Novel methods for the interactive design of complex objects and animations

Ulysse Vimont

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Mathématiques - Informatique**

Arrêté ministériel du 7 août 2006

Présentée par

Ulysse VIMONT

Thèse dirigée par **Marie-Paule CANI** et
co-encadrée par **Damien ROHMER**

préparée au sein du **Laboratoire Jean Kunzmann**
dans **l'École Doctorale MSTII**

Nouvelles Méthodes pour la Modélisation Interactive d'Objets Complexes et d'Animations

Thèse soutenue publiquement le **1er décembre 2016**,
devant le jury composé de :

Mme. Joëlle THOLLOT
Professeur, Grenoble INP, Présidente
M. Tamy BOUBEKEUR
Professeur, Telecom ParisTech, Rapporteur
M. Loïc BARTHE
Professeur, Université Paul Sabatier, Rapporteur
M. Michael WIMMER
Professeur Associé, Technische Universität Wien, Examinateur
Mme. Marie-Paule CANI
Professeur, Grenoble INP & Ensimag, Directeur de thèse
M. Damien ROHMER
Maître de Conférence, CPE Lyon, Co-encadrant de thèse

À ma famille, présente et future.

Abstract

As virtual content continually grows in quantity and quality, new challenges arise. Amongst others, generating and manipulating 3D shapes and animations have become intricate tasks. State of the art methods attempt to hide this complexity through complex tools, which exploit content semantics for running optimization procedures, yielding constraint matching outputs. However, the control offered by such methods is often indirect, object-specific, and heavy, which imposes long trial-and-error cycles and restrains artistic freedom.

The focus of this thesis is twofolds: First, improving user control through interactive and direct content manipulation; Second, enlarging the spectrum of manipulable content with innovative or generic content representations.

We introduce three new mehods related to 3D shapes design: A part-based modeling tool allowing to generate assembly shapes with semantic adjacency constraints; A painting tool for distributing objects in a 3D scene; And a grammar-based hierarchical deformation paradigm, enabling the interactive deformation of complex models. We also propose two methods related to the design of animated contents: A vectorial editing tool to synthesize consistent waterfall scenes; And finally a sculpting method enabling to design new liquid animation from examples.

Résumé

L'accroissement de la demande en contenu virtuel, tant en termes qualitatifs que quantitatifs, révèle de nouveaux défis scientifiques. Par exemple, la génération et la manipulation de formes 3D et d'animation sont particulièrement difficiles. Les méthodes modernes contournent ces difficultés en proposant des approches basées sur des algorithmes d'optimisation. Ces derniers utilisent des connaissances a priori sur les données à manipuler afin de générer de nouvelles données satisfaisant des contraintes dictées par l'utilisateur. De tels outils présentent le désaventage d'être indirects, couteux, et non génériques, ce qui limite la liberté artistique de l'utilisateur en le contraignant à de nombreux essais.

Les objectifs de cette thèse sont pluriels. D'une part, elle vise à améliorer le contrôle de l'utilisateur en proposant des méthodes de manipulation interactives et directes. D'une autre, elle cherche à rendre ces méthodes capables de manipuler des contenus plus variés en proposant des outils novateurs et génériques.

Plus précisément, cette thèse introduis trois méthodes de modélisation d'objets 3D. La première est une méthode basée exemple de génération d'objets composites caractérisés par l'adjacence de leurs sous-parties. La seconde propose une interface de types "peinture" pour décrire les distributions d'objets dans une scène 3D. La troisième étend le princides des grammaires génératives à la déformation d'objets hiérarchiques. Nous proposons également deux méthodes de modélisation d'animation. La première offre de modéliser des scènes natuelles de cascades grâce à des controlleurs vectoriels. La seconde permet de sculpter une animation de liquide en manipulant directement ses éléments spatio-temporels saillants.

Remerciements

Je remercie mes encadrants Marie-Paule et Damien, qui m'ont aidé dans la tâche difficile de réaliser cette thèse.

Je remercie également tout mes collaborateurs : Arnaud, Pierre-Luc, Han, Sylvain, Antoine, Niloy, Bedrich, Stefanie, Pierre, Michael, et Chris.

Je remercie tout ceux qui m'ont cotoyé pendant cette thèse, à l'Inria et ailleurs, ceux avec qui j'ai partagé mes journée be travail et souvent beaucoup plus : Guillaume, Léo, Thomas, Thomas, Aarohi, Adela, Matthieu, Quentin, Kevin, Armelle, Estelle, Laura, Even, Pablo, Gregoire, Robin, Camille, Tibor, Romain, Romain, Catherine, Sanie, Florence, Laurent, Harold, Vincent, Cedric, Moreno, Lucian, Richard, Vineet, Ali, Remi, James, Rémi, Rémi, François, Jean-Claude, Olivier, et encore beaucoup d'autres que j'oublie (ne le prenez pas mal).

Je remercie toutes les personnes avec qui j'ai partagé des moments en dehors de la thèse, colocs, voilocs et amis : Benoît, Vanessa, Hélène, Anaïs, Clement, Lydia, Nacho, Céline, Lucie, Marion, Loïc, Antoine, Ninon, Olivier, Seb, Guillaume, Guillaume, Cyrielle, Lorie, Pauline, Frank, Cassandre, Rebecca, John, Robin, Lorena, Marc, Gabi, Steff, Christophe, Thibaud, Agathe, David, Jerôme, Lorène, et encore une fois j'en oublie la moitié.

Enfin, je remercie ma famille pour son soutient constant et indéfectible, et Sandra pour absolument tout.

Contents

1 Introduction and Motivations **1**
- 1.1 What is virtual content? . . . 2
  - 1.1.1 Uses . . . 2
  - 1.1.2 Data representation . . . 3
  - 1.1.3 Creation . . . 4
- 1.2 Why is 3D shape creation a hard task? . . . 4
  - 1.2.1 Size of virtual objects . . . 5
  - 1.2.2 Shape space versus mesh space . . . 6
  - 1.2.3 Path in the mesh space . . . 7
- 1.3 The case of animation . . . 7
  - 1.3.1 Frame-based parameter interpolation . . . 7
  - 1.3.2 Simulation . . . 8
- 1.4 Overview of this thesis . . . 8
  - 1.4.1 Contributions . . . 8
  - 1.4.2 Publications . . . 9
  - 1.4.3 Outline . . . 10
  - 1.4.4 Video . . . 10

2 Previous work **11**
- 2.1 Introduction . . . 12
- 2.2 Static Objects Generation . . . 13
  - 2.2.1 Procedural modeling . . . 13
  - 2.2.2 Inverse procedural modeling . . . 17
  - 2.2.3 Sketch-based modeling . . . 17
  - 2.2.4 Example-based modeling . . . 18
  - 2.2.5 Limitations of generation methods . . . 20
- 2.3 Static Objects Deformation . . . 20
  - 2.3.1 Low-level deformation methods . . . 20
  - 2.3.2 Structure-aware shape deformation . . . 22
  - 2.3.3 Sketch-based deformation . . . 24
  - 2.3.4 Limitations of deformation methods . . . 25
- 2.4 Animation Design . . . 25
  - 2.4.1 Fluid animation modeling . . . 25
  - 2.4.2 Fluid animation control . . . 26
- 2.5 Conclusion . . . 29

3 Design of Static Objects **31**
3.1 Introduction . . . 32
3.1.1 Complex object framework . . . 32
3.1.2 Coherency . . . 34
3.1.3 Example-based modeling of complex objects . . . 35
3.2 Part-Based Modeling . . . 36
3.2.1 Introduction . . . 37
3.2.2 Overview of the solution . . . 37
3.2.3 Main contribution: Geometrical Sub-Object Deformation . . . 42
3.2.4 Results . . . 43
3.2.5 Discussion . . . 46
3.3 Worldbrush . . . 48
3.3.1 Vectorial analysis . . . 48
3.3.2 World painting . . . 50
3.3.3 Main contribution: RDF Interpolation . . . 53
3.3.4 Results . . . 55
3.3.5 Discussion . . . 55
3.4 Deformation Grammars . . . 61
3.4.1 Introduction . . . 61
3.4.2 Deformation Grammars . . . 62
3.4.3 Bilateral Grammar Rules . . . 64
3.4.4 Results . . . 67
3.4.5 Discussion . . . 71
3.5 Conclusion . . . 75
3.5.1 Controllability versus complexity trade-off . . . 75
3.5.2 Smart tools versus smart shapes . . . 75

4 Design of Animation **77**
4.1 Introduction . . . 78
4.1.1 Temporality of animations . . . 79
4.1.2 Preservative structures . . . 79
4.2 Waterfall scenes . . . 81
4.2.1 Motivation . . . 81
4.2.2 Overall system . . . 83
4.2.3 Main contribution: waterfall classification . . . 95
4.2.4 Results . . . 96
4.2.5 Discussion . . . 100
4.3 Fluid Sculpting . . . 100
4.3.1 Introduction . . . 101
4.3.2 Overview . . . 102
4.3.3 Feature extraction . . . 103
4.3.4 Feature representation . . . 109
4.3.5 Sculpting Tools . . . 110
4.3.6 Results . . . 111
4.3.7 Discussion . . . 113
4.3.8 Conclusion . . . 115
4.4 Conclusion . . . 117

5 Conclusion **119**
5.1 Summary of this thesis 120
5.2 Future work 120
5.2.1 Continuous deformations as an animation 120
5.2.2 Deformation diffusion using preservative structures 121
5.2.3 Deformation of preservative structures 121

A Bibliography **123**

B List of Figures **141**

C List of Tables **143**

D List of Algorithms **144**

Chapter 1

Introduction and Motivations

Contents

1.1 What is virtual content? 2
1.1.1 Uses 2
1.1.2 Data representation 3
1.1.3 Creation 4
1.2 Why is 3D shape creation a hard task? 4
1.2.1 Size of virtual objects 5
1.2.2 Shape space versus mesh space 6
1.2.3 Path in the mesh space 7
1.3 The case of animation 7
1.3.1 Frame-based parameter interpolation 7
1.3.2 Simulation 8
1.4 Overview of this thesis 8
1.4.1 Contributions 8
1.4.2 Publications 9
1.4.3 Outline 10
1.4.4 Video 10

RECENT years have seen the emergence of *digitization* as a major change in our customs: communication, entertainment, access to information and consumption have been impacted at large extends. One of the facets of this major trend is the increase in the demand for virtual content.This chapter explains the challenges of virtual content design, and relates those challenges to the contributions of this thesis.

1.1 What is virtual content?

In our context, we call "virtual content" the data used for representing objects and scenes in computers. It is the main data considered in the Computer Graphics field. This section describes the contexts in which this content is used, how it is represented, and the different ways to create it.

1.1.1 Uses

Virtual content aiming at describing object geometry was first developed in the domain of Computer Assisted Design (CAD), as for instance in the work of Bézier (1966) for the design of smooth curves and surface. 2D data such as Bézier curves are easily represented on a standard display such as a screen. When given 3D data, it is natural to try to visualize them as well. *Visualization* consists in transforming 3D data into 2D images. This can be performed using various methods, such as rasterization or ray tracing for example. This transformation can be done for two purposes: better understanding the underlying data, or actually getting images. The first setup is known as *scientific visualization*. It aims at representing data such that some underlying structure or associated phenomenon may appear in the clearest and most comprehensive way. It may be used for instance in cases where 3D data are dense and intricate, such as medicine or geography; or when it is not possible to see it directly, as the data produced by an electron microscope or a multi-spectral telescope. Note that in scientific visualization, the core structure is the original data, not the resulting image, therefore any modification of the data should be avoided to preserve its accuracy. In contrast, the second setup – which I call *entertainment visualization* – aims at producing the image in itself, not at understanding a specific phenomenon. This resulting image should be pleasant for the viewer, be plausible when depicting a real phenomenon, or may even drive an artistic concept. Computations may need to be interactive when dealing with creation interfaces, games, or navigation for instance. To achieve such goals, data may need to be adapted and modified before the rendering step. Note that entertainment visualization is widely spread in the special effect industry, in the area of computer-animated films, and in video games.

If not for visualization, 3D data also find their place in industrial applications such as architecture, serious games, and mechanical design. These contexts require virtual content to describe not only the shape of objects, but also other properties such as the constitutive materials and their mechanical behaviors. These data are used within numerical *simulations*, which virtually reproduce the evolution of the model under given laws prescribed by other scientific fields (e.g. mechanics). Such simulations yield results which are transposable to real objects, such as physical stress resistance.

Nowadays, real-time imaging and simulation capabilities allow interactive applications such as surgical training. This speed allows for even more intricate real-virtual relationships:

For example, *augmented reality* allows to interleave on-the-fly video streams with specific data. Such techniques are used many different contexts from video games (see Niantic (2016) for example) to medical intervention assistance (see Manescu et al. (2013) for an example of VR-assisted hadron therapy).

Fast 3D content acquisition technologies such as LIDARs allows to create virtual content from real world scenes at interactive rates. Such advances benefit previous applications and open new ones: For instance, it enables embedded computer systems to use a *virtual representation* of the world surrounding them. This permits such systems to interact with their environment for example by grasping objects or planning routes. This allows robots to be autonomous, as in self-driving cars (Lee et al. (2014)) or UAV (i.e. drones, see Tisdale et al. (2009)).

Finally, real objects can also be created from virtual content directly, for instance using *3D printing* technology. Such techniques are particularly well adapted to per-unit production scenarios such as prosthetics fabrication, as explained by Rengier et al. (2010).

Scope of this thesis. This manuscript primarily focuses on virtual content creation for entertainment visualization. More precisely, it describes several methods aiming at facilitating the generation of virtual content from an *artistic* stand point. This means that the virtual objects we will be manipulating are not associated with real object – as it would be the case for CAD for instance.

Although this context partially frees us from accuracy and realism constraints, an object model should still convey the "idea" that its creator (the 3D artist) intended. For example, considering the model of a real-life object, some constraints have to be respected for it to "look like" what it represents. This notion is further explored in Section 3.1.1. Besides, other constraints should be taken care of: Methods described here should be interactive and allow for very different object representations and interaction modes.

Now, let us see how virtual content is represented.

1.1.2 Data representation

The "virtual content" described in the previous section is a very general notion, since it fits the needs of different applications. Even when designed for entertainment visualization only, such data may represent various properties: surfacic appearance, shape, or motion for instance.

More precisely, surfacic appearance models (such as BRDFs) belong to the sub-field of rendering and are out of the scope of this thesis. We will instead focus on shapes and motions, which are themselves wide notions and can be represented using different structures: Shapes are often represented as polygonal meshes, sometimes associated with a subdivision scheme, but other useful structures exist such as implicit surfaces, NURBS, or height-fields; Motion data is most of the time encoded as animation curves prescribing changes over time of appearance or shape model parameters, up to a whole new model at each time step.

In the remainder of this thesis, we will not focus on data representation, except when it is necessary to the understanding of the contribution. Besides, most methods presented in this manuscript leverage object semantics in order to offer ways of interacting with its representation. Semantic data can usually be represented as tags associated to objects or object parts. It is generally added as a post-process on top of geometrical or animation

data. Semantic data inference is out of the scope of this thesis: In the following, unless stated, we assume that semantic information is provided as part of the input data used by the presented methods.

Let us now investigate the process of virtual content creation.

1.1.3 Creation

Virtual content creation is usually performed by skilled professional artists using modeling software such as Maya (2016), ZBrush (2016), BlenderFoundation (2014), or Neobarok (2016). Such software usually offer two main features: primitive insertion tools and shape modeling tools. Primitive insertion tools propose to add a pre-defined object model into the scene: for example canonical solids (e.g. platonic polyhedron, spheres, cones, or cylinders), or procedurally generated content (e.g. l-systems for plants or parametrized human models). Shape modeling tools enable artists to modify a shape in various ways, such as: picking an object subpart, affine transforming the selected parts, subdividing, smoothing, or assembling it with other shapes.

In practice, artists start by identifying mentally a shape to obtain, and might create some sketches for prototyping their creation. This goal can be fuzzy, allowing for artistic leeway; or precise, leaving no room for inaccuracies. In both cases, artists decide what primitive to start from, and what chain of edits will lead them to their goal given the editing tools they have. The first result is then matched against artistic requirements (e.g. "This fish should look more aggressive"). This will lead the artist to identify modifications on the shape (e.g. "Teeth should be bigger and eyes smaller"), and produce a second version of it. This iterative process is repeated until the result converges to a state where artistic requirements are matched.

In summary, the characters and the sceneries used in virtual worlds need to carry an emotional charge. It might be clear from the beginning that a character must look mean, the practical appearance of this character emerges through several iterations. The output of a production iteration might even influence the actual goal by modifying the initial requirements. Hence *artistic content creation is not a linear process.*

Starting from this assessment, **this thesis presents contributions aiming at easing the artistic creation process**. We propose methods for handling virtual content directly while making abstraction of its complexity. They enable artists to favor expressiveness over handling technical requirements inherent to virtual content manipulation.

1.2 Why is 3D shape creation a hard task?

> *What remains hard is modeling. The structure inherent in three-dimensional models is difficult for people to grasp and difficult too for user interfaces to reveal and manipulate. Only the determined model three dimensional objects, and they rarely invent a shape at a computer, but only record a shape so that analysis or manufacturing can proceed. The grand challenges in three-dimensional graphics are to make simple modeling easy and to make complex modeling accessible to far more people.*
>
> — Robert F. Sproull (1990)

Two and a half decades after this observation, numerous research papers in Computer Graphics still start by stating that 3D modeling is a complex and intricate task. Despite a recent popularization, this task is indeed reserved to skilled artists in industrial contexts. Such professional have learned to handle complex 3D creation softwares. As stated before, even skilled artists need time to create a shape, and several shape creation or modification iterations are needed for converging toward a requirement-matching result. This makes 3D models creation (as well as other types of virtual content) a costly task in a production budget (e.g. a special effect sequence in a film).

But what makes this task so difficult? Indeed, building a 3D model means setting the parameters of a virtual object representation. As discussed in the previous section, various structures are available for representing a same object (e.g. mesh or implicit surface). However, choosing a representation is not a practical difficulty since most of modeling software impose a given structure and/or propose efficient conversions from one to another. This section investigates the challenges of 3D shape creation and relates them to the notion of *shape space*.

1.2.1 Size of virtual objects

As stated earlier, the most popular structure used for representing virtual objects is the mesh. A mesh is a piece-wise planar surface approximation over polygonal regions (usually triangular or quadrangular). It is usually defined as a list of 3D coordinates representing the positions of the polygons' vertices, and a list of indices representing the adjacency of vertices within polygons. Vertices and faces are called mesh primitives;

Let $\mathcal{O}$ be an object of size L, and λ be the mean shortest distance between small-scale details on the surface of $\mathcal{O}$. The surface of $\mathcal{O}$ can be decomposed in non-overlapping regions $\{p_i\}$ containing a single small-scale detail. There is an order of magnitude of $\left(\frac{L}{\lambda}\right)^2$ such regions, each having a surface of λ^2.

Let $\mathcal{M}$ be a mesh representing $\mathcal{O}$ with N vertices. According to the Nyquist-Shannon theorem, each small-scale patch p_i must be sampled with at least two vertices. This yields that $N > 2.\left(\frac{L}{\lambda}\right)^2$. Note that for aesthetic reasons (such as representing curved areas without angles), the actual number of vertices in a mesh is usually much higher than this theoretical lower bound. Still, this gives an idea of the size of a mesh in terms of number of vertices.

Virtual worlds often reproduce real world features, including terrains, characters, and buildings. Such objects have big scale ranges, i.e. have big $\frac{L}{\lambda}$ ratios. Accordingly, meshes representing such object will have large numbers of vertices, increasing quadratically.

Lots of vertices means lots of 3D coordinates to set for the artist creating the mesh. The complexity of the 3D shape creation task can partially be explained by this great number of degrees of freedom in the model. Luckily, editing tools allow to simplify this task by allowing to modify several degrees of freedom at once: proportional editing, virtual sculpting, adaptative sampling, and texture-based displacement map are some examples.

Still, these tools rely on the artist performing per-primitive edit operations. For this reason, high quality meshes keep requiring lots of efforts.

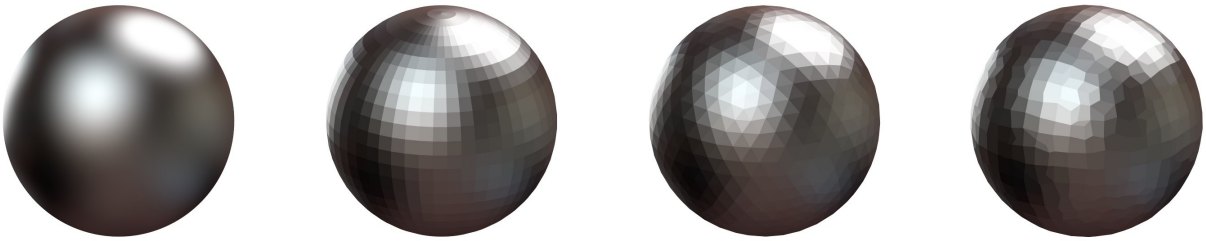
1.2.2 Shape space versus mesh space

Given two shapes, any human can assess their visual similarity. This gives the intuition that a metric space for shapes does exist. We call this space the *shape space*. Any concept – e.g. "fish" – can be identified with a subset of the shape space where shapes are identified as fish shapes by a human. The more precise the concept – e.g. "tuna" – the smaller the subset (in the sense of set inclusion). The notion of shape space is very close to Plato's *theory of Forms* (see Carpenter and Fine (2008)).

The problem with the shape space is that it fully relies on human subjectivity for its definition. As a result, each human has its own shape space, which might look alike on a coarse scale (all fish have fins) but might differ on details (marine biologists might have richer and more precise ideas of how fish shapes are than the average human). Besides, if shapes can exist in the human mind, they have no unique or natural (i.e. human-independent) representations. That is the work of the artist to transform a concept into an actual shape representation, using their own shape space and shape representation creation skills. These two attributes may qualify the *style* of the artist.

Given a representation structure – mesh for instance – we can define a representation space – a mesh space – of all possible representations. Note that the mesh space defined here is similar to the "shape space" introduced by Kilian et al. (2007) but without fixing the mesh topology.

The mesh space is human-independent, quantifiable, and more importantly it can be mapped onto the shape space, constituting a practical interface to it. However, this mapping is only injective: a single shape can have very different representations, as shown in figure 1.1.



(a) (b) (c) (d)

Figure 1.1 – Different meshes can represent the same shape. Here the same sphere (a) is represented through a UV tessellation (b), an icosahedron subdivision (c), and a cube-marched spherical implicit field (d).

There are various explanations for the above discrepancy. Given a shape, different tessellations are possible, which all have their benefits. For example: quad-based tessellations are better suited to skinning animation, triangle-based tessellations are more generic. Even inside each tessellation scheme, primitives can be placed differently depending on the shape features to be captured. Finally, the same shape can be represented at different resolutions (e.g. for efficiency), yielding different representation. Creating a shape representation imposes to take many design decision which will influence the output. The knowledge required to make these decisions is part of the expertise that makes the artistic task so intricate.

1.2.3 Path in the mesh space

As stated earlier, mesh design is iterative. Each iteration leads from an input mesh M_0 (a primitive or non final representation) to an output mesh M_1 (improved representation). Each of these endpoints is represented as a point in the mesh space. Both are connected by a path representing the sequence of edits made by the artist. Denning et al. (2015) presented a technique for summarizing and visualizing such paths for educational purposes.

Given identical sets of endpoints, different artists will create different mesh paths connecting them. Here again, the artist will use their knowledge and the tools offered to them for performing the edits. The high number of possibilities once again requires some expertise to find the best editing path, which contributes to the difficulty of the mesh creation task.

1.3 The case of animation

The previous section explained the difficulties raised by static 3D shape creation. This section explores the challenges of animation creation.

Animations are based on static shapes: The evolution of the intrinsic parameters of the shape representations through time creates the animation. Therefore, creating an animation always starts with creating at least one static shape, following the process described above.

The previous section relied on the 3D data quantity and ambiguity for explaining the complexity of 3D shape creation. Animated content possesses an extra dimension compared to static 3D content. The challenges of animation creation are therefore a superset of the ones described above.

Two animation design paradigms exist: frame-based parameters interpolation, and simulation. The remaining of this section details the challenges posed by these two alternative setups.

1.3.1 Frame-based parameter interpolation

In this setup, the artist specifies the values of some model parameters at specific time steps called key-frames. These values are interpolated over time using various schemes: nearest neighbor value, linear or cubic interpolation for example.

Frame-based parameter interpolation offers two main advantages: A high degree of control over the animation; And fast computations (parameter interpolation is direct and generally allows for interactive display).

The parameters set by the artist are often not directly those of the visual model (such as vertices positions). Instead, deformation tools based on bounding boxes, proxy geometry, skeletons, or cages allow to set visual parameters on simplified model and to automatically transfer the resulting deformations to the whole mesh. More sophisticated deformations are also possible, such as ad-hoc procedural scripts.

The function associating input parameters to the actual deformed shape is called the rigging. The space of the rigging input parameters has been conveniently called *rig space* by Hahn et al. (2012). The rig-space is a reduced base for deformations in the mesh space. For a given mesh, it represents a parametrization of the sub-space surrounding it in the mesh space.

Once the 3D shape model has been created and rigged, the artist control their mesh in the rig space. This makes it easier since it possesses much fewer dimensions. Even then, values have to be set for each of its degrees of freedom at each key-frame. This still represents a large amount of data to be created by the artist. The data creation itself is hard, since it requires the artists to design interpolation curves which will result in a motion which is hard to anticipate.

Besides, these parameters still represent purely static shapes: limbs positions and orientations for example. Key-framing fundamentally imposes to set such static parameters for later interpolating them. This goes against the temporal nature of animation and once again require lots of knowledge and experience from the artist to create such content.

1.3.2 Simulation

The simulation paradigm relies on a physical interpretation of the mesh data: it is considered to be the surface of a liquid or of a solid for example. From this physical interpretation, a motion is derived through the laws of physics: Navier-Stokes or Newton equations of motion respectively. Such simulations are costly in terms of computations and therefore in terms of time. On the other hand, they are able to produce realistic results in terms of both shapes and speeds.

Here, the artist has much fewer degrees of freedom than before: They can only influence the simulation by setting initial conditions (initial position, shape, and velocity for example) and physical parameters (such that fluid viscosities or object masses).

The non-linearity of the laws of mechanics makes it impossible to precisely predict the behavior of a complex mechanical system through a long period of time. Therefore, the artist has to run lots of simulations, modifying the degrees of freedom incrementally until converging to the desired result.

1.4 Overview of this thesis

The goal of the work presented here it to propose alternative editing paradigms, specific methods, or practical solutions to the challenges of creating both static and animated content. The common denominator of these approaches is their attempt to offer an editing interface to the shape space rather than focusing on the representation space. Content is not considered at the vertex level or at the key-frame level. Instead, semantic segmentations are used for processing object parts as high-level primitives. This enables the artist to focus on the relationships between object parts (object's constraints) and their modeling intent (external constraints) rather than on managing the structure representing the object.

1.4.1 Contributions

3D modeling This thesis first introduces a framework for decomposing the *consistency* of a *complex objects* along its hierarchical structure. This framework is then used for presenting the other contributions regarding 3D modeling: A *sub-structure deformation method* based on linear blend skinning, allowing to perform sub-structure substitution in a part-based modeling method; A *distributions descriptor interpolation* method based on optimal transport, allowing to create new distributions and distribution gradients

inside a distribution painting framework; And a *grammar-based framework for sculpting hierarchical objects*.

Animation modeling This thesis also introduces the notion of *temporality* of an animation, resulting in a temporality-based *classification of animations*. The novel concept of *preservative structure* derives from this classification. Preservative structure editing emerges as a paradigm for animation modeling, and is illustrated into two different methods: A waterfall network editing framework based on a *novel quantitative waterfall classification* is presented; We finally present a general liquid animation editing framework relying on a preservative structure called *space-time features* set, computed from raw input animation data and allowing to perform direct edits on the animated content.

1.4.2 Publications

The contributions presented in this manuscript have been first introduced in the following papers:

Liu, H., Vimont, U., Wand, M., Cani, M.-P., Hahmann, S., Rohmer, D., and Mitra, N. J. (2015). Replaceable substructures for efficient part-based modeling. *Computer Graphics Forum, Proceedings of Eurographics 2015*, 34(2):503–513

Emilien, A., Vimont, U., Cani, M.-P., Poulin, P., and Benes, B. (2015). Worldbrush: Interactive example-based synthesis of procedural virtual worlds. *ACM Transactions On Graphics (TOG), Proceedings of SIGGRAPH 2015*, 34(4):106

Vimont, U., Rohmer, D., and Cani, M.-P. (2016). Deformation grammars: Hierarchical constraint preservation under deformation. *Submitted at Computer Graphics Forum*

Emilien, A., Poulin, P., Cani, M.-P., and Vimont, U. (2014). Interactive procedural modelling of coherent waterfall scenes. *Computer Graphics Forum*, 34(6):22–35

Manteau, P.-L., Vimont, U., Rohmer, D., Cani, M.-P., and Wojtan, C. (2016). Space-time sculpting of liquid animation. *To appear in the proceedings of Motion In Games 2016*

Most of this work was done in collaboration with other researchers, to whom I express my gratitude. In parallel of this thesis, I also collaborated with Sylvain Meylan on DNA model generation. This work will not be discussed here; It was published in the following paper:

Meylan, S., Vimont, U., Incerti, S., Clairand, I., and Villagrasa, C. (2016). Geant4-dna simulations using complex dna geometries generated by the dnafabric tool. *Computer Physics Communications*, 204:159–169

1.4.3 Outline

The remainder of this manuscript is organized into four chapters.

Chapter 2 gives an overview of the state of the art in static and animated shape modeling.

Chapter 3 presents the notions of *complex object* and *consistency* as a general framework for semantic model generation and deformation. It then details three methods allowing to design static shapes in innovative ways: a part-based method for designing constrained shape assemblies, a painting metaphor for creating objects distributions, and a generic grammar-based framework for sculpting hierarchical objects.

Chapter 4 deals with the modeling of animated content, and more precisely with the control of liquid animations. It relies on a *classification of animations* from which follows the notion of *preservative structure*. It is used inside a waterfall network editing method, and a general liquid animation editing framework.

Finally, Chapter 5 concludes this thesis and draws perspectives for future work.

1.4.4 Video

Most results presented in this thesis are best seen in video. I compiled the videos accompanying the articles corresponding to the contributions presented in this manuscript in the following web page:

https://team.inria.fr/imagine/article-videos/

Alternatively, this web page can be accessed through the following QR-code:

Chapter 2

Previous work

Contents

2.1 Introduction . . . 12
2.2 Static Objects Generation . . . 13
2.2.1 Procedural modeling . . . 13
2.2.2 Inverse procedural modeling . . . 17
2.2.3 Sketch-based modeling . . . 17
2.2.4 Example-based modeling . . . 18
2.2.5 Limitations of generation methods . . . 20
2.3 Static Objects Deformation . . . 20
2.3.1 Low-level deformation methods . . . 20
2.3.2 Structure-aware shape deformation . . . 22
2.3.3 Sketch-based deformation . . . 24
2.3.4 Limitations of deformation methods . . . 25
2.4 Animation Design . . . 25
2.4.1 Fluid animation modeling . . . 25
2.4.2 Fluid animation control . . . 26
2.5 Conclusion . . . 29

This chapter describes how previous methods have addressed the challenges of complex object and animation modeling. Object modeling has evolved since the beginning of Computer Graphics. Recent advances in computers hardwares in the last two decades have opened new possibilities. Numerous works have addressed the initial problems and opened new ones.

Since the subject addressed in this thesis is large, the review of previous work will be focused on recent and/or relevant work. References to more comprehensive literature reviews are inserted; Please refer to those for a more extensive presentation of the state of the art on a particular subject.

This chapter is organized as follows: Section 2.1 presents the trends among virtual content manipulation methods; Section 2.2 describes the object generation literature; Section 2.3 focuses on object deformation as a creative tool; Section 2.4 deals with object animation, with an emphasis on fluid animation and control; Section 2.5 concludes this chapter and draws the directions that this thesis explored.

2.1 Introduction

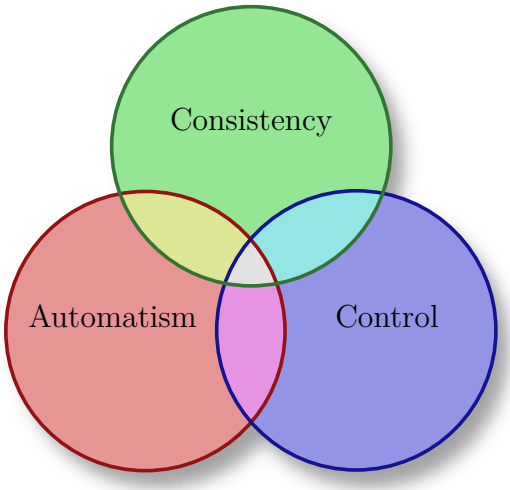


Figure 2.1 – The methods presented in this chapter propose a compromise between three antagonistic aspects of modeling.

Since 3D modeling and animation are hard, numerous methods have attempted to ease these tasks in various ways.

One way to achieve easy modeling is to build a method in charge of generating the geometry. Such methods may integrate constraints for creating highly realistic and detailed models. This the typical case of procedural methods. They fall in the red and/or yellow part of the diagram represented in Figure 2.1. The problem of such methods is that they fail to incorporate direct user control (the blue disk of the diagram). Indeed, considering that such methods attempt to create specifically constrained result aiming at realistic appearance with low degrees of freedom, user input might be in contradiction with the method's structure. The input parameters of the procedural method may be seen as a specific language in which any user intent must be translated. This problem is further discussed in Section 2.2 § Procedural modeling.

Inverse methods are an automation of this translation step. Starting from a user intent, such methods try and find suitable input parameters for the (forward) procedural method. These approaches require first a suitable way for the users to formulate their expectations, and second, a appropriate metric to compare this expectation to the output of the forward method.

Both forward and inverse methods perform the model creation task without the user being able to intervene within the actual generation process. In order for the creation process to actually be creative, metaphor methods aim to let the user participate through the model synthesis. This participation often relies on user inputs mimicking traditional media: drawing, painting, sculpting (hence the term "metaphor").

Example-based methods may or may not allow many user expectations to be taken into account. Their specificity is to settle what "realism" means through instances of objects considered as realistic. Such objects (i.e. examples) are used by the method for creating new objects in the same "style", or the same definition of realism.

The following sections use these axes as a method classification regarding generation, deformation and animation of 3D models. The frontiers between method families might be fuzzy, as between some inverse procedural and example-based methods. In such cases I tried to pick the option favoring clarity.

2.2 Static Objects Generation

This section focuses on static generation methods, i.e. methods aiming at creating 3D models without concern on their animation, as opposed to methods presented in Section 2.4. These methods also differentiate themselves from deformation methods presented in Section 2.3 as they create a model which is not intended to be re-used as an input of the method;

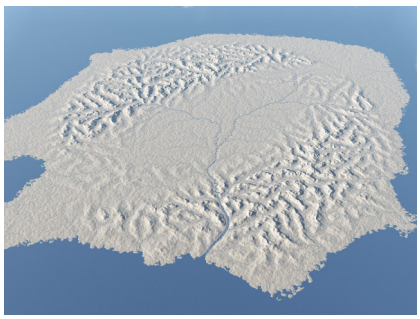
Various static object generation paradigms are presented: forward and inverse procedural modeling (Sections 2.2.1 and 2.2.2), sketch-based modeling (Sections 2.2.3), and example-based modeling (Sections 2.2.4). Each paradigm is illustrated with relevant references and analyzed through the scope of its utility from a user perspective. Generation methods have global strength and weaknesses which are analyzed and commented (in Section 2.2.5).

2.2.1 Procedural modeling

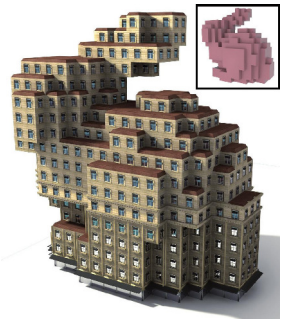
Procedural generation aims at creating a representation of an object using a mathematical description of the property to represent. In our case, this property is the shape of the object.

One of the first and most general-purpose procedural generation method was created by Perlin (1985); It consists in a mathematical function casually called Perlin noise, which can look either smooth or noisy depending on some parameter value. This function has been used for representing terrains, clouds, and other virtual world features in the early ages of Computer Graphics, as explained by Ebert et al. (2002).

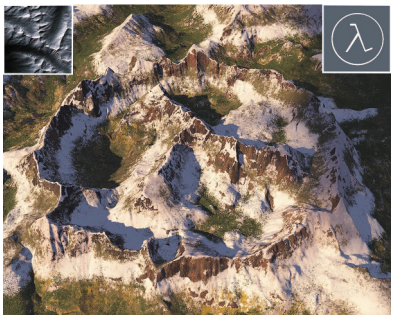
A virtual object representation has to satisfy some constraints in order to be understood and accepted as valid (this point is developed in Section 3.1.2). As a result, more special-purpose methods have been developed, which better satisfy constraints of the various



(a) Terrain procedurally generated by Génevaux et al. (2013) using hydrological principles.



(b) The method of Talton et al. (2011) is used for controlling a procedural model (constraint in inset).



(c) The method of Zhou et al. (2007) is both example-based (left inset: example data) and sketch-based (right inset: user stroke).

Figure 2.2 – Static generation methods can be classified into families: procedural (a), inverse procedural (b), example-based or sketch-based (c).

objects populating virtual worlds. Some objects have been studied extensively: terrains, ecosystems, and cities.

This section presents methods grouped by the type of object they generate. Other references regarding virtual world modeling can be found in the survey of Smelik et al. (2014).

2.2.1.1 Terrains

Virtual terrains have received a lot of attention in computer graphics research. Here are presented a sub-set of references relating to terrain generation. A more comprehensive survey of geological modeling methods has been done by Natali et al. (2013).

