adapt objects, we plan to use a Metropolis-Hastings algorithm similar to the synthesis algorithm (Section 6.2.3). Instead of starting from a random arrangement, we will

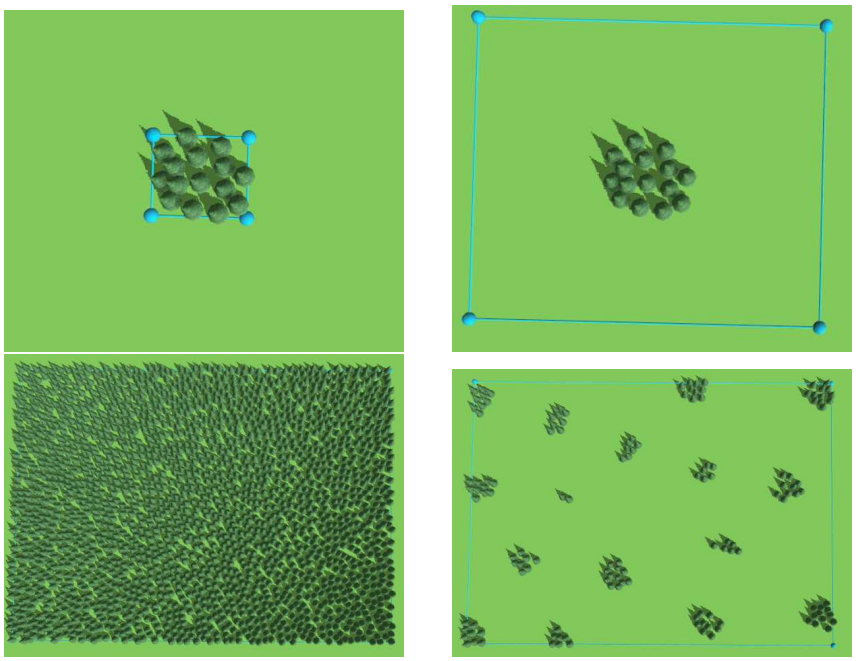


Figure 6.10: Left: Copying (top) and pasting (bottom) while selecting closely the objects; the synthesized color is a dense distribution. Right: Copying (top) and pasting (bottom) with a larger selection region; the color is a distribution of clusters.

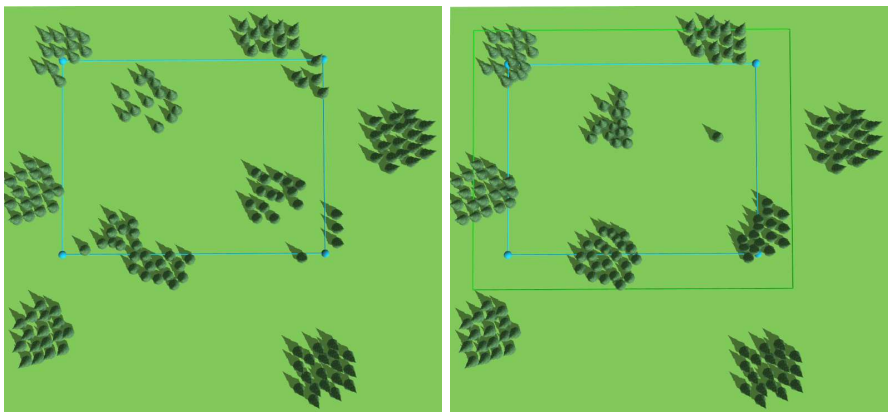


Figure 6.11: Left: Pasting without an influence region. Right: Improving with influence region. Note how the new clusters complete the existing ones whereas in the left scene they make the scene more inconsistent by ignoring them.

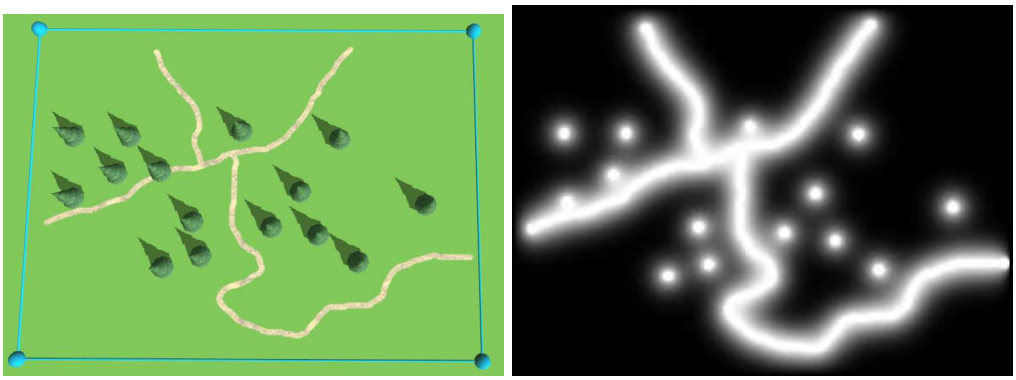


Figure 6.12: Left: Example scene with some trees and a road graph. Right: Energy computed and rasterized from the scene.

start from the existing objects. Instead of performing only birth-and-death perturbations, we can add a displacement perturbation by choosing randomly an existing point p and moving it to a new random location p':

$$X' = X \setminus \{p\} \cup \{p'\}.$$

Moreover, to favor smaller displacements, the probability function f can be multiplied by a Gaussian kernel depending on the distance $\|p - p'\|$.

6.4.3 Seamcarving-based Scaling

In standard software, scaling a part of a scene deforms element arrangement, where spacing between objects is uniformly changed, thus not preserving scene properties. We rather interpret the scaling of a region as a need for a larger region, respecting the arrangement properties in the original one.

Our approach to achieve this is inspired by seamcarving-based image scaling [AS07]. Seamcarving techniques seamlessly deform images, by finding paths (cuts) of least-energy in the original image (i.e., traversing smoothly varying pixels) and adding new pixels with interpolated colors or removing pixels along a cut, for instance, to enlarge or shrink the image.

When the user scales a region, the algorithm finds a cut that minimizes the scene deformation. We then only deform the objects near the path, which allows us to preserve the rest of the scene.

Energy Map. To find the best cut within the scene, we use the standard approach of seamcarving, i.e., computing the path of minimal energy within the scene, in the direction perpendicular to the deformation. We need to define an appropriate energy function, expressing the cost of cutting the scene at a given location, in the case of 3D distributions (Figure 6.12). We construct this energy from the vectorial data by rasterizing a repulsion energy that is inversely proportional to the distance to the nearest objects:

$$e(\mathbf{p}) = \frac{1}{\min_{o_i \in O}\{d(\mathbf{p}, o_i)\}},$$

where o_i is one object (point in a distribution or arc in a graph) in all objects O of the scene.

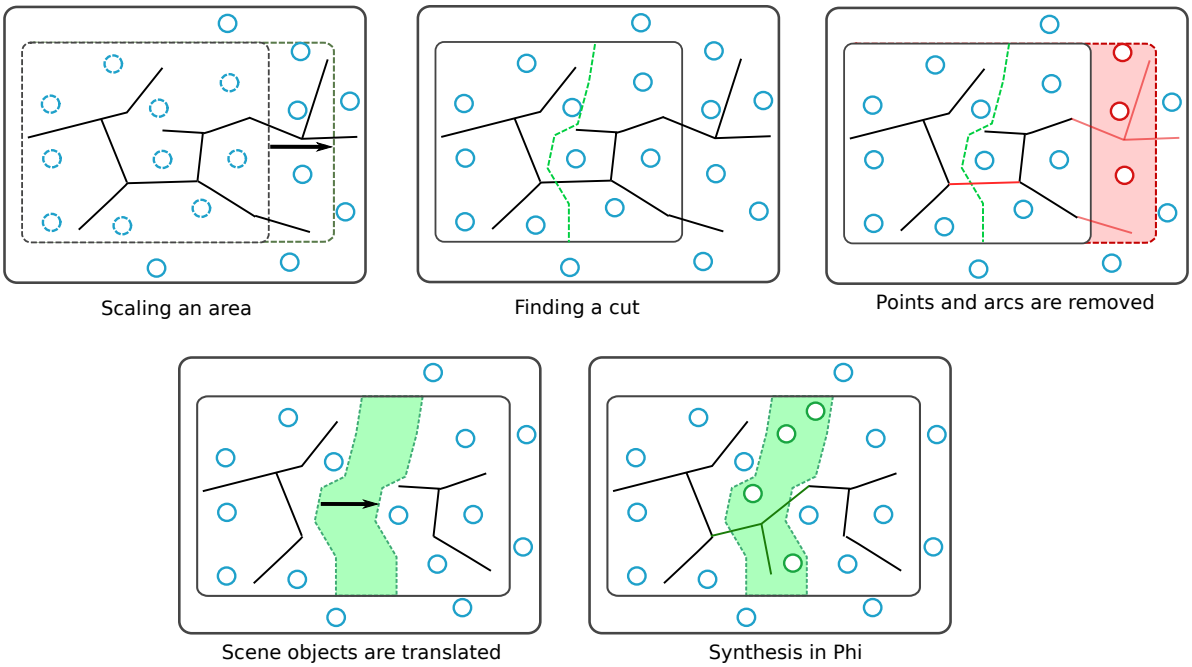


Figure 6.13: Enlarging a region: the user drags the region contour to enlarge it. The algorithm finds the best cut in the scene. The objects overlapping the displaced region are removed, as well as the arcs traversed by the cut. The objects on one side of the cut are translated on the scene depending on the scaling direction. New objects are synthesized in the empty space left by this displacement.

Enlarge. To enlarge a scene, we remove the objects overlapping the newly expanded region, and synthesize objects in the free space created by their displacement (Figure 6.13).

Shrink. To shrink a scene, we remove the objects around the cut, and synthesize objects in the space created by the region whose size is shrunk (Figure 6.14).

6.4.4 Color Interpolation and Gradient

To augment the analogy with painting software, we should provide tools to merge colors (where colors are properties of distributions, as stated in Section 6.1), in order to create gradients and brushes with alpha masks. In this section, we first detail how we perform histogram and color interpolations, and then detail the new gradient tool.

Color Interpolation. We perform histogram interpolation using optimal mass transport of the analyzed interaction histograms [Rea99], which is based on the inverse cumulative density function. Contrary to classical interpolation, this method performs a natural interpolation of histogram shapes (Figure 6.15). Indeed, a naive interpolation of two Gaussian shapes does not result in a Gaussian shape, but in a fade-in, fade-out phenomenon, while with this interpolation the result will be a Gaussian shape with parameters interpolated from the two example shapes.

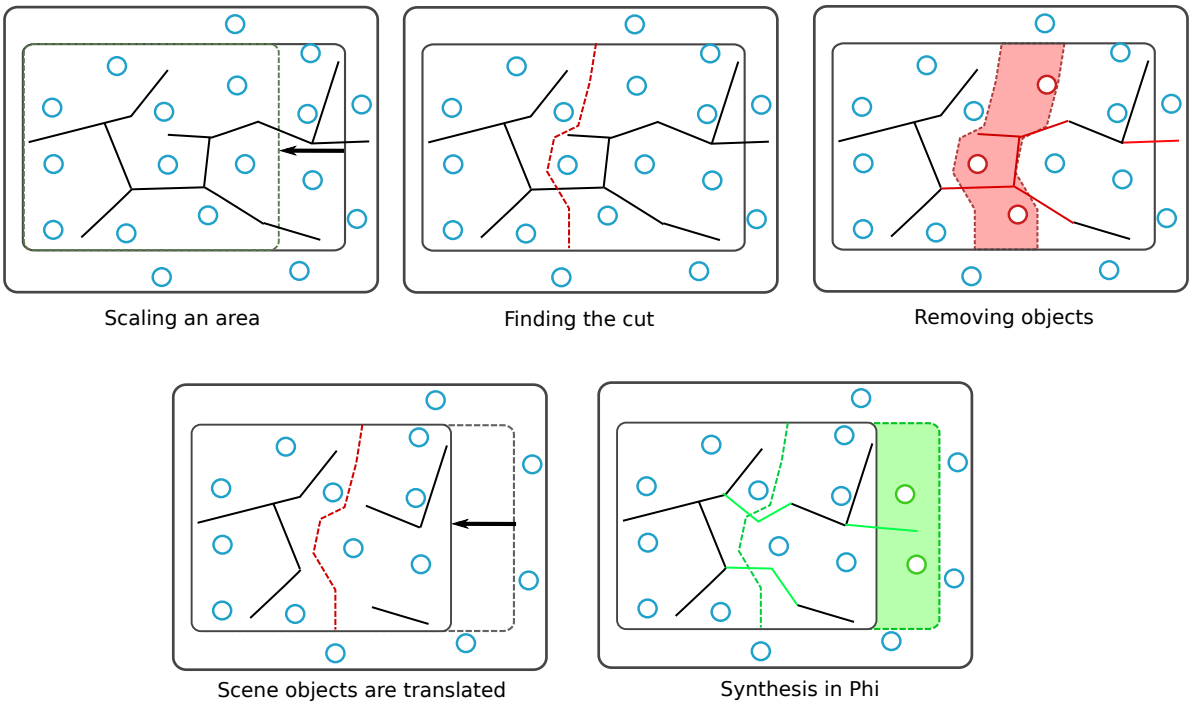


Figure 6.14: Shrinking a region: the user drags the region contour to shrink it. The algorithm finds the best cut in the scene. The objects in the cut region are removed. The objects are translated on the scene depending on the cut size and scaling direction. New objects are synthesized in the empty space at the right.