In most cases terrain are represented as height fields, as in the work of Prachyabrued et al. (2007) presenting a method for generating stylized 2D maps. However, height-fields do not allow to represent natural elements such as caves, overhangs, or arches. Gamito and Musgrave (2001) offers to solves this issue by introducing horizontal displacements generating overhangs, while Peytavie et al. (2009a) proposes a layer-based representation allowing any topological configuration. Besides, particular land features do not fit in those representations and have to be modeled independently: For example, Peytavie et al. (2009b) presents a convenient tool for modeling rock piles, and Beardall et al. (2007) introduces a volumetric weathering model for creating goblins.

A virtual terrain has scales typically ranging from a few meters or less, up to several kilometers. The ratio of these scales (around 10^4) combined with the height-fields representation makes terrain files too big to handle with standard methods. Geiss (2007) and Vanek et al. (2011) offers a GPU terrain structure allowing detailed terrain to be generated in real time. Rusnell et al. (2009) and Génevaux et al. (2015) propose feature-based terrains representations allowing to overcome scale range issues.

Erosion is one of the key phenomenon of the evolution and the appearance of a terrain. This phenomenon is not considered by standard fractal-based methods. Kelley et al. (1988) and Musgrave et al. (1989) first introduced erosion simulation in computer graphics. Roudier et al. (1993) added the sedimentary deposition phenomenon to this scheme. Chiba

et al. (1998) proposed a velocity-field model for erosion influence. Beneš and Forsbach (2001) introduced the influence of several layers of rock with different erosion behaviors and extended this method in Beneš and Forsbach (2002), Beneš and Arriaga (2005), and Benes et al. (2006). Neidhold et al. (2005), Krištof et al. (2009), Mei et al. (2007), Šťava et al. (2008) proposed improvements of the method for making it interactive, allowing to indirect control of the erosion process. Most of these methods rely on liquid simulation, which is only valid at small scale. As a result, large-scale phenomenon such as dendrite landscapes are not generated by those methods. Pytel and Mann (2013) takes a different approach based on hypsometric analysis which produces multi-scale results. Other approaches achieving such results are based on explicit river models.

Rivers have a major influence on the appearance of a terrain. This influence is not independent of that of erosion, but explicitly considering rivers as terrain primitive yields very different results compared to erosion simulation. Prusinkiewicz and Hammel (1993) first used rivers as a terrain generation primitive to be used in fractal schemes. Since, Belhadj and Audibert (2005) introduces a bottom-up approach for creating a terrain from ridges and rivers. Teoh (2009) extends this idea by proposing a river network generation algorithm. Derzapf et al. (2011) offers a fast solution allowing large terrain generation. Génevaux et al. (2013) rely on a novel implicit structure for defining a terrain through hydrological concepts (see Figure 2.2a).

Finally, Cordonnier et al. (2016) combines tectonically uplift with a geological river influences for creating realistic terrains and mountains.

2.2.1.2 Ecosystems

Virtual worlds usually contain virtual life. When different types of life live in the same system, it is called an ecosystem. Deussen et al. (1998) first proposed a set of tools for creating ecosystems based on a simulation. Beneš et al. (2011) describes a method for populating urban environments with plants. However, most work focuses on individual plant modeling.

L-systems have originally been introduced by Lindenmayer (1968). This generative plant model has since been extensively used and improved, as shown by Prusinkiewicz and Lindenmayer (2012). de Reffye et al. (1988) integrated various elements such as leaves, flowers and fruits in the model. Měch and Prusinkiewicz (1996) added environment interaction to the development model. Power et al. (1999) and Boudon et al. (2003) proposed tools for interactively manipulate the model and browse the shape space it parametrizes. Beneš et al. (2009) introduced an interactive model for plant generation with limited resource management (such as water or light). Peyrat et al. (2008) used L-systems as a leave generation model. Pirk et al. (2012) proposed an interactive tree model adaptation method allowing the tree to adapt to its environment. Apart from trees, other plant have been the subject of procedural modeling methods: for example Desbenoit et al. (2004) uses a Diffusion Limited Aggregation model to represent lichen growth.

2.2.1.3 Cities

Ecosystem modeling allow to create natural sceneries. Man-made sceneries have also been studied. Merrell and Manocha (2011) and Lars Krecklau (2011) offer methods for modeling interconnected structures such as bridges and roller coasters. However, cities

have received the biggest interest from the research community as shows in the survey of Vanegas et al. (2010b).

Parish and Müller (2001) introduced the concepts of city procedural modeling. Vanegas et al. (2009) added the possibility for the method to handle post-generation constraints. Scharl (2010) introduced a method for choosing and placing residential houses given a parcel layout. Lipp et al. (2011) proposed an interactive layer-based city layout method. Vanegas et al. (2012b) focused on the parcel subdivision of residential areas. Steinberger et al. (2014a) presented heavy optimizations allowing to generate a whole city model on the GPU and to visualize it in real time.

Buildings are sub-parts of cities, and have been studied on their own. Wonka et al. (2003) first proposed a grammar-based building generation framework, which has since been extended by Müller et al. (2006), Larive and Gaildrat (2006), and Schwarz and Müller (2015). Lipp et al. (2008) extended the grammar-based paradigm by providing visual rules. Whiting et al. (2009) introduced physical constraints into the process. Kelly and Wonka (2011) enriched facade exteriors with an extrusion-based method. Steinberger et al. (2014b) proposed a parallel implementation allowing the building models to be generated on the GPU.

Tutenel et al. (2011) introduced a method for mixing different room layouts in a same building, and Barroso et al. (2013) offered to mix building parts generated with different grammar rules. Some sub-parts of buildings have been studied in particular: Ilcik et al. (2015) details a layer-based facade generation algorithm. Merrell et al. (2010) and Leblanc et al. (2011) designed methods for creating room layouts inside residential houses and building floors respectively. Merrell et al. (2011) proposed a method for generating furniture layouts inside rooms from design guidelines. Zheng et al. (2016) introduced a method for adapting chair models to body poses according to ergonomics principles.

Apart from buildings, roads have also been studied on their own. Chen et al. (2008a) introduced a road-based city modeling method. Galin et al. (2010) and Galin et al. (2011) proposed procedural road generations methods for connecting cities.

Rural environments are somehow more constrained by their environment than cities. Bruneton and Neyret (2008) created a vectorial description for such environments, and Emilien et al. (2012) proposed a method for generating countryside village.

Various models might be difficult to use altogether for generating a single world model because of badly interacting models. Smelik et al. (2011) proposes a way of combining outputs of different methods into a consistent virtual world model using a declarative approach.

2.2.1.4 Limitations

We saw that many methods address different problems and sub-problems in very specific ways. Some attempts were made for integrating all these methods inside a single ubiquitous system (see Smelik et al. (2011)). This system should use each method in its specific scale and phenomenon. However, most systems take specific inputs, and produce non-predictable and non-constrained outputs. As a result, procedural methods are under-used in real-case production scenarios.

Another explanation for this under-usage is the counter-intuitiveness of most procedural tools: The *black box* effect. The input is often a set of, sometimes obscure, parameters, and it is uneasy for a user to know how to modify these to reach a given goal. Inverse methods offer a different paradigm which circumvents this issue.

2.2.2 Inverse procedural modeling

The concept of inverse methods is simple: Given a procedural model and a goal to achieve, inverse methods attempt to find the input parameters of the model which will produce the closest to the goal object. In order to achieve this, a comparison metric has to evaluate the distance between the output of the model and the goal. This distance is used as a cost function in an optimization procedure. In a way, it is an automation of the creation process described in Section 1.

Bokeloh et al. (2010) presented a symmetry analysis method allowing to build a shape grammar from raw input 3D models. Šťava et al. (2010) proposed an inverse procedural model for 2D L-systems from vectorial drawing inputs, and Benes et al. (2011) extended it for defining the input with vectorial guides. Vanegas et al. (2010a) introduced an algorithm for generating the 3D model of a building from a set of calibrated input pictures. Talton et al. (2011) generalized inversed procedural modeling to any grammar-based forward procedural model using a Metropolis algorithm (see Figure 2.2b). Vanegas et al. (2012a) and Šťava et al. (2014) addressed city and tree modeling respectively using this paradigm. Ritchie et al. (2015) optimized Monte Carlo Markov Chain algorithm (MCMC) by allowing incompletely generated models to be evaluated.

Inverse procedural modeling methods assume that the result is already known before it is created, which is not always the case. Besides, starting from a goal and letting the method perform the generation does not leave much artistic leeway. Another modeling paradigm is much closer to traditional artistic design process: sketch-based modeling.

2.2.3 Sketch-based modeling

Sketch-based modeling relies on a metaphor offering an intuitive modeling interface. It can naturally be used for designing 2D vector graphics. Kazi et al. (2012) proposed a method allowing to design such content while offering the advantages of computer-generated data. A review of sketch-based modeling for 2D applications has been made by Hurtut (2010).

Sketch-based modeling has also been used for generating 2D content on 3D objects. For example, Sun et al. (2013) introduced a sketch-based texturing method, and Schmid et al. (2011) used strokes for creating expressive paint-like volumetric texturing.

Here, we focus on sketch-based modeling of 3D shapes. More in-depth analysis of such methods are presented in Olsen et al. (2009) and Cook and Agah (2009).

3D sketch-based modeling was first introduced by Igarashi et al. (1999) with the Teddy system, allowing the design of arbitrary 3D shapes from strokes. This system was later extended by Karpenko and Hughes (2006), Schmidt et al. (2007), and Bernhardt et al. (2008).

The Harold system proposed by Cohen et al. (2000) offered an original way to create 2D billboards in 3D space, resulting in a doodle-like scene. But in most cases, sketch-based 3D modeling techniques focus on the design of a single object of a given type: Chen et al. (2008b), Wither et al. (2009), and Longay et al. (2012) proposed methods for creating trees; Wither et al. (2008) introduced a method for creating clouds; McCrae and Singh (2009) presented a method for creating road networks; Schmidt et al. (2009) analyses 3D scaffold strokes for reconstructing the geometry of man-made objects; Gain et al. (2014) proposed a city modeling sketch-based system; And Entem et al. (2015) recently introduced a method for creating 3D shapes of animals.

2.2.3.1 Terrains

Among all objects which have been studied in particular in sketch-based modeling terrains are again well represented. Watanabe and Igarashi (2004) first introduced a method for deforming a plan from 3D stokes, creating mountains. This approach has since been extended by Gain et al. (2009), Hnaidi et al. (2010), and Bernhardt et al. (2011) for faster and more realistic results.

dos Passos and Igarashi (2013) and Tasse et al. (2014) proposed first-person set-ups where strokes are directly interpreted as mountain profiles for creating the terrain.

de Carpentier and Bidarra (2009) proposed patch-based height-field brushes for painting terrains from above.

Finally, Zhou et al. (2007) proposed a top-view set-up where strokes are analyzed for matching ridges in example height-field patches (see Figure 2.2c). This method has since been extended by Tasse et al. (2012) for taking height constraints into account. Gain et al. (2015) also uses real-world data and mixes it with constraints diffusion.

2.2.3.2 Limitation

Sketch-based methods allow either to used strong hypothesis for interpreting a single drawing, or to progressively model an object with few hypothesis. The second setup which is very artist-oriented. One of the major draw-backs of those methods is that they usually only rely on user inputs for creating the object. This limits the complexity of the object to be generated.

Some recent methods differ from the others in that they use real-world data for generating the output. They also comply with an other model generation paradigm: example-based modeling.

2.2.4 Example-based modeling

Example-based techniques rely on existing instances of the object to model for creating new ones. They can rely on standard machine leaning techniques (as in the method of Talton et al. (2012), which creates a grammar from a set of examples), or on ad-hoc descriptors (as proposed by Gal et al. (2007) for generating shape matching 3D assemblies from object collections). As such, some example-based techniques also comply with the inverse procedural paradigm, where the target has similarities with the set of examples.

This paradigm is very useful for creating variations in an object family and for mixing pieces of objects. Recently, some methods attempted to take benefit of big datasets for creating new custom shapes.

2.2.4.1 2D textures

The example-based paradigm has been successfully used in texture synthesis, as explained by Wei et al. (2009). For example, Ashikhmin (2001) proposed a method for synthesizing natural textures. As stated in Section 2.2.1, terrains can be represented as height-fields, which are essentially textures. Brosz et al. (2006) and Pang and Zhao (2009) took advantage of this for proposing example-based terrain synthesis methods inspired by the texture synthesis literature.

A texture can also be discrete, it is then referred to as a 2D arrangement or distribution. Ijiri et al. (2008), Hurtut et al. (2009), Jenny et al. (2010), and Landes et al. (2013) introduced methods for synthesizing such arrangements from example in the context of 2D vector graphics.

2.2.4.2 3D distributions

3D distributions have been studied more recently. Yu et al. (2011) introduced a method for arranging pieces of furniture in a room from example displays. Yeh et al. (2012) used MCMC for generating constraints-matching 3D distributions. Guerrero et al. (2015) proposed a method for learning shape placement rules on a polygon and to propagate these rules on other polygons of the scene.

The method presented in Section 3.3 rely on example-based distribution analysis and synthesis method integrated inside a paint-based distribution modeling framework.

2.2.4.3 Part-based modeling

Some methods propose to decompose an object into parts, and to use parts from different example objects for creating a new one. This is called part-based modeling.

Funkhouser et al. (2004) first introduced this concept in a method allowing to copy and paste salient mesh parts from one mesh to another. Kraevoy et al. (2007) extended this concept by proposing an automatic replaceable sub-part detection algorithm. Schmidt and Singh (2010) proposed to handle mesh details by representing them as displacements (note that this method is extended to mesh sequence in Section 4.3). Takayama et al. (2011) mixed both approaches by creating a resilient parametrization of the rest shape. The method presented in Section 3.2 introduces a way to re-combine object parts while preserving their type adjacency.

Xu et al. (2012) proposed an evolutionary algorithm allowing to mix objects of the same family for generating variations. Chaudhuri et al. (2011) and Kalogerakis et al. (2012) offered probabilistic approaches allowing it to select relevant parts in a large set of objects. Zheng et al. (2013) introduced the notion of symmetric functional arrangements as a key component of semantically valid part assembly.

Jain et al. (2012) proposed a method for interpolating two structurally similar objects (i.e. generating in-betweens) by progressively substituting parts of the second into the first. Alhashim et al. (2014) extended this concept by making this interpolation continuous and handling topology changes.

2.2.4.4 Transfer

Another way to take advantage of example objects is to apply some properties of a given object to another. This is called transfer. Brouet et al. (2012) introduced a method for transferring garment designs from one character to another. Dicko et al. (2013) proposed a method for transferring anatomical 3D data from one character to another. Ma et al. (2014) presented an automatic method for segmenting details and structure allowing it to perform style transfer on various input data.

2.2.4.5 Limitations

Example-based modeling methods require examples of what the user wants to model for them to work. This is both a strength – since it allows the user to benefit from potentially existing models – and a weakness – since it also limits the scope of reachable results.

But static generation techniques are also limited on a larger scope, as explained next.

2.2.5 Limitations of generation methods

All previous methods are dedicated to high quality model creation. More precisely, these methods attempt to maximize the quality of the created model compared to the time required to generate it.

On the other hand, detailed models are heavy in terms of data size, and uneasy to manipulate. In particular, deforming a model might "break" some constraints that this model was supposed to satisfy, making it non consistent. This would not be a problem if models were ready to use: There would be no need to manipulate them before using them. However, generation methods do not usually integrate all required constraints of all possible use cases of the model they create – for the reason that those use cases are not always predictable. For instance a city model might need to be deformed for the sake of the story taking place in it, of a character model might undergo some changes for it to fit inside cloth models. In both cases, a city model must look like a city and a character model must look like a character.

For this reason, model generation methods are often associated with deformation methods. Such methods are presented in the next section.

2.3 Static Objects Deformation

The idea behind deformation methods is that an object model does not necessarily have to be built from scratch in one single pass. The creative process often starts from an initial object, and evolves toward a given goal, as with some sketch-based methods (see section 2.2.3). This very goal can also evolve through the creation process.

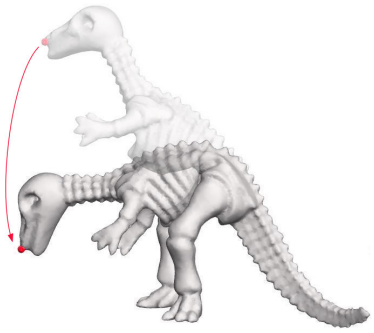
If not for creating an object, deformation methods can be used for modifying an existing object – e.g. an object created by another artist, generated using a static generation method, or downloaded from an online shape repository. This deformation aims at adapting the model to a particular purpose, and should preserve some of its attributes. The set of attributes to be preserved depend on the method used for performing the deformation.

2.3.1 Low-level deformation methods

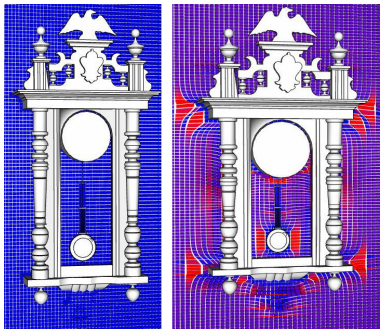
What we call here low-level deformation methods are methods focused on the object representation at the representation level.

2.3.1.1 Sculpting tools

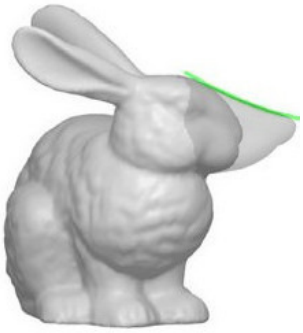
When sculpting clay, the starting point is the initial clay piece, which is progressively transformed into the final result. Sculpture is therefore an attractive metaphor for artistic shape modeling.



(a) The as-rigid-as-possible deformation scheme of Sorkine and Alexa (2007) allows to set vertex position constraints (in red) while respecting the local shape of the model.



(b) The method of Kraevoy et al. (2008) permits the non-homogeneous resizing of a man-made objects while maintaining the circularity of its parts.



(c) Zimmermann et al. (2008) proposed a method for sketching object deformations by modifying their silhouette.

Figure 2.3 – Low-level deformation method focus on shape-preserving deformation (a) while structure-aware deformation methods analyze the high-level structure of a shape for preserving its consistency (b). On the other hand, sketch-based deformation method emphasize intuitive shape edits (c).

Ferley et al. (2000) introduced a volumetric shape representation suited for virtual sculpture and sub-object copy-paste. McDonnell et al. (2001) created a sculpting framework including haptic feed-back. Angelidis et al. (2006) proposed a volume-preserving space deformation method aimed at sculpting meshes. von Funck et al. (2006) pursued this path by introducing a volume- and smoothness-preserving space deformation method.

Volume preserving space deformation methods allow to handle a volumetric property while using a much more efficient surfacic representation. Stanculescu et al. (2011) extended this possibility by proposing a topology-varying surfacic structure called quasi-uniform meshes.

Sculpting tools often define space deformation fields which are usually independent from the structure of the object. This tends to damage this structure. Handle-based techniques offer an alternative solution which solves this issue, as shown next.

2.3.1.2 Handle-based

Handle-based methods rely on the following setup: Given a rest shape represented as a mesh, the user tags a set of vertices called handles; Handles are moved to a pose position; The method automatically computes new positions for all the other (non handle) vertices by running a deformation-minimizing optimization. A survey of such methods was done by Botsch and Sorkine (2008).

Sorkine et al. (2004) proposed such a method through a Laplacian-based encoding of vertex neighborhood. This method was later extended by Igarashi et al. (2005) for real-time deformation of 2D shapes; And Sorkine and Alexa (2007) pursued with a fast 3D solution (see Figure 2.3a).

Botsch and Kobbelt (2005) proposed a handle-based deformation method working with arbitrary 3D topologies. Sumner et al. (2005) introduced an example-based algorithm for handle mesh deformation.

Operating mesh deformation at the vertex level can be tiresome for large meshes. For easing the process, proxy deformation methods propose to deform a higher level representation of the object.

2.3.1.3 Proxy geometries

A proxy geometry is a simplified version of the representation of an object. Its components are linked to the actual representation of the object, usually through weights. Those weights can often be computed automatically. The idea behind proxy geometry is that a deformation can be applied to it, and will be forwarded to the object representation. It allows to manipulate far less degrees of freedom than by deforming the model directly.

A cage is a set of volumetric cells which approximate the model. Each cell contains a sub-part of the model which will be influenced when the cage is deformed. Nieto and Susín (2013) presents a survey of modern cage-based deformation methods. Modern cage-based deformation methods attempt to preserve the model smoothness at cage interfaces, such as presented by Gonzàlez et al. (2013).

Unlike cages, skeleton components (called bones) are usually contained inside the object representations. Bones are suited for representing nearly-cylindrical geometries, such as the limbs of a character. It is often used for posing character models, and through pose interpolation for animating them (see Section 2.4). Skeleton-based mesh deformation is called skinning, and offer multiple possibilities, as shown in the SIGGRAPH course of Jacobson et al. (2014). Modern skinning methods tend to preserve geometrical properties, such as volume (see Vaillant et al. (2013)).

Shape approximations can also be used as proxy geometries for deformation, as shown by Thiery et al. (2013).

2.3.1.4 Limitations

By definition,low-level deformation methods do not take any object-specific data into consideration for deforming it. It makes them very generic, but somehow limited: some objects have specific constraints which cannot be ignored for correctly deforming them. Structure-aware shape deformation methods attempt to take such constraints into account.

2.3.2 Structure-aware shape deformation

As stated by Mitra et al. (2014), structure-aware shape deformation methods often rely on a two-step pipe-line: The first step aims at computing a set of features on an input model. In the second step, the identified features are automatically preserved while the user deforms the object. In the following, methods are sorted by category of feature preservation criteria.

2.3.2.1 Geometric details

Some authors call “detail” a small scale geometric component of a shape, as opposed to the “main shape” of the object. For example, the scales on the skin of a dragon can be considered as details due to their scale and repetition. Such details are often of a given size and shape which should be preserved, even though the main shape is deformed.

Chen and Meng (2009) proposed to represent geometrical details on an object as layers of geometric textures. This distinction allows them to apply non homogeneous scaling on the base object, and example-based texture synthesis on the detail texture. Alhashim et al. (2012) extended this method for arbitrary space deformations thanks to an adapted parametrization. Miao and Lin (2013) proposed to use local saliency measure of Lee et al. (2005) to perform a saliency-preserving local deformation. Dekkers and Kobbelt (2014) extended the notion of seam carving (already used in image-based deformation by Avidan and Shamir (2007) and Dong et al. (2009) for example) to mesh geometry, allowing meshes to be deformed while preserving salient features. Rohmer et al. (2015) Proposed a method for duplicating and removing geometric details through shape deformation.

Detail preservation methods are usually targeted at deforming organic shapes. Let us now explore other categories of objects.

2.3.2.2 Salient features

Man-made objects often exhibit salient features such as sharp edges. The shape of these features is usually essential to the functionality of the object, and therefore is seems natural to aim at preserving it through deformation.

Kraevoy et al. (2008) first proposed a non-homogeneous resizing method preserving circular features of man-made object models (see Figure 2.3b). Gal et al. (2009) introduced a deformation method relying on a feature-based shape representation called wires. Stanculescu et al. (2013) proposed a set of feature tags used inside rules defining how features are topologically transformed when deformed. Note that this introduces the notion of deformation behavior, that is later explored in Section 3.4.

Man-made objects often have another characteristic structure: linear arrangement.These structures have been studied through structure-aware deformation methods as we will see now.

2.3.2.3 Linear arrangements

Repetitive patterns in man-made objects often take the form of linear arrangements. Such structures can naturally be stretched along their repetition axis without changing their spatial frequency by duplicating the repeated element.

Owada et al. (2006) introduced a sketch-based interface for deforming repetitive pattern elements. Bokeloh et al. (2011) introduced the notion of "sliding docker" for proposing a deformation method based on axial controllers. Bokeloh et al. (2012) extended these notions to two-dimensional linear arrangements. Milliez et al. (2013) offered an original sub-part multiple-rest state algorithm for deforming one dimensional structures.

When the elements composing a man-made object are not as structured as in a linear arrangement, other deformation methods have to be devised.

2.3.2.4 Components manipulation

Component-based deformation methods propose to manipulate one or multiple sub-parts of an object while maintaining geometrical relationships between them.

Zheng et al. (2011) computes specific controllers for the components of complex 3D models, allowing the user to deform the right degrees of freedom for each component while maintaining inter- and intra-parts consistency. Controllers can be grouped, forming a

hierarchy. This method closely relates to the contributions presented in Section 3.4 of this manuscript.

Guerrero et al. (2014) proposed an edit propagation method allowing to preserve geometric relationships in 2D vector graphics (which can be extended to 2D layout of 3D objects).

2.3.2.5 Datasets exploration

Big datasets containing many different models of a same class of object are now a commonplace. Datasets exploration methods use such repositories for extracting structural knowledge about the object and devising object-specific deformations.

Ovsjanikov et al. (2011) introduced a method for exploring shape repositories through continuous variations. Yumer et al. (2015) proposed an innovative method for associating quantitative semantic ratings to a shape, and proposed a shape deformation paradigm based on modifications of these ratings.

2.3.3 Sketch-based deformation

Sketch-based deformation methods aim at inferring a 3D object deformation from a user-sketched screen-space stroke. The idea behind such interaction is that it is more intuitive for the user to describe deformation over the projected representation of the model rather than in the full 3D space.

Singh and Fiume (1998) proposed a sketch-based method for deforming 3D models based on pre-defined "wires", which are structural curves on the object. Apart from this method, most sketch-based deformation techniques do not rely on other data than the shape to deform. A distinction can be made between contour deformation and skeleton deformation.

2.3.3.1 Contour deformations

Also called "over-sketching", contour deformation methods interpret the user sketch as a silhouette constraint. The two major challenges are then to identify the object contour part to deform, and the deformation to apply for satisfying the constraint. Such methods are more suited for creating details on shapes.

Nealen et al. (2007) and proposed a method in which both the region to deform and the constraint have to be sketched. Zimmermann et al. (2007) and Zimmermann et al. (2008) extended the previous method by automatically computing the region to deform based on the constraint (see Figure 2.3c).

2.3.3.2 Skeleton deformations

Skeleton deformation methods do not interpret user strokes as silhouette constraints. Instead, they consider them as constraints on the mean shape of a portion of the object. Such methods are better suited to prescribing large scale deformation, such as limb positions on character models.

Kho and Garland (2007) devised a two-stroke method, where the first stroke defines the region of the shape to be deformed, and the second stroke describes the shape constraint.

Guay et al. (2013) rely on animation skeleton for prescribing single stroke deformations used for posing.

2.3.4 Limitations of deformation methods

Numerous object deformation methods managed to propose solutions to many sub-problems, in terms of property to preserve through edits or in terms of deformation input setup. As with object generation, all of these methods bring a valuable contribution, but all those contributions are hard to put together. The work presented in Section 3.4 proposes a method for mixing multiple deformation methods in a single hierarchical object.

The last method we discussed deals with *poses*. Several poses for a same shape are interpolated for creating an animation. Hence, deformation methods can often be used for creating animations. However, the following section presents a type of animation that can not be represented by pose interpolation: fluid animation.

2.4 Animation Design

"Animation" itself is an extremely large notion, ranging from the dynamic facial expressions on a virtual character to the motion of a collapsing building. Here, we focus on *fluid animation*; i.e. the motion of liquids and gas.

Unlike most objects in Computer Graphics, fluids are by definition very variable in their shape. This raises particular challenges in terms of modeling and control, as we will see now.

2.4.1 Fluid animation modeling

As with static shapes, animation modeling can be tackled from several angles. The difficulty of handling animated data makes automated methods beneficial. As a result, procedural and simulation methods are prominent. Note that in the context of fluid animation, "procedural" means non simulation-based procedural.

2.4.1.1 Procedural

The range of shapes and behaviors that can be taken by fluids makes it hard to model their animated geometry. Instead, the velocity field can be modeled procedurally: For example, Bridson et al. (2007) proposed a method based on Perlin noise (Perlin (1985)) for generating an incompressible velocity disturbance.

However, under certain hypothesis, fluids can be represented quite simply. The most common example is the representation of waves using height-fields. Various procedural models were proposed for modeling animated deep water waves: Hinsinger et al. (2002), Tessendorf (2004), and Nielsen et al. (2013). Recently, Jeschke and Wojtan (2015) and Horvath (2015) proposed wave models suited to shallower waters.

The method presented in Section 4.2 uses a procedural model for representing waterfalls.

These models are efficient but uneasy to integrate with other object (e.g. a boat) and limited (it only represents waves). For these reasons, most fluid models rely on simulation, as we will see now.

2.4.1.2 Simulation

Since the seminal work of Stam (1999) proposing to solve Navier-Stokes equations in a stable way, many fluid simulation methods have been proposed. Liquids in simulations are often represented as particles, as in the SPH model (see the survey of Ihmsen et al. (2014) for more details), or as a coupled particle-grid structure in the FLIP model (as in the method of Ferstl et al. (2016)). Concerning smoke, see the survey of Huang et al. (2015).

2.4.1.3 Limitations

Both procedural and simulation fluid models can produce detailed and realistic results. On the other hand, the output of such methods is very hard to control.

Control is a central point in modeling, and the next section describes methods attempting to provide it.

2.4.2 Fluid animation control

When using procedural modeling tools, control can be performed through the setting of input parameters. As for static object generation method, parameter setting is non-intuitive and tiresome. However, to my knowledge, no inverse procedural modeling method offers to find those parameters automatically. Besides, as stated above, the limited scope of procedural fluid models makes that simulation is used in most cases.

Simulations are notoriously hard to control. However, some work has been done in that direction: Since the introduction of *space-time constraints* by Witkin and Kass (1988), direct editing of simulation has been addressed in different contexts. Rigid bodies simulations have been addressed by Popović et al. (2000), Chenney and Forsyth (2000), and Twigg and James (2007); Whereas deformable objects simulations have been tackled by Wojtan et al. (2006), Barbič et al. (2009), Barbič et al. (2012), Schulz et al. (2014), and Li et al. (2014). However, few work address fluid surfaces, due to their constantly changing shape and topology that makes the output geometry inaccessible to standard deformation tools. This section focuses on two main methods for designing fluid animations: by *controlling the simulation*, and by *editing the animation*.

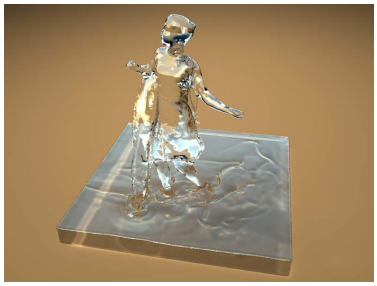
2.4.2.1 Fluid simulation control

The general approach for controlling a simulation is based on trial and error. Besides, as mentioned in Chapter 1, this process has severe drawbacks which make the design of fluid animation especially tedious. To overcome these limitations, several methods propose to guide the fluid behavior by using geometric proxies which are easier to control than the high resolution fluid simulation data itself.

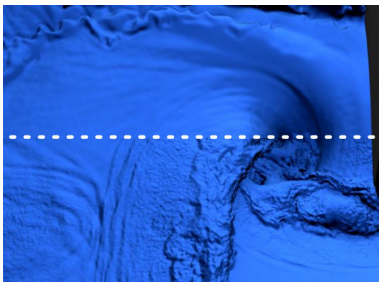
Parameter tuning The classical forward method consisting in tuning the input parameters of the simulation has been little developed.

Foster and Metaxas (1997) introduced a method based on simulation parameter tuning, but the range of results achievable through this method is very limited. Brousset et al. (2016) proposed a wave simulation controlling method based on intuitive wave parameters.

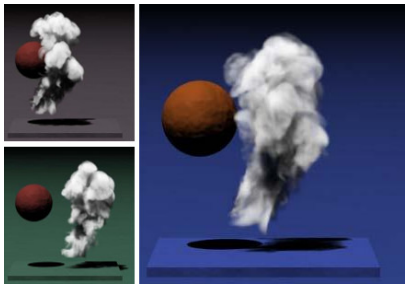
Changing parameter values is not always the easiest way to achieve a creative result with fluid simulation. Inverse methods presented after introduce more intuitive controls.



(a) The method of Raveendran et al. (2012) allows the user to control liquid simulation using animated meshes (here, the mesh of a dancer).



(b) Mercier et al. (2015) proposed a method for adding details to a liquid animation, enhancing the resolution of the inputs (up: original animation, down: up-sampled animation).



(c) Thürey (2016) offers to interpolate between multiple simulation-generated animations for creating new ones.

Figure 2.4 – Fluid control is often performed using external animation data for generating forces (a), or by increasing the resolution of an existing animation (b). However, some novel methods propose to use animation data directly without re-simulating (c).

Key-frame-based control Key-frames are commonplace in animation modeling outside fluids. Transposing them in the context of fluid modeling seems natural.

Some methods were proposed by Treuille et al. (2003), Mihalef et al. (2004) and McNamara et al. (2004). However, matching a particular shape at a particular time of the animation is not the only nor the most desirable controlling method that can be desired.

Surface-based control Artists can use an animated triangle mesh to specify a moving target shape for the fluid. This is usually done by adding artificial attraction forces to fluid particles or grid based on the distance to the mesh surface.

Such approaches have been successfully developed to drive smoke by Fattal and Lischinski (2004), Hong and Kim (2004), and Shi and Yu (2005a). Liquids simulations have also been addressed by Shi and Yu (2005b) and Raveendran et al. (2012) (see Figure 2.4a).

These methods require to run a full fluid simulation in order to see the results. That is not the case of resolution enhancement methods.

Resolution enhancement As an extension to surface-based control, the attracting surface itself can be define using a low-resolution fluid simulation. To achieve this, the artist quickly sets up a coarse simulation and uses the output geometry to guide the main features of a full resolution simulation.

This resolution enhancement approach has been applied to smoke simulation using optimization by Nielsen et al. (2009) and Nielsen and Christensen (2010). Yuan et al. (2011) used patterns extracted as skeleton, and Huang and Keyser (2013) used sparse sampling. Huang et al. (2011) proposed a method for constraining a fine-scale smoke simulation to reproduce the features of a coarse-scale one.

For liquid simulations, Nielsen and Bridson (2011) proposed to restrict the high resolution simulation to a thin layer around a guiding coarse animation defining the constraint.

In order to add details without running extra simulations, simulation results can be processed for adding procedural or physically-based animated details. Narain et al. (2007) proposed a texture-based method for adding details to a liquid mesh sequence. Kim et al. (2013) and Mercier et al. (2015) extended this approach for creating geometrical details on top of such animations (see Figure 2.4b). Horvath and Geiger (2009) developed a resolution-enhancement method for controlling fire animations.

Trajectory-based control Fluid trajectories, for their part, may also be a control asset. Kim et al. (2006) proposed to control these trajectories using user-defined velocity fields, while Yang et al. (2013) introduced a control method based on distance fields. Thürey et al. (2006) and Madill and Mould (2013) proposed to use specific control particles.

Feature-based control Although each of these approaches are able to successfully guide a fluid simulation, they do not enable direct control of the resulting fluid. Designing precise timing or feature scaling would therefore still require iterative trial-and-error steps to converge toward a desired animation. Few attempts have been made to enable direct control on the simulation.

Schpok et al. (2005) proposed to extract and parametrize features such as vortices, uniform advection, sinks, and sources to allow the user to modify the parameters in a smoke simulation.

In the context of liquid simulation, Pan et al. (2013) proposed a method to deform wave shapes by sketching their profiles. This approach enables direct spatial deformation but does not allow temporal editing, and the simulation needs to be re-computed from the modified frame onwards.

Data-based control Videos are sometimes useful as examples of desired results. Bhat et al. (2004) proposed a method for directly editing waterfall video using flow strokes. However videos are 2D. But some methods propose to use videos for constraining a simulation, yielding video-matching 3D animated fluid representations.

Wang et al. (2009) introduced a model for reconstructing a liquid animation from a depth video. Gregson et al. (2014) proposed a method for achieving this reconstruction for smoke using multiple view-points. Okabe et al. (2015) proposed an extension using a single view-point.

Limitations Simulation control methods still offer indirect control mechanisms. Besides, re-running a simulation until a desired output is generated is often sub-optimal, since it makes it impossible to select a given region in space and time for using it in another context.

2.4.2.2 Fluid animation editing

Animation editing methods aim to directly modify the result of a simulation without the need to re-simulate. In contrast with simulation control, animation editing is often a faster approach to slightly modify an existing animation.

Very few works have been proposed for the edit of fluid animations. For smoke and explosion animations, Pighin et al. (2004) proposed to parametrize density and temperature

fields from the simulation using advected radial basis functions. The parametrized data are then deformed using a trajectory-based editing tool.

For liquid animations, Raveendran et al. (2014) proposed a semi-automatic method to match two animations and smoothly blend between them. Thürey (2016) extended this concept using a 5D optical flow parametrization allowing it to interpolate between multiple animations of liquid and smoke (see Figure 2.4c). These methods can quickly produce in-between frames and explore the "animation shape space" between multiple animations.

However, the interpolation paradigm limits itself to space defined by pre-computed data, and do not offer tools for creating completely new animations. This limits the range of achievable results and the expressiveness of the method.

Section 4.3 introduces a method for analyzing general liquid animations in order to propose high-level edit operations.

2.5 Conclusion

To conclude, there is an enormous amount of research into all three axes presented here. While producing good results independently, most of those methods are often hard to combine in a same pipe-line. From an artist stand-point, this limits the usability of such methods.

Besides, methods offering high-quality results – procedural static object generation techniques and physically-based fluid simulation – often offer little user control. Once more, artists might be driven away from using such methods because of their lack of controllability.

This thesis proposes direct methods for generating and/or manipulating complex virtual content through direct tools. When this is possible, these methods re-use state-of-the-art work and integrate these in a more usable framework.