A color is defined by a set of interaction histograms computed from an example analysis. To interpolate colors, we interpolate independently each histogram.

Gradient. Using the color interpolation described previously, we provide a gradient tool that takes two colors as input, and enables the creation of objects with a color interpolated at different levels within the active region.

The active region φ is segmented in N independent regions $\varphi_0, ..., \varphi_{N-1}$. The algorithm successively performs a synthesis in the regions φ_k, $k \in [0, N-1]$, while taking into account the surrounding objects in $\mathcal{X}$ and the newly created objects in $\{\varphi_i, i < k\}$ (Figure 6.17). The color used in each region is interpolated from the initial and final colors:

$$c_k = interpolate\left(s\left(\frac{k}{N-1}\right), c_{initial}, c_{final}\right),$$

$s(t) : [0,1] \mapsto [0,1]$ is the gradient shape function. In our prototype we provide linear, bilinear, and radial shapes. Note that N can be a fixed value or depend on the region size.

6.5 Synthesis-based Painting

The key concept of our method is to paint distributions and structures directly in a scene. We show in this section how to use the operations presented previously to design a new interactive paint-like

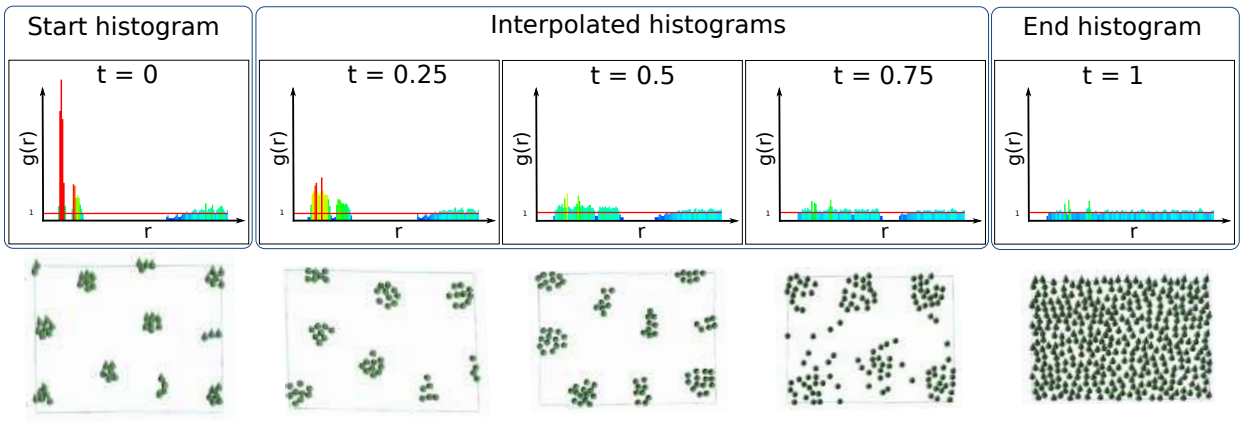


Figure 6.15: Top: Interpolation of two *rdf* histograms. The histogram shapes are deformed, thus naturally interpolating the interaction properties. The initial histogram corresponds to a point distribution with dense clusters, while the final histogram corresponds to a random distribution. The interpolated histograms show clusters enlarging and point interaction becoming less and less attractive to finally become a random distribution. Bottom: Point distributions generated using the different histograms.

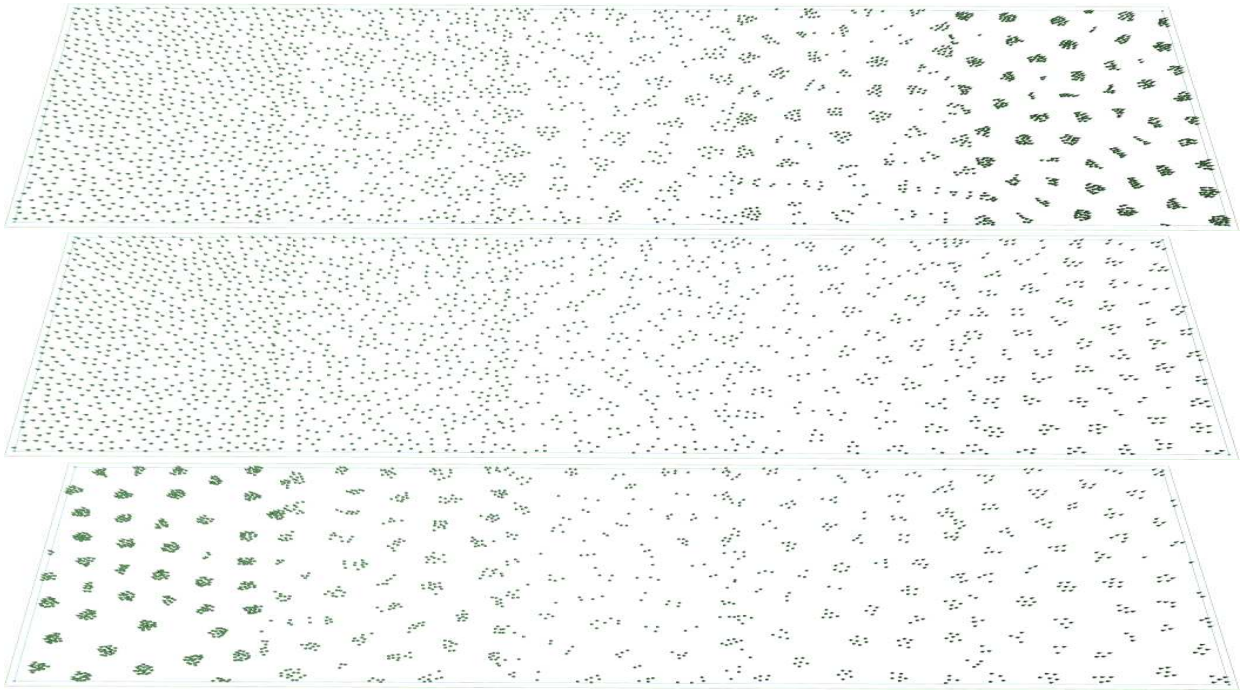


Figure 6.16: Examples of gradients. From top to bottom: Gradient from a uniform distribution to a dense cluster distribution, gradient from a uniform distribution to a sparse cluster distribution, and gradient from a dense cluster distribution to a sparse cluster distribution.

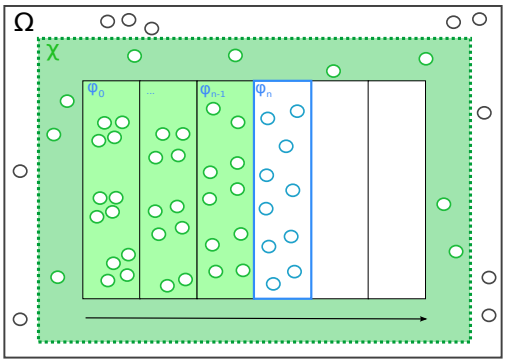


Figure 6.17: Gradient synthesis. At the k, the points are synthesized in φ_k with interpolated color c_k, while being influenced by $\mathcal{X} \cap \{\varphi_i, i < k\}$.

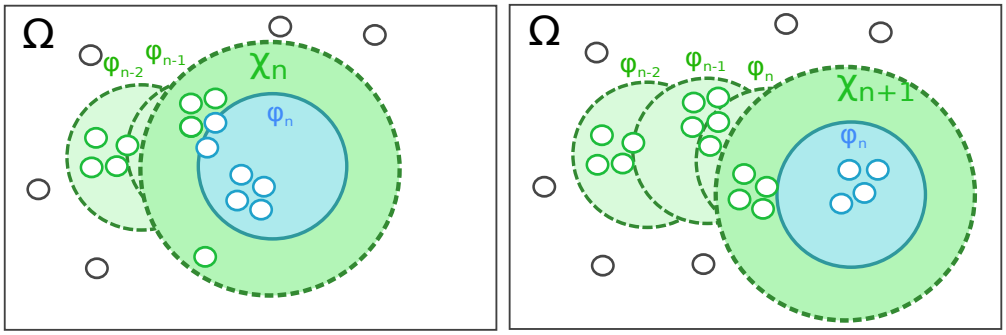


Figure 6.18: Two successive steps of a brushing gesture. Synthesis of the blue objects in φ_n while taking into account $\mathcal{I}$. The new clusters respect the distance properties with the existing ones.

tool for modeling virtual worlds.

6.5.1 Brush

The brush is a special case of synthesis where the synthesis region evolves during the brushing gesture. When the user performs a brushing gesture, we compute several synthesis steps, each of them taking into account the previously synthesized regions, similarly to the gradient tool (Figure 6.18).

The tool works as follows: during a single brushing gesture, all synthesis regions $\{\varphi_k\}$ are kept in memory. When the user has moved sufficiently far along the brushing region, we perform a new synthesis step: all existing objects in φ_n are removed, except if they were created at a previous step, and then we synthesize new objects in φ_n with an influence region $\mathcal{I} = \mathcal{X}_n \cup \{\varphi_k, k \leq n\}$.

Pipette. The pipette tool is equivalent to the copy tool, except that it picks the color in a circular region of the same radius as the brush.

Eraser. The eraser simply removes all objects within the circular region.

Alpha. We started working on alpha masks, based on color interpolations. We plan to interpolate between the existing color with the brush color, with an alpha coefficient that varies along the brush

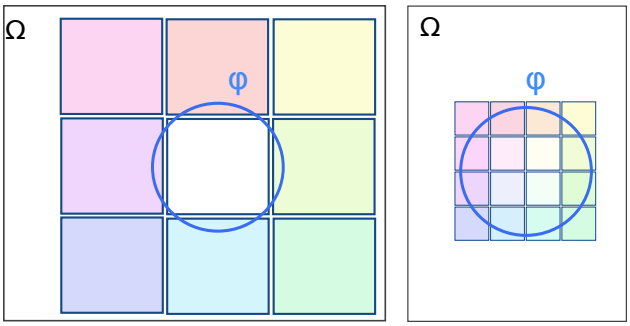


Figure 6.19: Left: Several *colors* are analyzed around the blur region φ. Right: Synthesis within φ using various interpolated *colors*.

radius. Using those masks, we aim to provide a brush tool whose colors vary depending on the distance to the brush center.

6.5.2 Blur

Another common tool in traditional drawing softwares is blur. Traditionally, this tool smoothly blends colors. Consequently applying a blur to a scene means to smoothly blend point distributions and graph structure properties.

Improvements. We plan to provide a blur tool working as follows: first, we perform several analyses in the brush surrounding, then we synthesize new objects within the brush region, and use different colors computed from various interpolations of the analyzed regions (Figure 6.19).

6.6 Preliminary Results and Discussion

In this section we present preliminary results. Most of the tools described in this chapter have been implemented in an interactive world modeling prototype.

While we already reached good results on unit test examples, more work is required to improve robustness to more complex sceneries, such as islands with mountains, rivers, roads, houses, and trees. Particularly, the graph algorithm still requires more research to add other constraints, such as topological statistics, to better reproduce graph examples.

We changed several times the synthesis algorithm. An early solution applied energy minimization on point distributions, with an energy computed from analyzed histograms. The results proved very positive on point distributions (Figure 6.21) but lacked robustness to support complex scenes and graphs. Indeed, our solution worked mainly with histograms with many local maxima. Since we switched to the Metropolis-Hastings algorithm as presented in Section 6.2.3, we did not adapt all our tools, but are confident in this new approach.

Interface. The user navigates in the virtual world with camera controls similar to Blender [Ble14]. Figure 6.20 shows the editor window.

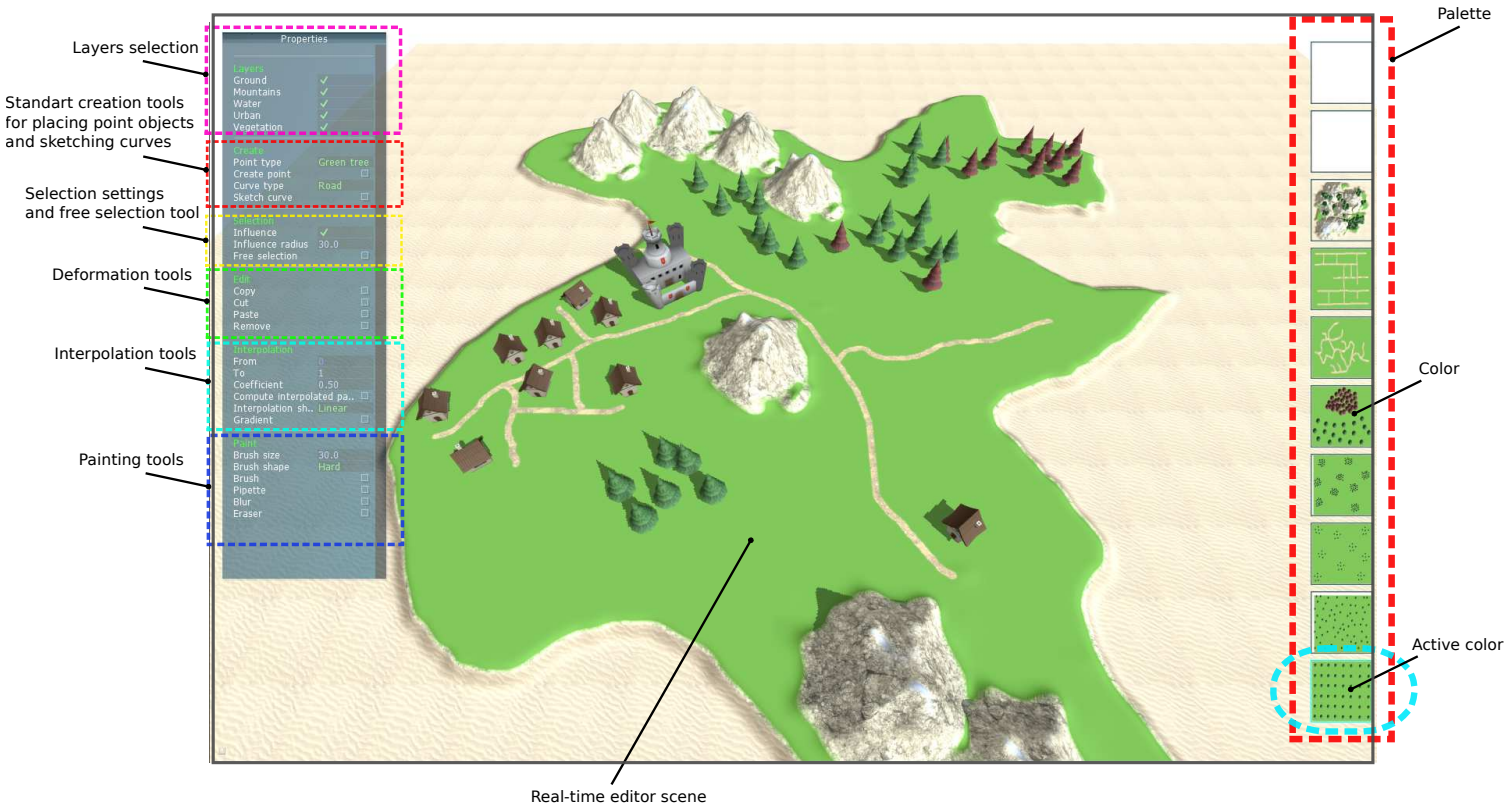


Figure 6.20: Editor interface. On the left, tool widgets let the user modify the tool settings. In the center, the scene is edited and visualized in real time. On the right, the palette displays schematically the colors currently available.

Implementation. The prototype is implemented in C++, using OpenGL and GLSL Compute Shaders. Computations are performed on an NVidia 660GTX GPU and an Intel® Xeon® E5-1650 CPU, running at 3.20 GHz with 16 GB of memory. Our algorithms use only one CPU thread.

Synthesis. Figures 6.21 and 6.22 show examples generated with our algorithms. When all unit tests will be validated, larger and more complex scenes will be designed to test the robustness of our solution. The graph synthesis algorithm still needs topological constraints in order to have a valid road or river network.

Performance. The scenes in Figures 6.21 and 6.22 were analyzed in 0.005 to 0.041 seconds and synthesized in 0.124 to 0.709 seconds. However, they remain simple examples and computation time may increase with scene complexity, even if we did not do any optimization yet.

Comparison to Other Methods. Our objective being to provide interactive editing tools, our major concern is the trade-off between computation time and synthesis fidelity. Although our method compares relatively well to state-of-the-art arrangement synthesis methods, more detailed comparisons will be performed with our final algorithm.

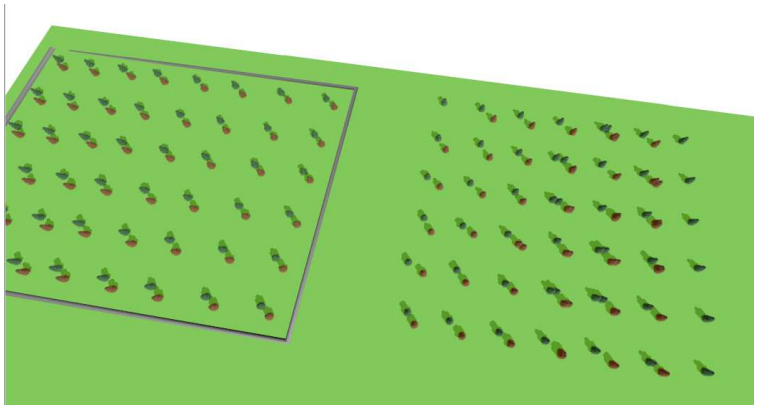


Figure 6.21: Example of a synthesized point distribution using an oriented radial density function.

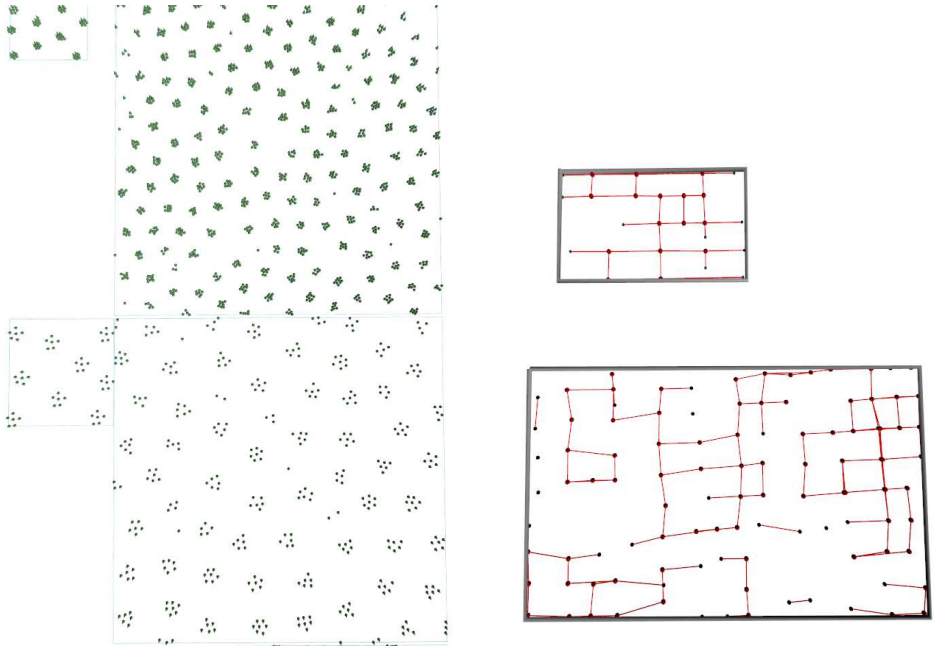


Figure 6.22: Examples of synthesized scenes. The smaller boxes contain the original examples and the larger ones, the synthesized scenes.

User Study. We started our work with a pre-research user study about the deformation and synthesis of vectorial scenes. In this study, 19 persons from various genders, ages, and artistic skills were asked to manually enlarge nine different vectorial scenes printed on paper, and to manually fill the empty space in ten other vectorial scenes (Appendix A). This user study allowed us to validate the set of interactions to analyze and guided the interaction matrix and the synthesis algorithm.

We did not more fully evaluate our prototype yet, but we believe that users will be attracted by our painting metaphor for modeling virtual worlds.

Limitations. The first limitation comes from the interaction matrix, since it needs to be given as input. The matrix used in our prototype is always the same, but it may not be appropriate for

particular cases. Moreover, it implies to know the object types and categories. A more flexible approach would analyze and synthesize scenes without a priori knowledge, like in recent example-based synthesis techniques [HLT*09] or in inverse procedural methods [YYW*12].

Another possible problem is that object distributions may have various properties within the same analyzed scene. However, it is possible to add more granularity to the method by segmenting the scene in cells and computing colors independently within each of them.

Our synthesis algorithms work well for natural scenes with chaotic arrangements, but are not adapted to complex sceneries with meaningful arrangements, such as 2D vectorial art. Inverse procedural modeling could provide a more generalized approach to synthesize arbitrary sceneries.

6.7 Conclusion

We presented in this chapter a more artist-centered procedural-based editor for vectorial maps, combining intuitive user editing and example-based procedural generation. Instead of providing a new procedural method and making it interactive, we focused from the beginning on an interactive editor similar to interactive painting editors, and worked to develop the underlying models for distribution and graph syntheses, enabling the user to convey his intent without taking care of details. Our preliminary results are promising and we believe our prototype will soon be fully operational. However, more work is needed to better formalize the algorithms and make them robust to complex cases with a large number of objects of different categories.

One big advantage of our approach is that it can be used with any type of example-based synthesis or inverse procedural method. We believe this approach could lead to a new kind of interactive modeler where modeling is a tight coupling between intuitive and interactive editing, example-based synthesis, and procedural modeling.

Chapter 7

Conclusion

In this thesis, we explored several approaches to improve the process used to model virtual worlds. While each chapter had its conclusion and suggested some future directions of investigation, we summarize here our main contributions and present more general recommendations for future work in this area.

Contributions

The work presented in this thesis can be divided into two main types of contributions to virtual world modeling: firstly, we introduced new methods for modeling new types of virtual world elements; secondly, we focused on improving approaches for interactive editing of virtual worlds.

In this thesis, we introduced two methods for the modeling of virtual world elements that were not yet treated: villages and waterfalls. Indeed, we presented the first method for procedural village generation. Our method relies on a particle-based seeding and a growth simulation. By intertwining building and road generations, and providing dynamic interest maps, we were able to generate a variety of plausible villages, ranging from fortified villages on a cliff to fisherman villages on a coast. This system is highly flexible and we believe that it is an important direction to follow for urban and non-urban environment generation. We also presented the first method for interactive modeling of waterfall sceneries. Our method enables the creation of a coarse waterfall network, and the system automatically handles the generation of a detailed waterfall scene that preserves hydraulic consistency.

The second major focus of our work was rethinking the interactive modeling of virtual worlds, in order to answer the question formulated in our introduction: "How can we improve the interactive design of virtual worlds to better match user needs and computer capabilities?". We studied this question by proposing several approaches combining intuitive user controls and procedural generation, gradually increasing their interactive aspect:

- We combined intuitive parameters with procedural generation to design villages on arbitrary landscapes. The main interest of our method is that it enables the emergence of complex village patterns adapted to strong environment constraints while remaining easy to use. How-

ever, while powerful and flexible, the indirect control of the modeling process hinders the creation of a particular scene.

- We combined interactive coarse editing with procedural generation of details to design coherent waterfall sceneries. The user creates a coarse waterfall network on an existing terrain, while the algorithm automatically generates a detailed scenery and deals with the environment constraints and the hydraulic properties of the network. This work helps to show the importance and the need for tools that provide direct control over what an artist would like to design, and that automate the long and laborious tasks.