Chapter 3

Design of Static Objects

Contents

3.1 Introduction 32
3.1.1 Complex object framework 32
3.1.2 Coherency 34
3.1.3 Example-based modeling of complex objects 35
3.2 Part-Based Modeling 36
3.2.1 Introduction 37
3.2.2 Overview of the solution 37
3.2.3 Main contribution: Geometrical Sub-Object Deformation . . . 42
3.2.4 Results 43
3.2.5 Discussion 46
3.3 Worldbrush 48
3.3.1 Vectorial analysis 48
3.3.2 World painting 50
3.3.3 Main contribution: RDF Interpolation 53
3.3.4 Results 55
3.3.5 Discussion 55
3.4 Deformation Grammars 61
3.4.1 Introduction 61
3.4.2 Deformation Grammars 62
3.4.3 Bilateral Grammar Rules 64
3.4.4 Results 67
3.4.5 Discussion 71
3.5 Conclusion 75
3.5.1 Controllability versus complexity trade-off 75
3.5.2 Smart tools versus smart shapes 75

Chapter 1 highlighted the difficulties arising from 3D shape modeling, whereas chapter 2 presented some existing solutions allowing to overcome these issues in some cases. This chapter extends the possibilities of interactive complex shape design by presenting three novel methods. Each of these enable to reuse already existing objects or scenes in order to create new ones. Therefore they belong to the vast family of example-based techniques. Their respective core idea can be summarized as follows:

— Using existing shape segmentation for automatically learning connection rules and generating new valid shapes (Section 3.2);
— Analyzing local distribution properties of groups of objects for re-synthesizing it at user-selected locations through a painting metaphor (Section 3.3);
— Decomposing general deformations along complex objects hierarchies for a generally coherent deformation behavior under sculpting interactions (Section 3.4).

Methods are presented in chronological order of inception, but more importantly in ascending order of user control, which is at the core of recent development in Computer Graphics research. Besides, they fit into a common framework, which is presented next (Section 3.1).

3.1 Introduction

This section introduces the notions of *complex object* and *coherency*. Initially introduced by Vimont et al. (2016) for defining deformation grammars, this framework is used here as a reference frame for the study of 3D shape modeling.

3.1.1 Complex object framework

In the remainder of this manuscript, we call *simple object* a geometric object in the classical sense. It is fully defined by the *internal parameters* of its visual representation: For example, it can be a triangular mesh defined using the positions of its vertices and their indexing into faces.

The notion of *complex object* recursively extends the notion of simple object by adding a set of *hierarchical parameters* which are references to *sub-objects*. Sub-objects can themselves be simple or complex. The internal parameters of a complex object do not always correspond to a visual representation: For instance a heap of stones may include internal parameters such as a maximum slope or number of stones, while the visual representation may be only be stored in its children - e.g. simple objects representing the individual stones. In the remaining of this chapter, we call *element* any complex or simple object.

Given an element ε, its sub-objects (if any) are called its *children*, noted $C(\varepsilon)$. Accordingly, ε has a *parent* noted $P(\varepsilon)$ such that $\varepsilon \in C(P(\varepsilon))$.

In our formalism, each element ε is associated with a *semantic type* $t(\varepsilon)$: For example the type "stone" for a simple object representing a stone.

Notations relative to complex objects are summarized in table 3.1.

3.1.1.1 Creating a complex object

Complex objects are general enough for representing a wide variety of 3D shapes, as shown in Figure 3.1. The idea behind this formalism is to order any complex shape into a

Symbol	Description
ε	element
$C(\varepsilon)$	children set of element ε
$P(\varepsilon)$	parent of element ε
$t(\varepsilon)$	type of element ε

Table 3.1 – Notations relative to complex objects.

meaningful hierarchy allowing interactions at different levels. The way this hierarchy is created is not discussed here: It can be described manually by a user or may result from the use of a procedural modeling tool to build the object, such as a shape grammar (see Figure 3.2) or a L-system for a tree, as described by Prusinkiewicz and Lindenmayer (2012). Alternatively, this hierarchy could be retrieved from an input shape using hierarchical segmentation method such as the one of Attene et al. (2006).

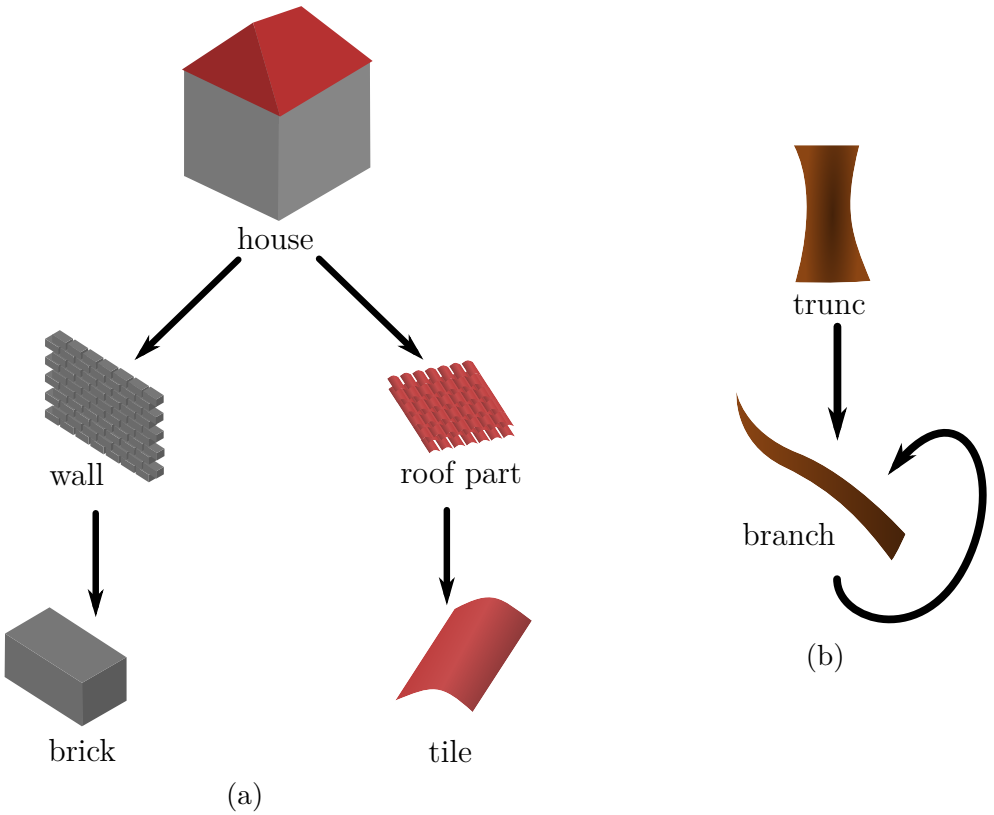


Figure 3.1 – Complex objects can be used for organizing various objects into hierarchies, such as houses (a) and trees (b). Here is a simplified view where only the type of each sub-object is shown (a house has in fact multiple walls, and a tree has multiple branches and sub-branches). Note that in the house example, only the simple objects have a visual representation (the house is made of bricks and tiles), while all objects store a trunk or branch model in the tree example.

3.1.1.2 Example

Let us expand a concrete and naive example of a complex object: a tree. It is constituted of a cylindrical trunk and branches subdivide at their extremity into a few smaller branches, as seen in Figure 3.4a. This tree can be represented as a complex object ε_0, using a large

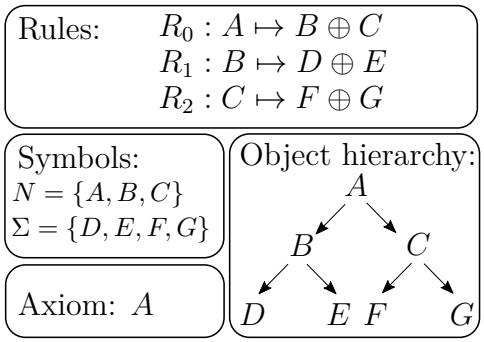


Figure 3.2 – Standard use of grammars to define hierarchical shapes.

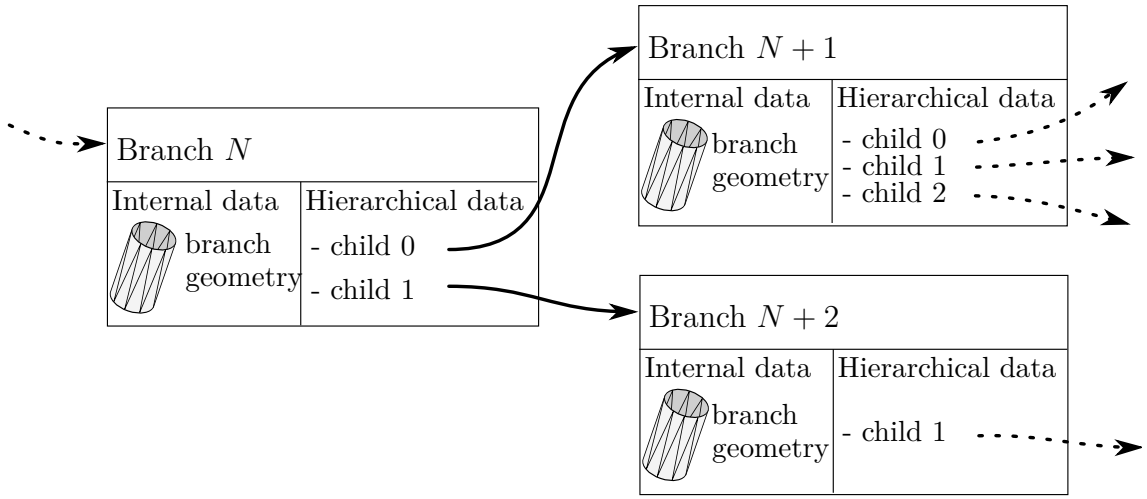


Figure 3.3 – A tree model can be organized hierarchically into a complex object. This allows to consider individually each level of the hierarchy as well as the relationships between levels.

cylinder (for the trunk) as internal representation – in this case this representation is also visual – plus a set of references to the main branches. Each branch is itself represented as a cylinder plus a set of references to children branches, as shown in Figure 3.3. The smallest branches at the extremities are simple objects, with only a visual representation but no sub-branch; The other ones are complex objects. All are of the same semantic type as ε_0.

3.1.2 Coherency

When dealing with virtual content in general (and 3D shapes in particular), one implicit phenomenon is at the root of the experience: The user (video game player, engineer, movie viewer for example) accepts the virtual model as a valid representation or visual model of what is supposedly displayed on the screen. For that, car models must *look like* cars, characters like characters, and landscapes like landscapes. This stands even though the actual cars, characters, and landscapes do not exist and might not even correspond to any real-life object.

This last point leads us to postulate that *model validation* should not fully rely on the geometrical representation. Also important is a set of semantic constraints that the model must satisfy to look like what it is representing: Cars must have wheels in contact with the ground, characters must have limbs and move, landscapes must be huge and

static. These constraints are highly important to ensure that the model will be correctly interpreted and accepted.

3.1.2.1 Terminology

Of course this notion has been called differently in previous works: structure (Bokeloh et al. (2012), Mitra et al. (2014)), soundness (Whiting et al. (2009)), manufacturability (Brouet et al. (2012)), and more (design, validity, consistency). In this chapter, we use the word *coherency* for designating this semantic link between an object and its representation.

In the framework of complex objects, the coherency of an element can be expressed as a function of its parameters. More precisely, two sub-notions can be differentiated: *Internal coherency*, which relies on internal parameters of the element; *Hierarchical coherency*, which relies on hierarchical parameters of the element.

3.1.2.2 Example

In the case of the simple tree model already discussed, the internal coherency of a branch could be for instance the cylindricity of the internal triangular mesh that represents it. The hierarchical coherency of a branch could be that all its sub-branches start from its extremity, or that their radius should be smaller.

3.1.3 Example-based modeling of complex objects

Section 3.1.1 defined a very general framework for considering hierarchical 3D shapes, while Section 3.1.2 explained what makes these shapes usable. This last point highlights the usefulness as well as the difficulty of modeling complex objects: They are ubiquitous and rely on a vast set of constraints to be coherent. It becomes therefore attractive to re-use complex object models once created: this is the goal of example-based techniques. However, naively modifying such models might break one of the constraints they were previously satisfying, making them non coherent.

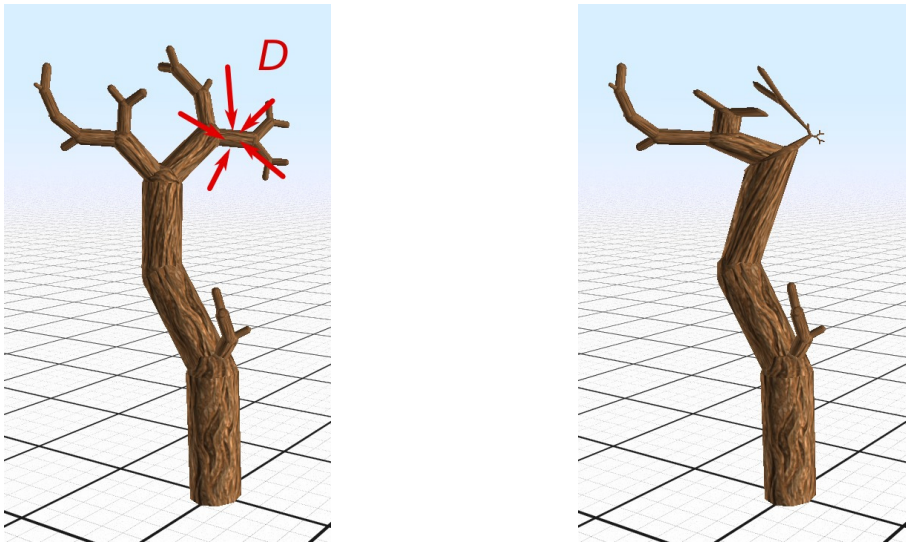
Performing the synthesis or the deformation of a complex object model without breaking its coherency is the challenging goal explored by this chapter.

3.1.3.1 Example

To illustrate the difficulty of deforming a complex object while maintaining its coherency, let us consider the former tree example.

Let us consider a geometrical deformation $d : \mathbb{R}^3 \to \mathbb{R}^3$. Directly applying d on the tree mesh (by displacing individual mesh vertices) make the whole model loose its coherency: Branches are not cylinders anymore, as can be seen in Figure 3.4. In some cases, this prevents the model to be recognized as a tree.

The notion of *shape space* explained in Section 1.2 offers another explanation of this difficulty. A classical deformation tool without knowing of the shape space of the object it is deforming will unavoidably take the object model out of its shape sub-space. On the other hand, the notion of coherency gives some conditions on a model for belonging to the shape sub-space. These conditions can be used for designing coherency-preserving deformations. Alternatively, they can be used for projecting a model back on the shape space after it has been deformed.



(a) (b)

Figure 3.4 – An initial complex object (a) is deformed by the user using a local deformation tool applied at the top right. (b): The object is deformed as a whole without any constraint being maintained; Note how the geometry branches gets degenerated.

Section 3.4 presents a general hierarchical deformation interpretation framework allowing to maintain coherency through such deformation. But before that, we explore two other example-based methods that respectively tackle example based modeling of assembly shapes and object distributions.

3.2 Automatic Synthesis: Part-Based Modeling

The work presented in this section results from an international collaboration with Han Liu (KAUST), professor Michael Wand (Universiteit Utrecht), professor Stefanie Hahmann (Université Grenoble Alpes/Inria), and professor Niloy J. Mitra (UCL). The resulting paper was published in the CGF journal (see Liu et al. (2015)) and presented at the Eurographics conference in 2015 (`www.eurographics2015.ch`):

Liu, H., Vimont, U., Wand, M., Cani, M.-P., Hahmann, S., Rohmer, D., and Mitra, N. J. (2015). Replaceable substructures for efficient part-based modeling. *Computer Graphics Forum, Proceedings of Eurographics 2015*, 34(2):503–513

This section is organized as follows: Sub-section 3.2.1 roots the problem of part-based modeling in the framework of complex objects; Sub-section 3.2.2 explains the solution offered in this work; Sub-section 3.2.3 details the geometrical part of the method, which is my main contribution to this work; Sub-section 3.2.4 shows results of this method and draws its limitations and perspectives. For a review of related works, please refer to Section 2.2.

3.2.1 Introduction

This work addresses the problem of part-based modeling of 2-levels complex objects. The top level of the hierarchy is a single complex object ε_0 with no visual representation. The bottom level is composed of the parts of ε_0 and carries its visual representation.

The main assumption on the object can be formulated as a strong topological top-level hierarchical coherency: Two bottom-level elements ε_i and ε_j should be connected (i.e. in contact) only if the pre-recorded type connectability table R allows it: $R(t(\varepsilon_i), t(\varepsilon_j)) = true$.

3.2.1.1 Shape graph and docking rules

The coherency formulation supposes that bottom-level elements can be connected to one another similarly to jigsaw puzzle pieces. This connection is at once topological (semantic types corresponds to connectible pieces) and geometric (the shapes of the elements indeed form a connection). The geometrical connection zone on each element is called a *docking site*. On the topological side, these connections can be represented as a graph, which we call *shape graph*, noted $G = (V, E)$ (see Figure 3.6). The semantic type of an element $t(\varepsilon)$ translates into a constraint on having a given set of docking sites, and therefore the connectability table can be represented as a boolean matrix where each cell represents a *docking rule*, i.e. an element type pair docking site matching (see Figure 3.10). These rules constitute what we call a *tiling grammar*.

3.2.1.2 Part-based modeling

When creating the 3D shape of a complex object, it might be hard to ensure that the set of all constraints between its elements – i.e. its coherency – is respected. As explained by Mitra et al. (2014), part-based modeling offers to use a preexisting coherent complex object model as input, and to mix and match its elements in a coherency-preserving manner in order to generate an output.

3.2.2 Overview of the solution

The cornerstone of this approach is to identify substitutable substructures inside pairs of annotated input complex objects. Once identified, such substructures can be replaced by one another, creating a novel complex object, as illustrated in Figure 3.5.

3.2.2.1 Input specifications

The input of this method is composed of segmented 3D models representing complex objects such as element assemblies. The model in itself is a triangulated mesh, and each element is supposed to be represented as a singular connected component inside the mesh. Each part has a label which represents its functionality.

Another semantic information is required as input: connections between parts. Two parts can be "connected" to one another if they share some user-defined functional relationship. Connections between parts are stored as a graph where the nodes represent the elements (with associated label) while the edges represent connections between elements. See Figure 3.6.

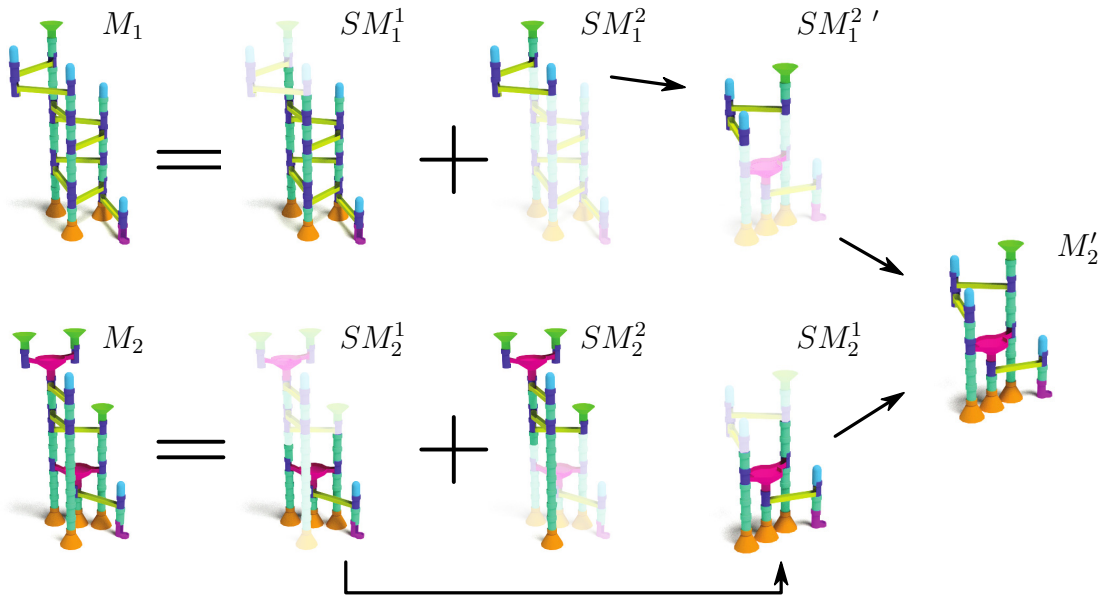
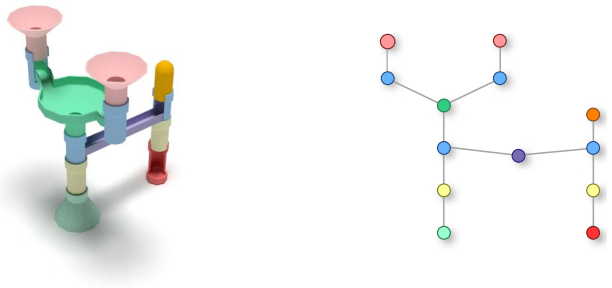


Figure 3.5 – Pipe-line of the part-based modeling method: Starting from a pair of segmented complex objects (M_1, M_2) as input, our method identifies matching sub-objects (SM_1^2, SM_2^2). After deforming SM_1^2 into $SM_1^{2\prime}$, it can be assembled with SM_2^1 for creating a new coherent complex object M_2'.



(a) Shape M (b) Shape graph G

Figure 3.6 – For a given 3D model M, the shape graph G expresses topological information which will further be used by the method. Parts of M correspond to nodes of G, while part connections correspond to edges.

The shape graph will later be used as a coarse-level shape abstraction allowing to treat the problem of finding replaceable substructures in the topological domain. The geometry of the substructures will be taken into account later. The de-correlation of both aspects of this intricate problem allows to tackle the topological sub-problem without geometrical constraints which makes it easier. This constitutes both the benefits and the drawbacks of this approach, as will be discussed in Section 3.2.4.

Symbol	Description
M	complex object model
SM	replaceable sub-structure
V	shape graph nodes
E	shape graph edges
$G = (V, E)$	shape graph
S	sub-graph
G^*	dual shape graph
dS^*	matching cut
G'	transformed graph
M'	transformed object
$\mathcal{T}_{SM,SM'}$	geometrical transformation from SM to SM'
R	rule table

Table 3.2 – Notations relative to replaceable sub-structures.

3.2.2.2 Topological processing pipeline

The most challenging part of the method is the replaceable sub-structure identification. Thanks to shape graphs, this problem boils down to a particular case of subgraph matching. However, this problem is still NP-hard (see Cook (1971)).

As explained in Figure 3.7, the method presented here circumvents this complexity by using a heuristic approach:

— Let G_1 and G_2 be two shape graphs respecting the coherency rules R.
— Compute G_1^* and G_2^* their dual shape graphs.
— Two equivalent circuits ∂S_1^* and ∂S_2^* are identified inside G_1^* and G_2^*. ∂S_1^* and ∂S_2^* have the same length, and contain dual nodes of the same type in the same order.
— In the primal space, ∂S_1^* and ∂S_2^* corresponds edge sequences which isolate two subgraphs (a.k.a. *cuts*), noted $S_1 \subset G_1$ and $S_2 \subset G_2$.
— By construction, S_1 and S_2 satisfy the same border constraints, they are said *replaceable*. A novel pair of shape graphs can be generated by replacing S_1 with S_2 inside G_1 and S_2 by S_1 inside G_2, yielding G_1' and G_2' respectively. The replaceability of S_1 and S_2 insures that G_1' and G_2' still abide by R.

Dual graph Given a shape graph G, we consider its dual graph G^* such that nodes of G^* correspond to edges of G, and carry a type composed of the two types of its extremity vertices. Two nodes of G^* are connected if and only if:

— their primal counter-parts are incident; and
— their primal counter-parts are consecutive in the *local ordering system* of the adjacent node.

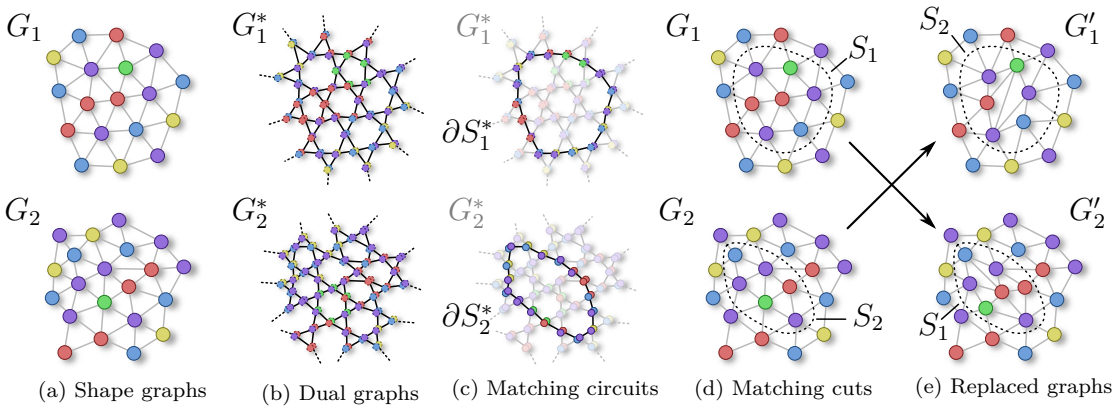


Figure 3.7 – From left to right: Two input shape graphs G_1 and G_2 are analyzed through their duals G_1^* and G_2^*. Identical circuits ∂S_1^* and ∂S_2^* are found, which yields a pair of cuts in the primal space. Those cuts isolate a pair of sub-graphs S_1 and S_2 which are replaceable by construction: They can be substituted, generating the two output graphs G_1' and G_2', which are coherent shape variations of the original.

The *local ordering system* of a node is a way of ordering edges adjacent to a node n. A plane P approximating the centers of its 1-neighborhood is computed using principal component analysis (PCA); Projected edge centers are clock-wise ordered and consecutive corresponding dual nodes are connected in G^* (see figure 3.8).

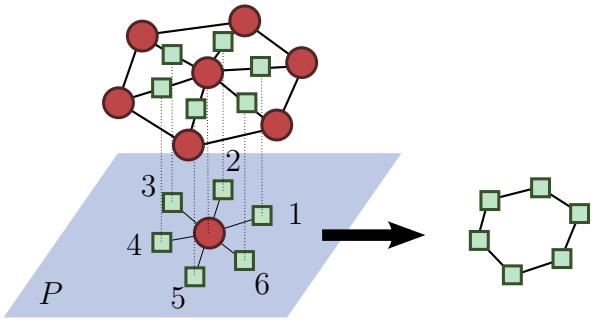


Figure 3.8 – Shape graph nodes are projected onto the local neighborhood approximation plane P and clock-wise ordered (left). Consecutive dual nodes (i.e. primal edges) in that ordering are connected in the dual shape graph (left).

This particular type of dual graph is closely related to the notion of *line graph* in graph theory, which is commonly used in graph isomorphism problems, as shown by Whitney (1932).

Replaceable subgraphs are characterized by pairs of corresponding cuts that still respect all of the learned rules (i.e. matching) when the interior of one is exchanged with the interior of another. We call the resultant operations *(grammar-consistent) subgraph substitution*.

$$\partial S_1^* = \partial S_2^* = ($$  $$)$$

Figure 3.9 – Two graphs are *replaceable* if and only if their frontier cuts are equivalent. Here, both cuts ∂S_1^* and ∂S_2^* from graphs of Figure 3.7 are equivalent.

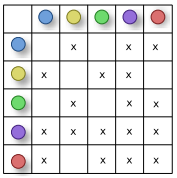
$R =$ 

Figure 3.10 – Substituting replaceable subgraphs implicitly preserves the coherency of the object. Here an adjacency matrix represents the coherency of the graphs of Figure 3.7.

Dual graph algorithm The search algorithm can be more conveniently formulated using dual shape graphs pair (G_1^*, G_2^*). In this configuration, we search for a pair of isomorphic subgraphs, which is a standard NP-hard problem. However, we have some extra constraints:

— Both subgraphs must be circuits; Such subgraphs are isomorphic if and only if they have the same cardinality.

— Node types should appear in the same order along the two circuits.

This allows us to devise Algorithm 1: For each pair of nodes $(v_1, v_2) \in V_1^* \times V_2^*$ such that $t(v_1) = t(v_2)$ it recursively depth-first searches for type-matching neighbors. The algorithm therefore explores similar paths between both graph, backtracks when encountering different node types, and outputs the path when it is closed. The early determination of path mis-match practically allows to discard many invalid circuits.

Algorithm 1 Matching cut depth first search.
This algorithm is called for all pairs of nodes in $V_1^* \times V_2^*$ taken as initial cuts.

Input: Two dual node sequences $C_1 = (v_1^i)_i$, $C_2 = (v_2^j)_j$ of same size L and node types $t(v_1^i) = t(v_2^i) \forall i \in [1 \dots L]$
if C_1 and C_2 form cycles **then**
  Return success
end if
Collect all unvisited adjacent nodes of v_1^L and v_2^L, denoted as N_1 and N_2;
if $|N_1| = 0 || |N_2| = 0$ **then**
  No matching cut can be found, return failure
end if
for all $n_1^i \in N_1$ **do**
  label n_1^i as visited
  for all $n_2^j \in N_2$ **do**
    label n_2^j as visited
    if $t(n_1^i) == t(n_2^j)$ **then**
      $C_1 \leftarrow n_1^i$, $C_2 \leftarrow n_2^j$
      if DFS(C_1, C_2) **then**
        Return success
      end if
      pop n_1^i from C_2
      pop n_2^j from C_1
    end if
  end for
end for

Robustness to real-world imperfections Exact matching can fail in practice due to imperfect input such as inconsistent annotations, imprecise segmentations, or small shape variations. In particular, we have observed in our experiments that the ordering of non-manifold graphs using PCA-based tangent-plane projection is not always reliable.

We therefore permit limited violations of the ordering by introducing *dummy edges* that cover sparse outliers. We add a potential dummy edge to the dual graph between all nodes with a graph distance of two (nodes with distance one directly connected), irrespective of grammatical constraints, thereby short-cutting over local defects. We minimize the number of violations by exhaustively searching graphs with up to $q \in \{0, 1, 2, 3 \ldots\}$ violations. This search is exponential in q. In our experiments, we used $q \leq 2$.

3.2.3 Main contribution: Geometrical Sub-Object Deformation

Once the subgraphs to be substituted have been identified, the actual sub-structure (i.e. the elements associated to the nodes of the subgraph) replacement is still to be performed.

In the following, we consider two input complex objects O_1 and O_2 along with their respective shape graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Let $S_1 \subset G_1, S_2 \subset G_2$ denote the replaceable subgraphs, and SM_1 and SM_2 the corresponding sub-structures. We look for a geometrical transformation $\mathcal{T}_{SM_2,SM_1}$ such that $\mathcal{T}_{SM_2,SM_1}(SM_2)$ fits into O_1, yielding O_1'. Note that $\mathcal{T}_{SM_1,SM_2} = \mathcal{T}^{-1}_{SM_2,SM_1}$ can be computed in the exact same way.

3.2.3.1 Boundary constraint

In order for SM_2 to fit in place of SM_1, we consider only boundary geometrical constraints: The shape of the boundary of SM_2 should match the one of SM_1, and the rest of the sub-structure should nicely blend inside this boundary.

For $k \in \{1, 2\}$, let us consider an edge $e^i_k \in \partial S^*_k$. This edge connects two vertices: one belongs to SM_k, noted v_1, and the other belongs to $G_k \setminus SM_k$, noted v_2. We note BB_1 and BB_2 the axis-aligned bounding boxes of the elements associated with v_1 and v_2 respectively. We associate e^i_k to an orthonormal frame $f(e^i_k)$ representing the position and the orientation of the connection between v_1 and v_2. It is computed as follow:

— The origin $\mathcal{O}$ of $f(e^i_k)$ is computed as the center of the bounding boxes intersection $BB_1 \cap BB_2$;
— The X axis of $f(e^i_k)$ is aligned with the line connecting bounding box centers $(\widehat{BB_1}, \widehat{BB_2})$;
— The Z axis is the normalized projection of $(0, 0, 1)$ on the plane $(\mathcal{O}, X)$;
— The Y axis is computed directly as $Y = Z \times X$.

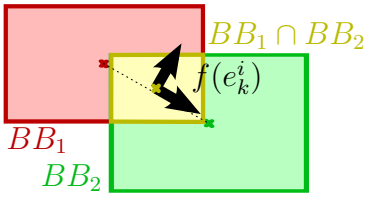


Figure 3.11 – Boundary frame position and orientation are defined from the bounding boxes of adjacent elements.

3.2.3.2 Skinning weights computation

We note $t_{f(e_2^i),f(e_1^i)}$ the rigid transformation from $f(e_2^i)$ to $f(e_1^i)$. $\mathcal{T}_{SM_2,SM_1}$ is computed as a per-vertex linear combination of those rigid transformations using classical linear blend skinning. Skinning weights are computed per vertex using normalized inverse distances to frame origins.

Rigid element case. Non-homogeneous weighting over an element's vertices induces a non-rigid transformation of this element. For allowing given elements to be rigidly transformed, the user has the possibility to tag them; This results in the weights of all vertices of the element to be averaged.

Now that the substructure substitution pipe-line has been fully described, let us now look upon the results of the method.

3.2.4 Results

This section shows results of the method and opens a discussion on its limitations.

Replacing complex objects sub-structures allows to generate variations. We support three main modes of replacement, which are described next: *Cross-object replacement*, *In-object replacement*, and *Structural variation*.

3.2.4.1 Cross-object replacement

This first mode of replacement happens when matching substructures are substituted between two different complex object models. This is the standard replacement mode. Figure 3.12 shows synthesized shape variations obtain from a pair of input ball track models. Once a given cross-model replacement has been performed, two output objects are generated, which can in turn be used as input. The number of potential input therefore grows quadratically, allowing for a wide variety of shape variations.

Figure 3.13 shows that a family of complex objects (here: tractor models) can be combined amongst one another. This multiple inputs yields again a broad variety of outputs.

3.2.4.2 In-object replacement

Here, matching substructures are detected and replaced inside the very same object. The method is the same as for cross-object replacement, where the first model is duplicated for creating the second one. In this case, care has to be taken not detect trivial cases: when both subgraphs are actually similar. Figure 3.14 presents results of in-object replacement on castle, playgrounds and racetracks models. An interesting case arises when one substructure is also a substructure of the other substructure: $S_1 \subset S_2$. Here, replacing S_2 with S_1 re-introduces S_1 into M, which again allows to replace S_1 with S_2 iteratively.

3.2.4.3 Structural variation

Finally, complex objects can be topologically combined (subgraphs are indeed replaced) while keeping their element's geometry. This is called structural variation. Examples of this replacement mode are shown in Figure 3.15.

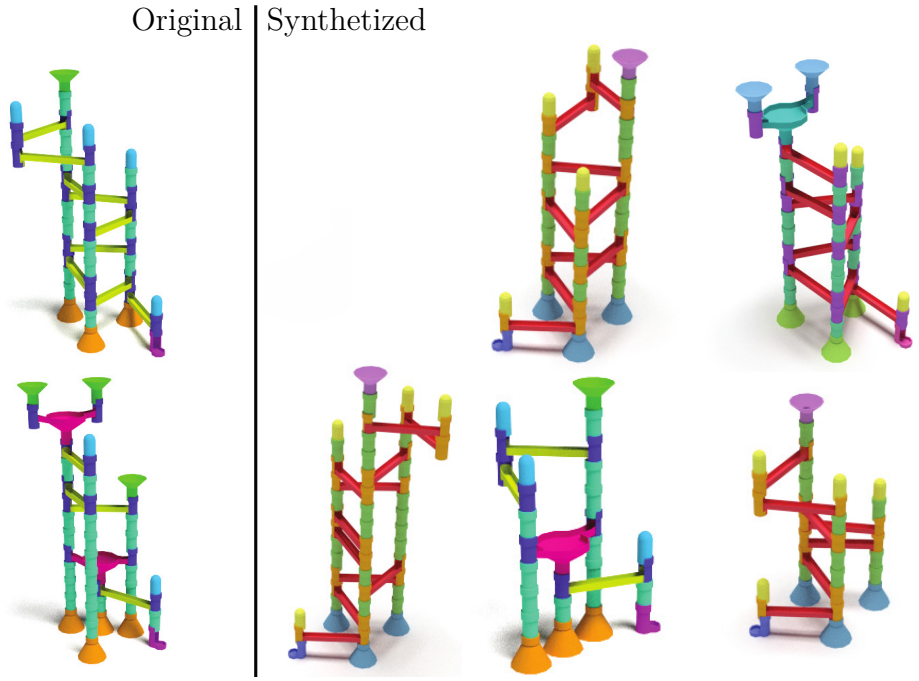


Figure 3.12 – Starting ball track models, replaceable subgraphs result in plausible synthesis results.

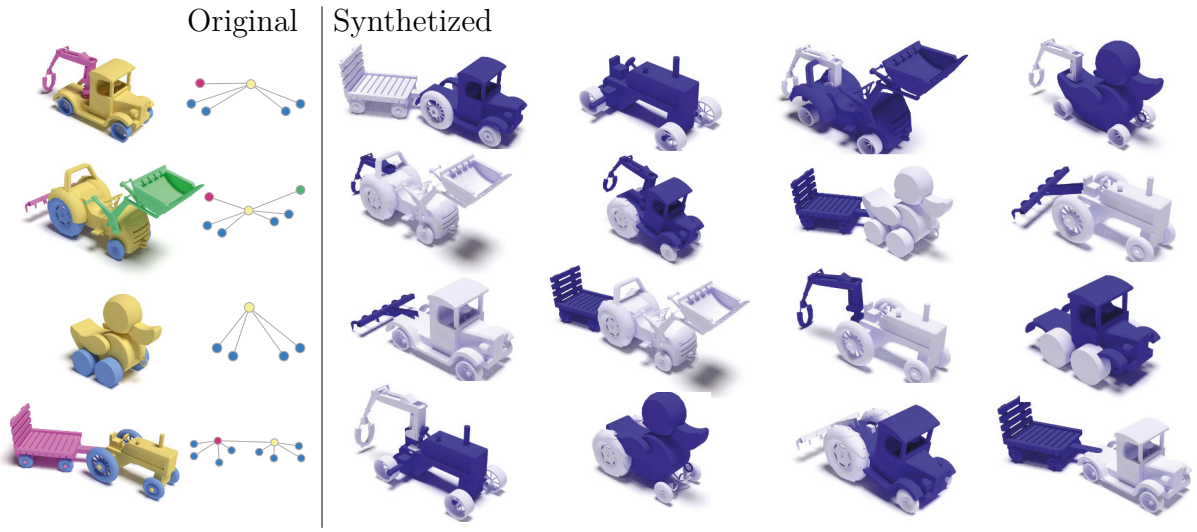


Figure 3.13 – Starting tractors models, replaceable subgraphs result in plausible synthesis results.

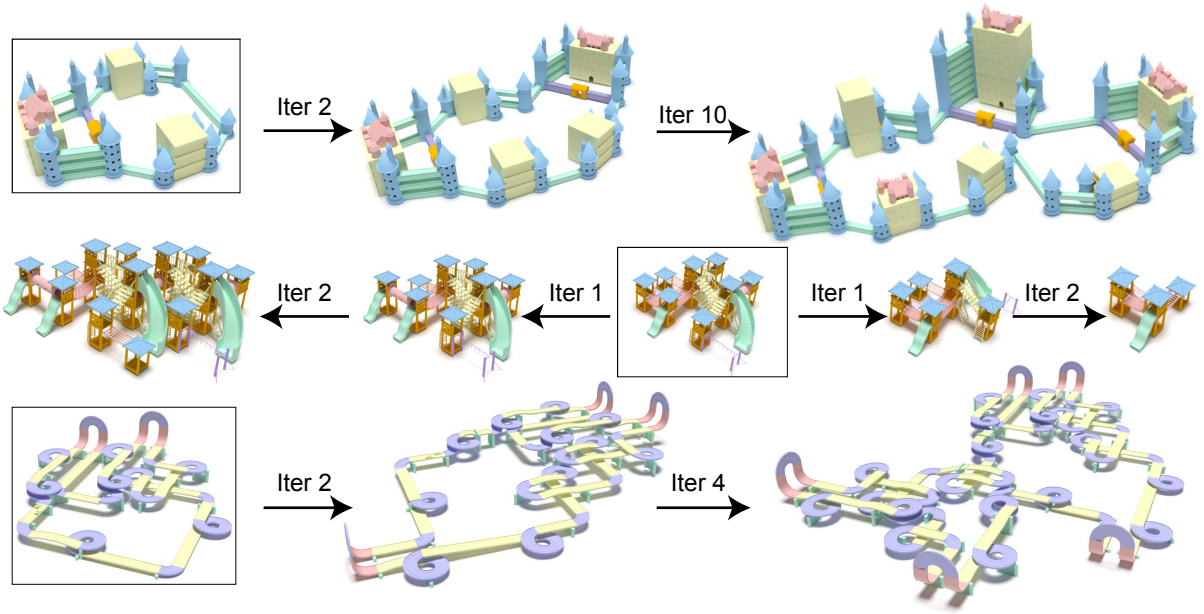


Figure 3.14 – Starting from single castle, playground, racetrack models, matching subgraphs are progressively found and replaced, generating new coherent complex objects.

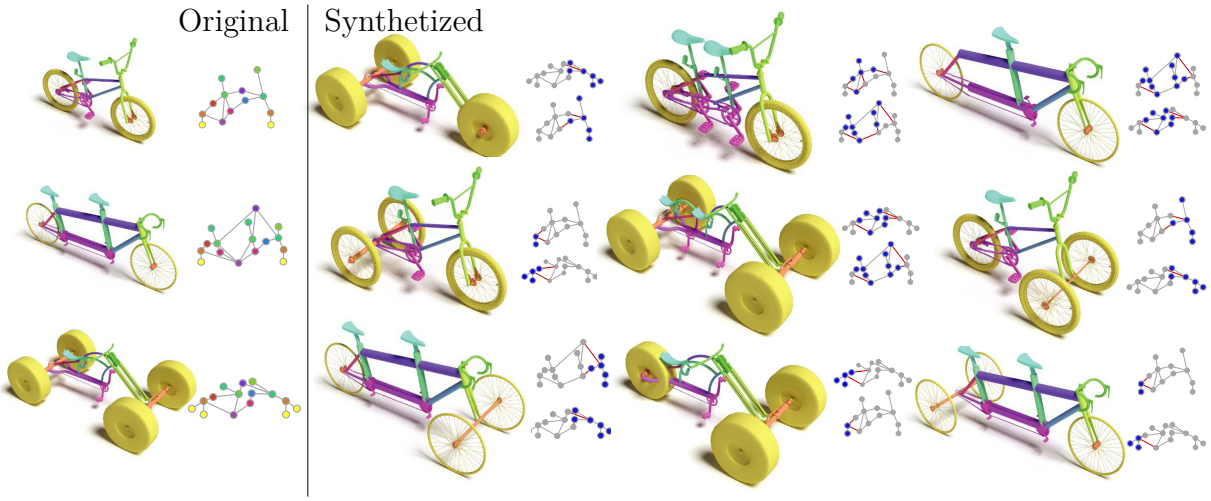


Figure 3.15 – Structural variations (right) are created using the element already present in the initial objects (left).

3.2.5 Discussion

The method presented in this section has several limitations, which are discussed here.

3.2.5.1 Local coherency guarantee

First, the only coherency guarantee offered by this method is local: Global coherency can be violated since it is not captured by the shape graph, as shown in Figure 3.16.

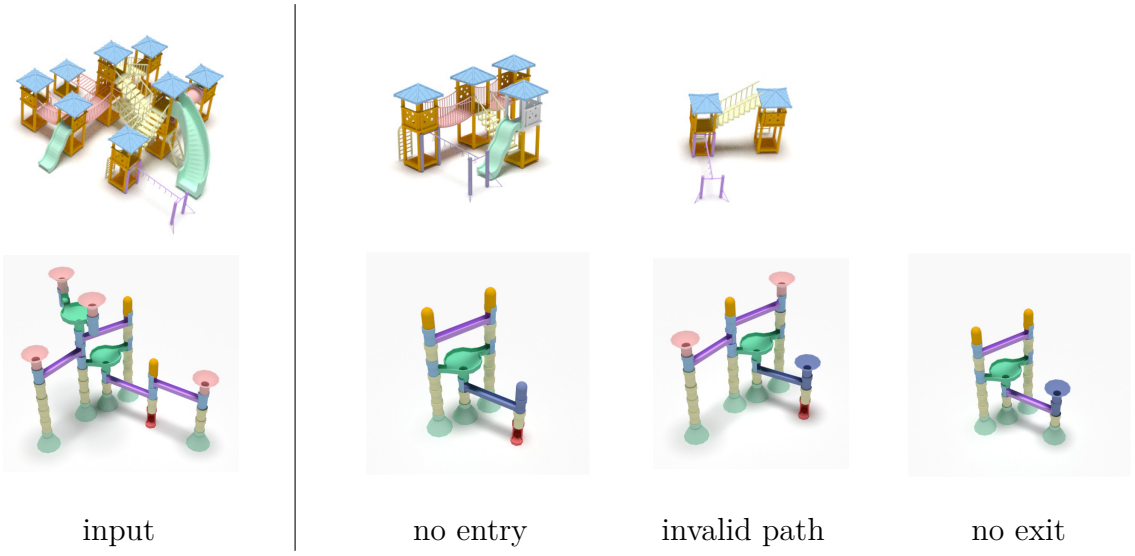


Figure 3.16 – Various synthesis results marked as invalid due to violation of global constraints (e.g., no entry, no exit, or no connection between entry-exit).

3.2.5.2 Sensibility to input data

Second, as any example-based method, ours only allows to reproduce cases observed in the example data. This is usually overcome by using an extensive example data-base, supposed to contain all coherent cases. This is however not possible for larger examples due to algorithmic complexity: The NP-hard nature of the underlying graph isomorphism problem constraints the method to rely on a heuristic algorithm which complexity makes it intractable on large example.

In consequence, the method might lack the amount of variation necessary for building interesting objects in complex cases. Dummy edges and permitted violation intend to improve this point, but they also discard the coherency of the output object if used too extensively. The experiments have shown that the likelihood for an input example to contain interesting replaceable sub-structure is not as high as initially expected.

Besides, the topological description of the model based on the shape graph is very dependent on the ordering of the edges extracted from the PCA, which makes it uncertain for highly non planar cases.

3.2.5.3 Topological versus geometrical coherency

Third, the replaceable sub-structure identification is only based on topological attributes: In many cases, the actual feasibility of the replacement relies on the geometry of the sub-structure.

For example, even when the topological matching allows it, putting huge sub-structure in place of a tiny one creates important geometrical distortion of the corresponding parts. A geometrical deformation penalty should be added, discarding such subgraph replacements. Such geometrical constraints are not handled by the method. Besides, discarding topologically valid solutions might restrict the solution space, which might become problematic.

3.2.5.4 User control

Last, the method offers little user control: Examples are generated, and the user can only select the one they prefer. There is no possibility for them to express their needs, neither in terms of geometry nor functionality.

The next section describes a method which allows for much more user-control through a painting interface.

3.3 Painting of Object Distributions and Graphs: Worldbrush

The work presented in this section results from a collaboration with Arnaud Émilien (Université Grenoble Alpes/Inria), professor Pierre Poulin (Université de Montréal), and professor Bedrich Benes (Purdue University). The resulting paper was published in the ACM TOG journal (see Emilien et al. (2015)) and presented at the Siggraph conference in 2015 (http://s2015.siggraph.org/):

Emilien, A., Vimont, U., Cani, M.-P., Poulin, P., and Benes, B. (2015). Worldbrush: Interactive example-based synthesis of procedural virtual worlds. *ACM Transactions On Graphics (TOG), Proceedings of SIGGRAPH 2015*, 34(4):106

This section is organized as follows: Sub-section 3.3.1 describes the complex object model based on correlated distributions used in this method; Sub-section 3.3.2 explains how this model is used inside a painting tool; Sub-section 3.3.3 details the distribution interpolation method, which is my main contribution to this work; Sub-section 3.3.4 shows results of this method and draws its limitations and perspectives. For a review of related work, please refer to Section 2.2.

Note that beside my main contribution detailed in Section 3.3.3, my participation to this project ranged from the map rendering mode to input data creation (see results in Section 3.3.4).

3.3.1 Vectorial analysis

As in the previous section (see 3.2), the method we propose here still uses the example-based paradigm. However, it differs on almost every other aspects: It deals with element distributions instead of assemblies, which allows to represent more organic scenes as opposed man-made objects; This allows the coherency definition to be less stringent and more efficient to compute; In turn this allows more user control and interactivity. All these attributes fit into an artist-friendly painting metaphor.

3.3.1.1 Virtual world vectorial description

A virtual world (also call virtual scene) is described as a multi-level complex object. At the top of the hierarchy stands the virtual world itself. Its children are distributions. Each distribution is composed of a set of elements associated to positions, and optionally orientations and connections (for graphs). Here are the types we consider:

1. *ground*
2. *island*
3. *mountain*
4. *hill*
5. *river*
6. *road*
7. *castle*
8. *house*
9. *farm*
10. *big rock*
11. *pine tree*
12. *red tree*
13. *rock*
14. *small rock*
15. *grass*

The list above also defines an order between element types, from low to high. This order is used for simplifying world synthesis: It is done in ascending order, which for

example allows not to consider trees while creating a castle.

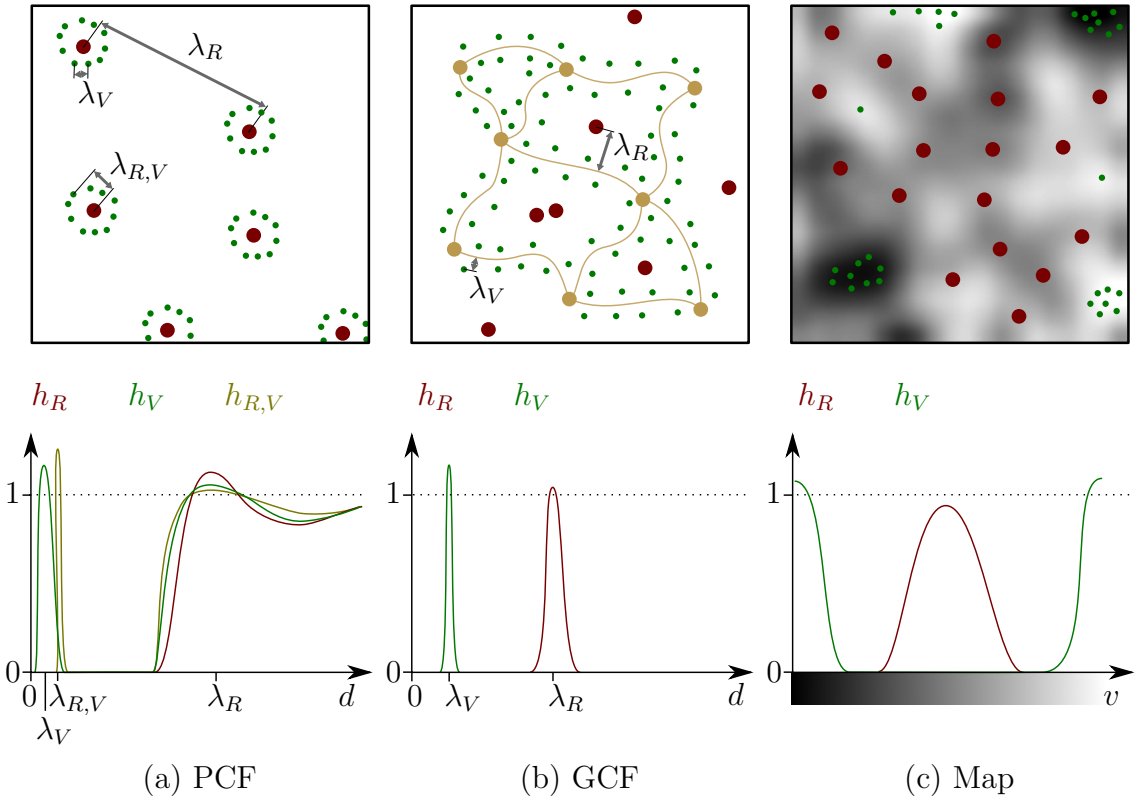
3.3.1.2 Natural scene coherency

As for replaceable sub-structures, this method relies on a particular notion of coherency. It is defined by the *distribution* of the positions of all the elements of the scene.

These arrangements are usually represented by *pair correlation function* (PCF). Such PCF are the normalized histograms of the distances between element positions. They are computed within each element class (the set of elements of the same type), as well as between element of a class versus the elements of lower classes.

Two other arrangement descriptors can be used: Point distributions can be described relatively to graphs using point to graph distance histograms; Point distributions can be compared to external 2D data through value histograms. Those descriptors are illustrated in Figure 3.17.

We call the set of descriptors for a homogeneous zone of the scene a *color*. The learning phase of this method, presented in the painting metaphor as the *pipette* tool, aims at acquiring a *color*. It consists of the computation of selected descriptors in a user-defined zone, and storing the resulting *color* in the *color* container that we call the *palette*.



(a) PCF (b) GCF (c) Map

Figure 3.17 – Element distributions are described through three types of histograms: (a) Pair Correlation Functions represent distances repartition across and inside isolated element types; (b) Graph Correlation Functions represent distances repartition from isolated element types to their closest graph neighbor; (c) Map Correlation Functions represent isolated element likelihood depending on external parameters.

3.3.2 World painting

This section explains how the world model described in the previous section is used through painting tools. Almost all of these tools rely on the same base: the synthesis algorithm.

3.3.2.1 Synthesis algorithm

Once a *color* C has been computed and selected in the *palette*, it can be used for synthesizing an arrangement similar to the one it was extracted from into a new zone D. For this we use the same method than Geyer and Møller (1994) and Hurtut et al. (2009): Metropolis-Hasting Algorithm 2, which proceeds as follow: Starting from a randomly initialized arrangement X_0, a fixed number T of iterations are performed. An iteration t consists in generating a variation X' of X_t. The modification is randomly picked between two possibilities: *Element birth*, where $X' = X \cup \{u\}$ and u is a random position in D. *Element death*, where $X' = X \setminus \{u\}$ and $u \in X$. Both modifications are associated with an acceptance ratio R; a modification is accepted (i.e. $X_{t+1} \leftarrow X'$) with the probability R. In the *element birth* case, the acceptance ratio is computed as follow:

$$R_b = \frac{f_C(X')}{f_C(X)} \tag{3.1}$$

where f_C is a color-dependent evaluation function which will be discussed later. In the case of death, we compute the acceptance death as follow:

$$R_d = 1 - \frac{f_C(X)}{f_C(X')} \tag{3.2}$$

Note that the expression for R_b and R_d are different here than in the original articles of Hurtut et al. (2009) and Emilien et al. (2015). Some experiments have led me to this new formula for R_d. It arises by considering an element death, written as $D : X' = X \setminus \{u\}$ as the complementary event of the element birth, written as $B : Y' = Y \cup \{u\}$. The simple variable change $Y' = X$ and $Y = X'$ is equivalent to considering a *death* (u is removed from X) as a birth *birth* (u is added to Y). However it yields very different results than what was done before, due to the balance between birth and death ratio values. In fact, using such balanced ratios, I've found that there is no need for normalization anymore.

3.3.2.2 Evaluation function

Metropolis-Hasting algorithm relies on the computation of acceptance ratios. Such ratios are composed of an evaluation function f_C where:

$$f_C(X) \propto \prod_{x \in X} \prod_{t(Y_k) \leq t(x)} \prod_{y \in Y_k} h_C^{t(x),t(y)}(d(x,y)) \tag{3.3}$$

where $h_C^{t(x),t(y)}$ is a correlation function between categories $t(x)$ and $t(y)$ according to color C. It is set to a normalized histogram that measures interaction between such categories. d is the Euclidean distance normalized by the width of the bins used in the histogram, as described by Öztireli and Gross (2012).

Algorithm 2 : Modified Metropolis-Hasting algorithm for distribution synthesis

Randomly initialize output arrangement $X_0 = X$ such that $f(X_0) > 0$
for all time-steps $t \in [1, T]$ **do**
  alter current arrangement X_t by randomly
  Element birth:
    Add random element u: $X' = X \cup \{u\}$
    Compute acceptance rate $R = \frac{f(X')}{f(X)}$
  Element death:
    Remove random element u: $X' = X \setminus \{u\}$
    Compute acceptance rate $R = 1 - \frac{f(X)}{f(X')}$
  Accept perturbation with probability R $(X_{t+1} \leftarrow X')$
end for

The evaluation function is called many times during Algorithm 2, and its cost is in $\mathcal{O}(\|X\|^2)$, which makes it very slow. Fortunately, $f_C(X)$ being expressed as a product, the birth acceptance ratios can be simplified as:

$$R_b = \prod_{t(Y_k) \leq t(u)} \prod_{y \in Y_k} h_C^{t(u),t(y)}(d(u, y)) \tag{3.4}$$

which is in $\mathcal{O}(\|X\|)$. The death acceptance ratio can be simplified in the same way. This results in a much faster algorithm.

3.3.2.3 Editing tools

Some tools only differ in the definition of the synthesis and influence zones. Figure 3.18 illustrates these differences:

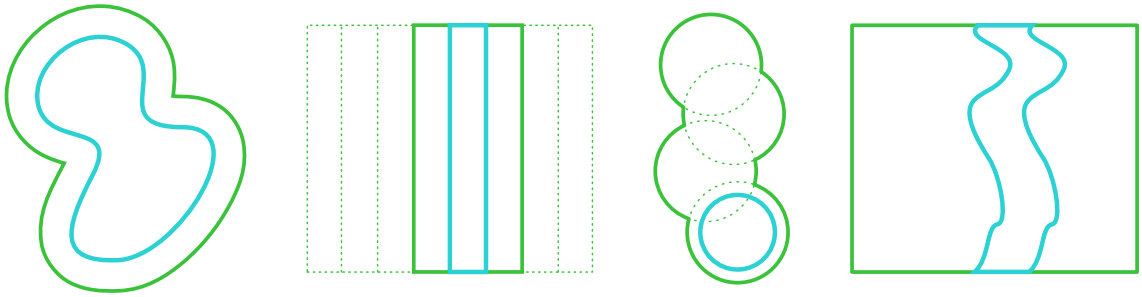
Paste : The user can manually select a zone to synthesize in using a rectangle controller or a closed spline. See Figure 3.19 and Figure 3.20.

Gradient : The user can select a rectangle region for synthesizing a *color* gradient. This zone is split into slices inside which an interpolated *color* is used for synthesis. See Figure 3.22.

Brush : The user moves its cursor on the virtual world surface; A circular synthesis zone centered around each trajectory sample is progressively generated. See Figure 3.21.

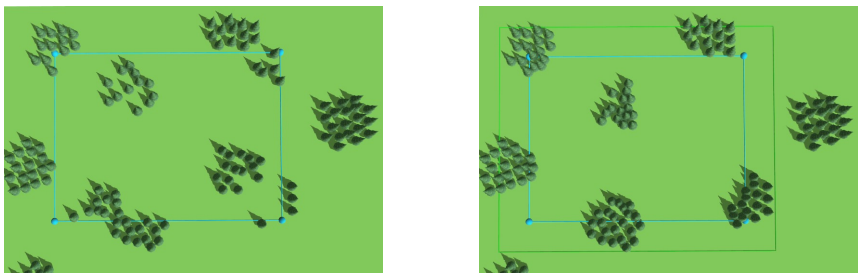
Stretch : Given a rectangular zone, a cut orthogonal to the stretching direction is found. It is extruded in the stretch direction for defining the synthesis zone. See Figure 3.23.

The **move** tool is different: when a zone is selected for moving, its color is computed and maintained through the motion despite the change of environment. This is done through a modified Metropolis-Hasting algorithm allowing a single operation on the element consisting of displacing it while keeping every other element untouched. If a better position is found this way for any element, it is kept. On the other hand, if the probability of any point is below a threshold and no better position if found, it is temporarily discarded.



(a) Paste (b) Gradient (c) Brush (d) Strech

Figure 3.18 – Each tool has its proper synthesis (in cyan) and influence zone (in green). The user-defined synthesis zone is enlarged for creating the influence zone while *pasting* (a). Each synthesis zone uses the neighboring zones for influence in the *gradient* tool (b). The union of previous influence zones is used in *brushing* (c). The extruded cut is used for synthesis while the whole selected zone serves as influence in *stretching* (d).



(a) (b)

Figure 3.19 – Pasting consists in synthesizing a distribution matching the currently selected *color*. It can be done without influence: only elements inside the selected zone are accounted for during synthesis, which might create discontinuities (left). An influence region can be used for avoiding such discontinuities (right). It is used for neighbor look-up, but not for seeding new elements.

Figure 3.20 – Village synthesis: *Color* from an exemplar (left) is used to generate a larger village (right).

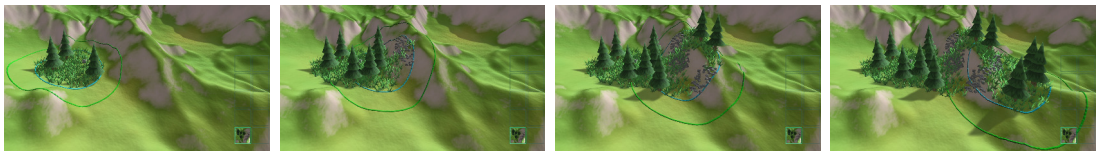


Figure 3.21 – Brushing consists in synthesizing the distribution inside a moving zone. Here, from left to right, a gesture is progressively decomposed into circular synthesis zones, creating an intuitive painting tool.

3.3.3 Main contribution: RDF Interpolation

Once acquired, a given *color* can be used through the tools mentioned above. However in some circumstances, the desired color is not available to acquire in the initial scene. For that reason, our method offers to create new *colors* from existing ones through *color interpolation*.

3.3.3.1 Interpolating density function

The idea behind *color* interpolation is quite intuitive: mixing two input *color* for creating a third one, like it is made with paint. However, mixing distribution *colors* is more complicated than mixing classical colors: interpolating linearly the *color* histogram leads to unwanted results.

This comes from the use we have of histograms: peaks in the functions create high values of correlation, which in turns allows the corresponding attribute (element distance, graph distance, or map value, see Section 3.3.1) to be used by a candidate during generation. When interpolating linearly two histograms with different peaks, the linear interpolation still presents all features of both initial histograms, only weighted by the interpolation coefficient, as shown in Figure 3.25.c. Linearly interpolated correlation function will lead to results presenting both characteristics of input samples in various proportions instead blending those characteristics into one another.

The result we aim for is the one of Figure 3.25.d: features of the initial histograms are re-located and blended into the output. This result is obtained through mass transport (see Bonneel et al. (2011)).

3.3.3.2 Mass transport solution

Solving mass transport problems require to use an optimization procedure, which makes it too slow for our purpose, even using a parallel implementation. Fortunately, Read (1999)

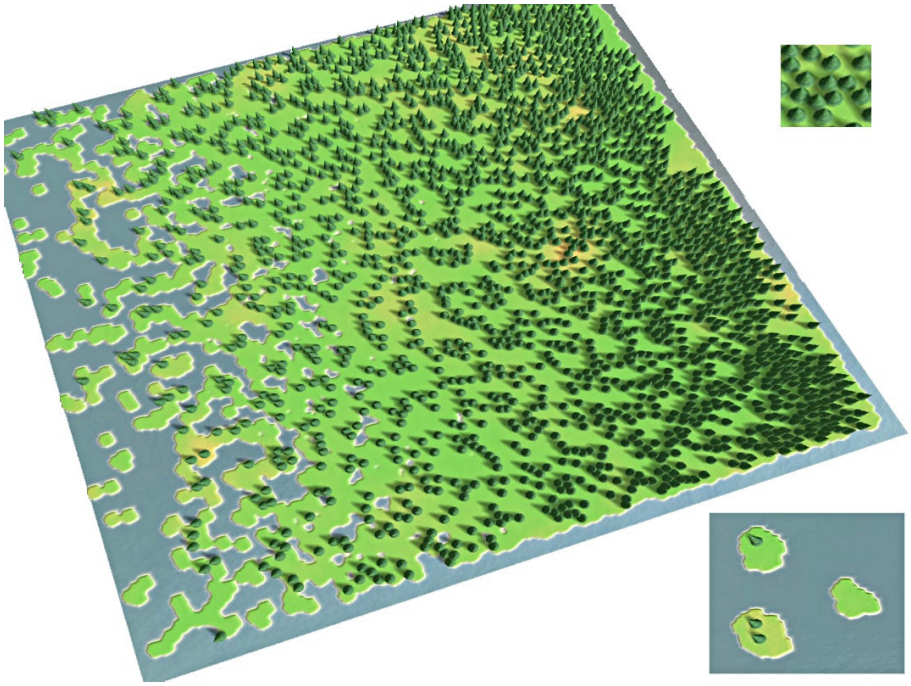
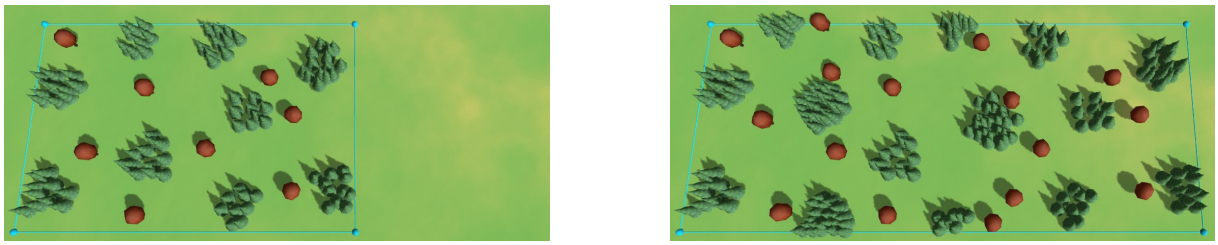


Figure 3.22 – An example of the linear gradient tool applied with the two example *colors* in insets.



(a) (b)

Figure 3.23 – Seam-carving-based stretching. Top: Initial arrangement. Bottom: New trees have been seamlessly inserted in the empty region, preserving the visual appearance of the distributions.

describes a very fast method for interpolating one dimensional histograms known as *inverse cumulative density functions* (ICDF). This method is explained in Figure 3.26: Given two input histograms f and g, we consider their integrals F and G, computed as cumulated sums; Integrals are then normalized: $\hat{F} = \frac{F}{\|F\|}$ and $\hat{G} = \frac{G}{\|G\|}$; The interpolated normalized integral is computed through inverses: $\hat{H}_t^{-1} = (1-t).\hat{F}^{-1} + t.\hat{G}^{-1}$; De-normalization is performed using an interpolated amplitude: $H_t = ((1-t).\|F\| + t.\|G\|).\hat{H}_t$; A simple differential operation allows us to retrieve h_t.

3.3.3.3 *Color* interpolation results

Color interpolation yields fast and conclusive results, as shown in Figure 3.27.

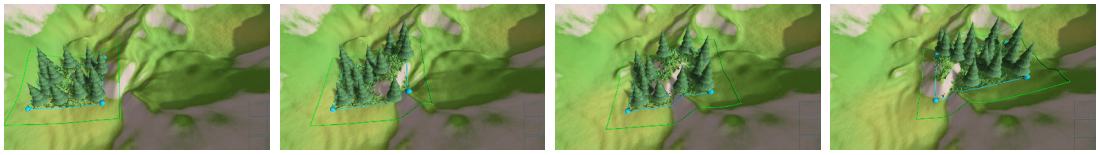


Figure 3.24 – Left to right, top to bottom: Moving a selection while maintaining constraints, and favoring object displacement rather than births and deaths, to increase temporal coherence.

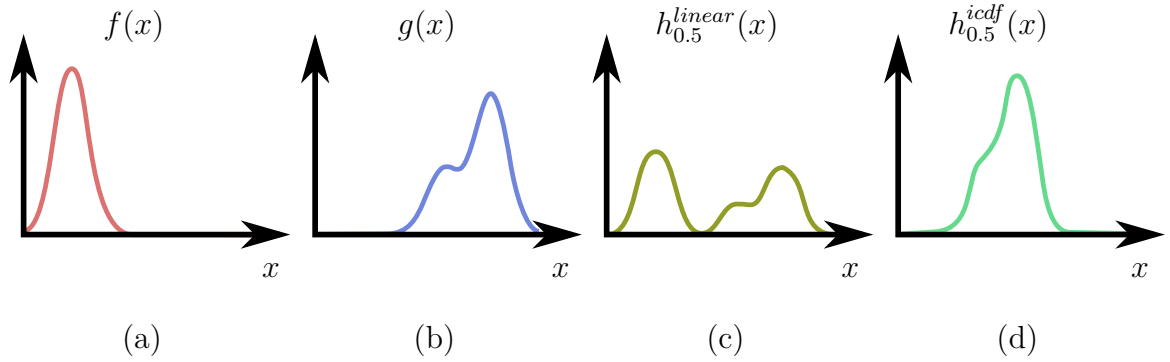


Figure 3.25 – Two input histograms (a) and (b) are interpolated linearly (c), resulting in a mix of features. Optimal transport on the other hand allows to morph features of the first histogram into those of the second one (d). Both interpolations use the same parameter $t = 0.5$.

3.3.4 Results

This section describes some of the results obtained with the method.

Figure 3.28 presents a wide scene fully created with our tool by an artist: Object distributions were created by hand on small areas, corresponding *colors* were learned using the pipette tool and used for painting the whole scene.

Figure 3.29 showcases the use of an existing input scene (here a map of the Middle Earth designed by Tolkien (1955)). In this case, the input data is an image (bottom left), which has been converted into a vectorial representation (upper left) by hand.

3.3.5 Discussion

The method presented here has several limitations, which are discussed here.

3.3.5.1 Input of an interaction matrix

One limitation is of practical interest: for the method to work, the user has to provide an *interaction matrix*. This matrix specifies which element type depends on which other (which is unilateral and imposes the order of generation), and specifies the property to be analyzed for creating the corresponding *color* (pair correlation function, graph distance, or map correlation). Despite being a large constraint on the usability of our method, this input follows the analysis the user makes of their data and is therefore unavoidable. However, only few possibilities make sense in the context of world synthesis, which drastically reduces the possible choices.

The method of Hurtut et al. (2009) is faced with the same issue of having to identify which descriptor is best suited for discriminating elements. Their approach is to compute all kinds of descriptors on the input data and then to select those which are the most

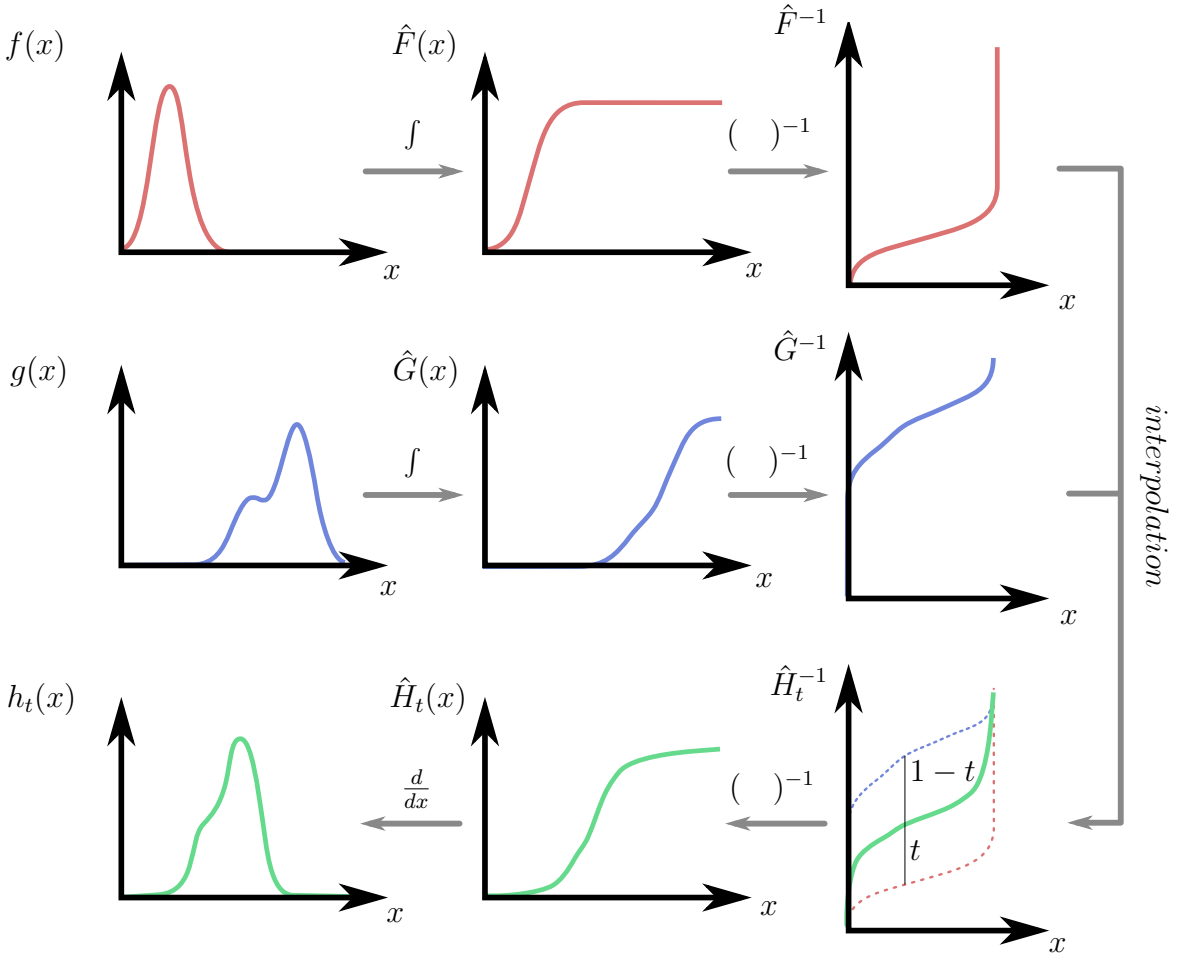


Figure 3.26 – Inverse cumulative density function interpolation: Both input histograms f and g are integrated, normalized, inversed, and interpolated with parameter t. The resulting function processed the other way: inversed, de-normalized (using an interpolated amplitude), and differentiated, for obtaining the in-between h_t.

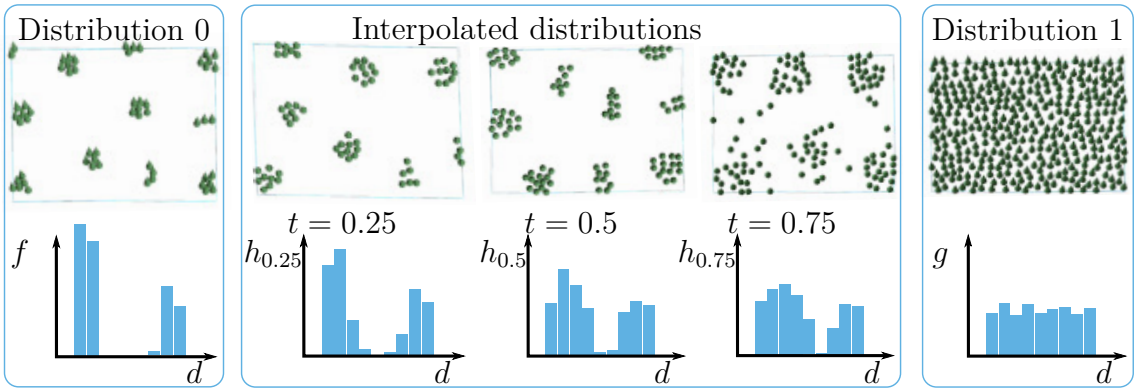


Figure 3.27 – *Colors* are interpolated using mass transport on pair correlation functions.