- We combined interactive sketch-based editing with procedural deformation to edit complex terrains. The method enables artists to draw a detailed sketch of the desired mountain silhouette from a first-person viewpoint, and the procedural algorithm deforms the existing terrain to match these constraints. This approach highlights the need of methods to edit complex sceneries with intuitive fine controls, often managing 3D intentions from 2D manipulations from the current viewpoint.

- Finally, we combined interactive painting with example-based analysis and synthesis to edit virtual maps. The user can edit scenes composed of object distributions and graphs with tools inspired from traditional painting software. This method emphasizes the need for generic editing tools, truly coupling the simplicity and controllability of classical manual editing tools with the power of procedural and example-based modeling.

Recommendations for Future Work

To conclude this thesis, we outline in this section some recommendations for future work to continue improving on the interactive design of virtual worlds to better match user needs and computer capabilities.

Interactivity First. Interactivity is often considered as a feature of a procedural method. Instead of creating a new procedural method for modeling virtual world elements and then trying to make it interactive, we started from classical interactive scene editing methods and introduced example-based synthesis to enhance the editing experience. We believe virtual world modeling methods should be intrinsically based on intuitive and interactive editing, such as sketch-based or painting interfaces, and that the procedural model should be built on top of it.

Multi-scale Interactive Modeling. The user should not have to choose between coarse or fine controls. Interactive modeling methods should focus on multi-scale editing approaches enabling the editing at several levels of granularity and to smoothly transition between them according to user intentions.

Iterative Procedural Modeling. A method coupling interactive editing and procedural generation needs to face the problem of preserving previous or concurrent edits from the user. For instance, if the user manually edits the scene and then changes some procedural parameters, a scene that is completely regenerated would lose all previous modifications. Ideally, interactive modeling methods should handle both manual and procedural iterative modifications.

Generic Interactive Inverse Procedural Modeling. While example-based synthesis techniques enable the editing of arbitrary distributions and graphs, it is limited to structures and distributions that can be statistically analyzed and generated. If we cannot extract appropriate statistics, or more generally, properties, no procedural methods will be able to generate them. Interactive methods should propose inverse procedural techniques as generic as possible, but also be interactive so the user can better specify his intents to handle the complexity and variety of virtual worlds.

Artist-centered Design. Interactive editing methods should provide users with advanced tools enabling them to focus on what they want, and should use algorithms to generate the missing elements while ensuring global consistency. This goal can only be achieved by focusing research on interactive procedural generation and by always putting the artist at the center of the process.

The domain of interactive design of virtual worlds will continue to be a major research topic in the coming years, because of artists' increasing needs. We believe that methods coupling procedural modeling with intuitive user control are an answer to these needs.

Appendix A

User Study

In this appendix, we detail the pre-research user study about the deformation and synthesis of vectorial scenes, conducted to guide our algorithm choices in Chapter 6. This user-study allowed us to validate the set of interactions to analyze and guided the interaction matrix and the synthesis algorithm. The remaining of this appendix is the user study as it has been filled by 19 persons from various genders, ages, and artistic skills.

User Study
Return to Arnaud, IMAGINE TEAM, INRIA

Complete the following informations :
Age : .....
Sex : .....
Expertise : (1) (2) (3) (4) (5 - Expert)

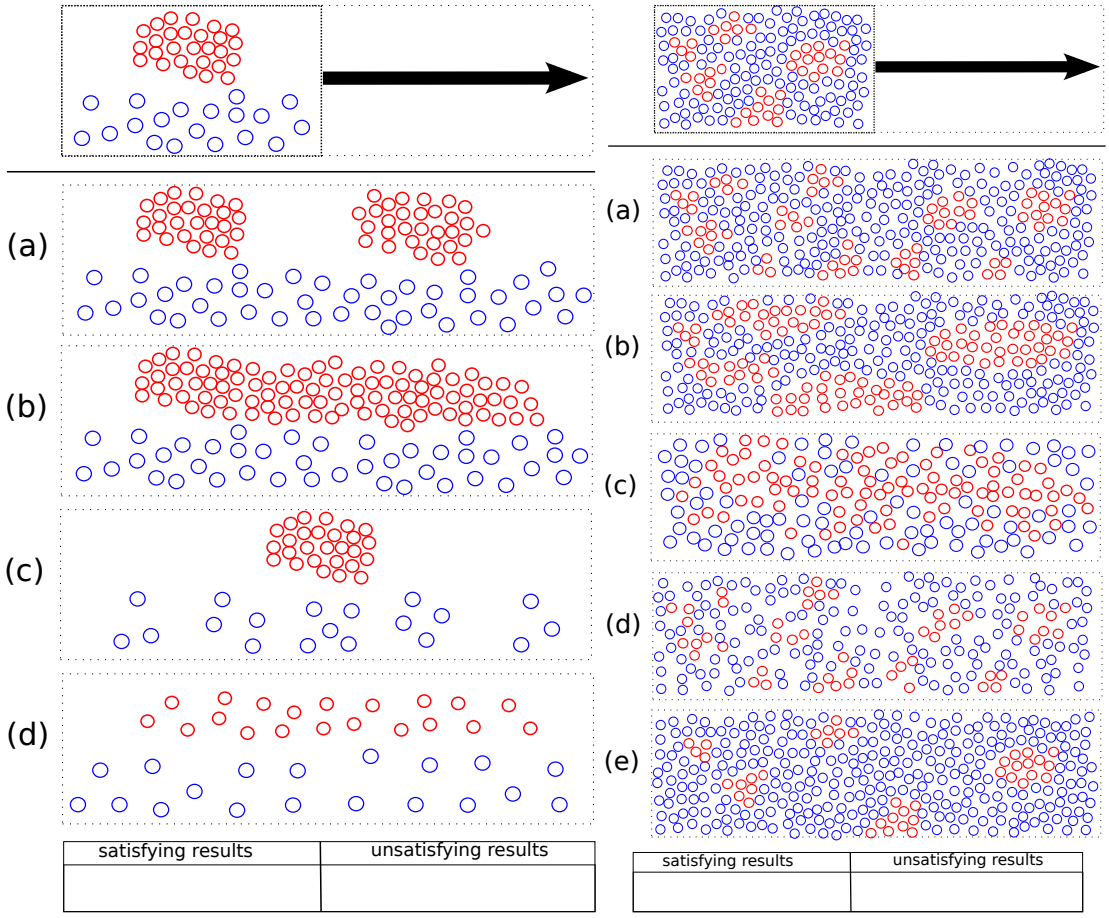
A.1 Choose the Best Deformation

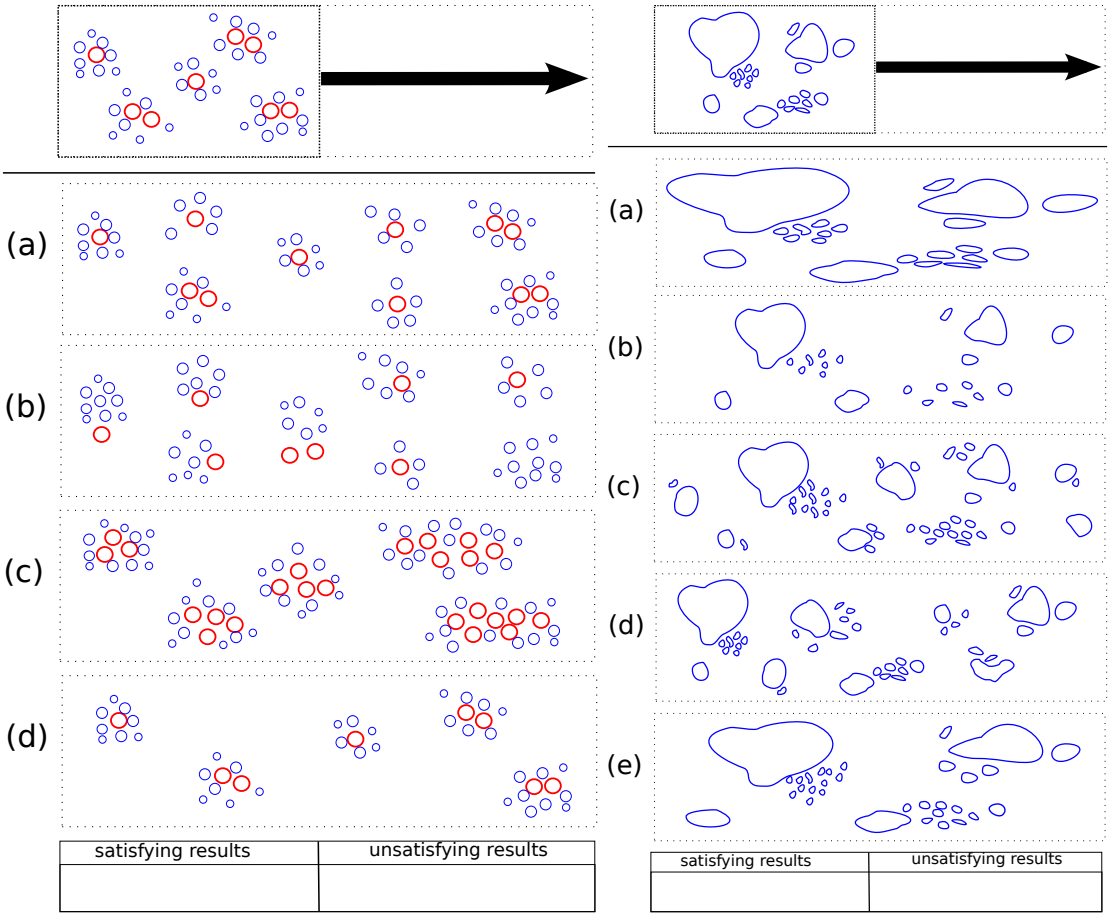
Situation: The top box contains a vectorial scene that you have just created by hand. Unfortunately the scene is too small, you have to make a larger scene.

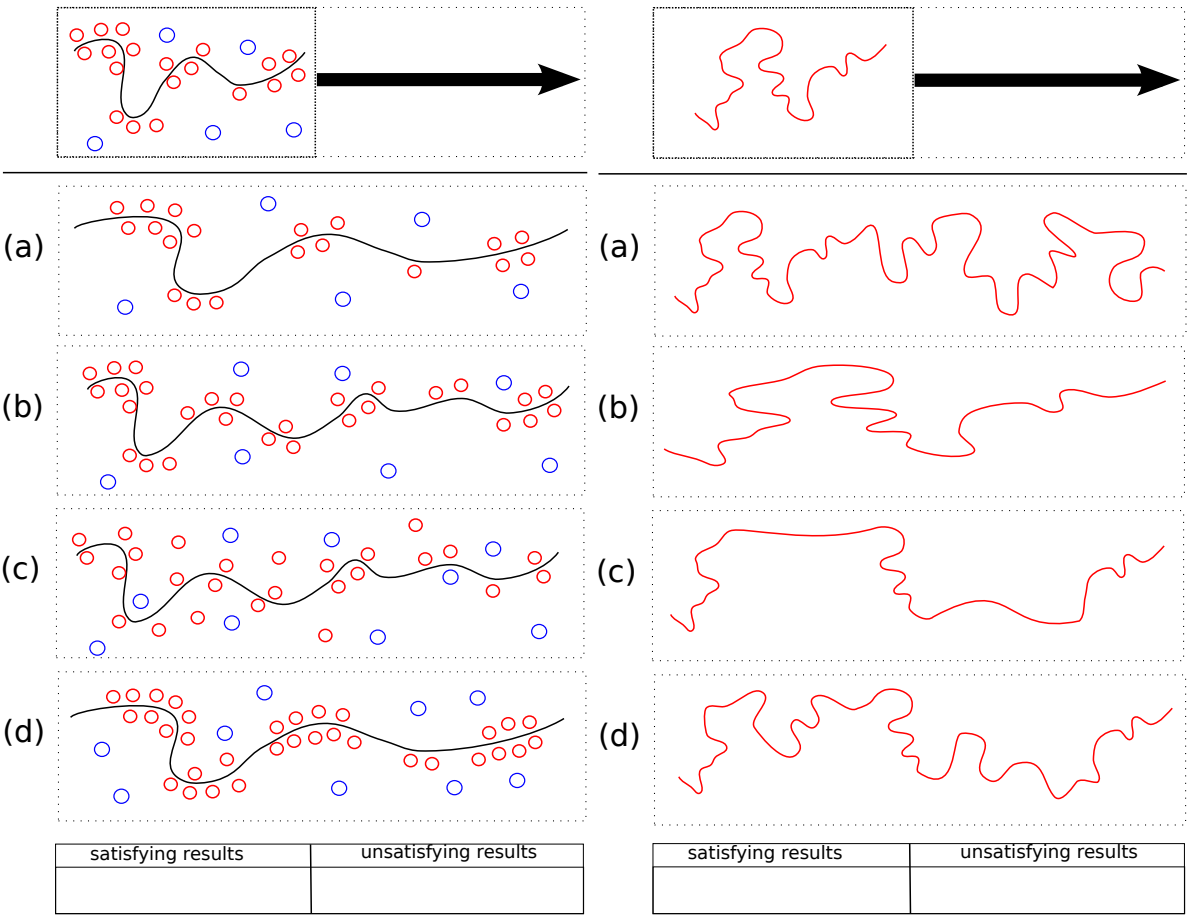
The boxes a,d,c,d and (sometimes) e represent the result given by a smart deformation tool, Sort them by preference, **as you would like a tool to deform your scene**, or list them in "non satisfying results" if you think they are not satisfying. If you think some results x and y are equivalent, write them (x,y).