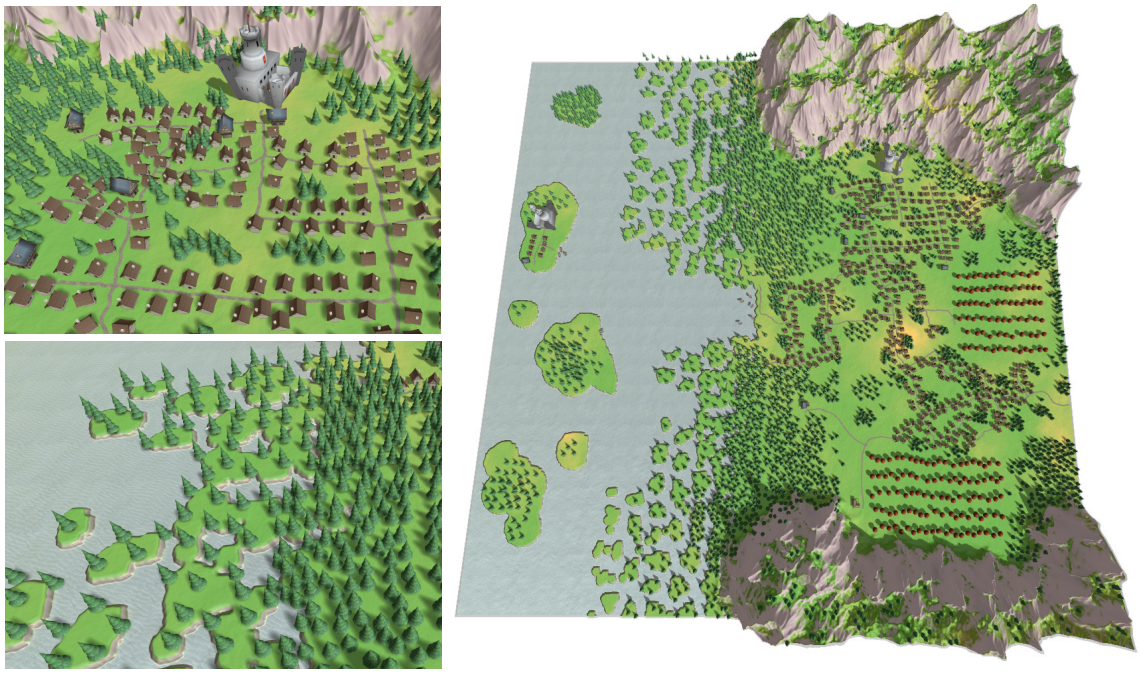


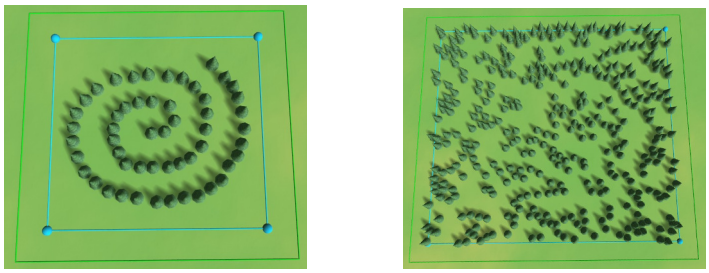
Figure 3.28 – Starting from an empty scene, the user defined some *colors* by hand and used those for painting the whole content. On the left are shown close-ups on details of the scene.

Figure 3.29 – A map of Middle Earth (bottom left) has been converted to a vectorial description (upper left) by an artist and used as input in our method. After learning *colors* of the input and using those for painting new element, the user created a new map (right).

relevant. The same approach could be used with our method by computing several possible descriptors and choosing the best one based on significance criteria. For example, in the case of histograms, significance could be computed using the cumulated difference between a given histogram and a constant 1-valued reference.

3.3.5.2 Distributions descriptor issues

Figure 3.30 presents a case in which a structured element distribution fails to be correctly reproduced through copy-paste. This comes from the inability of a PCF to correctly represent structured input, except for blue noise or clustered distribution.



(a) (b)

Figure 3.30 – Failure case: The *color* computed from a structured element distribution (left) in not relevant: The pasting of such color results in an almost random distribution (right). This comes from the PCF distributions descriptor not capturing spiral patterns: Left and right examples have the same PCF, despite being visually different.

On the one hand, a more elaborated distribution descriptor could be conceived, which discriminates inputs based on their structure. Such a descriptor could be used as a correlation function in the acceptance ratio computation of the Metropolis-Hasting algorithm, allowing to generate relevant output distribution. However speed is a critical aspect of interactive modeling, and PCF performs well on this panel because of its second order nature (pairs of points are compared) and the possibility to simplify ratios, yielding a time-linear sample estimation. It is likely that a more complex descriptor would not have such good properties, discarding the possibility to use it for interactive applications. Besides, such a descriptor should capture a wide variety of structures while being scale-independent, which is complex and ambiguous. For example, given a spiral shaped element distribution, which larger distribution should have the same evaluation? The answer to this question relies on the semantic understanding of what the distribution represents, which is far from our current methodology.

On the other hand, representing blue noise or clustered distributions can be done with a much simpler model than PCF. For example, a piecewise Strauss correlation function as the one represented in Figure 3.31 allows to represent the same distributions as the full histograms we use, and has only 6 parameters (compared to more than 100 in a histogram). These parameters can be learned from the input distribution as it is done by Hurtut et al. (2009), and later re-used during generation.

Besides a compact memory representation, such a model would be much easier to interpolate: It would only require to interpolate input parameters, instead of mass-transporting a full histogram. This is due to parameters sampling the abscissa of the function; Such parameters move the features horizontally when interpolated, which is equivalent to morphing function features.

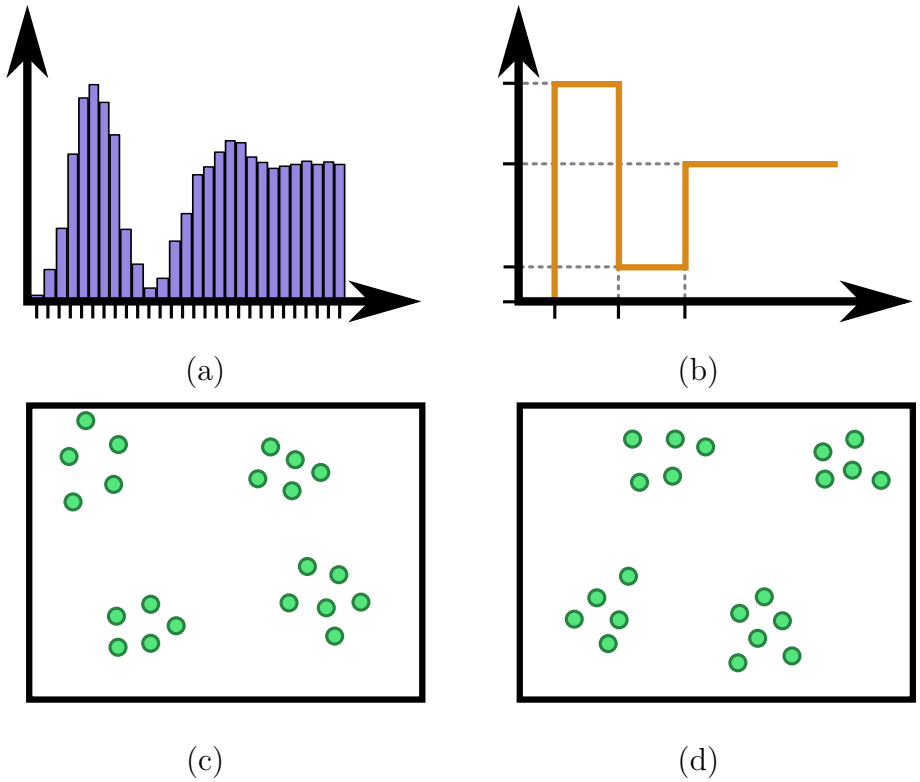


Figure 3.31 – A PCF containing typically a hundred parameters (a) is used for synthesizing a clustered distribution (c). A qualitatively similar result (d) can be obtained using a much simpler Piece-wise Strauss model described by less than 10 parameters (b).

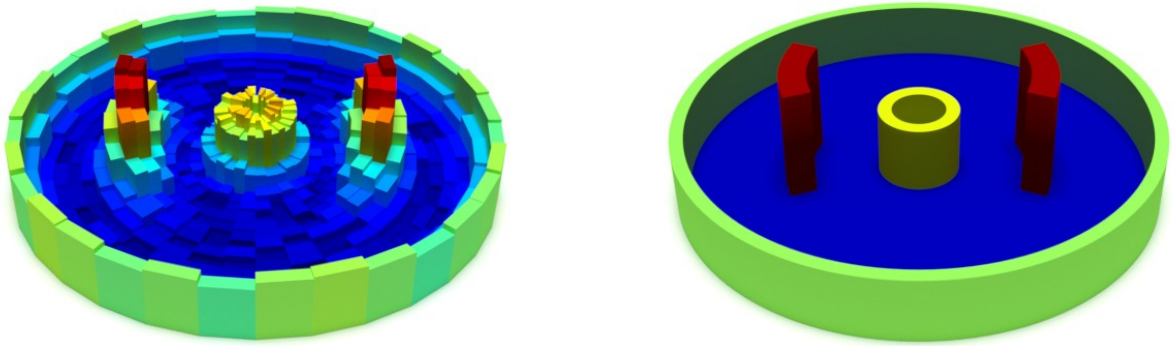
For 2D PCF, no equivalent procedural correlation function exists; this would be an interesting area to look into. My intuition is that a piece-wise constant 2D function over partial concentric rings (see Figure 3.32) should be a proper 2D extension of the piecewise Strauss correlation function. The resulting descriptor should be easy to interpolate, which is currently not possible in 2D using ICDF. Alternatively, another method of Bonneel et al. (2015) allows to interpolate 2D histograms in a fast manner. Such a method could be used for blending complex bi-dimensional *colors*.

3.3.5.3 Content nature

Worldbrush could easily be extended for handling other type of content than virtual worlds such as general vectorial textures or ecosystems for example. However, theoretical limitations exist on the nature of the content one can paint using this approach:

3D distributions are present in virtual worlds such as in bird flocks, clouds, stars, or tree leaves. In addition, 2D and 3D distributed elements can also be animated, which adds yet another dimension to the problem. Our method is limited to 2D element distribution and handling such content requires using other distributions descriptors and other interaction metaphors. Indeed,the distribution descriptors we use are defined in 2D; They might be extended to higher dimensions, but then the cost of a sample estimation will increase. Besides, the painting metaphor itself relies on 2D tools; Extensions to 3D have been proposed, as in the recent work of Schmid et al. (2011).

Even when dealing with 2D content, our method shows limitations: when this content is hierarchical. More precisely, editing 2D content in a coarse-to-fine way is not problematic.



(a) (b)

Figure 3.32 – 2D histograms made of constant pieces over circular rings might be a suitable model for representing 2D interactions between distributed elements.

On the other hand, fine-to-coarse operation sequences are to be avoided, since large-scale edits might over-write previous small-scale ones. Artistic freedom requires a mix of both paradigms, which is not directly possible with Worldbrush.

The remaining of this manuscript deals with other types of content: Section 3.4 details a method for sculpting 3D hierarchical content; And Chapter 4 take care of animated content.

3.4 Sculpting of Complex Hierarchical Objects: Deformation Grammars

The work presented in this section represents the major outcome of this thesis concerning static object design. The resulting paper was submitted at the Computer Graphics Forum journal and is accepted with minor revisions:

Vimont, U., Rohmer, D., and Cani, M.-P. (2016). Deformation grammars: Hierarchical constraint preservation under deformation. *Submitted at Computer Graphics Forum*

This section is organized as follows: Sub-section 3.4.1 introduces the problematic of complex object sculpting; Sub-section 3.4.2 explains what deformation grammars are and how they help to deform complex objects; Sub-section 3.4.3 extends deformation grammar for allowing bottom-up deformation behaviors; Sub-section 3.4.4 shows results of this method and draws its limitations and perspectives.

For a review of related work, please refer to Section 2.3.

3.4.1 Introduction

Previous sections have proposed solutions for generating specific types of complex objects based on specific coherency definitions. Despite being useful in specific contexts, those methods lack generality: They solve an editing problem while making lots of assumptions on the nature and the structure of the object to be edited. These object-specific approaches have the advantage to benefit from knowledge about the object, which drive the approach in term of coherency formulation as well as in terms of interaction tools or modeling metaphor.

Here, we focus on the other end of this spectrum by creating a method which relies on few assumptions about the object while allowing a wide variety of interactions. Namely, the object is only supposed to be a complex object as defined in Section 3.1. We offer to interact with using any deformation tool. This is thoroughly explained in Section 3.4.2.1; It intuitively corresponds to a set-up where an initial coherent complex object is progressively modified while maintaining its coherency rather than plainly synthesized.

We claim that the following features are essential for an artist-driven deformation tool, and address them specifically:

— The coherency of the whole model should be maintained throughout deformation. The user should be able to select the coherency criteria for each type of element and at different scales, in order to fully express their intent.

— A hierarchical model should be editable at different scales, ranging from local to global ones.

— The artist should be able to apply the edits in the order they wish, not only from coarse to fine scales.

Our solution is based on the new concept of *deformation grammars*. The latter enable to define deformation interpretation rules and allow us to freely deform a complex object while maintaining its coherency.

3.4.2 Deformation Grammars

This section introduces deformation grammars as an efficient tool to setup coherency-preserving deformations for complex objects.

3.4.2.1 Definitions: Object deformation

A *deformation* is any function which maps the values of an object's parameters. For example, if the object is visually represented using a mesh, an example of deformation is $d : \mathbb{R}^3 \to \mathbb{R}^3$. d can be used to change all the vertices positions at once.

Classically, applying a deformation d to a complex object G (which we call *object deformation* and noted using the couple $D = (G, d)$) is performed by applying the deformation independently to the visual representations of each of the object's sub-parts. In contrast, our formalism enables us to redefine the application of a deformation in a hierarchical way, a first step for enabling us to preserve the consistency of complex objects through deformations. The hierarchical decomposition is done as follows:

An *element deformation* is defined as the edit of the element's internal parameters, which we call the *internal deformation*, followed by the element deformations applied to its children (if any), which we call *hierarchical deformation*. Using the element formalism for G, where $G = \varepsilon_0$ is the highest element in the hierarchy, this enables us to rewrite the object deformation of G in a hierarchical way, as follows:

$$\begin{cases} D^0 = (\varepsilon_0, d) \\ D^i = (\varepsilon_i, d) = D^i_{internal} \oplus D^i_{hierarchical} & \forall \varepsilon_i \in E \\ D^i_{hierarchical} = \bigoplus\limits_{\varepsilon_j \in C(\varepsilon_i)} D^j \end{cases} \tag{3.5}$$

where $\oplus$ stands for the *independent application* of element deformations on a set of elements. This operator also allows to combine any internal element deformation with other deformations.

The hierarchical definition of deformations in Equation 3.5 is instrumental for allowing various sets of coherency constraints to be maintained when deforming complex objects. Section 3.4.2 explains how we express this deformation propagation process using deformation grammars, and how to use it for preserving the full coherency of the object.

3.4.2.2 Definition: Grammar

Formal grammars are widely used in Computer Graphics for representing hierarchical processes. More specifically, shape grammars such as the one in Figure 3.2 are often used to generate the static geometry of complex objects as in Müller et al. (2006), Emilien et al. (2012), Schwarz and Müller (2015). In this work, we extend the scope of formal grammars to handle the hierarchical deformation of complex objects.

A *deformation grammar* models a deformation behavior for a complex object under a set of deformations. It is defined as any formal grammar by:

— a set of non terminal symbols N.
— a set of terminal symbols Σ
— an axiom $A \in N$
— a set of production rules $P = \{R_i\}$

Symbols. A symbol is an element deformation $D = (\varepsilon, d)$, where $\varepsilon \in E$ and d is an arbitrary deformation. A terminal symbol is the deformation of the internal parameters of an element, while a non-terminal symbol is a regular deformation. Symbols can be assembled using the independent application operator $\oplus$.

In our example of a tree model (see Section 3.1.2.2), a terminal symbol is the geometric transformation of a single branch, while a non-terminal symbol is the deformation of a branch and all its descendants.

Axiom. The axiom is a non terminal symbol created by the user. It represents the object deformation intent, such as a free-form deformation interactively generated through a sculpting tool. It is the initial symbol which will be decomposed into other symbols, until only terminal symbols remain.

In our example, the axiom is the deformation that the user wants to apply to the tree model. It is processed as a deformation of the highest element ε_0 in the object's hierarchy (i.e. the trunk, see Equation (3.5)) and decomposed hierarchically using the grammar rules.

Rules. A rule is the substitution of a symbol by a $\oplus$ of other symbols. In other words, it is the interpretation of an element deformation into the deformations of its components. It is defined with respect to a type of element $t(\varepsilon)$ and a type of deformation $t(d)$:

$$R(t(\varepsilon), t(d)) : D = (\varepsilon, d) \mapsto D' \tag{3.6}$$

where D' is an element deformation preserving the coherency of elements of the type $t(\varepsilon)$. Following Equation (3.5), we define D' as follows:

$$D' = D_{internal} \oplus D_{hierarchical} \ . \tag{3.7}$$

$D_{internal} = (\varepsilon, d')$ is a terminal symbol; It is a deformation that applies to the internal parameters of the element ε and preserve its internal coherency. $D_{hierarchical}$ is a non-terminal symbol; It calls for the independent application of deformations of the children of ε that preserve the hierarchical coherency of ε:

$$D_{hierarchical} = \bigoplus_{e \in C(\varepsilon)} D^{\varepsilon,e}(e) \ , \tag{3.8}$$

where $D^{\varepsilon,e} = (e, d_e)$ is an element deformation that preserves the hierarchical coherency between ε and e. It will be further processed for element e by the rule $R(t(e), t(d_e))$ for preserving the coherency of e (see Equation (3.6)).

The rules, in the form of Equation (3.6), are defined by the user for each type of deformation and each type of element, by specifying $D^{\varepsilon}_{internal}(\varepsilon)$ and $D^{\varepsilon,e}$ used in Equations (3.7) and (3.8). They enable to control the behavior of a complex object under arbitrary deformations, and in particular to preserve the coherency of the object, as illustrated next.

3.4.2.3 Example

Let us come back to our example of the tree model and detail the process of creating a deformation grammar. The structure of the object has been described in Section 3.1.1 and the associated coherency constraints in Section 3.1.2.

We consider a sculpting deformation behavior using a free-form deformation $d : x \in \mathbb{R}^3 \rightarrow d(x) \in \mathbb{R}^3$. The deformation is controlled by the user's mouse displacement, applying a weighted local translation in the view plane.

Defining a grammar rule boils down to define $D_{internal}$ and $D_{hierarchical}$ (see Equations (3.6, 3.7)).

Let us start with $D_{internal}$, which preserves the *internal coherency* of a branch. We call p_{start} and p_{end} the extremities of the branch geometry. In order to preserve the cylindrical geometry of the branch, $D_{internal}$ needs to be an affine transformation whose matrix M can be expressed as:

$$\mathrm{M} = \mathrm{T} \times \mathrm{R} \times \mathrm{S}$$

where:

— $\mathrm{T} = \mathrm{trans}\left(d(p_{start}) - p_{start}\right)$

— $\mathrm{R} = \mathrm{rot}\left(\frac{d(p_{end})-d(p_{start})}{\|d(p_{end})-d(p_{start})\|}, \frac{p_{end}-p_{start}}{\|p_{end}-p_{start}\|}\right)$

— $\mathrm{S} = \mathrm{scale}\left(\frac{p_{end}-p_{start}}{\|p_{end}-p_{start}\|}, \frac{\|d(p_{end})-d(p_{start})\|}{\|p_{end}-p_{start}\|}\right)$

and:

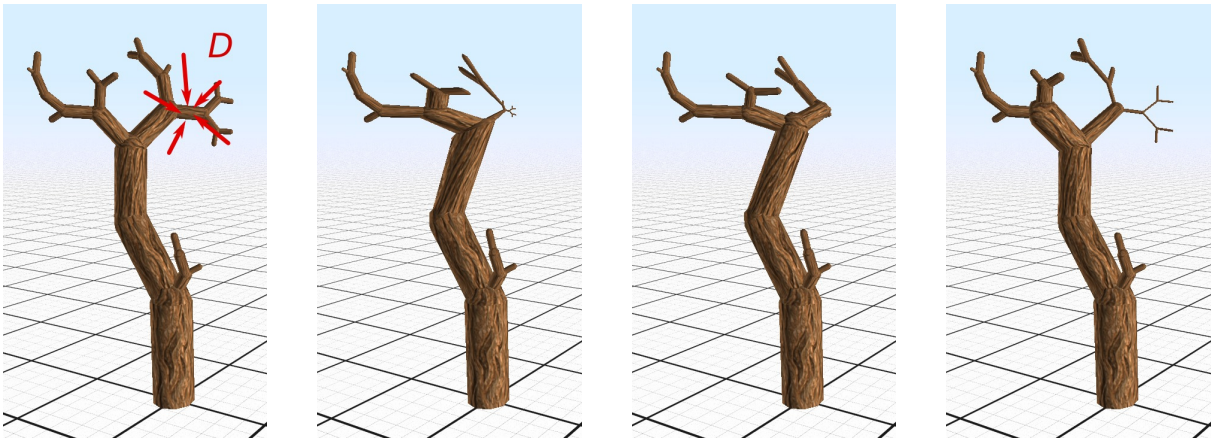
— $\mathrm{trans}(x)$ is the translation of vector x;

— $\mathrm{rot}(a, b)$ is the rotation from vector a to vector b;

— $\mathrm{scale}(a, s)$ is the scaling of axis a and magnitude s

In the current case, $D_{internal}$ does not disconnect a branch from its initially connected children. However, in the general case, one can define $D_{hierarchical}$ in such a way that it re-connects a branch with its children. According to Equation 3.8, it requires to define the deformation from a branch ϵ to its child e:

$$D^{\epsilon,e}_{hierarchical} = \mathrm{trans}(p^{e}_{start} - p^{\epsilon}_{end})$$

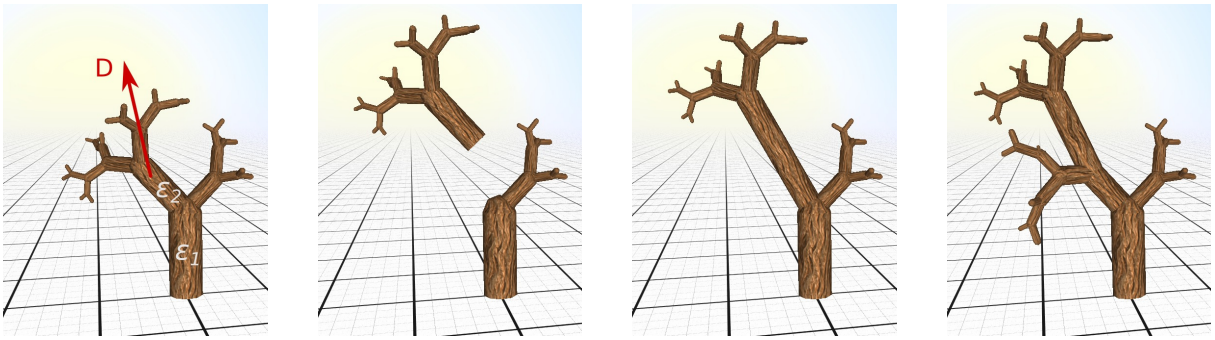


(a) (b) (c) (d)

Figure 3.33 – Deforming a tree model (as in Figure 3.4) using deformation grammar results in coherent tree shapes, according to our tree definition.

3.4.3 Bilateral Grammar Rules

As stated in Section 3.4.1, an object should be editable at different scales (i.e. by editing parts at different levels of the hierarchy) in an arbitrary order. But the deformation



(a) (b) (c) (d)

Figure 3.34 – Bilateral grammar rules allow us to deform a complex object at any level of the hierarchy. (a): An initial model has one of its elements deformed. (b): Result when a local edit is applied to a sub-branch without the use of bilateral grammar rules. (c): Using our bilateral grammar rules maintains coherency, here by ensuring that the edited branch stays in contact with its parent. (d): An alternative rule is used to automatically split elongated branches and generate new sub-branches.

grammars defined so far starts from a deformation of the top level element ε_0 and propagate down the hierarchy to preserve coherency. Figure 3.4b shows that applying the deformation on another element than ε_0 breaks the object coherency at the parent level as the parent does not interpret the deformation. Still, as already stated, local edition should be allowed at any time of the edition in order to enable fine user control.

In order to maintain the object's global coherency during the deformation at any hierarchical level, deformation grammars needs to handle upward coherency management.

3.4.3.1 Closed-loop problem

Let us consider a tree model with two branches ε_1 and ε_2 such that $\varepsilon_2 \in C(\varepsilon_1)$, and ε_2 receives a deformation d from the user.

A naive solution to allow for upward coherency management consists in creating an *ascending grammar rule* $R\left(t(\varepsilon_2), t(d)\right)$ generating a deformation of ε_1. For example, we could have $R_{ascending}\left(t(\varepsilon_2), t(d)\right) : D = (\varepsilon_2, d) \mapsto D' = (\varepsilon_1, d)$, where D' is a non terminal symbol. But D' would be interpreted into a deformation of ε_2 following Equation 3.8. This results in a loop creating non-terminal symbols, and never converging to terminal ones. The deformation operation does not terminate in this case.

3.4.3.2 Deformation emitter specification

Our solution to this problem is to incorporate the deformation *emitter* inside symbols, which are now defined as:

$$D = (\varepsilon, d, S) \in \Sigma \cup N \tag{3.9}$$

where $\varepsilon \in E$ is the deformed element, d the deformation and $S \in E \cup \{0\}$ is the emitter of the deformation. By convention, deformations directly generated by the user are associated with the emitter $S = 0$. Rules can now make use of this new information to avoid the closed-loop problem, as explained next.

3.4.3.3 Emitter-specific grammar rules

The objective is now to enable the propagation of the interpretation of a deformation up and down the hierarchy of a complex object, but only when needed, and in particular while avoiding loops. To this end, we modify the rules defined using Equation (3.6) into emitter-specific grammar rules encoding whether the deformation of an element may influence or not other elements. In our solution, this influence only depends on the type of relation t between the emitter element and the deformed element, where $t(\varepsilon, S)$ can take the values *none, child, parent, self*, as defined in Section 3.1. We therefore define our emitter-specific grammar rules by replacing Equation (3.6) by

$$R(t(\varepsilon), t(d), t(\varepsilon, S)) : D = (\varepsilon, d, S) \mapsto D' \ . \tag{3.10}$$

where the interpreted deformation D' is generally defined as $D' = D_{up} \oplus D_{down}$, where

$$D_{up} = (p(\varepsilon), d, \varepsilon) \ , \tag{3.11}$$

is an element deformation applied on the parent of ε, propagating therefore the interpretation upward in the hierarchy until $\varepsilon = \varepsilon_0$, and D_{down} an element deformation applied to the children, and used to propagate the deformation downward to the leaves. Depending on the relation type $t(\varepsilon, S)$, only one of these deformations needs to be applied, in order to ensure the preservation of coherency while avoiding the closed-loop propagation problem. This is achieved using:

— $R(t(\varepsilon), t(d), parent) : D \mapsto D_{down}$,
where $D_{down} = D_{internal} \oplus D_{hierachical}$. This corresponds to the basic case (Equation (3.5) of a deformation transmitted by from a parent to its children in the hierarchy.

— $R(t(\varepsilon), t(d), child) : D \mapsto D_{up} \oplus D_{down}$,
where $D_{down} = (S, d_{interpreted}, \varepsilon)$ is the interpreted deformation preserving the hierarchical coherency of ε. This enables to coherently transform the other children when one of them has been edited, while still propagating the deformation to the parent for preserving higher level coherency.

— $R(t(\varepsilon), t(d), none) : D \mapsto D_{up}$.
This enables the deformation to reach a common ancestor, where the consistency will be preserved between ε and its parent.

— $R(t(\varepsilon), t(d), self) : D \mapsto D_{internal}$.
This enables internal consistency to be maintained using a terminal symbol.

3.4.3.4 Example

Let us now extend the example of Section 3.4.2.3 in order to allow for bilateral deformation interpretation. We consider an affine deformation d applied on a branch $\varepsilon_2 \neq \varepsilon_0$ of a tree. We need to handle those according to

$$R(t(\varepsilon_1), t(d), child) : D(\varepsilon_1, d, \varepsilon_2) \mapsto D' = (\varepsilon_2, d', p(\varepsilon_1)) \tag{3.12}$$

We define d' as the affine transformation which displaces one extremity of S while preserving the other one:

$$\begin{cases} d'(p^{\varepsilon_2}_{start}) = d(p^{\varepsilon_2}_{start}) \\ d'(p^{\varepsilon_2}_{end}) = p^{\varepsilon_2}_{end} \end{cases} \tag{3.13}$$

On Figure 3.34c, we can see that the interpreted deformation keeps ε_1 and ε_2 connected, which respects the coherency of the tree. Figure 3.34d shows an alternative rule which splits elongated branches and generates new sub-branches.

Let us consider the case where a branch $\varepsilon_2 \neq \varepsilon_0$ receives an affine deformation emitted by the user $D_0 = \{\varepsilon_2, A, 0\}$. First we need to notify the parent of ε_2, $\varepsilon_1 = p(\varepsilon_2)$. This is the role of $D_{up} = \{\varepsilon_2, A, \varepsilon_1\}$. In order to preserve the hierarchical coherency of ε_1 while deforming ε_2, D_{up} needs to be interpreted as follows:

$$D_{up} \mapsto D_1 = \{\varepsilon_2, \mathrm{B}, \varepsilon_1\} \tag{3.14}$$

where B is the affine transformation which preserves $p^{\varepsilon_2}_{start}$ while deforming $p^{\varepsilon_2}_{end}$ as A:

$$\mathrm{B} = \mathrm{T} \times \mathrm{R} \times \mathrm{S} \times \mathrm{T}^{-1} \tag{3.15}$$

where:

— $\mathrm{T} = \mathrm{trans}\,(p^{\varepsilon_2}_{start})$

— $\mathrm{S} = \mathrm{scale}\left(\frac{p^{\varepsilon_2}_{end} - p^{\varepsilon_2}_{start}}{\|p^{\varepsilon_2}_{end} - p^{\varepsilon_2}_{start}\|}, \frac{\|A\,p^{\varepsilon_2}_{end} - p^{\varepsilon_2}_{start}\|}{\|p^{\varepsilon_2}_{end} - p^{\varepsilon_2}_{start}\|}\right)$

— $\mathrm{R} = \mathrm{rot}\,(p^{\varepsilon_2}_{end} - p^{\varepsilon_2}_{start}\ ,\ A\,p^{\varepsilon_2}_{end} - p^{\varepsilon_2}_{start})$

The bilateral deformation grammar can be expressed using those three rules:

— $R(branch, affine, none) : (\varepsilon, A, S) \mapsto (p(\varepsilon), A, \varepsilon)$

— $R(branch, affine, child) : (\varepsilon, A, S) \mapsto (S, B, \varepsilon)$

— $R(branch, affine, parent) : (\varepsilon, A, S) \mapsto (\varepsilon, A, \varepsilon) \bigoplus_{e \in C(\varepsilon)} (e, A, \varepsilon)$

3.4.3.5 Persistent editing

Enabling to apply deformations at different levels of the hierarchy greatly increases the user's freedom. With this method, small scale edits may, however, be overwritten by subsequent higher level modification, leading to the loss of specific user changes.

Bilateral deformation grammars allow us to seamlessly solve this issue by keeping track of previously locally edited elements. Once an element ε is locally edited by the user, it can be tagged as *persistent*. Grammar rules can then take this tag into account for preserving these elements.

Therefore any global deformation applied later on the object will not modify the previously edited element enabling the user to iterate between global and local deformations as desired.

3.4.4 Results

This section develops several possible applications, inspired by state of the art deformation methods, to demonstrate the versatility of our framework. All the examples were implemented using different deformation grammar rules, within the same interactive sculpting software. We also refer the reader to the video accompanying this work.

3.4.4.1 Grammar creation

Independently of the category of complex object to be deformed, the creation of a new deformation grammar proceeds as follows

1. Design the hierarchy of the object, or use the hierarchy inherited from a previous procedural generation method;
2. For each type of element in the object, identify its internal and hierarchical coherency rules;
3. Based on steps 1 and 2, identify the deformation types applicable to each type of element;
4. Create a set of downward rules for each pair of element types and associated deformation type;
5. Optionally enrich the set of rules with upward rules allowing to maintain upward hierarchical coherency.

Since a rule is needed for every tuple $(t(element), t(deformation), t(relation))$, the number of rules to design is directly correlated to: the number of element types; for each element type, the number of possible deformation types; for each element type, the number of possible children types.

The example grammars presented in this section contain between two and ten rules. Each rule typically takes 10 to 30 minutes to be designed by an experienced user.

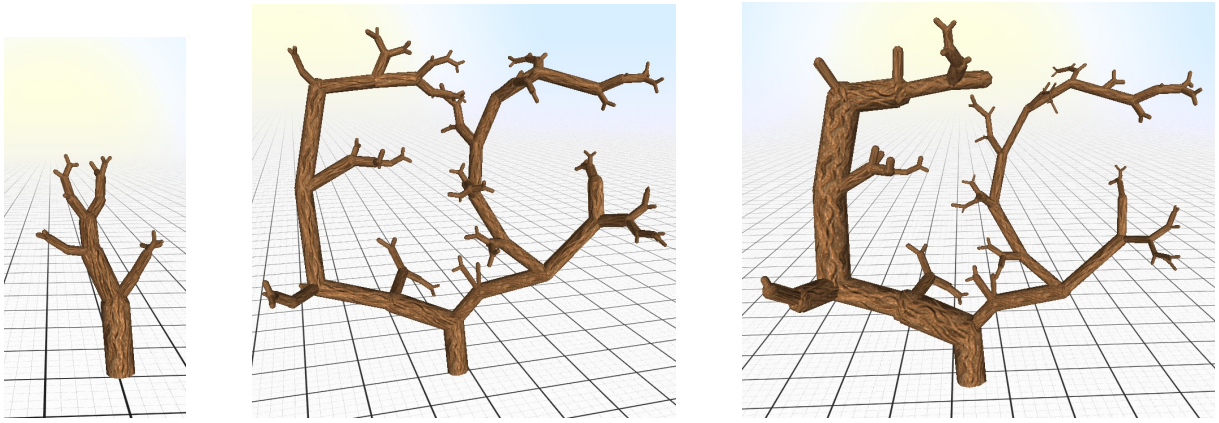
3.4.4.2 Organic shapes

We start by demonstrating our deformation grammar on complex objects representing organic shapes.

Figure 3.35 shows three steps of an interactive tree modeling session. In this example, we use the rules given in Section 3.4.2.3 to ensure that branches remain cylindrical and adjacent. We added a bilateral grammar rules enabling to prevent self-intersection between the different object parts. This rule applies the initial deformation to the uppermost branch of the hierarchy ε_0 with the initial target branch as emitter. The deformation is propagated down the hierarchy only if it does not generate intersections between the sub-branches. Branches longer than a threshold are split and new branches appear at the junction between consecutive branches at the same hierarchical level, using a call to a local L-system generation. Note that our interface makes possible to *dynamically change the deformation behavior* by activating or deactivating specific rules at run time: This is used, for instance, for interpreting a subsequent free-form deformation as a radius change only in Figure (3.35c).

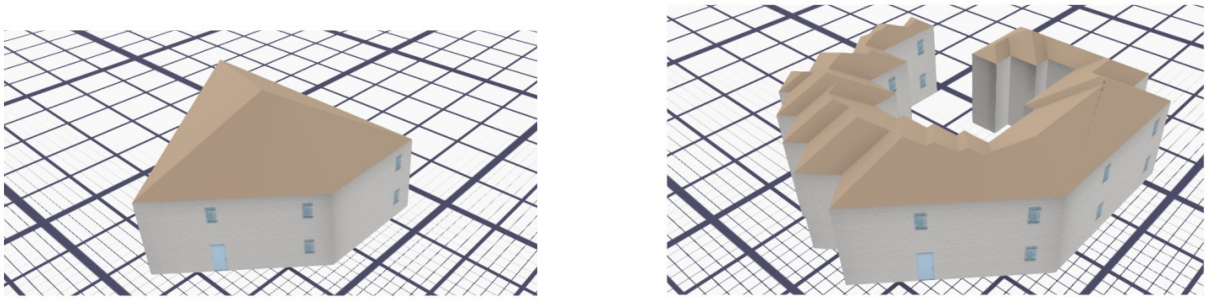
3.4.4.3 Man-made objects

In order to show the versatility of the application of our deformation grammars, we set-up rules to model a deformable house. The house is a hierarchical heterogeneous object whose hierarchy is the following: A house object is composed of floors and a roof. Each floor is composed itself of walls and windows. In this example, the coherency properties are the following: Adjacent walls must be orthogonal, the maximum height of each floor is bounded, and the roof is positioned at the top of the last floor. See Figure 3.36 (second column).



(a) (b) (c)

Figure 3.35 – Deformation grammars allow us to freely deform organic shapes such as trees.(a): Initial tree;. (b): Deformed tree; Note that the geometry of the branches is non degenerated and that junctions are evenly distributed. (c): Tree deformed while free form deformations are interpreted as a radius changes; Dynamic rule changes allow us to manipulate the object in a context-specific manner.



(a) (b)

Figure 3.36 – An initial house (a) is deformed by the user while preserving properties typical of man made objects such as wall orthogonality and floor linear arrangement (b).

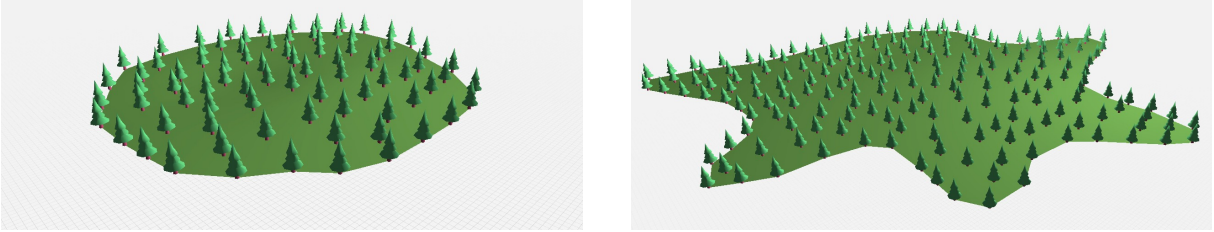
This example also illustrates the possibility to *element specific deformation interpretation*, i.e. rules depending on the nature of the deformation: For example, a vertical translation of the roof is propagated to the house object and interpreted as a global vertical scaling, which allows the roof to be supported by the walls at every time. In turn, a house scaling is interpreted as a scaling of the last floor. A floor higher than a given threshold splits into two floors on top of each other, an in return a floor too low is merged with the underlying one. A translation of a wall piece is interpreted a horizontal wall extrusion, which also results in a re-generation of the roof thanks to our bilateral grammar rules.

3.4.4.4 Object distributions

Objects distributions are hard to deform because of the inter-object constraints, such as non penetration and relative positions (see Emilien et al. (2015)). They can be represented as complex objects. Usually, the parent element in the hierarchy does not have any visual representation, but stores the distribution parameters to be maintained as internal parameters. The objects in the distribution are its children. We implemented three

different examples in order to illustrate the ability of deformation grammars to maintain the coherency in the case of distributions.

The first example, shown in Figure 3.37, is a forest, i.e. a distribution of trees. Naively applying a user-defined deformation to this forest would either create empty regions between adjacent trees, or make some of them too close to each other. We aim at preserving the visual density of trees. This requires to merge trees that are too close, and to create new ones in large empty spaces.
Let us consider that the initial trees are associated to an underlying Delaunay triangular mesh whose vertices are the tree positions. Displacing the trees is expressed as the deformation of the mesh. Our solution for maintaining the visual appearance of the distribution is based on quasi-uniform meshes Stanculescu et al. (2011), which are re-expressed as a specific case of our deformation grammar, as follows: Mesh vertices are maintained at a distance d such that $\frac{d_{detail}}{2} < d < d_{detail}$ (where d_{detail} is a constant learned from the input distribution). The edge collapse and split operations used to maintain the distribution's coherency trigger the elements merging and splitting respectively, which are new types of deformations.



(a) (b)

Figure 3.37 – (a) Initial 2D distribution of trees. (b) Deformed distribution with newly inserted trees in the stretched regions.

We also show a similar example in Figure 3.38, where houses can split or merge based on the same rules. But this time, splitting a large house results in creating several smaller ones, while merging has the opposite effect. Such effect could be used, for instance, for compacting a village while preserving the number of inhabitants it can house.

The third example, shown in Figure 3.39, is a volumetric distribution representing a flock of birds. Elements are merged when they come close to each other (using an element-specific merge transformation), therefore avoiding any intersection. In this case, a grid-based acceleration structure was used to compute element neighborhood.

3.4.4.5 Color transformation

Deformation grammars are not limited to geometrical transformation interpretation: As an illustration, we used the deformation grammar framework to design a color definition tool on distributions as shown in Figure (3.40).

3.4.4.6 Heterogeneous distributions

One of the main advantages of our deformation grammar formalism is its ability to seamlessly handle heterogeneous distributions. Therefore, once a deformation behavior has been described in our framework, it can be further reused as a sub-elements of a larger

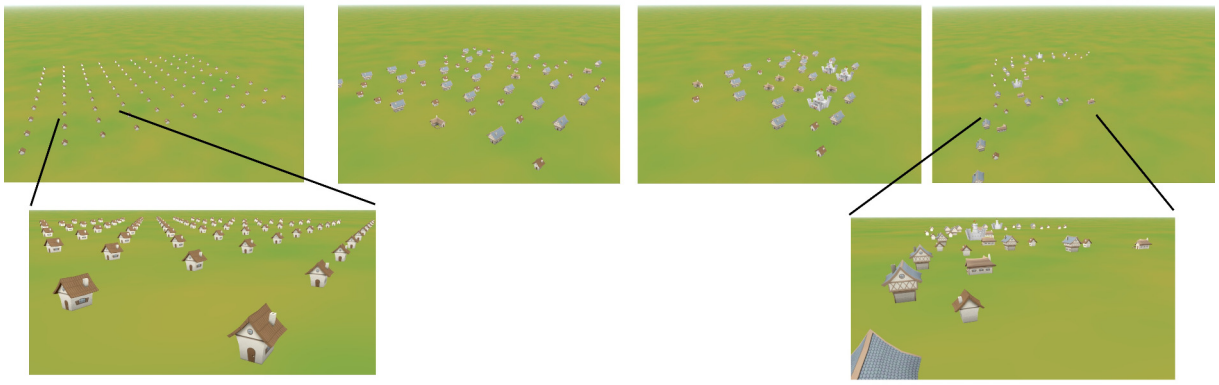


Figure 3.38 – An initial distribution of houses (in left) is interactively deformed by the user using space deformation (middle and right figures). Closed-by houses are merged into larger one to model the increased density.

scene, and interact with other elements. For instance, outdoor scenes such as the one illustrated in Figure 3.41, are defined by assembling the deformation behaviors of the tree distributions, bird flocks, and houses where the root element is the entire scene. The rule interpreting the deformation of the whole scene simply forwards input deformations to its children (the house, the bird flock, and the tree distribution), and prevents them to inter-penetrate. Each element can be either globally deformed, or individually while still preserving the individual and global coherency.

3.4.4.7 Persistent editing

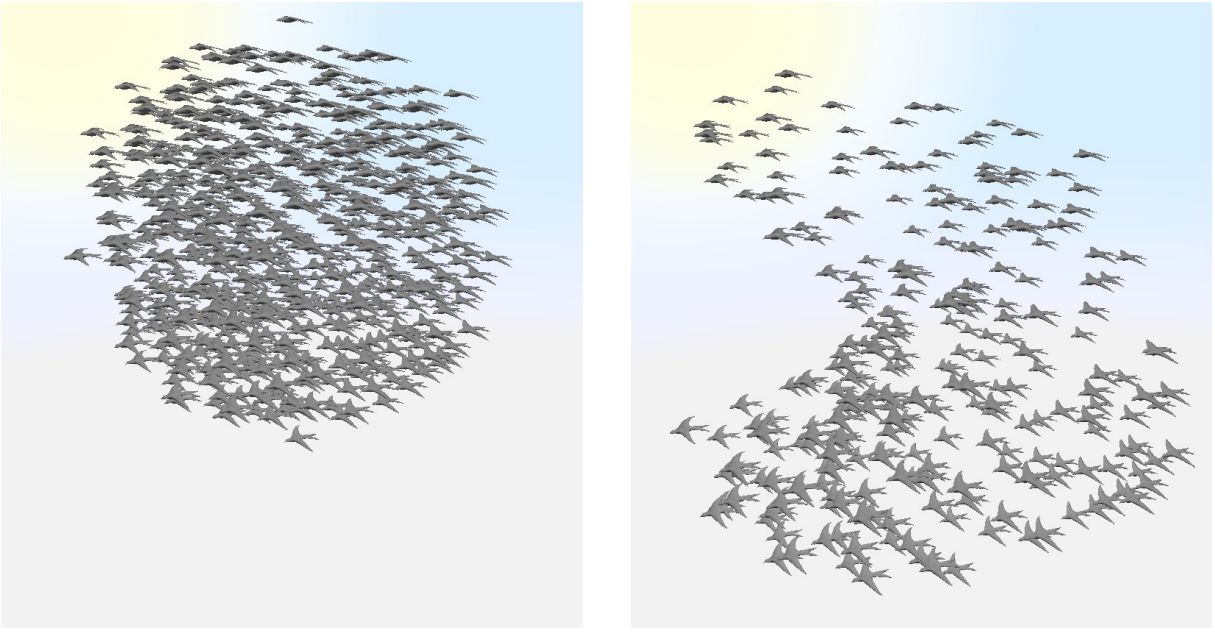
Figure (3.42) illustrates persistent editing. In this case, a forest tree is locally deformed by the user (Figure 3.42b). Next, a global deformation is applied on every tree. If the persistent edit is not applied, the trees may be re-dispatched for maintaining tree density, therefore destroying all previous manual editing operations (Figure 3.42c). Instead, using our persistent editing method on a similar global deformation enables us to preserve the local aspect of the tree while still allowing it to be translated, and other trees to be deformed (see Figure 3.42d).

3.4.5 Discussion

In this section we first compare results obtained with deformation grammars with results of state-of-the-art methods targeted at particular object types. We then discuss the advantages and drawbacks of using deformation grammars.

3.4.5.1 Comparison to state of the art methods

Two methods aiming at deforming complex objects can be relevantly compared to ours in terms of results: Emilien et al. (2015) with the Worldbrush system (described in details in section 3.3) and Longay et al. (2012) with their TreeSketch system.. Each of these methods focuses on a particular type of complex object: distributions of elements and trees, respectively. Although we do not provide in our implementation the specific user interfaces dedicated to trees and elements distributions, enabling to achieve the high



(a) (b)

Figure 3.39 – The user can freely deform this bird flock while maintaining some properties: All bird instances are similar, and non inter-penetrating.

quality interaction provided in these prior works, our deformation grammar would enable to capture similar deformation behaviors: Indeed, all the sub-elements used in these two works only require to me moved, built, and deleted while maintaining specific rules. On the one hand, integrating the histogram preservation as a coherency criteria would enable to interactively deform a 2D distribution of elements similarly to WorldBrush. On the other hand, implementing new deformation types such as element painting, and new branch behavior for trees, would enable to model the deformation behavior of TreeSketch. One of the advantages of using our deformation grammar in such cases would be to fully integrate these two different behaviors within a single framework. Then, within the same scene, the user could seamlessly design the distribution of trees of a forest, while being able to control each of the trees similarly to the approach in TreeSketch.

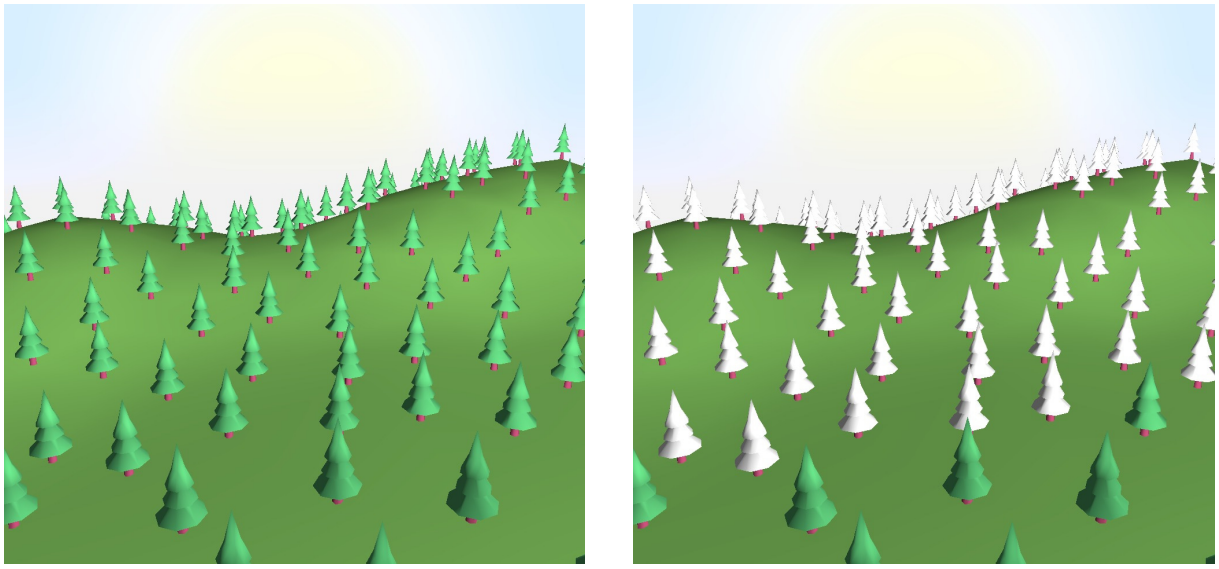
3.4.5.2 Suitability for deforming procedurally generated objects

As shown in Section 3.4.4.2, deformation grammars are particularly well suited to interact with shapes defined using shape grammars. These objects are hierarchical by nature and they can be generated dynamically, enabling us to easily set rules that add or delete parts of the object when the latter is deformed. The tree example of Section 3.4.2 and 3.4.3 uses this principle for generating new branches after splitting elongated ones.

3.4.5.3 Challenges of the faithfulness versus consistency trade-off

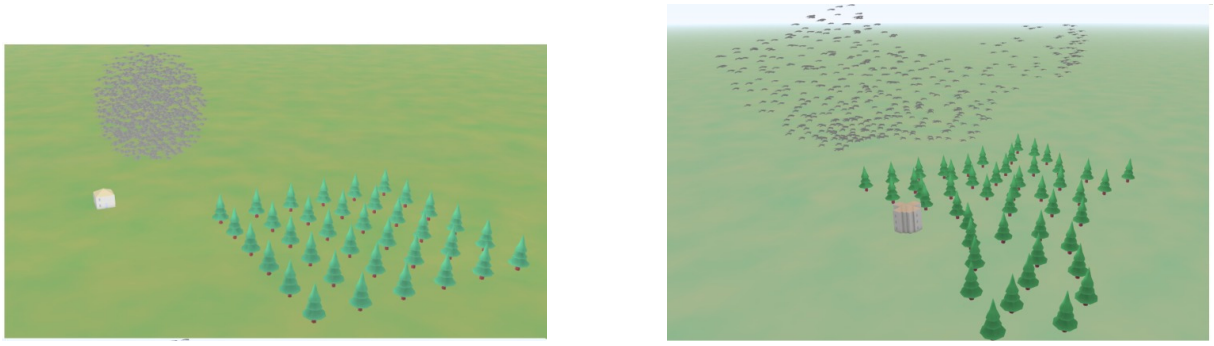
We call *faithfulness* of a deformation interpretation the difference between the non interpreted and the interpreted deformation.

The faithfulness is positively correlated with the predictability of the deformation behavior, and therefore to the intuitiveness of the deformation tool: The more the



(a) (b)

Figure 3.40 – The user can paint an original set of trees (a) with another color (b). The change of color is interpreted by the rule such that only the foliage is affected, not the trunk.



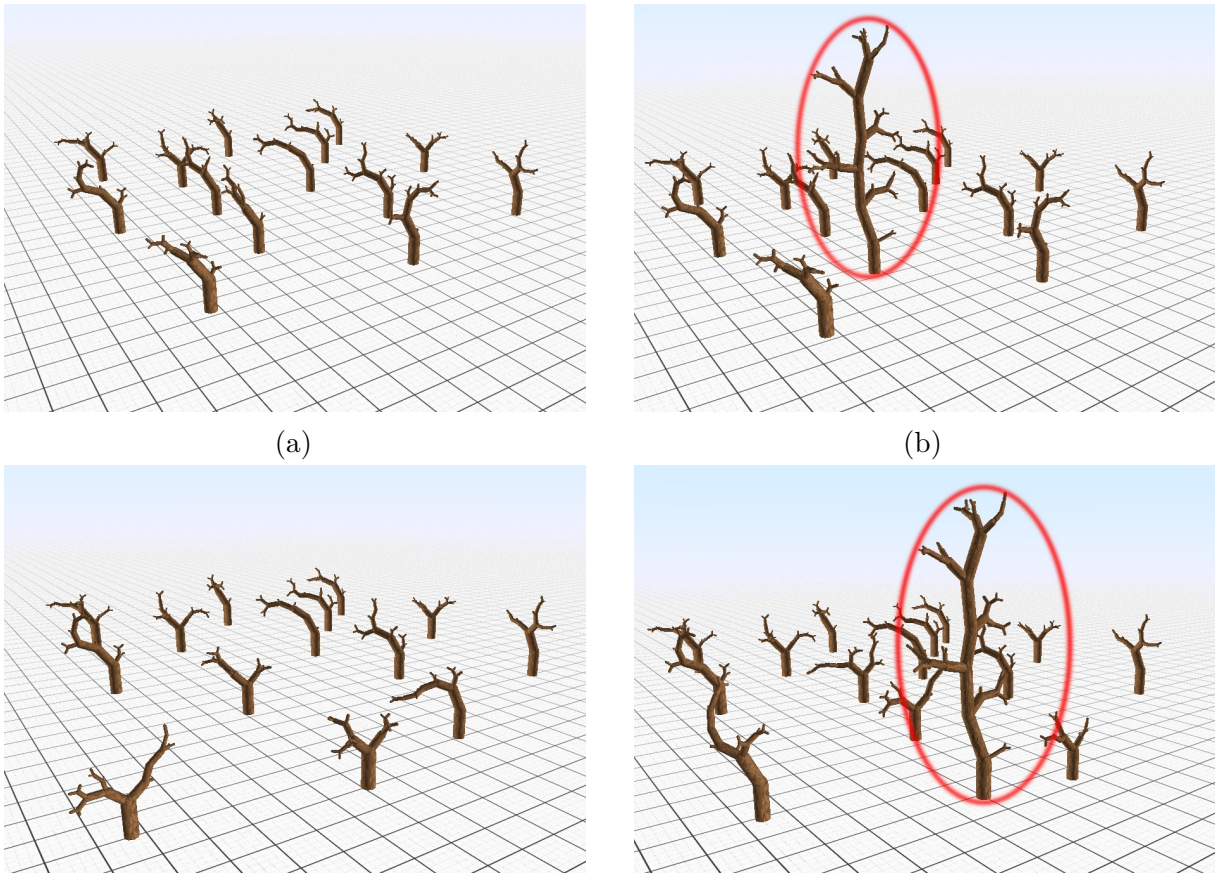
(a) (b)

Figure 3.41 – A single deformation gesture allows to deform various objects at once , preserving different coherencies.

interpreted deformation corresponds to the input deformation (i.e. the deformation interpretation is faithful), the more the object will behave in the way the user expects it to. On the other hand, a perfectly faithful deformation interpretation will probably alter the coherency of the object, losing all interest. A compromise is to be found between coherency preservation and interpretation faithfulness.

The problem with this trade-off is that its value can be application-specific. Even in some applications, depending on the phase, of conception, different trade-offs could be required. For example, a faithfulness-oriented in the shape exploration phase of the object conception could allow more freedom to the artist, whereas it might become less necessary in the final phase.

Ideally, a deformation behavior could be designed, which let the user choose a trade-off (through a slider or a switch). This would require the deformation behavior to handle the change of trade-off as a deformation. Pushing the trade-off toward consistency would therefore require to enhance the object, which might prove difficult.



(a) (b)

(c) (d)

Figure 3.42 – (a): Initial set of trees. (b): Local editing of a single tree. (c): Global deformation without persistent editing leads to a loss of the previous user edits. (d): Global deformation with our persistent editing method enables to preserve the local aspect of the previously deformed tree.

3.4.5.4 Over-constrained coherency

Related to the faithfulness, an object with many constrains may not provide enough degrees of freedom to be deformed as expected. As interpreted deformation will fall inside some very limited deformation space, which may result into a deformation behavior of little interest. For instance, a cube constrained to stay cubic will only allow uniform scaling deformations which may be considered too restrictive by the user.

Note, however, that even with restrictive individual behaviors, the deformation of complex heterogeneous objects combining different types of elements at different levels of a hierarchy will still look rich and expressive.

3.4.5.5 Stochasticity

Stochasticity allows grammars to choose which of several applicable rules to use according to a random law Ritchie et al. (2015). This property is heavily used with shape generation grammars for allowing various shapes to be generated from a single input.

In the case of deformation grammars, the variation of the deformation behavior is not desirable because the intuitiveness of the deformation interpretation relies on its

predictability. Besides, such random behavior prevent the use of undo/redo patterns, which is very useful in design processes. Therefore, we did not explored this track.

3.5 Conclusion

In this chapter, we discussed three methods for designing static 3D shapes from example. These methods allow us to create new content while fulfilling semantic constraints; This constitutes a progress compared to previous artist-oriented 3D shape design processes.

3.5.1 Controllability versus complexity trade-off

The two first methods presented in this chapter propose to model a complex object given a predefined coherency. They are at two opposite of an interesting scale: the controllability versus complexity scale.

The first method identifies replaceable substructures inside complex objects (see Section 3.2). It relies on a complex coherency synthesizing valid complex object models which are sparse in the shape space and hard to create. It leads to a method having a very low user controllability.

The second method offers a painting interface to element distributions (see Section 3.3). It relies on a simple and easy to compute coherency which allows real-time and local user edits. This makes the method having a high user control at the cost of the complexity of its outputs.

Finally, the third method formalizes hierarchical object deformation in a grammar-based framework (see Section 3.4). It compromises between those two by allowing arbitrary deformation (which means high controllability) on arbitrary object (including complex ones). This is done at the cost of not automatically performing coherency preservation: this method performs no "learning" on the input object. The user is required to provide rules which tells how to preserve coherency, which implicitly contains the definition of this coherency. This is in accordance with Figure 2.1, which tells us that any method has to compromise between automatism, coherency, and control.

3.5.2 Smart tools versus smart shapes

In the future, I imagine the "smartness" necessary to more intuitive modeling interfaces will not be found in "smart tools" but rather in "smart shapes". Smart tools are capable of analyzing the shape they model or deform in order to preserve its coherency. However, this requires for the architect to build one tool per type of object, and for the user to learn to use as many tools. As discussed in Section 1, this is part of the difficulties of 3D shape modeling: the large body of knowledge necessary for mastering modeling tools.

In contrast, smart shapes can be arbitrarily deformed while maintaining artist-defined or auto-detected coherencies. This offers the advantage of separating the user (who can use the deformation tool they want) from the object (which can maintain its coherency automatically).

Ideally, such "smart shapes" should be able to represent arbitrary complex object, including spatiotemporal ones such as animations. Chapter 4 presents some advances in this specific direction.

Chapter 4

Design of Animation

Contents

4.1 Introduction 78
4.1.1 Temporality of animations 79
4.1.2 Preservative structures 79
4.2 Waterfall scenes 81
4.2.1 Motivation 81
4.2.2 Overall system 83
4.2.3 Main contribution: waterfall classification 95
4.2.4 Results 96
4.2.5 Discussion 100
4.3 Fluid Sculpting 100
4.3.1 Introduction 101
4.3.2 Overview 102
4.3.3 Feature extraction 103
4.3.4 Feature representation 109
4.3.5 Sculpting Tools 110
4.3.6 Results 111
4.3.7 Discussion 113
4.3.8 Conclusion 115
4.4 Conclusion 117

This chapter addresses the topic of *modeling animated virtual content*. The challenges arising from this task are recalled in Section 1.3: The high number of degrees of freedom of the animation data, and the difficulty of visualizing and interacting with it using 2D interfaces.

More precisely, this chapter restricts its concerns to *liquid animation*. This specific kind of animation is particularly challenging to model and control: It is usually cast as a simulation problem, as shown in Section 2.4. However, Section 1.3.2 recalls the control problems associated to this category of solutions: Only a few degrees of freedom are available to the user, and they are often counter-intuitive; Besides, the non-linear nature of liquid behaviors makes that edits of the simulation parameters can have unpredictable consequences. What is more, simulation-generated liquid animations are usually not temporally consistent representation-wise (each frame of the animation is represented with a different mesh with no correspondence between frames), which makes the editing impractical.

The two research contributions presented in this chapter are based on a facilitating paradigm that I call *preservative structure editing*. The principle is simple: instead of editing directly the raw data representing the animation, the user can manipulate a spatial and temporal consistent structure used as a proxy for interacting with the content. Similarly to the rig-space, the preservative structure constitutes a reduced basis for parameterizing the animation data.

This chapter is organized as follows: Section 4.1 presents more thoroughly the *preservative structure* paradigm as well as a classification of animated content allowing some simplifications on this structure in some cases; Section 4.2 illustrates this simplification in the case of time-limited dependency phenomena with an editing method for waterfall scenes; Section 4.3 extends the concept to fully time dependent phenomenon with a general structure for representing liquid animation; Finally, Section 4.4 concludes on the topic of animated content modeling and identifies several perspectives for *preservative structures* editing.

4.1 Introduction

Modeling animation in general is difficult in particular because animation data linking geometry and time is complex to manipulate. However, some animations may have a static component which can be modeled independently from their temporal component making them more easy to manipulate.

This section introduces the notion of *temporality* of an animation. Temporality allows us to distinguish between animations for which time and space are independent and animations for which they are intrinsically correlated. This notion can be used as an efficient mean for simplifying the representation and the modeling of phenomena falling in the first class.

Phenomena of the second class are more difficult to handle. In such case, this section also proposes the notion of *preservative structure* aiming at establishing an editable interface in cases where space and time are correlated.

4.1.1 Temporality of animations

Fluid simulations produce animation data which are typically represented as mesh sequences. Such data are heavy and uneasy to handle since they represent a full shape a each time step.

However some animated phenomena are in a stable state when their motion flow varies little over time. Consider the animation of a fire at the end of a torch: The appearance of this phenomenon at a given instant does not contain any information on that instant. An other way to say this is that given two random frames of the animation, it is impossible to tell which comes first. We call these kind of phenomena *stationary* by analogy with stationary textures (in our case the ergodicity is temporal rather than spatial).

Note that *static* (i.e. completely motionless) phenomena are a trivial sub-case of *stationarity*. The lack of relative temporal position implies that stationary phenomena have no natural beginning nor end. This makes good candidates for looping videos, as it has been exploited by Liao et al. (2013).

Now, consider the animation of a burning match. Each frame can be localized in time from the position of the sparkle on the stick. The bigger the unburned part of the stick is, the sooner the frame comes. We call these phenomenon *transient*. They represent a transition between two stable states: Completely unburned and completely burned in the case of the burning match.

A third category of phenomena can be identified: *periodic* phenomena, in which a frame can be localized in time modulo a given period at which the phenomenon repeats itself. It is for example the case of the day-night cycle: Frames can be ordered (e.g. using the sun angle) only if they come from the same day. More generally, a *transient* phenomenon where the system evolves toward a state it has taken before actually describes the cycle of a periodic phenomenon. Such phenomena are seldom within the domain of fluids, and will therefore not be considered in the remaining of this chapter.

We call *temporality* the property of an animation to be *stationary*, *transient*, or *periodic*. Note that the *temporality* of a phenomenon is linked to the duration of the corresponding animation: The torch fire will eventually die because of lack of combustible. The animation is stationary during the period of stability between the ignition and the extinguishment. If the animation exceeds these bounds, a frame can be partially localized in time (before/after ignition/extinguishment). In turn, the animation of the match can be considered stationary over very small periods of time during which a negligible length of stick will be burned.

Sections 4.2 and 4.3 respectively presents solutions to *stationary* and *transient* fluid animation modeling. Both solutions are based on *preservative structures*, presented next.

4.1.2 Preservative structures

Animated content can be seen as 3D content changing over time. This change must be subtle: to consecutive frames must present some similarities for the movement illusion to work (this is called beta movement). If two consecutive frames are too dissimilar, objects from the first are not recognized in the second. Therefore, the impression that the same object switched from one configuration to another disappear. It is replaced by the impression that an object vanished and that another appeared.

We define the *preservative structure* of an animation as a representation of what is preserved between consecutive frames. It thus encodes the correspondence between frames.

The preservative structure exists between each two consecutive frames, and is therefore a spatio-temporal object.

Preservative structures for stationary animations The notion of preservative structure is easy to understand on stationary phenomena. Indeed, a stationary phenomenon is always associated with an underlying static shape: this is actually its preservative structure. Let us consider some examples: The smoke escaping from a chimney is confined in an envelope, which is a *static* shape characteristic of this flow. The trajectory of a river can be seen as a *static* path.

An interesting example of preservative structure of periodic animation has been used by Jordao et al. (2014) for animation editing. This work uses crowd patches: a specific data representation for crowds where *static* shapes are the preservative structure. This static shape is edited (using the method of Milliez et al. (2013)) and "dressed" with animation.

An other example using preservative structure has been developed by Bhat et al. (2004) in the case of waterfall video sequences. This example is based on the optical flow of the video stream. Optical flow is a good example of preservative structure for 2D raster animated data. It also fits our restriction on stationary animations since the optical flow of videos for such phenomenon is almost constant.

In summary, preservative structures in the case of stationary phenomena are static shapes. Those shapes are related to the spatial component of the animation, independently of the temporal aspect. This spatial component can be edited using standard shape modeling tools.

Section 4.2 describes a method which uses the static paths of a waterfall network for creating a vectorial editing framework. Now, let us investigate what preservative structures are in the case of transient animations.

Preservative structures for transient animations In the case of transient animations, no time-independent structure can be found in the data. This is natural since by definition those animations do not have any time-independent component.

Since the changes between pairs of consecutive frames of the animation are not constant, the preservative structure is defined per frame: it is therefore fully spatio-temporal. More precisely, it encodes the correspondences within each pair of consecutive frames.

Once again, the optical flow of a video stream is an insightful analogy. We saw before that the videos of stationary phenomena have temporally constant optical flow (which might vary in space however). In turn the optical flow of a transient phenomenon video varies along time. Each frame of optical flow indicates for each pixel the displacement of the corresponding value in the animation. In the same way, the preservative structure of a transient animation indicates how frame data are transported across frames at each frame interval.

Section 4.3 describes a method which extracts a preservative structure from an arbitrary 3D liquid animation. This structure enables us to design spatio-temporal editing tools allowing to directly manipulate the main features of the animation. But first, let us explore the stationary case of 3D waterfall networks.

4.2 Vectorial editing of stationary animations: the case of waterfall scenes

The work presented in this section results from a collaboration with Arnaud Émilien (Université Grenoble Alpes/Inria) and Pierre Poulin (Université de Montréal). The resulting paper (Emilien et al. (2014)) was published in the CGF journal and presented at the Eurographics conference in 2015 (www.eurographics2015.ch):

Emilien, A., Poulin, P., Cani, M.-P., and Vimont, U. (2014). Interactive procedural modelling of coherent waterfall scenes. *Computer Graphics Forum*, 34(6):22–35

The remaining of this section is organized as follows: Section 4.2.1 introduces the problem of waterfall scene editing; Section 4.2.2 gives an overview of the method; Section 4.2.3 details one of my main contribution to this work: a quantitative waterfall classification; Section 4.2.4 presents some results of this method; Section 4.2.5 discusses the results and draws limitations of this work.

4.2.1 Motivation

Procedural modeling is a convenient paradigm for automatically modeling complex objects and scenes; It has been applied in many contexts such as terrains, trees, buildings, street networks, or cities generation (see Section 2.2). In addition to its efficiency for generating details, its power lies in its ability to ensure that certain constraints are respected—for instance from a physical, biological, or architectural viewpoint—making it much easier for the user to create consistent models.

However, when one has a very specific goal in mind, the intricate parameters of these automatic procedures can be very cumbersome to tune. In such situations, it would be advantageous to separate coarse-scale and fine-scale editing: The coarse scale modeling could easily be handled by the artist using adequate tools, while the fine-scale details could be automatically generated by the method, while ensuring both fine- and coarse-scale consistency. In this section, we apply this paradigm of interconnected procedural generation and interactive user-control to the modeling of animated waterfall scenes.

4.2.1.1 Waterfalls

Waterfall scenes as the one depicted in Figure 4.1 offer some of the most beautiful landscapes in nature. So much that hikers will spend hours walking through difficult terrain for their gratifying sight. Therefore, it seems legitimate to integrate such assets inside virtual worlds. However, waterfalls modeling presents two major difficulties: They might take various appearances, requiring different representations at once; Waterfalls and terrain shapes are intrinsically connected: The terrain guides the waterfall it supports while being carved by the passing water, creating a complex interaction. As a result, no easy-to-use method for designing waterfall scenes has been developed so far in computer graphics.

Given a terrain fit for supporting a waterfall, different solutions currently exist for modeling water bodies: One consists in using physically based simulation of fluids, which produces realistic flows; Unfortunately, the volumetric scale range of a waterfall requires

Figure 4.1 – Picture of *trou de fer* on *Ile de la Réunion* (courtesy of Serge Gélabert). Such beautiful sceneries are very difficult to reproduce in virtual worlds.

huge representations, which makes this solution intractable. Another solution is to manually create waterfall models with standard modeling tools, using manifold meshes and/or particles, and to position and animate them by hand. Both of these methods require a terrain suited for supporting a waterfall, which is impossible to check before the waterfall is modeled. Changing the terrain requires to completely recreate the whole waterfall model, and vice versa.

4.2.1.2 Contributions

In this work, we combine interactive and procedural methods to ease the process of designing consistent waterfall scenes. Our solution is based on a new interactive procedural model for flowing water networks. It enables users to easily shape complex waterfall scenes while automatically enforcing the consistency of the results in terms of hydraulic flow and terrain embedding. Our main contributions include the introduction of:

— a slope-flow diagram-based classification of waterfalls;
— three parametric models for designing waterfall elements;
— a procedural method for ensuring the waterfall network consistency;
— automatic methods for locally adapting water trajectories to the terrain and/or the terrain to the flow.

All these contributions combine nicely into a vectorial editing framework allowing an artistic approach to river and waterfall scenery design. The resulting scene could be either used as a synthetic environment for games or films, or as an initial setup for further refinements through physically-based simulation.

4.2.1.3 Preservative structure

Rivers and waterfalls are typical stationary phenomena. It seems therefore natural to aim at editing a static shape for modeling their animations. This shape is the preservative structure we are looking for.

Our goal is to offer some simple tools for editing the shape of the waterfalls trajectories, in the form vectorial controllers. Those controllers are used for generating a finer-scale representation of the waterfall network. This representation is converted into a texture-animated mesh, and the latter is used for adequately deforming the terrain so that flow consistency is preserved.

The challenges of waterfall scene editing rely in the inter-relationship between the terrain and the waterfall rather than in the animated nature of the waterfall. Still, it shows how stationary phenomena can be modeled using static shape modeling tools.

4.2.2 Overall system

The goal of this method is to leave the coarse-scale design of waterfall scenes in the hands of the user, while automatically generating consistent and detailed results. The pipe-line of our method is illustrated in Figure 4.2. Here is an overview of each step:

1. Given the navigable 3D view of a terrain, the user starts by creating a coarse representation of the waterfall network geometry (trajectories and flow of the water bodies) and topology (direction and connection of the bodies). For doing this, the user selects a controller type: *contact*, *free-fall*, and *pool*; Then he places 3D control points which are interpolated using Cardinal Splines. We call the resulting oriented vectorial **controller network** $\mathcal{U}$; The later is interactively checked and adapted for down-hill validity (all rivers must go down-hill with a minimum slope). This step is detailed in Section 4.2.2.1.
2. Each arc of the controller network corresponds to a waterfall. A custom hydraulic resolution procedure allows us to compute a flow in each of those arcs, yielding what we call the **hydraulic graph** $\mathcal{G}$. This step is detailed in Section 4.2.2.2
3. Each point of an arc of the hydraulic graph can be associated with a flow and a slope. These two parameters determine a class to which this point belongs, according to our quantitative waterfall classification described in Section 4.2.3. Each arc is divided into constant-class segments, and this class is used for computing fine parameters such as the width and the precise trajectory. We call the result the **waterfall network** $\mathcal{W}$. This step is detailed in Section 4.2.2.3.
4. Lastly, all data contained in the waterfall network are combined inside a geometrical representation called the **integration mesh** $\mathcal{M}$. The latter is used for defining terrain deformations and combined with segment types for creating parameter maps used for rendering (fine scale speed, rock and foam densities, and wave intensity). Finally, the integration mesh is used as a visual representation for the water surface; It is textured and animated from the fine-scale parameters computed previously. This step is detailed in Section 4.2.2.4.

Users give two kinds of information about the waterfall they create: its coarse type (by selecting the appropriate tool), and its coarse geometry (by positioning control points). Both informations are used inside the two-level quantitative classification scheme for yielding fine-scale information: a precise type (allowing to predict fine-scale parameters) and a precise geometry (through types-specific subdivision).

Note that the controller network is still editable at any time: each edit of the controller network triggers the processing of the whole pipe-line. This allows the user to iteratively modify the animated waterfall scene model.

The user can interactively visualize all the data produced by the system, as shown in Figure 4.3.