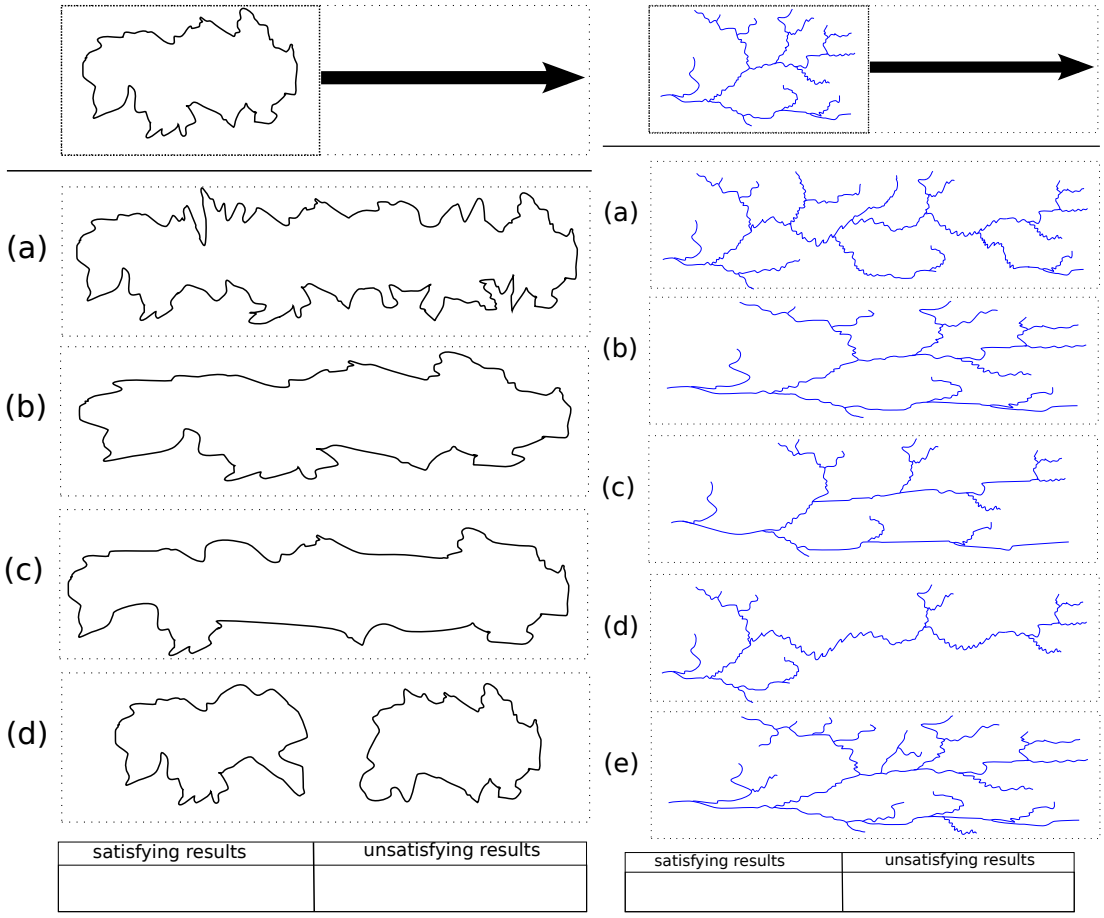
Example :

Satisfying results	Unsatisfying results
a, (d,c)	b, e


(a)
(b)
(c)
(d)
satisfying results
unsatisfying results
(a)
(b)
(c)
(d)
(e)
satisfying results
unsatisfying results

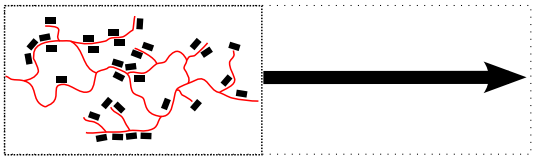

(a)
(b)
(c)
(d)
satisfying results
unsatisfying results
(a)
(b)
(c)
(d)
(e)
satisfying results
unsatisfying results


(a)
(b)
(c)
(d)
(a)
(b)
(c)
(d)
satisfying results
unsatisfying results
satisfying results
unsatisfying results

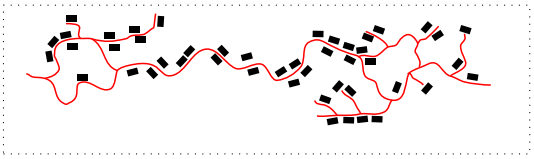


satisfying results	unsatisfying results

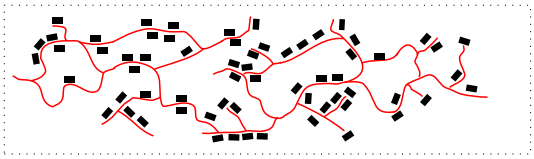
satisfying results	unsatisfying results



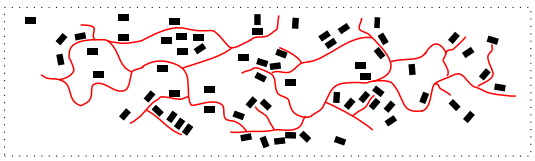
(a)



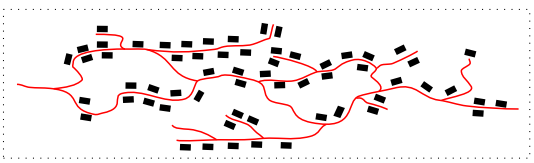
(b)



(c)



(d)