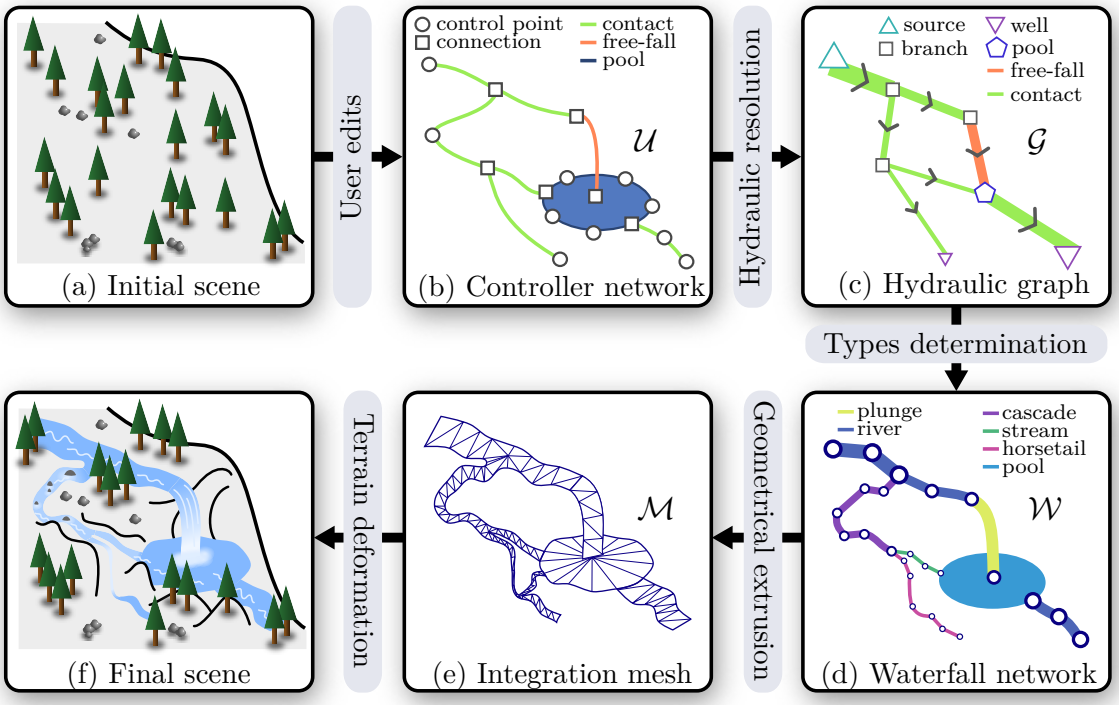
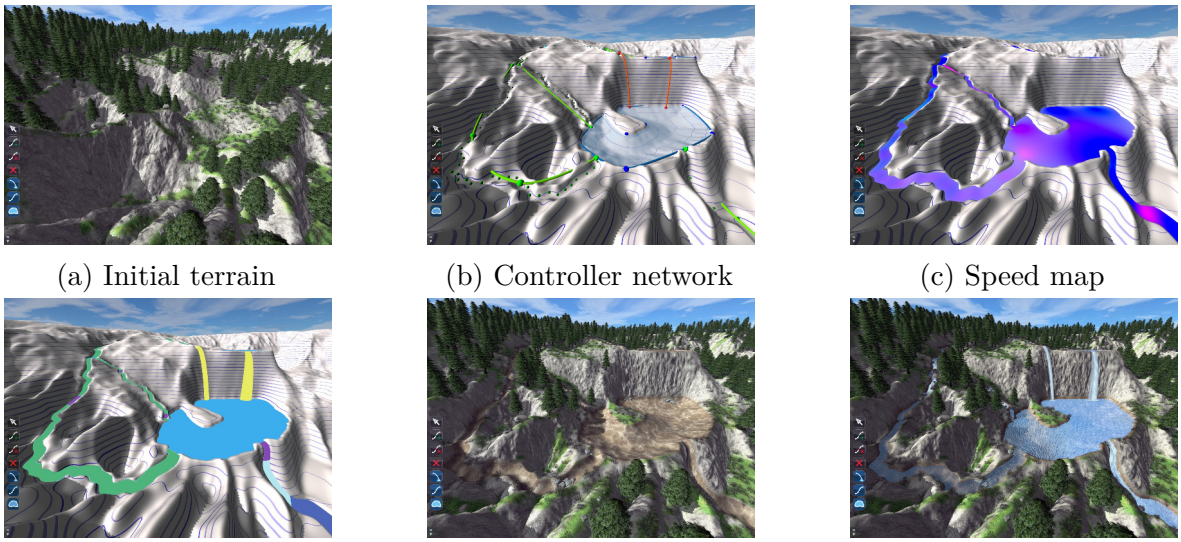


Figure 4.2 – Overview of our waterfall editing framework. An original terrain (a) is used as a support for creating a vectorial controller network $\mathcal{U}$ (b). A flow computation procedure transforms $\mathcal{U}$ into a hydraulic graph $\mathcal{H}$ (c). Flow and slope are combined for determining a local waterfall types (d) which let us infer the geometry of the river $\mathcal{M}$ (e). Finally, the terrain is deformed and integrated with the animated river (f).



(a) Initial terrain (b) Controller network (c) Speed map

(d) Waterfall types (e) Deformed terrain (f) Final result

Figure 4.3 – Interactive visualization of each step of our pipe-line.

4.2.2.1 Network creation

The user builds a controller network $\mathcal{U}$ composed of a set of vectorial elements $\{\mathcal{V}_i\}$ called *controllers*. A controller $\mathcal{V}_i$ is a cardinal spline associated to a controller type $\alpha_i \in \{$ *free-fall*, *contact*, *pool* $\}$. Cardinal splines are defined through control points; A control points can be shared between multiple controllers.

In terms of user interface, the user starts by selecting a controller type. Depending on the latter, different design tools are offered:

— A *contact* controller is created by positioning a series of 3D control points on the terrain (and optionally over and under it);

— A *pool* controller is first defined by a flat plane, on which is defined a closed contour with 2D control points;

— A *free-fall* controller is created by positioning two endpoints in 3D, the lower one being typically placed inside a *pool*.

Free-fall and *contact* controllers are oriented according the ordering of their control points: The first inputed control points are preceding (i.e. are considered up-stream of) the later ones.

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.

To ease the adaptation of the controller network to the underlying terrain, the user can project all control points onto the terrain. They can also define, if desired, 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 plane level is automatically set to the average height of all projected contour points.

Extremity control points of a controller can be re-used for starting a new controller, creating a junction. When a controller $\mathcal{V}_i$ starts at the control point where another controller $\mathcal{V}_j$ ends, we say that $\mathcal{V}_j$ precedes $\mathcal{V}_i$.

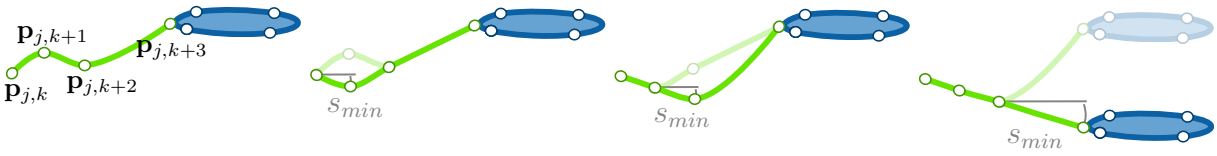
Minimum slope For $\mathcal{U}$ to be consistent, the system must ensure that each control point is higher than all its successors (in the same controller as well as in successive ones). This allows the water to flow "down" in the user-defined direction.

This is done by using some automatic correction of the user-defined positions, during a traversal of the controller network (see Figure 4.4): Starting with the sources of the network, and traversing it in a topological sorted order, we check that each control point maintains a given slope with each of its successors. If not, the failing control point is lowered to match the constraint. Pool contour planes are lowered if necessary, according to the full set of incoming controllers. At this stage, loops in the controller network are detected and deleted.

4.2.2.2 Hydraulic resolution

Starting from the controller network $\mathcal{U}$, we build the hydraulic graph $\mathcal{G}$ as an oriented graph representing the flow of our waterfall network. $\mathcal{G}$ is composed of:

— a set of typed nodes $\mathcal{N} = \{\beta_j\}$, where $\beta_j \in \{$ *source*, *well*, *branch*, *pool* $\}$ is a node type; and



(a) (b) (c) (d)

Figure 4.4 – Minimum slope constraints are imposed on each node of the controller network, to make it consistent with the user-defined flow direction. From left to right: An initial controller network (a) is progressively modified (b, c, d). The algorithm performs a breadth-first traversal of $\mathcal{U}$ and lowers every node which do not satisfy the minimum slope constraint.

— a set of weighted arcs $\mathcal{A} = \{a_j\}$, with $a_j = (\beta_j^0, \beta_j^1, \gamma_j)$, where β_k^0 is the origin, β_k^1 is the end, and γ_j is the flow passing through of the arc.

Graph construction A node in $\mathcal{G}$ is generated for each free-fall, contact controller extremity, and for each pool in $\mathcal{U}$. Pool controller nodes are given the type *pool*; Controller extremities associated to multiple free-fall and contact controllers are given the type *branch*; Other controller extremities are given the type *source* if they have no predecessor, and *well* otherwise. Arcs are created between corresponding nodes of $\mathcal{G}$ for all free-fall and contact controllers of $\mathcal{U}$.

Flow computation All water exchange in the network are supposed to be represented in $\mathcal{G}$. This means that the flow is constant through the arcs of the graph, and that the incoming flow of a node equals its outgoing flow. Expressing this at each node of the hydraulic graph leads to a system of interconnected equations.

For solving this system, the graph is traversed in dependency order from the sources to the wells, and a node type-dependent heuristic is used for progressively determining arc flows. This enables us to take into account the user specifications on the relative strength of the input flows, and to generate a solution that best matches the coarse geometric trajectories in terms of branching angles at each node, as explained next.

Branch flow repartition The flow of an outgoing arc at a branch node should depend on the angles between the inflow and outflow arcs, to capture the natural course of water. For instance, we expect that most of the flow should follow its own original direction.

Let $\{a_j\}$ be the set of input arcs of branch node β_i. Each a_j can be associated with an incoming direction represented as a normalized vector $\mathbf{u}_j$ (see Figure 4.5). Let $\{b_k\}$ be the set of outgoing arcs of β_i, and $\{\mathbf{v}_k\}$ their corresponding directions. We distribute the incoming flow $\sum_j \gamma_j$ to each output arc according to a normalized weight $\bar{w}_{j,k}$ defined as:

$$w_{j,k} = 1 + (\mathbf{u}_j \cdot \mathbf{v}_k) \qquad \bar{w}_{j,k} = \frac{w_{j,k}}{\sum_l w_{j,l}}. \tag{4.1}$$

The outgoing flow γ_k for arc b_k is computed as:

$$\gamma_k = \sum_j \gamma_j \bar{w}_{j,k}. \tag{4.2}$$

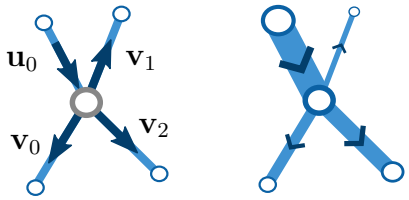


Figure 4.5 – Flow repartition in a branch. Left: Input and output directions. Right: Resulting flow exchange, drawn as segment thicknesses.

Pool flow repartition 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 could be unusable in an interactive system Zhu et al. (2011). 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.

4.2.2.3 Procedural fine-scale generation

In this section, we generate the 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 into fine-scale trajectories. Subdivided trajectories are waterfall segments. We compute the type of each segment using the classification from Section 4.2.3 and deduce visual parameter for each of them.

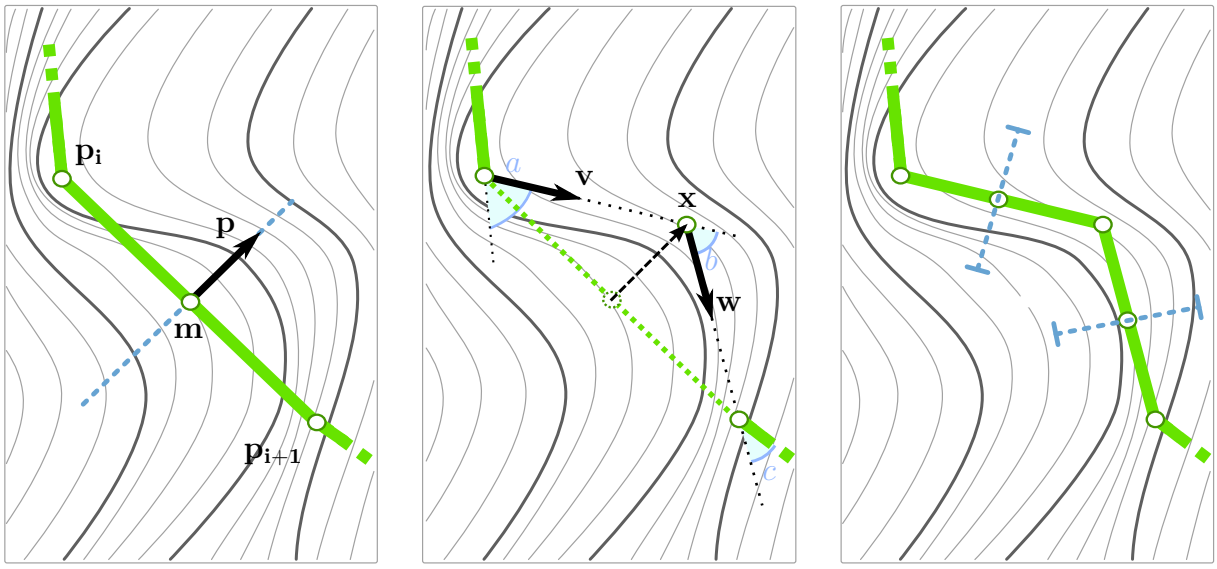


Figure 4.6 – Recursive subdivision process for waterfall network segments: (a) The segment $[\mathbf{p_i}, \mathbf{p_{i+1}}]$ is subdivided at its middle $\mathbf{m}$, which is moved along perpendicular direction $\mathbf{p}$. (b) The final position $\mathbf{x}$ is the point that minimizes our cost function $C(x)$. (c) The process is iterated until segments are smaller than the threshold l.

Fine-scale trajectories In order to procedurally refine the coarse-scale trajectories created by the user, we provide a fractal-like subdivision scheme based on midpoint displacement. This scheme adapts the fine-scale trajectories to the underlying terrain while matching user inputs (see Figure 4.6).

Let $a_i = (p_j, p_{j+1}, \gamma_i)$ be an arc of G. We subdivide this segment at its middle point $\mathbf{m}$ and move it along $\mathbf{p}$, the normalized vector orthogonal to $[p_j, p_{j+1}]$ in the top view plane. This creates two new segments, $V = [\mathbf{p_i}, \mathbf{m}]$ and $W = [\mathbf{m}, \mathbf{p_{i+1}}]$, and we note $\mathbf{v}$ and $\mathbf{w}$ their respective normalized directions. The new point can be written $\mathbf{x} = \mathbf{m} + \lambda\mathbf{p}, \lambda \in [-\tau, \tau]$, where τ is the maximal displacement amplitude. While the original midpoint displacement scheme randomly selects a value for λ, we choose the value which minimizes the following cost function:

$$C(x) = w_g.C_g(x) + w_a.C_a(x) + w_r.C_r(x) \tag{4.3}$$

where $C_g(x)$ is the gradient cost, $C_a(x)$ the angle cost, and $C_r(x)$ the random cost. The values w_g, w_a, and w_r are weight coefficients associated to those costs; We set $w_g = 1$, $w_a = 0.1$, and $w_r = 0.2$ for all our examples.

The **gradient cost** favors paths that go down the slope of the terrain; We define it as:

$$C_g(x) = \frac{1}{4}\left[\left|1 + \frac{\int_{[0,1]} \tilde{\mathbf{g}}(p^\theta_{p_i,x}).\mathbf{v}\ d\theta}{\|p_{i+1} - x\|}\right| + \left|1 + \frac{\int_{[0,1]} \tilde{\mathbf{g}}(p^\theta_{x,p_{i+1}}).\mathbf{w}\ d\theta}{\|x - p_i\|}\right|\right] \tag{4.4}$$

where vector $\tilde{\mathbf{g}}(x)$ is the normalized gradient of the terrain elevation at position x, and $p^\theta_{a,b}$ is a linear interpolation of a and b with coefficient θ. C_g will be minimal for paths descending the height gradient: $\tilde{\mathbf{g}}(x).v = -1$ if v is descending through the biggest slope.

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 $a = (p_{i-1}p_i, p_ix)$, $b = (p_ix, xp_{i+1})$, and $c = (xp_{i+1}, p_{i+1}p_{i+2})$ (in radians), created at the introduction of new point x (see Figure 4.6), as:

$$C_a(x) = \left(\frac{a}{\pi}\right)^2 + \left(\frac{b}{\pi}\right)^2 + \left(\frac{c}{\pi}\right)^2 \tag{4.5}$$

The **random cost** aims at creating meanders on flat terrains. It is based on a 2D Perlin noise. Note that is value is negligible compared to the gradient cost on non flat terrains (due to a low w_r).

The segments are recursively subdivided until their length is inferior to l. The **maximal displacement amplitude** is the half of the segment length: $\tau_i = \frac{\|p_{i+1} - p_i\|}{2}$. The detail size l is proportional to the flow: $l_i = \gamma_i/\sigma$. 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 $\sigma = 0.5$.

Waterfall Network Construction The waterfall network is composed of waterfall segments $\mathcal{S}_i = (\mathbf{s}_i, \gamma_i, \kappa_i, \delta_i, \epsilon_i, \zeta_i)$ where:

- $\mathbf{s}_i$ is the segment vector;
- γ_i the flow going through the segment;
- $\kappa_i \in \{$ *stream*, *horsetail*, *cascade*, *rapid*, *block*, *river*, *ribbon*, *plunge*, *ledge*, *cataract* $\}$ the waterfall type according to our classification (see Section 4.2.3);
- δ_i the speed of the flow; and
- ϵ_i and ζ_i respectively the width and depth of the riverbed (Figure 4.7).

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 is the total incoming flow; and

— ζ_j is the depth.

Each segment vector $\mathbf{s}_i$ is directly extracted from the subdivided trajectories computed previously, 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.

For each waterfall segment S_i, we know its slope s_i and its flow γ_i. These values automatically determine the type of waterfall segment κ_i, as explained in Section 4.2.3.

Note that the user may define a non-plausible waterfall controller, e.g., defining a free-fall on a flat terrain, or a contact on a vertical terrain, the waterfall segment may lie outside of the valid range of values in the slope-flow graph. In this case, the parametric model is still used by considering 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.

The final step in the generation of the waterfall network is to compute the properties of each segment 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 4.7). In this case, the flow can be expressed by:

$$\gamma = \frac{\delta \epsilon \zeta}{2} \tag{4.6}$$

Since we have one equation with three unknowns and complex inter-dependencies, a physically accurate solution should rely on strong assumptions, which would not be justified here. Instead, we decided to solve the system by providing simple and intuitive functions to compute these variables.

We first propose to solve for the speed as a simple linear function of slope s: $\delta = k.s$, where $k \in \mathbb{R}^+$. 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 that our slope constraint enforces that $s > 0$ everywhere for a waterfall segment. In our prototype, we used $k = 10$ and $f_d(\epsilon, \gamma) = \frac{\gamma}{2}$.

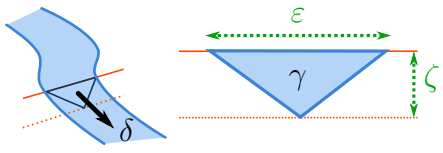
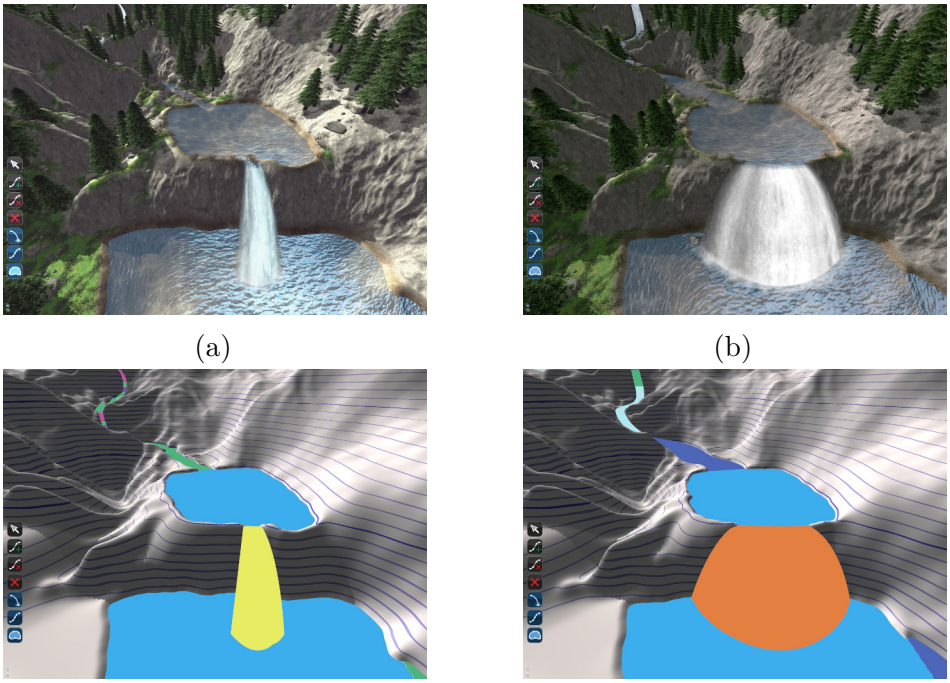


Figure 4.7 – In first approximation, waterfall segments are considered as having a triangular section and constant flow. This simplification is used for computing the segment width and depth from the flow.

Applying Equation 4.6 is now possible, which allows us to compute the width and to deduce the depth value. While simple, our model efficiently provides plausible results using intuitive parameters. In Figure 4.8, 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, as shown in Figure 4.8. 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.



(a) (b) (c) (d)

Figure 4.8 – The water flow has a direct impact on the visual appearance of the waterfall. This is due to the fall geometry (a bigger flow implies a larger fall) and to its appearance: a bigger flow means a different type, which is set to look different.

4.2.2.4 Geometrical embedding

The data computed until now are used for generating an integration mesh. It is used as a visual representation for the water surface, but also for creating parameter maps and for defining terrain deformation.

Integration mesh The integration mesh is a 3D mesh defining the waterfall surface. It is composed of the surface meshes for all waterfall elements, as shown in Figure 4.9. It is generated using the fine-scale network trajectories and the widths of the segments.

The mesh parts corresponding to contact elements are computed by extruding segment trajectories of their half width $\frac{\epsilon}{2}$ along their normal in the horizontal plane. Pool and branch meshes are simply triangulated from their contours. Free-fall meshes are extruded from the borders of their incoming segments and follow the free-fall element curve. We carefully handle the connections between mesh parts, namely between contact and branch meshes, contact and pool meshes, and contact and free-fall meshes.

Parameter maps creation To further animate and decorate the waterfall, we use waterfall type-dependent parameters. These parameters are encoded into parameter maps,

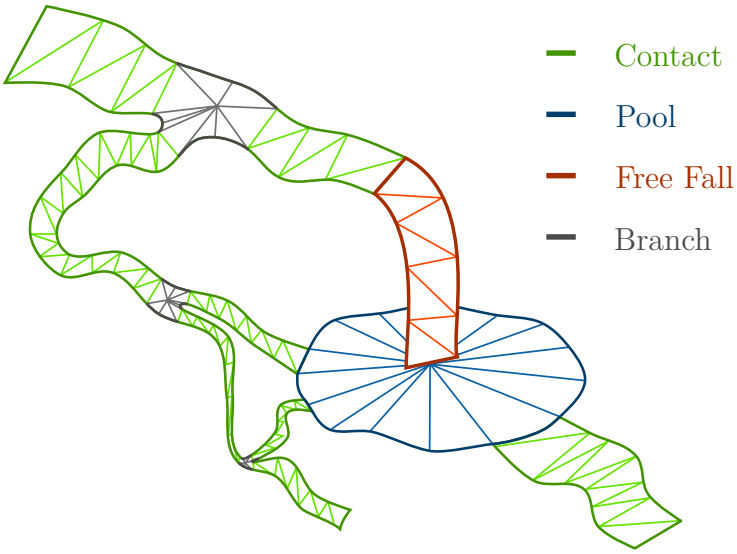


Figure 4.9 – The integration mesh is generated procedurally from the refined waterfall trajectories and flow data. Each element is generated independently while satisfying boundary constraints.

as shown in Figure 4.12. A parameter map is computed by rendering the integration mesh viewed from above into a texture, similarly to the road footprints of Bruneton and Neyret (2008). During this process, we render each sub-mesh in a gray-scale map, depending on its type and on the map that is being computed.

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.

Table 4.1 lists a set of parameter values depending on the waterfall type. By using these values as gray scale 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; The disturbance map relates to the amplitude of the waves on the water surface (see Figure 4.10).

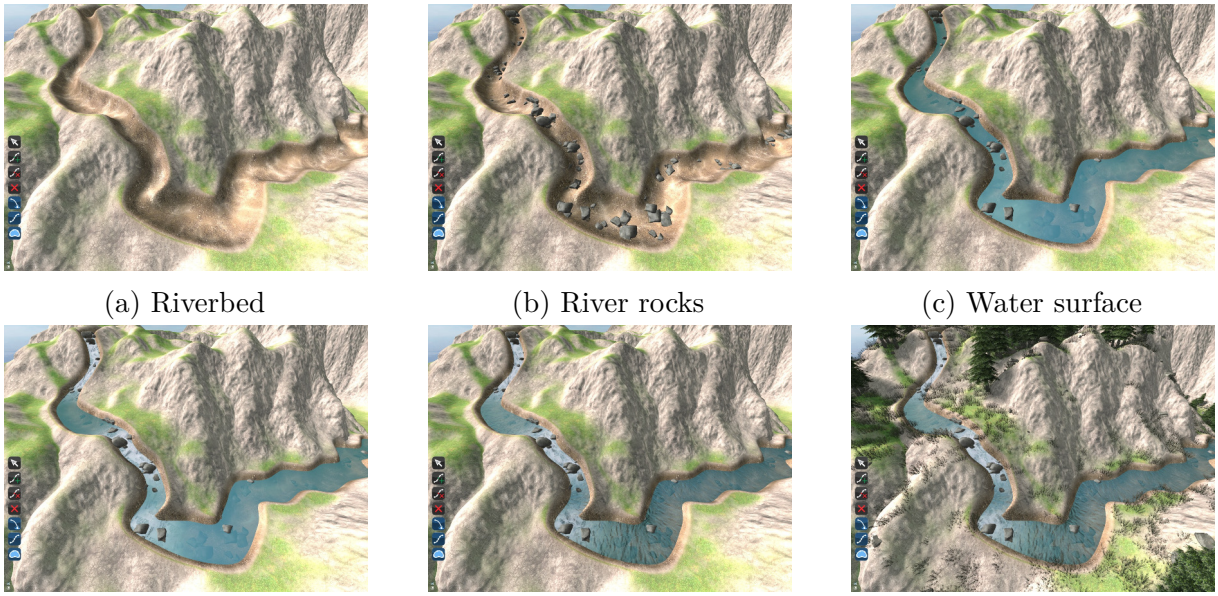
The variation of values depending on the waterfall type allows increases of the visual difference between them, and the overall appearance of the scene. Note that some filtering is applied to these maps to reduce visible transitions between different types.

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

Figure 4.11 shows how the speed of the water is computed depending on the type of the segment. We use different approaches to compute the surface speed of contacts, pools, and branches. For a **contact**, the speed at point x is given by:

$$\delta_x = \delta_c \left(1 - \frac{\|x - c\|}{\|b - c\|}\right) \tag{4.7}$$

where c and b are the projections of x along the cross section on the main axis and on the shore respectively. For a **pool**, a fixed number of 2D fluid simulation steps are evaluated (see Stam (1999)). For a **branch**, we use a standard interpolation method based on a standard technique of weighting by the inverse distance; Each point p_i is considered as a



(a) Riverbed (b) River rocks (c) Water surface

(d) Foam texture (e) Wave displacement (f) Final scene

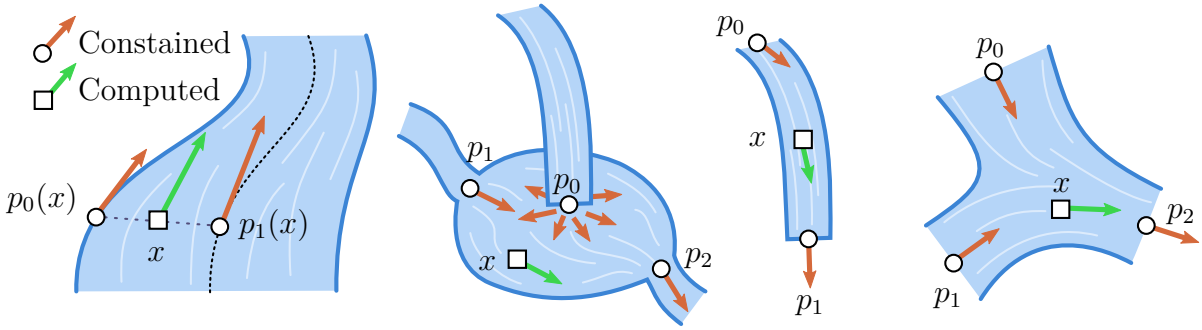
Figure 4.10 – Each parameter map is used for generating procedural details of the waterfall. Original details of the terrain are removed if they are inside the river bed.

velocity constraint δ_i. The speed within a branch is given by $\delta x = \sum_i \omega_i \delta_i$. Interpolation weights ω_i are computed using the equation:

$$\omega_i = \frac{\tilde{\omega}_i}{\sum_i \tilde{\omega}_i} \tag{4.8}$$

$$\tilde{\omega}_i = \prod_j \left(1 - \frac{\|x - p_i\|}{\|p_j - p_i\|}\right) \tag{4.9}$$

Finally **free-fall** are not rendered like the other parts in the speed map, but their mesh part is instead animated using a standard ballistic velocity model.



(a) Contact (b) Pool (c) Free-fall (d) Branch

Figure 4.11 – Each element of the integration mesh is associated to a speed computation procedure.

Terrain deformation The terrain is stored as a heightfield, on which a deformation map is applied to adapt it to waterfalls. This deformation map is computed using a vectorial constrained Poisson solution, as inspired by the terrain modeling method of

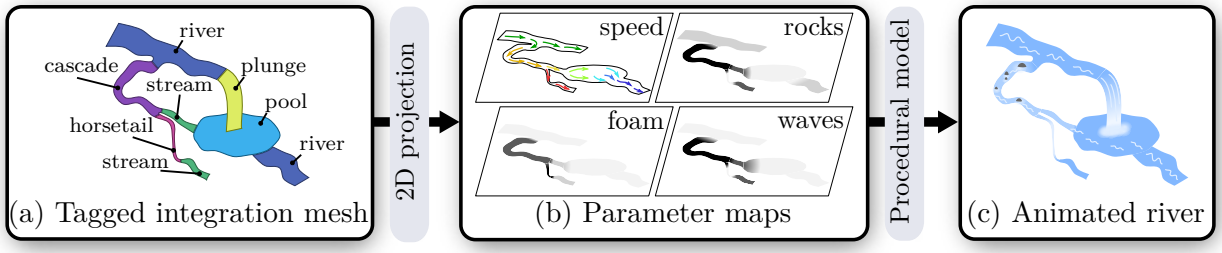


Figure 4.12 – The integration mesh $\mathcal{M}$ associated to the waterfall types (a) allows us to define parameter maps (b): Speed, rock density, foam, and wave intensity; These are used in a procedural model generating the animated details of the waterfall surface (c).

Hnaidi et al. (2010). However, our solution differs from theirs on three points: It propagates deformations (i.e., height differences) instead of heights; It generates three different types of constraints: riverbanks, riverbeds, and overhang; As a result of our approach, small details on the original terrain will remain on the terrain after deformation.

Riverbanks constrain the terrain to match the border of the integration mesh. The difference d is the value that will be diffused within our solver such that is matches 0 on $\partial\mathcal{M}$. We also diffuse a gradient constraint to ensure that the neighborhood of the terrain just outside of the waterfall is higher than its border (see Figure 4.13a). Please refer to Hnaidi et al. (2010) for more details about the diffusion algorithm.

The border constraints do not provide any control on the river profile. For this reason, **riverbeds** constraints are added. For a given point x located inside the waterfall border we create a height constraint based on its distance to $y \in \partial\mathcal{M}$, the nearest point of the border. A profile function f_d is applied on this distance for creating the river profile. The operation is repeated on a dense sampling basis (see Figure 4.13b). This defines a set of additional elevation constraints processed using our solver, as shown in Figure 4.14.

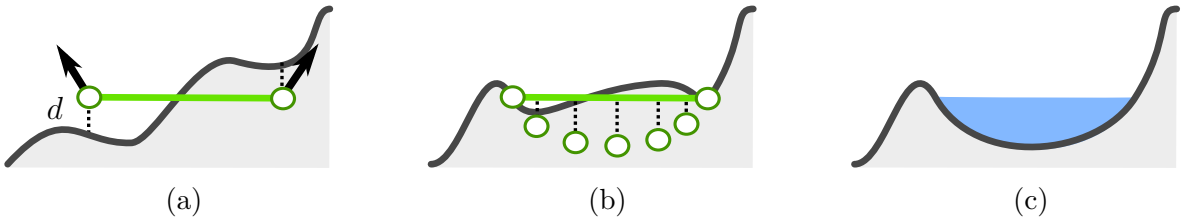
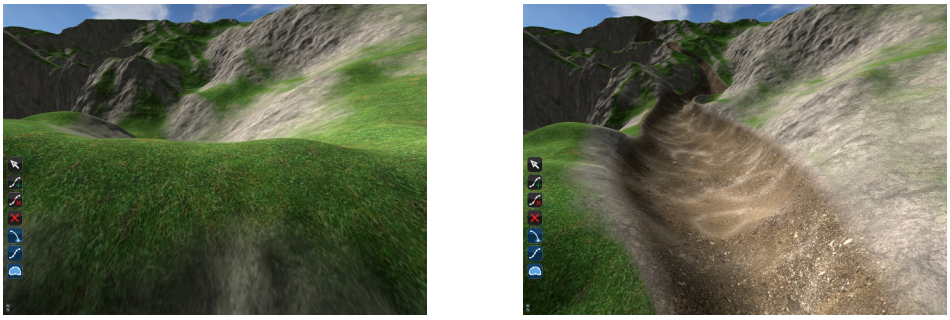


Figure 4.13 – The initial terrain (a) is deformed for satisfying boundary constraints (b) forming the riverbanks, as well as profile constraints (c) forming the riverbed. See also Figure 4.14

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 (2001). Along the border at the top of a free-fall, we set a displacement constraint λu, where u is the free-fall direction and λ a constant defined by the user. In addition, we set a displacement constraint $-\lambda u$ along the border of the receiving pool, under the free-fall (see Figure 4.15). As shown in Figure 4.16, these two constraints generate a flipped "S" curve. The top of the overhang is extended in the direction of the flow, and the bottom of the receiving pool extending into the cliff, simulating erosion and falling rocks.



(a) Original terrain (b) Carved river bed

Figure 4.14 – The river bed is created by deforming the terrain, changing the texture, and adding animated fake caustics.

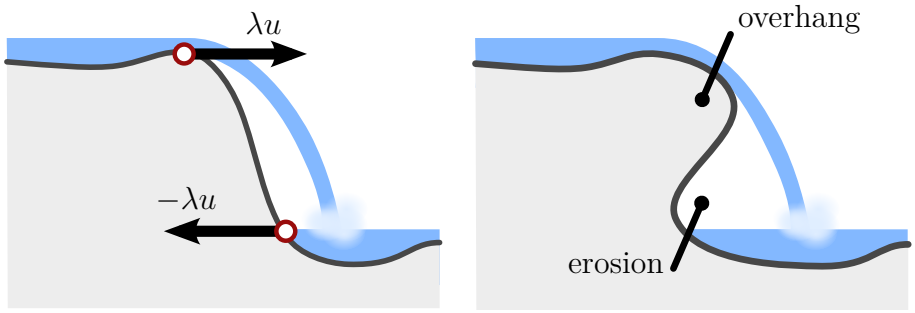
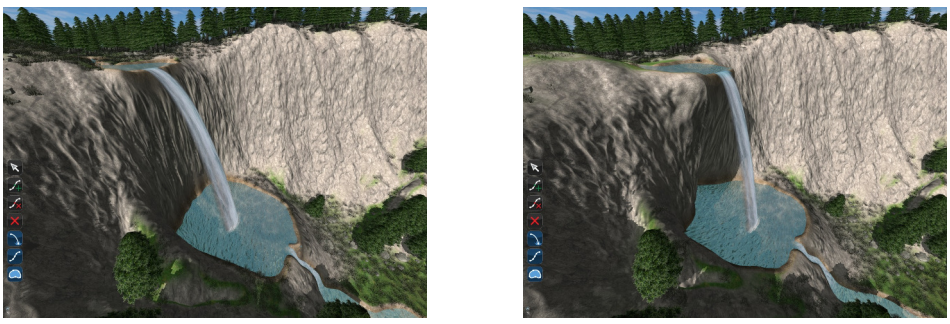


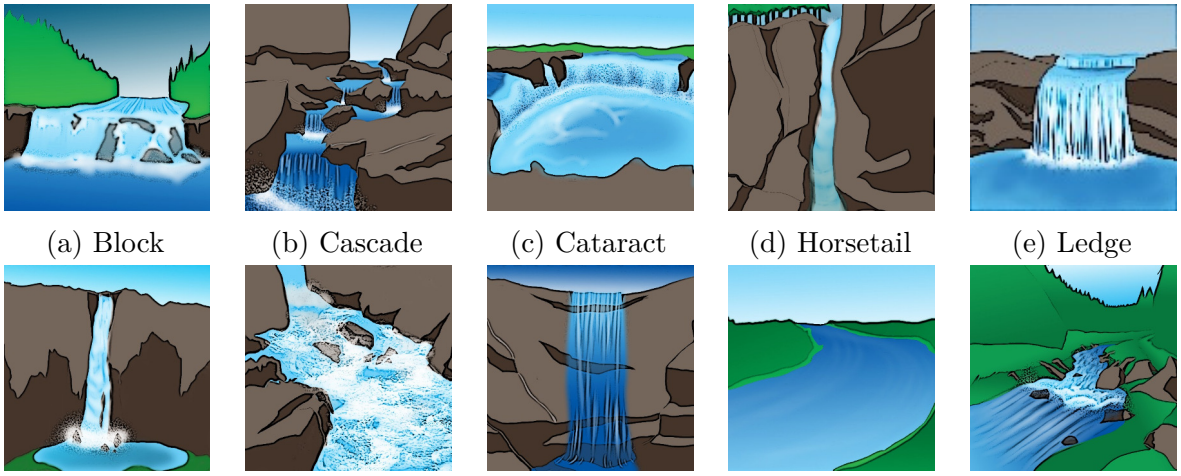
Figure 4.15 – Overhangs are a visually important feature of free falls due to erosion. We model this phenomenon with a horizontal displacement field proportional to the water speed at the top of the fall and to its opposite at the bottom. See figure 4.16.



(a) (b)

Figure 4.16 – Resulting waterfall without (a) and with (b) procedural overhang.