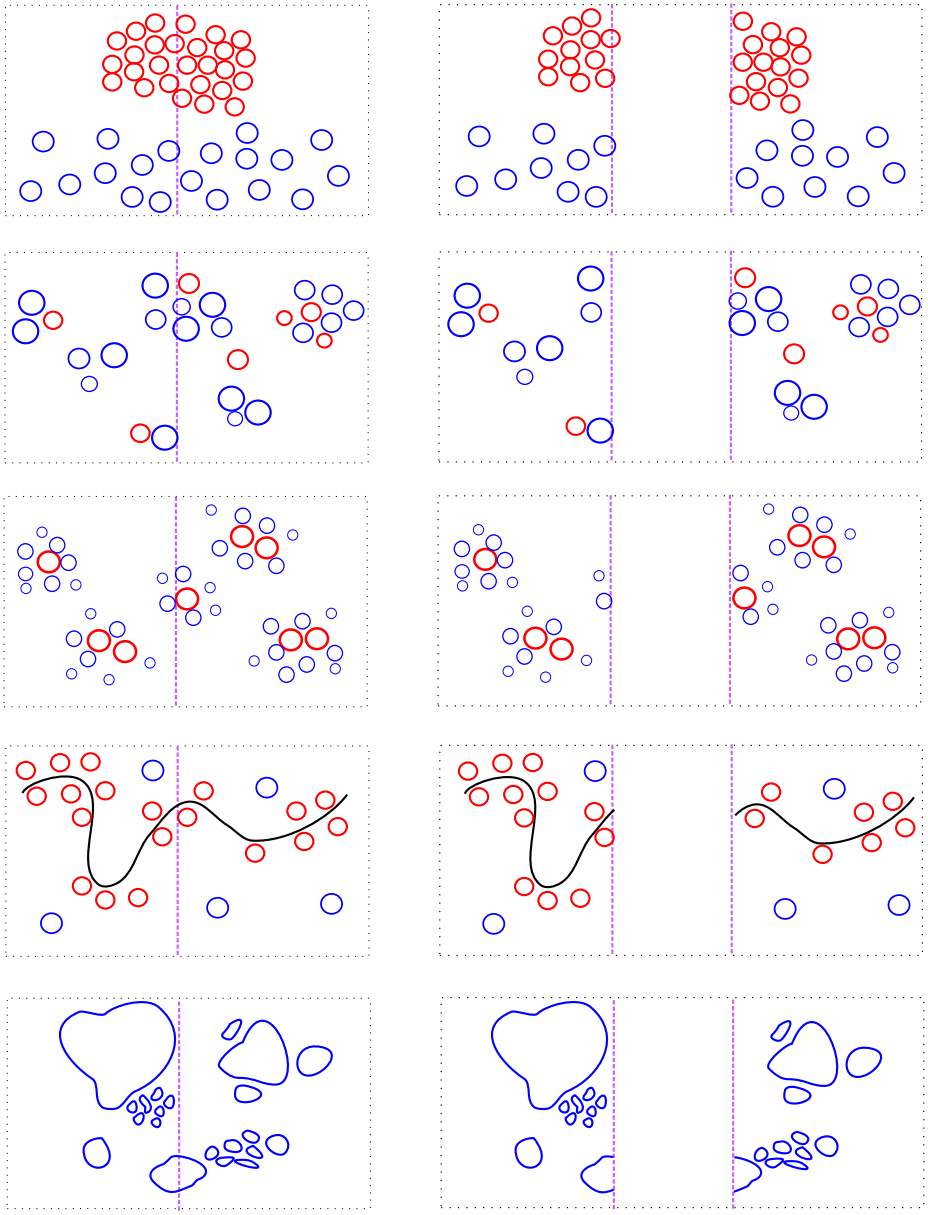


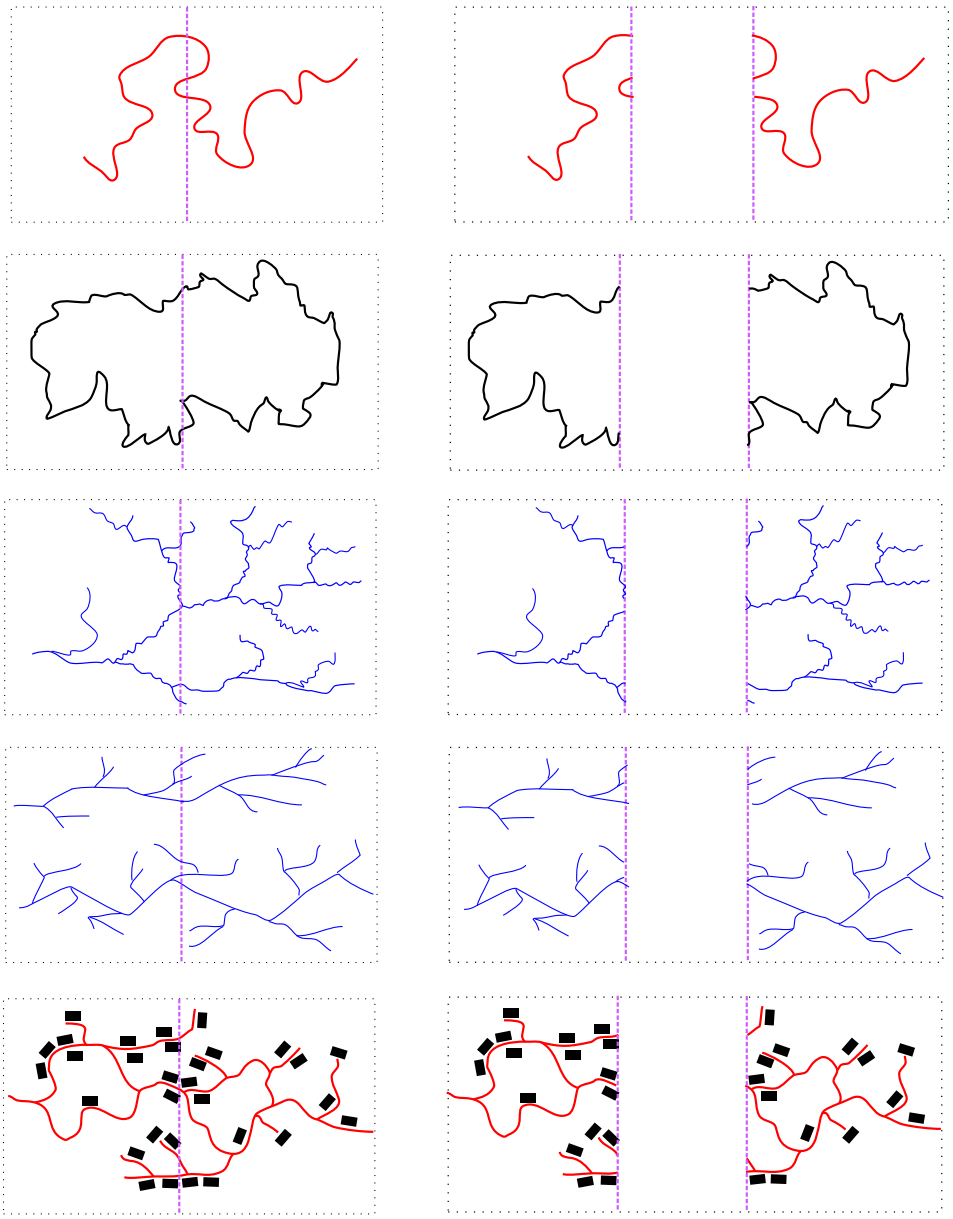
satisfying results	unsatisfying results

A.2 Fill the Holes

Situation: The boxes on the left contain vectorial scenes that you have just created by hand. The scene has been cut into two parts, and you have to fill the created holes.

Fill the holes of the boxes on the right as you would like them be filled.



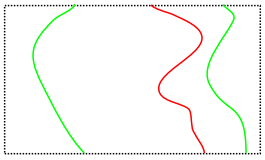


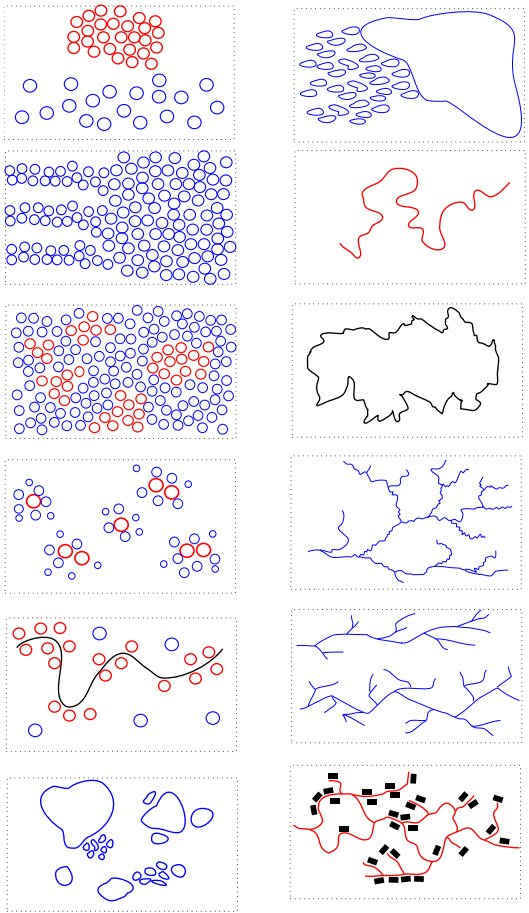
A.3 Cut the Scenes

Situation: The boxes contain vectorial scenes that you have just created by hand. You want to enlarge them by cutting them in two parts and filling the holes, like you have done in the previous section.

1 - Draw a line that cuts the boxes **vertically**, where you would cut yourself the boxes if you had to separate the scenes to enlarge the scene.
→ The line is not necessary a straight line.
→ You can cut (or not) the elements.
→ The parts are **not necessarily balanced**.

2 - Draw two other **vertical** lines (with another color) to cut again the two parts you just created.

→ Your lines shall not cross the first line. Example : 

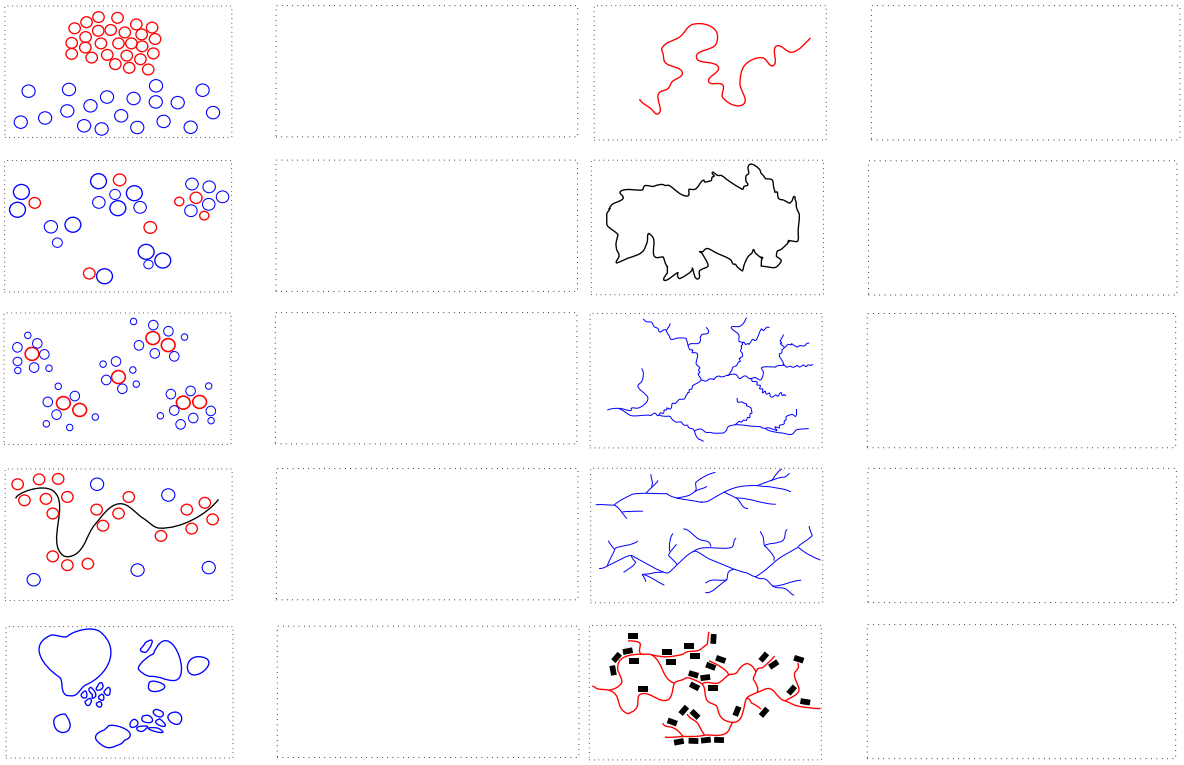


A.4 Deform a Scene

This section allows you to specify your requirements, especially if you where not satisfied by the results proposed in Section A.1.

You can deform the scene freely, to match the box on the right size. For instance you can:

- preserve what you would like to be preserved,
- preserve, stretch, modify, cut, complete, duplicate (or not) existing elements,
- **create (or not)** new elements.



Bibliography

[ABVA08] Aliaga D. G., Beneš B., Vanegas C. A., Andrysco N.: Interactive reconfiguration of urban layouts. *IEEE Computer Graphics and Applications 28*, 3 (2008), 38–47.

[AS07] Avidan S., Shamir A.: Seam carving for content-aware image resizing. *ACM Transactions on Graphics (SIGGRAPH) 26*, 3 (2007).

[Ash01] Ashikhmin M.: Synthesizing natural textures. In *Proceedings of the Symposium on Interactive 3D Graphics (I3D)* (2001), ACM, pp. 217–226.

[AVB08] Aliaga D. G., Vanegas C. A., Beneš B.: Interactive example-based urban layout synthesis. *ACM Transactions on Graphics (SIGGRAPH Asia) 27*, 5 (2008), 160:1–160:10.

[BA05] Beneš B., Arriaga X.: Table mountains by virtual erosion. In *Proceedings of the Eurographics Workshop on Natural Phenomena (NPH)* (2005), pp. 33–40.

[Bar99] Barry T. (Ed.): *A story of settlement in Ireland*. Routledge, 1999.

[BAŠ09] Beneš B., Andrysco N., Šťava O.: Interactive Modeling of Virtual Ecosystems. In *Proceedings of the Eurographics Workshop on Natural Phenomena (NPH)* (2009), pp. 9–16.

[BBT*06] Barla P., Breslav S., Thollot J., Sillion F., Markosian L.: Stroke pattern analysis and synthesis. *Computer Graphics Forum (Eurographics) 25*, 3 (2006), 663–671.

[BdBG10] Bangay S., de Bruyn D., Glass K.: Minimum spanning trees for valley and ridge characterization in digital elevation maps. In *Proceedings of the International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa (AFRIGRAPH)* (2010), ACM, pp. 73–82.

[Bei06] Beisel Jr R. H.: *International Waterfall Classification System*. Outskirts Press, 2006.

[BF01] Beneš B., Forsbach R.: Layered data representation for visual simulation of terrain erosion. In *Proceedings of the Spring Conference on Computer Graphics (SCCG)* (2001), IEEE Computer Graphics and Applications, pp. 80–86.

[BF02] Beneš B., Forsbach R.: Visual simulation of hydraulic erosion. In *WSCG Proceedings of the International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision* (2002), pp. 120–132.

[Ble14] BlenderFoundation: Blender, 2014.

[BMV*11] Bernhardt A., Maximo A., Velho L., Hnaidi H., Cani M.-P.: Real-time terrain modeling using cpu-gpu coupled computation. In *Proceedings of the Conference on Graphics, Patterns and Images (SIBGRAPI)* (2011), IEEE, pp. 64–71.

[BN08] Bruneton E., Neyret F.: Real-time Rendering and Editing of Vector-based Terrains. *Computer Graphics Forum (Eurographics) 27*, 2 (2008), 311–320.

[BSHK04] Bhat K. S., Seitz S. M., Hodgins J. K., Khosla P. K.: Flow-based video synthesis and editing. *ACM Transactions on Graphics (SIGGRAPH) 23*, 3 (2004), 360–363.

[BŠMM11] Beneš B., Šťava O., Měch R., Miller G.: Guided ProceduralModeling. *Computer Graphics Forum (Eurographics) 30*, 2 (2011), 325–334.

[BSW10] Bagar F., Scherzer D., Wimmer M.: A Layered Particle-Based Fluid Model for Real-Time Rendering of Water (EGSR). *Computer Graphics Forum 29*, 4 (2010), 1383–1389.

[BTHB06] Beneš B., Těšínskỳ V., Hornyš J., Bhatia S. K.: Hydraulic erosion. *Computer Animation and Virtual Worlds 17*, 2 (2006), 99–108.

[BW06] Bokeloh M., Wand M.: Hardware accelerated multi-resolution geometry synthesis. In *Proceedings of the Symposium on Interactive 3D Graphics and Games (I3D)* (2006), ACM, pp. 191–198.

[BWKS11] Bokeloh M., Wand M., Koltun V., Seidel H.-P.: Pattern-aware shape deformation using sliding dockers. *ACM Transactions on Graphics (SIGGRAPH Asia) 30*, 6 (2011), 123:1–123:10.

[BWS10] Bokeloh M., Wand M., Seidel H.-P.: A connection between partial symmetry and inverse procedural modeling. *ACM Transactions on Graphics (SIGGRAPH) 29*, 4 (2010), 104:1–104:10.

[BWSK12] Bokeloh M., Wand M., Seidel H.-P., Koltun V.: An algebraic model for parameterized shape editing. *ACM Transactions on Graphics (SIGGRAPH) 31*, 4 (2012), 78:1–78:10.

[CEW*08] Chen G., Esch G., Wonka P., Müller P., Zhang E.: Interactive procedural street modeling. *ACM Transactions on Graphics (SIGGRAPH) 27*, 3 (2008), 103:1–103:10.

[CHZ00] Cohen J. M., Hughes J. F., Zeleznik R. C.: Harold: A world made of drawings. In *Proceedings of the International Symposium on Non-photorealistic Animation and Rendering (NPAR)* (2000), ACM, pp. 83–90.

[Cla97] Clausen J.: Branch and bound algorithms-principles and examples. *Parallel Computing in Optimization* (1997), 239–267.

[CMF98] Chiba N., Muraoka K., Fujita K.: An erosion model based on velocity fields for the visual simulation of mountain scenery. *Journal of Visualization and Computer Animation 9*, 4 (1998), 185–194.

[CNX*08] Chen X., Neubert B., Xu Y.-Q., Deussen O., Kang S. B.: Sketch-based tree modeling using markov random field. *ACM Transactions on Graphics (SIGGRAPH Asia) 27*, 5 (2008), 109:1–109:9.

[Cry09] Crytek: Cryengine 3, 2009.

[CS07] Chang Y.-C., Sinha G.: A visual basic program for ridge axis picking on dem data using the profile-recognition and polygon-breaking algorithm. *Computers and Geosciences 33*, 2 (2007), 229–237.

[DD06] Danielsson M., Danielsson K.: *Waterfall Lover's Guide Northern California*. Mountaineers Books, 2006.

[DGA] Desbenoit B., Galin E., Akkouche S.: Simulating and modeling lichen growth. *Computer Graphics Forum (Eurographics) 23*, 3, 341–350.

[DGGK11] Derzapf E., Ganster B., Guthe M., Klein R.: River Networks for Instant Procedural Planets. *Computer Graphics Forum (Pacific Graphics) 30*, 7 (2011), 2031–2040.

[DHL*98] Deussen O., Hanrahan P., Lintermann B., Měch R., Pharr M., Prusinkiewicz P.: Realistic modeling and rendering of plant ecosystems. In *SIGGRAPH Comput. Graph.* (1998), ACM, pp. 275–286.

[DIP14] Dupuy J., Iehl J.-C., Poulin P.: *GPU Pro 5*. 2014, ch. Quadtrees on the GPU.

[DK14] Dekkers E., Kobbelt L.: Geometry seam carving. *Computer-Aided Design 46*, 0 (2014), 120–128.

[DMLG02] Dischler J.-M., Maritaud K., Lévy B., Ghazanfarpour D.: Texture particles. *Computer Graphics Forum (Eurographics) 21*, 3 (2002), 401–410.

[dPI13] dos Passos V. A., Igarashi T.: Landsketch: A first person point-of-view example-based terrain modeling approach. In *Proceedings of the International Symposium on Sketch-Based Interfaces and Modeling (SBIM)* (2013), ACM, pp. 61–68.

[dREF*88] de Reffye P., Edelin C., Françon J., Jaeger M., Puech C.: Plant models faithful to botanical structure and development. *SIGGRAPH Comput. Graph. 22*, 4 (1988), 151–158.

[DZPZ09] Dong W., Zhou N., Paul J.-C., Zhang X.: Optimized image resizing using seam carving and scaling. *ACM Transactions on Graphics (SIGGRAPH Asia) 28*, 5 (2009), 125:1–125:10.

[EBP*12] Emilien A., Bernhardt A., Peytavie A., Cani M.-P., Galin E.: Procedural generation of villages on arbitrary terrains. *The Visual Computer (CGI) 28*, 6-8 (2012), 809–818.

[EMP*02] Ebert D. S., Musgrave F. K., Peachey D., Perlin K., Worley S.: *Texturing and Modeling: A Procedural Approach*, 3rd ed. Morgan Kaufmann, 2002.

[EPCV13] Emilien A., Poulin P., Cani M.-P., Vimont U.: Design de cascades réalistes : une méthode pour combiner contrôle interactif et modèle procédural. In *AFIG 2013 - Journées de l'Association Française d'Informatique Graphique* (2013).

[EPCV14a] Emilien A., Poulin P., Cani M.-P., Vimont U.: Design de cascades réalistes : une méthode pour combiner contrôle interactif et modèle procédural. *Revue Electronique Francophone d'Informatique Graphique 8*, 1 (2014), 33–44.

[EPCV14b] Emilien A., Poulin P., Cani M.-P., Vimont U.: Interactive procedural modeling of coherent waterfall scenes. *Computer Graphics Forum* (2014). to appear.

[Fat11] Fattal R.: Blue-noise point sampling using kernel density model. *ACM Transactions on Graphics (SIGGRAPH) 30*, 4 (2011), 48:1–48:12.

[FFC82] Fournier A., Fussell D., Carpenter L.: Computer rendering of stochastic models. *Communications of the ACM 25*, 6 (1982), 371–384.

[GCZ*06] Guan Y., Chen W., Zou L., Zhang L., Peng Q.: Modeling and rendering of realistic waterfall scenes with dynamic texture sprites. *Computer Animation and Virtual Worlds 17*, 5 (2006), 573–583.

[Gei07] Geiss R.: Generating complex procedural terrains using the GPU. *GPU Gems 3* (2007), 7–37.

[GGG*13] Génevaux J.-D., Galin E., Guérin E., Peytavie A., Beneš B.: Terrain generation using procedural models based on hydrology. *ACM Transactions on Graphics (SIGGRAPH) 32*, 4 (2013), 143:1–143:13.

[GM01] Gamito M. N., Musgrave F. K.: Procedural landscapes with overhangs. In *Portuguese Computer Graphics Meeting* (2001), vol. 2.

[GMB06] Glass K. R., Morkel C., Bangay S. D.: Duplicating road patterns in south african informal settlements using procedural techniques. In *Proceedings of the International Conference on Computer Graphics, virtual reality, visualisation and interaction in Africa (AFRIGRAPH)* (2006), ACM, pp. 161–169.

[GMS09] Gain J., Marais P., Strasser W.: Terrain sketching. In *Proceedings of the Symposium on Interactive 3D Graphics and Games (I3D)* (2009), ACM, pp. 31–38.

[GPGB11] Galin E., Peytavie A., Guérin E., Beneš B.: Authoring Hierarchical Road Networks. *Computer Graphics Forum (Eurographics) 30*, 7 (2011), 2021–2030.

[GPMG10] Galin E., Peytavie A., Maréchal N., Guérin E.: Procedural Generation of Roads. *Computer Graphics Forum (Eurographics) 29*, 2 (2010), 429–438.

[HB95] Heeger D. J., Bergen J. R.: Pyramid-based texture analysis/synthesis. In *SIGGRAPH Comput. Graph.* (1995), ACM, pp. 229–238.

[HGA*10] Hnaidi H., Guérin E., Akkouche S., Peytavie A., Galin E.: Feature Based Terrain Generation Using Diffusion Equation. *Computer Graphics Forum (Pacific Graphics) 29*, 7 (2010), 2179–2186.

[HLT*09] Hurtut T., Landes P.-E., Thollot J., Gousseau Y., Drouillhet R., Coeurjolly J.-F.: Appearance-guided synthesis of element arrangements by example. In *Proceedings of the International Symposium on Non-Photorealistic Animation and Rendering (NPAR)* (2009), ACM, pp. 51–60.

[Hur10] Hurtut T.: 2d artistic images analysis, a content-based survey.

[HW04] Holmberg N., Wünsche B. C.: Efficient modeling and rendering of turbulent water over natural terrain. In *Proceedings of the International Conference on Computer Graphics and interactive techniques in Australasia and South East Asia (GRAPHITE)* (2004), ACM, pp. 15–22.

[IMH05] Igarashi T., Moscovich T., Hughes J. F.: As-rigid-as-possible shape manipulation. *ACM Transactions on Graphics (SIGGRAPH) 24*, 3 (2005), 1134–1141.

[IMIM08] Ijiri T., Měch R., Igarashi T., Miller G.: An Example-based Procedural System for Element Arrangement. *Computer Graphics Forum (Eurographics) 27*, 2 (2008), 429–436.

[JHH10] Jenny B., Hutzler E., Hurni L.: Point pattern synthesis. *The Cartographic Journal 47*, 3 (2010), 257–261.

[JJCH11] Jenny H., Jenny B., Cartwright W. E., Hurni L.: Interactive local terrain deformation inspired by hand-painted panoramas. *The Cartographic Journal 48*, 1 (2011), 11–20.

[KBKŠ09] Krištof P., Beneš B., Kriváne k J., Šťava O.: Hydraulic Erosion Using Smoothed Particle Hydrodynamics. *Computer Graphics Forum (Eurographics) 28*, 2 (2009), 219–228.

[KH06] Karpenko O. A., Hughes J. F.: Smoothsketch: 3D free-form shapes from complex sketches. *ACM Transactions on Graphics (SIGGRAPH) 25*, 3 (2006), 589–598.

[KIZD12] Kazi R. H., Igarashi T., Zhao S., Davis R.: Vignette: Interactive texture design and manipulation with freeform gestures for pen-and-ink illustration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)* (2012), ACM, pp. 1727–1736.

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

[KMN88] Kelley A. D., Malin M. C., Nielson G. M.: Terrain simulation using a model of stream erosion. *SIGGRAPH Comput. Graph. 22*, 4 (1988), 263–268.

[KSSCO08] Kraevoy V., Sheffer A., Shamir A., Cohen-Or D.: Non-homogeneous resizing of complex models. *ACM Transactions on Graphics (SIGGRAPH Asia) 27*, 5 (2008), 111:1–111:9.

[KW11] Kelly T., Wonka P.: Interactive architectural modeling with procedural extrusions. *ACM Transactions on Graphics 30* (2011), 14:1–14:15.

[LBW*14] Lu J., Barnes C., Wan C., Asente P., Měch R., Finkelstein A.: Decobrush: Drawing structured decorative patterns by example. *ACM Transactions on Graphics (SIGGRAPH) 33*, 4 (2014), 90:1–90:9.

[LD08] Lagae A., Dutré P.: A Comparison of Methods for Generating Poisson Disk Distributions. *Computer Graphics Forum 27*, 1 (2008), 114–129.

[Lew87] Lewis J. P.: Generalized stochastic subdivision. *ACM Transactions on Graphics 6*, 3 (1987), 167–190.

[LGH13] Landes P.-E., Galerne B., Hurtut T.: A Shape-Aware Model for Discrete Texture Synthesis. *Computer Graphics Forum (EGSR) 32*, 4 (2013), 67–76.

[LH04] Losasso F., Hoppe H.: Geometry clipmaps: Terrain rendering using nested regular grids. *ACM Transactions on Graphics (SIGGRAPH) 23*, 3 (2004), 769–776.

[LHP11] Leblanc L., Houle J., Poulin P.: Component-based modeling of complete buildings. In *Proceedings of Graphics Interface* (2011), pp. 87–94.

[Lin68] Lindenmayer A.: Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of Theoretical Biology 18*, 3 (1968), 280–299.

[LMS11] Löffler F., Müller A., Schumann H.: Real-time rendering of stack-based terrains. In *Vision, Modeling & Visualization* (2011), pp. 161–168.

[LRBP12] Longay S., Runions A., Boudon F., Prusinkiewicz P.: Treesketch: interactive procedural modeling of trees on a tablet. In *Proceedings of the Eurographics Symposium on sketch-based interfaces and modeling (SBIM)* (2012), pp. 107–120.

[LS12] Löffler F., Schumann H.: Generating smooth high-quality isosurfaces for interactive modeling and visualization of complex terrains. In *Vision, Modeling & Visualization* (2012), pp. 79–86.

[LSWW11] Lipp M., Scherzer D., Wonka P., Wimmer M.: Interactive Modeling of City Layouts using Layers of Procedural Content. *Computer Graphics Forum (Eurographics) 30*, 2 (2011), 345–354.

[LWSF10] Li H., Wei L.-Y., Sander P. V., Fu C.-W.: Anisotropic blue noise sampling. *ACM Transactions on Graphics (SIGGRAPH Asia) 29*, 6 (2010), 167:1–167:12.

[LWW08] Lipp M., Wonka P., Wimmer M.: Interactive visual editing of grammars for procedural architecture. *ACM Transactions on Graphics (SIGGRAPH) 27*, 3 (2008), 102:1–102:10.

[Man83] Mandelbrot B. B.: *The fractal geometry of nature*. W. H. Freeman, 1983.

[Mil86] Miller G. S. P.: The definition and rendering of terrain maps. *SIGGRAPH Comput. Graph. 20*, 4 (1986), 39–48.

[MKM89] Musgrave F. K., Kolb C. E., Mace R. S.: The synthesis and rendering of eroded fractal terrains. *SIGGRAPH Comput. Graph. 23*, 3 (1989), 41–50.

[MNB*14] Milliez A., Noris G., Baran I., Coros S., Cani M.-P., Nitti M., Marra A., Gross M., Sumner R. W.: Hierarchical motion brushes for animation instancing. In *Proceedings of the Workshop on Non-Photorealistic Animation and Rendering (NPAR)* (2014), ACM, pp. 71–79.

[MP96] Měch R., Prusinkiewicz P.: Visual models of plants interacting with their environment. In *SIGGRAPH Comput. Graph.* (1996), ACM, pp. 397–410.

[MWCS13] Milliez A., Wand M., Cani M.-P., Seidel H.-P.: Mutable Elastic Models for Sculpting Structured Shapes. *Computer Graphics Forum (Eurographics) 32*, 2 (2013), 21–30.

[MWH*06] Müller P., Wonka P., Haegler S., Ulmer A., Van Gool L.: Procedural modeling of buildings. *ACM Transactions on Graphics (SIGGRAPH) 25*, 3 (2006), 614–623.

[Nag97] Nagashima K.: Computer generation of eroded valley and mountain terrains. *The Visual Computer 13*, 9-10 (1997), 456–464.

[NWD05] Neidhold B., Wacker M., Deussen O.: Interactive physically based fluid and erosion simulation. *Proceedings of the Eurographics Workshop on Natural Phenomena (NPH)* (2005), 25–32.

[OBW*08] Orzan A., Bousseau A., Winnemöller H., Barla P., Thollot J., Salesin D.: Diffusion curves: A vector representation for smooth-shaded images. *ACM Transactions on Graphics (SIGGRAPH) 27*, 3 (2008), 92:1–92:8.

[OG12] Öztireli A. C., Gross M.: Analysis and synthesis of point distributions based on pair correlation. *ACM Transactions on Graphics (SIGGRAPH Asia) 31*, 6 (2012), 170:1–170:10.

[OSSJ09] Olsen L., Samavati F. F., Sousa M. C., Jorge J. A.: Sketch-based modeling: A survey. *Computers & Graphics 33*, 1 (2009), 85–103.

[Per85] Perlin K.: An image synthesizer. *SIGGRAPH Comput. Graph. 19*, 3 (1985), 287–296.

[PGGM09] Peytavie A., Galin E., Grosjean J., Merillou S.: Arches: a Framework for Modeling Complex Terrains. *Computer Graphics Forum (Eurographics) 28*, 2 (2009), 457–467.

[PHM93] Prusinkiewicz P., Hammel M. S., Mjolsness E.: Animation of plant development. In *SIGGRAPH Comput. Graph.* (1993), ACM, pp. 351–360.

[PLH*90] Prusinkiewicz P., Lindenmayer A., Hanan J. S., Fracchia F. D., Fowler D. R., de Boer M. J., Mercer L.: *The algorithmic Beauty of Plants*. Springer, 1990.

[Plu05] Plumb G. A.: *Waterfall Lover's Guide Pacific Northwest*. Mountaineers Books, 2005.

[PM01] Parish Y. I. H., Müller P.: Procedural modeling of cities. In *SIGGRAPH Comput. Graph.* (2001), ACM, pp. 301–308.

[PM13] Pytel A., Mann S.: Self-organized approach to modeling hydraulic erosion features. *Computers & Graphics 37*, 4 (2013), 280–292.

[PTMG08] Peyrat A., Terraz O., Merillou S., Galin E.: Generating vast varieties of realistic leaves with parametric 2Gmap L-systems. *The Visual Computer 24*, 7-9 (2008), 807–816.

[Rea99] Read A. L.: Linear interpolation of histograms. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment 425*, 1 (1999), 357–360.

[RPP93] Roudier P., Peroche B., Perrin M.: Landscapes synthesis achieved through erosion and deposition process simulation. *Computer Graphics Forum (Eurographics) 12*, 3 (1993), 375–383.

[SBC*11] Solenthaler B., Bucher P., Chentanez N., Müller M., Gross M.: SPH based shallow water simulation. In *Proceedings on the Eurographics Workshop in Virtual Reality Interactions and Physical Simulation* (2011), pp. 39–46.

[ŠBM*10] Šťava O., Beneš B., Měch R., Aliaga D. G., Krištof P.: Inverse Procedural Modeling by Automatic Generation of L-systems. *Computer Graphics Forum (Eurographics) 29*, 2 (2010), 665–674.

[SDKT*09] Smelik R. M., De Kraker K. J., Tutenel T., Bidarra R., Groenewegen S. A.: A survey of procedural methods for terrain modelling. In *Proceedings of the CASA Workshop on 3D Advanced Media In Gaming And Simulation (3AMIGAS)* (2009), pp. 25–34.

[SDZ*07] Sakaguchi R., Dufor T., Zalzala J., Lambert P., Kapler A.: End of the world waterfall setup for "Pirates of the Caribbean 3". In *SIGGRAPH Sketches* (2007), ACM.

[SPK10] Sibbing D., Pavic D., Kobbelt L.: Image Synthesis for Branching Structures. *Computer Graphics Forum (Pacific Graphics) 29*, 7 (2010), 2135–2144.

[ŠPK*14] Šťava O., Pirk S., Kratt J., Chen B., Měch R., Deussen O., Beneš B.: Inverse Procedural Modelling of Trees. *Computer Graphics Forum 33*, 6 (2014), 118–131.

[Sta99] Stam J.: Stable fluids. In *SIGGRAPH Comput. Graph.* (1999), ACM, pp. 121–128.

[STBB14] Smelik R. M., Tutenel T., Bidarra R., Beneš B.: A Survey on Procedural Modelling for Virtual Worlds. *Computer Graphics Forum 33*, 6 (2014), 31–50.

[STdKB10] Smelik R., Tutenel T., de Kraker K. J., Bidarra R.: Integrating procedural generation and manual editing of virtual worlds. In *Proceedings of the Workshop on Procedural Content Generation in Games (PCGames)* (2010), ACM, pp. 2:1–2:8.

[STdKB11] Smelik R., Tutenel T., de Kraker K., Bidarra R.: A declarative approach to procedural modeling of virtual worlds. *Computers & Graphics 35*, 2 (2011), 352 – 363.

[SWM11] Smith G., Whitehead J., Mateas M.: Tanagra: Reactive planning and constraint solving for mixed-initiative level design. *Computational Intelligence and AI in Games, IEEE Transactions on 3*, 3 (2011), 201–215.

[SYBG02] Sun J., Yu X., Baciu G., Green M.: Template-based generation of road networks for virtual city modeling. In *Proceedings of the Symposium on Virtual Reality Software and Technology (VRST)* (2002), ACM, pp. 33–40.

[SZZ*13] Sun Q., Zhang L., Zhang M., Ying X., Xin S.-Q., Xia J., He Y.: Texture brush: An interactive surface texturing interface. In *Proceedings of the Symposium on Interactive 3D Graphics and Games (I3D)* (2013), ACM, pp. 153–160.

[TEC*14a] Tasse F. P., Emilien A., Cani M.-P., Hahmann S., Bernhardt A.: First person sketch-based terrain editing. In *Proceedings of Graphics Interface* (2014), pp. 217–224.

[TEC*14b] Tasse F. P., Emilien A., Cani M.-P., Hahmann S., Dodgson N.: Feature-based terrain editing from complex sketches. *Computers & Graphics* (2014). to appear.

[Teo09] Teoh S.: Riverland: An efficient procedural modeling system for creating realistic-looking terrains. In *Advances in Visual Computing*, vol. 5875. Springer, 2009, pp. 468–479.

[TGM12] Tasse F., Gain J., Marais P.: Enhanced Texture-Based Terrain Synthesis on Graphics Hardware. *Computer Graphics Forum 31*, 6 (2012), 1959–1972.

[TLL*11] Talton J. O., Lou Y., Lesser S., Duke J., Měch R., Koltun V.: Metropolis procedural modeling. *ACM Transactions on Graphics 30*, 2 (2011), 11:1–11:14.

[VABW09] Vanegas C. A., Aliaga D. G., Beneš B., Waddell P. A.: Interactive design of urban spaces using geometrical and behavioral modeling. *ACM Transactions on Graphics (SIGGRAPH Asia) 28*, 5 (2009), 111:1–111:10.

[vBBK08] Šťava O., Beneš B., Brisbin M., Křivánek J.: Interactive terrain modeling using hydraulic erosion. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA)* (2008), pp. 201–210.

[VBHŠ11] Vanek J., Beneš B., Herout A., Šťava O.: Large-scale physics-based terrain editing using adaptive tiles on the GPU. *IEEE Computer Graphics and Applications 31*, 6 (2011), 35–44.

[VGDA*12] Vanegas C. A., Garcia-Dorado I., Aliaga D. G., Beneš B., Waddell P.: Inverse design of urban procedural models. *ACM Transactions on Graphics (SIGGRAPH Asia) 31*, 6 (2012), 168:1–168:11.

[vH11] van Hoesel F.: Tiled directional flow. In *SIGGRAPH Posters* (2011), ACM, pp. 19:1–19:1.

[VKW*12] Vanegas C. A., Kelly T., Weber B., Halatsch J., Aliaga D. G., Müller P.: Procedural Generation of Parcels in Urban Modeling. *Computer Graphics Forum (Eurographics) 31*, 2 (2012), 681–690.

[WBC08] Wither J., Bouthors A., Cani M.-P.: Rapid sketch modeling of clouds. In *Proceedings of the Eurographics Workshop on Sketch-Based Interfaces and Modeling (SBIM)* (2008), Alvarado C., Cani M.-P., (Eds.), pp. 113–118.

[WBCG09] Wither J., Boudon F., Cani M.-P., Godin C.: Structure from Silhouettes: a New Paradigm for Fast Sketch-based Design of Trees. *Computer Graphics Forum (Eurographics) 28*, 2 (2009), 541–550.

[Wei08] Wei L.-Y.: Parallel poisson disk sampling. *ACM Transactions on Graphics (SIGGRAPH) 27*, 3 (2008), 20:1–20:9.

[WI04] Watanabe N., Igarashi T.: A sketching interface for terrain modeling. In *SIGGRAPH Posters* (2004), ACM, p. 73.

[WLK*09] Wei L.-Y., Lefebvre S., Kwatra V., Turk G., et al.: State of the art in example-based texture synthesis. In *Eurographics, State of the Art Report* (2009), pp. 93–117.

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

[WWSR03] Wonka P., Wimmer M., Sillion F., Ribarsky W.: Instant architecture. *ACM Transactions on Graphics (SIGGRAPH) 22* (2003), 669–677.

[YM09] Yeh Y.-T., Měch R.: Detecting Symmetries and Curvilinear Arrangements in Vector Art. *Computer Graphics Forum (Eurographics) 28*, 2 (2009), 707–716.

[YNBH09] Yu Q., Neyret F., Bruneton E., Holzschuch N.: Scalable Real-time Animation of Rivers. *Computer Graphics Forum (Eurographics) 28*, 2 (2009), 239–248.

[YNS11] Yu Q., Neyret F., Steed A.: Feature-based vector simulation of water waves. *Computer Animation and Virtual Worlds 22*, 2-3 (2011), 91–98.

[YYW*12] Yeh Y.-T., Yang L., Watson M., Goodman N. D., Hanrahan P.: Synthesizing open worlds with constraints using locally annealed reversible jump MCMC. *ACM Transactions on Graphics (SIGGRAPH) 31*, 4 (2012), 56:1–56:11.

[ZHWW12] Zhou Y., Huang H., Wei L.-Y., Wang R.: Point sampling with general noise spectrum. *ACM Transactions on Graphics (SIGGRAPH) 31*, 4 (2012), 76:1–76:11.

[ZIH*11] Zhu B., Iwata M., Haraguchi R., Ashihara T., Umetani N., Igarashi T., Nakazawa K.: Sketch-based dynamic illustration of fluid systems. *ACM Transactions on Graphics (SIGGRAPH Asia) 30*, 6 (2011), 134:1–134:8.

[ZSTR07] Zhou H., Sun J., Turk G. b. b., Rehg J. b. b.: Terrain synthesis from digital elevation models. *IEEE Transactions on Visualization and Computer Graphics 13*, 4 (2007), 834–848.