4.2.3 Main contribution: waterfall classification



(a) Block (b) Cascade (c) Cataract (d) Horsetail (e) Ledge

(f) Plunge (g) Rapid (h) Ribbon (i) River (j) Stream

Figure 4.17 – Artist view of the ten types of visually distinguishable waterfalls we use in our classification. © Estelle Charleroy.

As we saw, a waterfall type κ_i is a fine-scale semantic information concerning waterfall segments $\mathcal{S}_i$. That type is essential in the creation of the waterfall segment representation, since it will be used in the parameter map generation step, which in turn dictates the final appearance of the waterfall. Distinctive waterfall segment appearances help to identify the waterfall state and create more impressive scenes.

Several classifications of waterfalls have been proposed by hydro-geologists and/or artists, such as the one depicted in Figure 4.17. There are two main classification systems: volume based , and shape based. The volume based classifications such as the one presented by Beisel (2006) sorts the waterfall in classes using their water flow as only descriptor. However, very different-looking waterfalls can have similar flows; This proves that it does not suffice for visually discriminating waterfalls. On the other hand, the geometry-based classifications such as the ones presented by Danielsson and Danielsson (2006) or Plumb (2005) only provide visual classes, without any quantitative way of determining the type of a precise waterfall part.

Our approach consists in combining those two classification schemes in order to get the visual distinctiveness of shape-based classifications and the deterministic procedure of volume-based methods. More precisely, the volume and other quantitative parameters are used as *inputs* to describe the type of a waterfall. In turn, each type is associated to appearance parameters that we are able to use as *outputs* for creating the waterfall representation.

On a coarse level, waterfalls parts can be separated into three categories: those flowing in contact with the terrain, those free-falling in a reception pool, and the reception pool itself. These categories correspond to the controller types α_i used for creating the controller network.

On a finer level, we need to distinguish sub-categories inside each coarse category. By studying the existing classifications and looking at many real cases, we noticed that the geometrical type of a waterfall mostly depends on two important parameters: the intensity of the water flow, and the slope of the terrain. Indeed, by reducing the flow intensity, a

		Input		Output			
	Type	**Slope**	**Flow**	**Foam**	**Rocks**	**Waves**	**Particles**
Free-fall	**Ribbon**	90°	10 %	0	0.3	0	0
	Plunge	90°	20 %	1	0.4	0	0.2
	Ledge	90°	50 %	0	0.5	0	0.5
	Cataract	90°	80 %	1	1	1	1
Contact	**Stream**	10°	20 %	0	0.2	0	0
	River	5°	50 %	0	0.2	0.1	0
	Rapid	10°	50 %	0.2	0.5	0.5	0
	Horsetail	50°	20 %	1	0	0	0
	Cascade	30°	30 %	0.5	1	1	0
	Block	50°	60 %	1	0	0.2	0

Table 4.1 – Properties for each type of waterfall. The coordinates (slope, flow) indicate the location in the classification graph of Figure 4.18. The slope values indicate an average inclination in degrees, and the flow values give more a percentage of a maximal flow. Foam, rocks, disturbance, and particles are parameters in the range $[0, 1]$ used for the procedural generation and for the rendering.

river becomes a stream, a cascade becomes a horsetail, and a cataract becomes a plunge. Increasing the slope also transforms a river into a rapid or a block.

This leads to the classification depicted in the flow/slope diagram of Figure 4.18, where all geometrical types are positioned as seeds of a Voronoï segmentation. Table 4.1 contains the seed coordinates we used in our prototype. These coordinate describe archetypes of each fine-scale waterfall type in terms of slope and flow. They were set by trials and errors to provide interesting behavior during the modeling sessions. The types of water flows on the left belong to contact waterfall-segments, and the ones on the right belong to free-fall waterfall-segments. However in our case, the coarse type is imposed by the user through a choice of controller type: If a waterfall segment falls outside of its category in the slope-flow diagram, the user is notified.

Each waterfall segment is assigned a type depending on its closest archetype neighbor in the slope-flow diagram. This property is used by our geometrical generation algorithm, as seen in the previous section.

4.2.4 Results

The system is implemented in C++, using OpenGL and GLSL Compute Shaders. The computations are performed on computer equiped with 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.

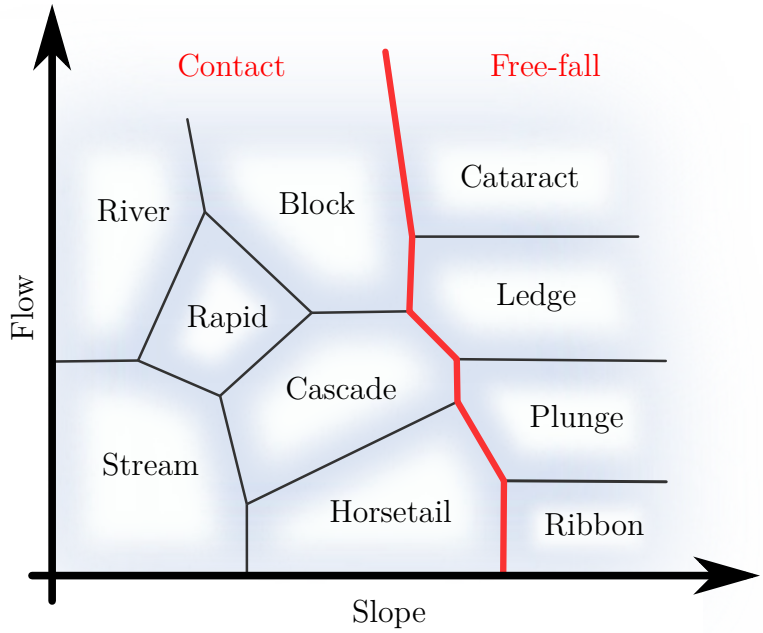


Figure 4.18 – Slope-flow diagram allowing to quantitatively discriminate waterfall types. Types regions are Voronoï cells, which seeds coordinate are given in Table 4.1.

Rendering The waterfalls in our editor are rendered in real time using the integration mesh and the parameter maps computed earlier (Figure 4.12). We use the technique of "tiled directional flow" (see van Hoesel (2011)) 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 4.3 and 4.21 show an overview of our system under editing, with different stages described in caption. The accompanying video (see Section 1.4.4) gives a much better understanding of our system in action, and illustrates several of the features described in the previous sections.

In Figure 4.19, 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 to the original photo, 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 4.20 shows a non-realistic scene created by an artist during a short modeling session. It shows that our system enable its users to create locally realistic scenes while leaving them free of the large-scale design.

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 19, all in under 30 minutes. We asked to reproduce the scene of Figure 4.19a with classical modeling tools, such as Maya (2016). It took them more than two days to reach an equivalent level of detail for both the waterfall and terrain deformation; the longest part being the manual deformation of the heightfield to match the waterfall mesh. Of course, Maya (2016) has not been designed to specifically model waterfalls, but this preliminary experiment shows how specialized tools can be beneficial.

		Computation times (ms)						
Fig.	n	$\mathcal{G}$	$\mathcal{W}$	$\mathcal{M}$	Terrain	Speed	Maps	Proc.
4.19	36	2	2	112	758	258	428	284
4.3	17	1	1	177	677	404	428	297
4.21	29	1	1	77	871	494	428	324
4.20	26	1	2	30	1115	482	428	302

Table 4.2 – Performances of our waterfall editing framework. 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 2048x2048 resolution, speed map generation, details map (foam, disturbance, and rocks) generation, and procedural detail generation.

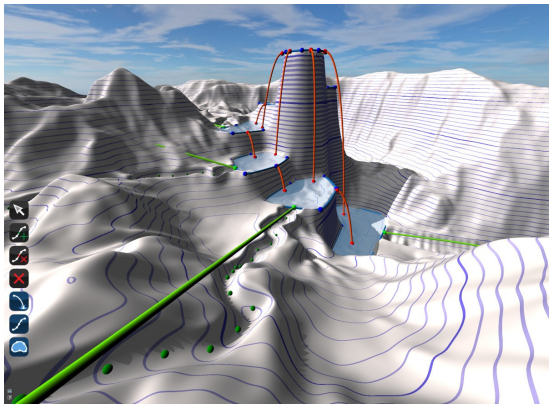
The artists were pleased by the ease of use of our system and its efficiency. However, they expressed a desire for finer control over the result.

Performance Our system generates a complex waterfall network over a terrain in a few seconds (see Table 4.2). The number of elements does not have a huge impact on the computation time. Indeed, most of our pipe-line uses a fixed size grid, so its complexity is independent of 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 varies noticeably.

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

(a) Initial goal (b) Artist result (c) Controller network

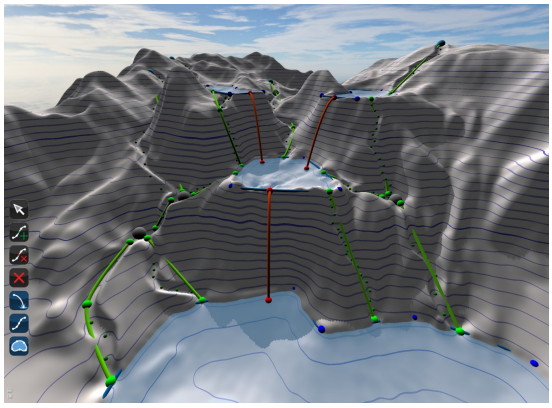
Figure 4.19 – The initial goal presented in Section 4.2.1 has been achieved by an artist using our tool in 10 minutes.



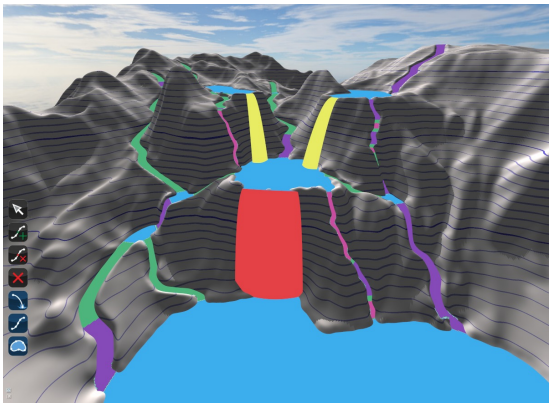
(a) Controller network

(b) Interactive rendering

Figure 4.20 – Our interactive framework also allows the user to design non-realistic sceneries with realistic details.



(a) Controller network



(b) Waterfall types

(c) Carved terrain

(d) Final result

Figure 4.21 – Switching easily between the layers of the system allows users to fully understand the tool they use.

4.2.5 Discussion

One limitation of our method comes from the trajectory subdivision algorithm. The recursive nature of this algorithm limits the adaptation of paths in complex cases: A locally good node position at a given step might be a bad global choice. 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 completely. A solution inspired by algorithms for procedural roads such as Galin et al. (2010), should be applicable to procedural river trajectories. Another possibility would be to allow user specified nodes to be moved automatically.

Another limitation comes from the terrain deformation technique. The terrain is only adapted locally, and therefore it does not preserve any global hydraulic or geologic properties. Indeed, a waterfall network can be created at an undesirable location on the terrain, failing to respect natural river paths shaped by the terrain slope. While this may lead to non plausible 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 enable us to create overhangs (see Gamito and Musgrave (2001)). With support for stack-based terrains as it is done by Peytavie et al. (2009a), our system could handle more complex terrain elements.

Our method is limited in space: Our terrain deformation method relies on a grid-based representation, which limits its application to relatively small terrains. In our examples, we used a 2048x2048 grid with a resolution of 10 pixels per meter (= 0.254 dpi). As a result, the terrain is limited to approximately 200x200 square meters due to GPU cache size. Adapting our terrain representation to the structure proposed by Génevaux et al. (2015) would be a solution.

Finally, our method is also limited in time. It relies on the assumption that waterfalls are stationary. This hypothesis only stands for short periods (typically some weeks or month). Waterfall networks evolve on a larger temporal scale: sources run dry or overflow, meanders increase or disappear, overhangs collapse. At this scale, waterfalls are a transient phenomenon. Our model could be adapted for allowing such variations of flow and trajectories. The next section describes a method able to handle such transient phenomenon through a particular preservative structure.

4.3 Transient Animation Modeling: Fluid Sculpting

The work presented in this section results from a collaboration with Pierre-Luc Manteau (Université Grenoble Alpes/Inria), Chris Wojtan (IST Austria), as well as my advisors Marie-Paule Cani and Damien Rohmer. The resulting paper was presented at the MIG conference in 2016 (`https://mig2016.inria.fr/`) and published in the conference proceedings:

Manteau, P.-L., Vimont, U., Rohmer, D., Cani, M.-P., and Wojtan, C. (2016). Space-time sculpting of liquid animation. *To appear in the proceedings of Motion In Games 2016*

The remaining of this section is organized as follows: Section 4.3.1 recalls the challenges of modeling transient liquid animations; Section 4.3.2 presents space-time features as a preservative structure suited to this problem; Section 4.3.3 explains how space-time features are computed from a raw mesh sequence; Section 4.3.4 deals with the *space-time features* representation; Section 4.3.5 details the tools we offer for manipulating *space-time features*; Section 4.3.6 shows results obtained with our method; Section 4.3.7 draws the limits of our approach and discuss its perspectives. A review of previous work tackling fluid animation modeling can be found in Section 2.4.

Note that this work has to main contributors who are both co-first authors of the resulting paper. Sections related to feature computation and representation (4.3.3 and 4.3.3) are my main contributions to this work.

4.3.1 Introduction

Due to advances in fluid simulation methods over the last two decades, liquid animation has become commonplace in 3D animation productions. The animations can be either highly realistic — for example showing plausible fluid dynamics and interactions with obstacles — or they can exhibit a more expressive behavior to convey specific artistic intentions. In both cases, it is essential for the artist to be able to control the simulation in order to achieve their goals.

Generally, the simulation control is achieved through the careful setting of a large number of parameters such as initial conditions, boundary conditions, viscosity, or external forces. Several reasons make the tuning of these parameters especially difficult. First, they only offer indirect control over the animation, which makes them quite non-intuitive. Second, it is usually not possible to have interactive visual feedback when modifying the parameters, due to the high computational cost of liquid simulation. Third, the inherently non-linear nature of fluid behavior makes it difficult to transfer parameter values from a low to a high resolution simulation. In consequence, achieving a desired effect requires a tedious trial-and-error loop, where computation is restarted multiple times from scratch with different parameters. In many cases, this process does not allow tight control over a sequence of waves and splashes with specific magnitudes or shapes and occurring in a specific order.

In this work, we propose a significantly different approach. Instead of controlling a simulation, we propose an interactive sculpting system enabling to edit pre-computed liquid animations. Our system is based on a copy/edit/paste approach: The user can select coherent and visually important space-time parts of a liquid animation, such as waves or droplets, that we call *space-time features*; These space-time features can then be edited in both space and time in order to change their size, orientation, trajectory or speed. Finally, the edited space-time feature can be inserted into any destination animation at a specific position and time set by the user.

To enable the use of arbitrary liquid animations computed using varying simulation techniques, we based our editing framework on generic inputs; our method allows input mesh sequences without point-wise correspondences between frames, and with arbitrary changes of topological genus between two consecutive time steps. Also, we focused on three requirements to make our method useful in realistic cases. First, the selection of

the effect in the original simulation must be as simple and straightforward for the user as possible. Therefore, once space-time features have been computed, the user can select them using a simple click on the surface. Secondly, pasting the selected effect onto the final animation should be handled automatically, with seamless adaptation of the pasted fluid effect to the destination surface. Finally, the pipeline of selection, copy, edit, and paste steps should be computed efficiently in order to enable interactive user feedback.

The key contributions of our work are as follows:

— A semi-automatic method to tag salient regions in a liquid animation.
— An algorithm that extracts coherent *space-time features* from a mesh sequence with tagged vertices.
— A *space-time feature* representation independent from the original animation.
— A set of editing operations that allow the extraction, manipulation, and insertion of *space-time features* into an animation.

The main challenge of this work is to build a preservative structure from a highly non coherent space-time signal. The animations we manipulate are transient, and the very nature of liquids makes it hard to establish correspondences between frames of the animation. A drop can merge into or emerge from another body of water, a wave can vanish or reflect. These dramatic changes of both geometry and topology mean that the objects the user want to manipulate are evanescent; This is typically not the case in rigid or character animation where a manipulable object – e.g. a character limb – is guaranteed to exist over the whole animation. Space-time features are the ubiquitous preservative structure that we use for retrieving local coherence from mesh sequence input data.

4.3.2 Overview

In this work, we focus on editing liquid animations. To be independent from the simulation method, we take as input a sequence of meshes without any correspondences between the mesh vertices from one frame to another. Due to the arbitrary topology of the meshes and to the temporal coherence to be maintained for numerous geometric details, editing each frame with a shape modeling tool would represent a tremendous amount of work. Instead, we propose to build and manipulate a higher level representation of the liquid animation that we call *space-time features*. A space time feature is a sub-part of the animation, i.e. a sequence of sub-parts of the liquid surface. It establishes a connection between successive frames of the animation: it is therefore a preservative structure.

Our editing pipeline generalizes standard sculpting tools such as those presented by Ferley et al. (2000): cut/copy/edit/paste. It is made of three steps which are illustrated in Figure 4.22. The first step extracts space-time features from the animation. As these features represent regions that deform over time, it would be too tedious for a user to define them by hand. We propose a semi-automatic method to detect salient regions in a liquid animation from which space-time features will be automatically computed. The user can then easily select them using picking: a click at a specific location at a given frame in time results in the automatic selection of the associated feature with its full range in space and time. The second step computes representations of the selected space-time features that are independent from the input animation. They enable space-time features to be transferred from one animation to another. Finally, the last step consists of editing the space-time features and pasting them back into an animation.

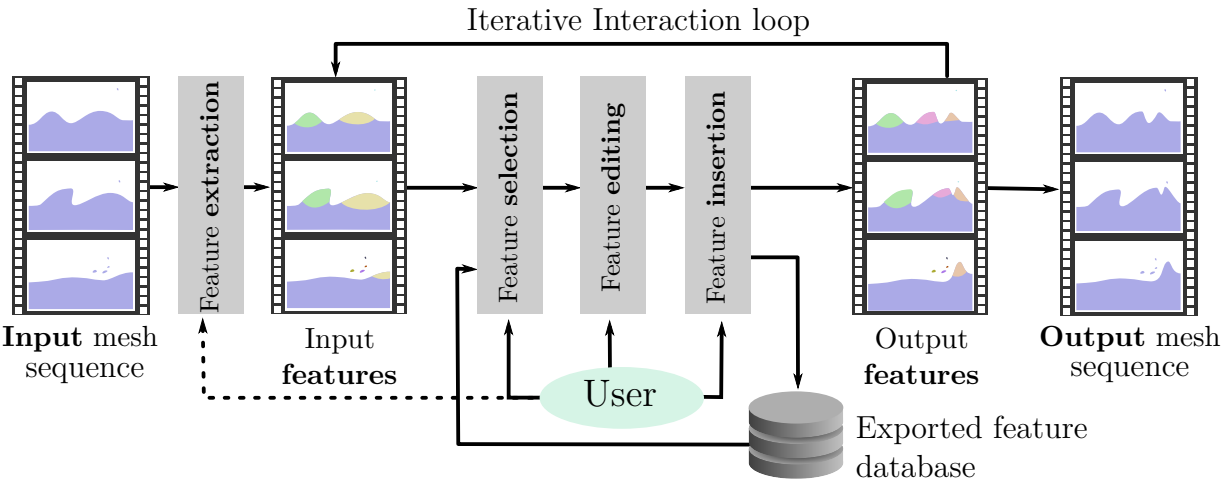


Figure 4.22 – Pipeline of our method: An input fluid animation is given as a mesh sequence. It is pre-processed into a higher-level space-time feature representation. This representation allows the user to iteratively select features from the animation and edit them before inserting them back to the animation. Alternatively, features can be saved and re-imported in this animation or a different one.

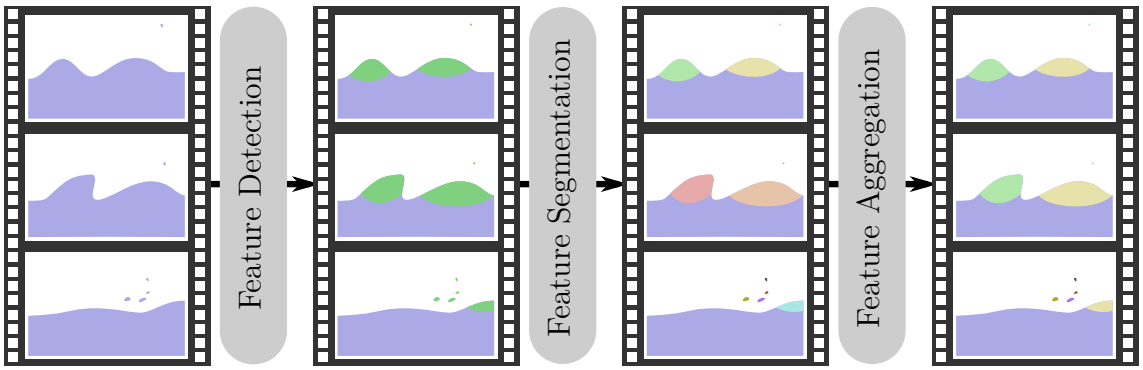


Figure 4.23 – Feature extraction process, from left to right: an initial mesh sequence representing a fluid animation is subjected to a feature detection process, followed by a segmentation step, which results in a frame feature representation. A final aggregation step allows us to build a temporally coherent feature structure.

4.3.3 Feature extraction

In the feature extraction step, our method defines the space-time features that the user would like to manipulate. This process is divided into three steps, as described in Figure 4.23: detection, segmentation, and aggregation. While detection is semi-automatic (it is interleaved with user interaction to define customized regions of interest throughout the animation), segmentation and aggregation are fully automatic.

Notation The input of our method is a mesh sequence over the time steps t that we note $M = (M^t)$, where M^t is a manifold triangular mesh. We note $T(.)$ the temporal length (i.e. the number of frames) and $L(.)$ the characteristic spatial length of any space-time sequence (mesh sequence or feature). Given a triangular mesh X, we call N_X the set of its vertices and P_X the set of its faces. A vertex can carry attributes. We note $A(n, X)$ the value of the attribute A at the vertex n of the mesh X. In the following, we will note

$pos(n, X)$, $norm(n, X)$, and $curv(n, X)$ for positions, normal, and curvature respectively. $\Delta_A(n, X)$ designates the Laplace-Beltrami operator applied to the attribute A at vertex n of the mesh X. A comprehensive list of notations can be found in Table 4.3.

4.3.3.1 Mesh part manipulation

Most of the operations we propose for manipulating mesh sequence rely on a particular structure that we call mesh part. Intuitively, it represents a localized region on a triangular mesh. Space-time features are represented as sequences of such mesh parts. The remaining of this section defines more rigorously this structure as well as the operations it can undergo.

We define R a part of mesh $X = (N_X, P_X)$ as a subset of its vertices: $R \subset N_X$.

The border of R, noted ∂R, is defined by:

$$\partial R = \{n \in R | \exists n_i \in neib(n), n_i \notin R\} \tag{4.10}$$

where $neib(n)$ is the set of neighbors of n in X:

$$neib(n) = \{n' \in N_X | \exists p \in P_X, n \in p \cup n' \in p\} \tag{4.11}$$

Mathematical morphology operators. R being itself a set, usual set operations can be performed on it such as union, intersection and difference. More geometric operations can also be defined: R can be *eroded* into $ero(R)$ using the following relation:

$$ero(R) = \{n \in R | \nexists n_i \in neib(n), n_i \notin R\} \tag{4.12}$$

which is equivalent to $ero(R) = R \setminus \partial R$.

Reciprocally, R can be *dilated* into $dil(R)$ using the following relation:

$$dil(R) = \{n \in N_X | \exists n_i \in neib(n), n_i \in R\} \tag{4.13}$$

Dilation can be extended to an arbitrary order k:

$$dil^k(R) = \underbrace{dil(\dots dil(}_{k \text{ terms}} R) \dots) \tag{4.14}$$

The same stands for erosion:

$$ero^k(R) = \underbrace{ero(\dots ero(}_{k \text{ terms}} R) \dots) \tag{4.15}$$

We call *opening* of order k the mesh part defined as:

$$ope^k(R) = dil^k(ero^k(R)) \tag{4.16}$$

and *closure* of order k the mesh part defined as:

$$clo^k(R) = ero^k(dil^k(R)) \tag{4.17}$$

These operations will be used in the detection phase. For a detail overview of mesh part mathematical morphology, we refer the reader to the work of Serra (1986).

4.3.3.2 Detection

The detection phase aims at defining a sequence of regions of interest $R = (R^t)$ on M. A region R^t is represented as a mesh part.

To let the user easily and intuitively define R, we propose a semi-automatic tool. This tool is based on two key components that we describe in detail below. Combined together they let the user define R in a coarse-to-fine manner: First, curvature analysis is used to automatically detect salient features at each frame and initialize R. Then, topological filtering allows us to interactively adjust R. We also added a painting tool that allows the user to fine-tune each R^t if needed by locally removing or adding vertices from R by clicking.

Multi-resolution curvature analysis. We chose a curvature criteria to extract features as it is a natural asset for detecting waves and ripples in liquid animations. Moreover, the intimate relationship between surface curvature and liquid surface dynamics had already led previous work to use curvature as a tool to enrich liquid simulations, for example with splashes (see Takahashi et al. (2003)), foam (see Ihmsen et al. (2012)), textures (see Narain et al. (2007)), and fine-scale turbulences (see Mercier et al. (2015)).

Curvature is computed at each vertex n of the animation meshes M^t using the following formula:

$$curv(n, M^t) = norm(n, M^t) \cdot \Delta_{pos}(n, M^t) \tag{4.18}$$

Vertices are colored with respect to their curvature magnitude, enabling the user to interactively observe the curved regions and their deformations on the fluid surface while playing the animation (see Figure 4.24a). Then we provide two sliders that the user can interactively tune to filter the curvature and select meaningful regions. These sliders represent:

— A number of iterations β of Laplacian diffusion on the curvature values. We define the i-th iteration of the Laplacian curvature diffusion as:

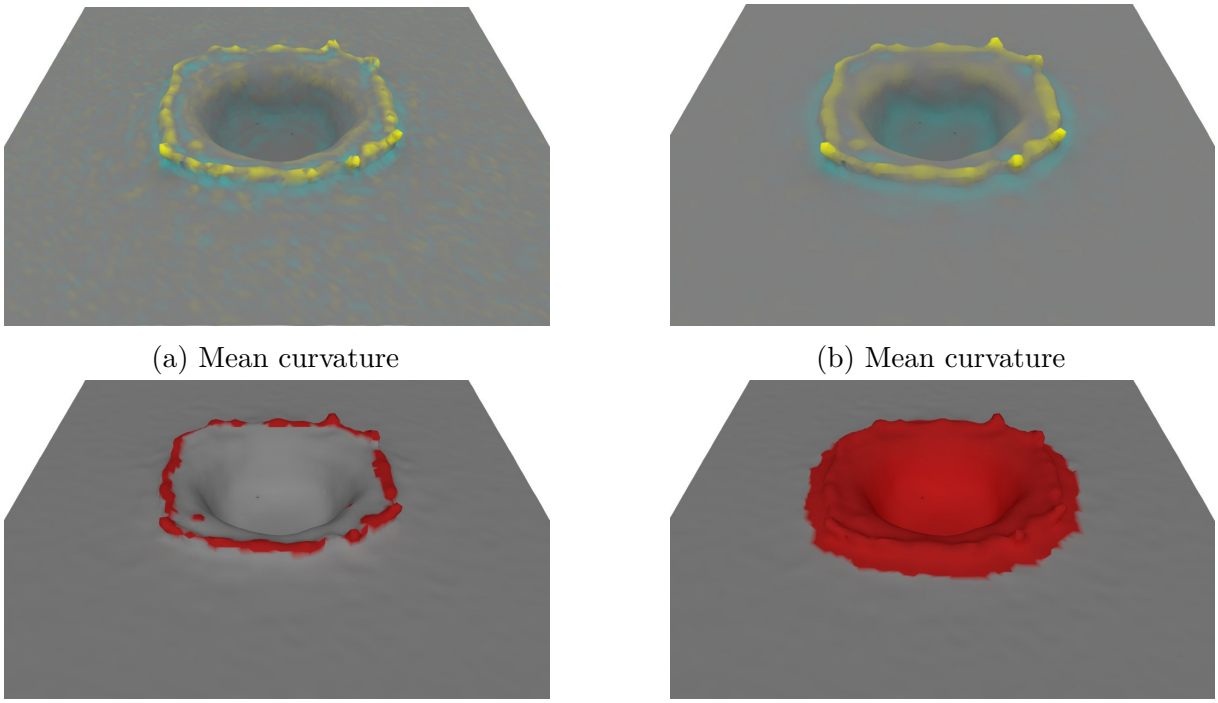
$$curv^{i+1}(n, M^t) = curv^i(n, M^t) - \lambda.\Delta_{curv^i}(n, M^t) \tag{4.19}$$

with $curv^0(n, M^t) = curv(n, M^t)$ and $i \in [0, \beta]$. In our experiment, we used $\lambda = 1$ as a diffusion factor. Laplacian diffusion of the computed curvature values is used to decrease the spatial frequency of the curvature function over the surface. This allows the user to select broader regions in an efficient way without actually smoothing the geometric details on the mesh (see Figure 4.24b).

— A threshold γ on the curvature of R. All the vertices whose curvature is above γ are added to R. This allows the user to control the extent of R (see Figure 4.24c). In the end, we can mathematically define a region of interest for a frame t as:

$$R^t = \{n \in N_{M^t} | curv^\beta(n, M^t) > \gamma\} \tag{4.20}$$

Topological filtering. In many cases, curvature-based selection is not sufficient to extract meaningful animation features. For instance, in Figure 4.24c, the user might want to select the whole crown splash and not only its contour as it has been done with the curvature analysis tool. To remedy these issues, we use the mathematical morphology operators presented in Section 4.3.3.1. They allow the user to interactively and easily refine the regions of interest detected by the curvature analysis.



(a) Mean curvature (b) Mean curvature

(c) Curvature thresholding (d) Topological closure

Figure 4.24 – Curvature analysis-based feature detection.

We propose two main tools:
— *Erosion* for disconnecting, reducing or removing parts of R; and
— *Dilatation* for connecting and enlarging parts of R.
Both tools can be combined for performing *openings* and *closures* of arbitrary degree on R. In practice, these tools were particularly useful for selecting regions such as the interior part of the circular wave in Figure 4.24d, achieved with a closure.

4.3.3.3 Segmentation

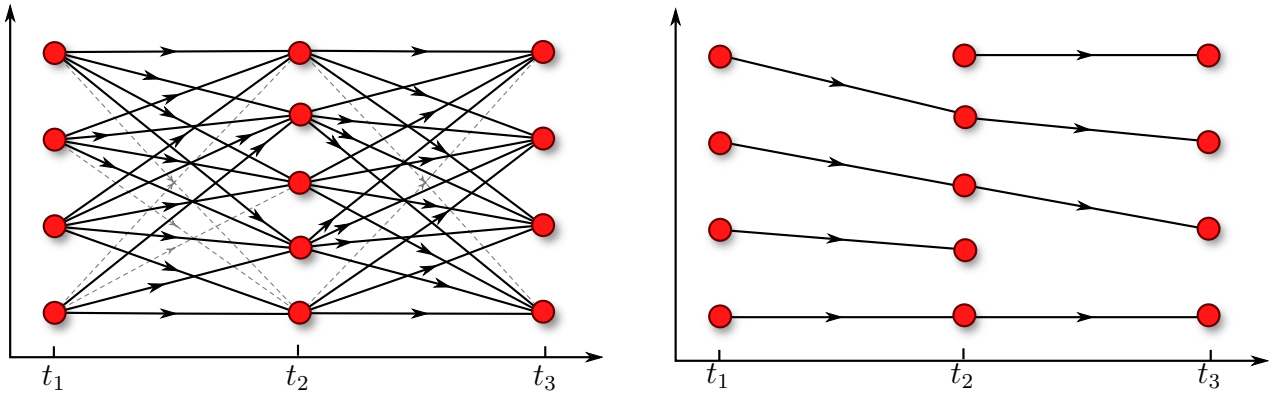
Once R has been computed, the segmentation step decomposes each R^t into connected components $(C_k^t)_{k,t}$, where k is the index of the component. Mathematically, a region of interest for a frame t can be defined as the disjoint union of its connected components:

$$R^t = \bigsqcup_k C_k^t \quad | \quad \forall t \in [0, T(M) - 1] \tag{4.21}$$

The decomposition is computed using the straightforward breadth-first search on each frame in parallel. We call each C_k^t a *frame feature*.

4.3.3.4 Aggregation

Finally, the aggregation step extracts temporally coherent sequences of *frame features* that we call *space-time features*. The process is divided into two steps as illustrated in Figure 4.25. First, we build a graph of all possible *frame feature* connections, and then we compute a vertex-disjoint path cover of that graph. Temporal coherency of the resulting paths is enforced by minimizing a geometric matching cost described below. We call the resulting paths *space-time features*.



(a) Initial DAG (b) Vertex-disjoint path cover

Figure 4.25 – Left: Frame features (red dots) are assembled into a directed acyclic graph as described in Section 4.3.3.4. Each edge of the graph carries a cost computed with Equation (4.22). Edges whose cost is over a user-defined threshold (gray dashed lines) are discarded. Right: a vertex-disjoint minimum-cost path cover has been computed based on Algorithm (3). The extracted paths represent space-time features.

Graph construction. We build a directed acyclic graph $G = (V_G, E_G)$ representing the possible connections between frame features (see figure 4.25a). The set of nodes V_G is made of the frame features $(C_k^t)_{k,t}$ while the set of edges E_G is made of oriented edges e_{ij} linking each pair of consecutive frame features C_i^t and C_j^{t+1}.

Edge cost computation For every edge $e_{ij} \in E_G$, we compute a cost measure ω_{ij}. This measure relates to the geometrical matching between its two endpoints v_i and v_j. We divided ω_{ij} into three terms:

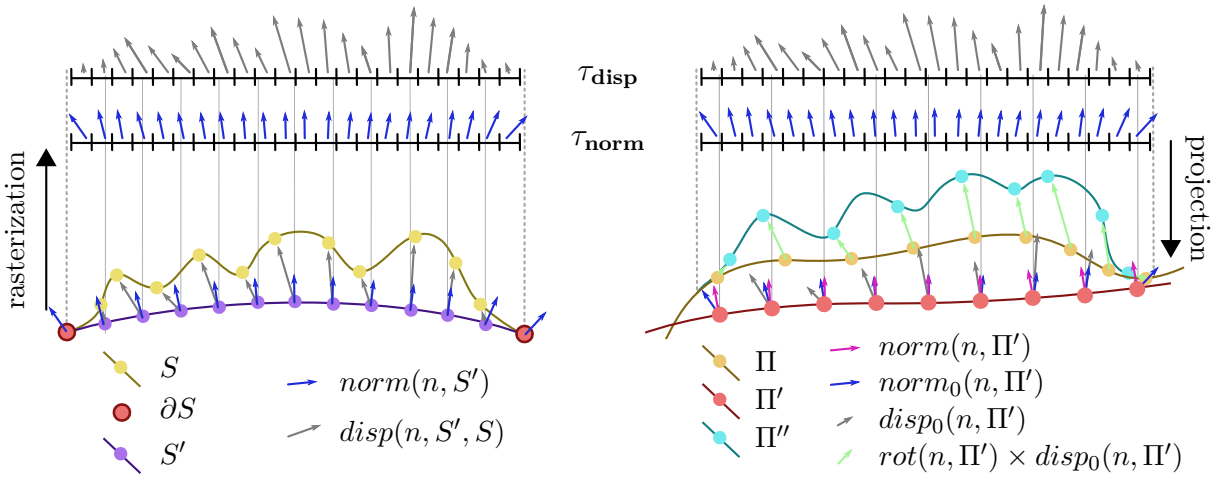
— d_{ij}: The distance between the centers of mass of v_i and v_j.
— s_{ij}: The difference of the surface area between v_i and v_j.
— v_{ij}: The difference of volume between v_i and v_j. v_{ij} is computed only if both v_i and v_j are closed.

The edge cost ω_{ij} is a weighted sum of these terms, normalized by the appropriate power of $l = L(M)$, the characteristic size of the bounding box of M:

$$\omega_{ij} = \omega_d \left(\frac{d_{ij}}{l}\right)^2 + \omega_s \left(\frac{s_{ij}}{l^2}\right)^2 + \omega_v \left(\frac{v_{ij}}{l^3}\right)^2 \tag{4.22}$$

For all the examples of this paper we used $(\omega_d, \omega_s, \omega_v) = (0.6, 0.2, 0.2)$. We chose to favor the closeness between frame features and consider difference of surface and volume equally. After the cost computation, we discard edges whose cost is below a threshold ϵ that we set to $0.3 \times l$ in our examples. Higher thresholds lead to fewer edges in the graph and more disconnected paths.

Vertex-disjoint path cover computation. To the authors' knowledge, there is no standard algorithm for computing minimum weight vertex-disjoint path cover. We propose an algorithm based on Kruskal (1956) algorithm for computing minimum spanning trees: All vertices are first copied from the input graph to the output one; edges of the input graph are considered in ascending order of cost and added to the output graph if they



(a) Copying (b) Pasting

Figure 4.26 – (a): The displacement representation of a mesh part S is built from the sampling of the displacement field transporting S' toward S, and the normal field of S'. (b): This representation can be inserted back into a mesh part Π by projecting a displacement and a normal on vertices of Π'. The difference of normals between S' and Π' is used for orienting the displacements, which are in turn used for generating the deformed surface Π''.

satisfy a given topological condition. In Kruskal's algorithm, the condition is that the edge does not form a cycle in the output graph. In ours, the condition is that both of its endpoint vertices have strictly fewer than two neighbors. This allows us to ensure that the resulting path cover will be vertex-disjoint.

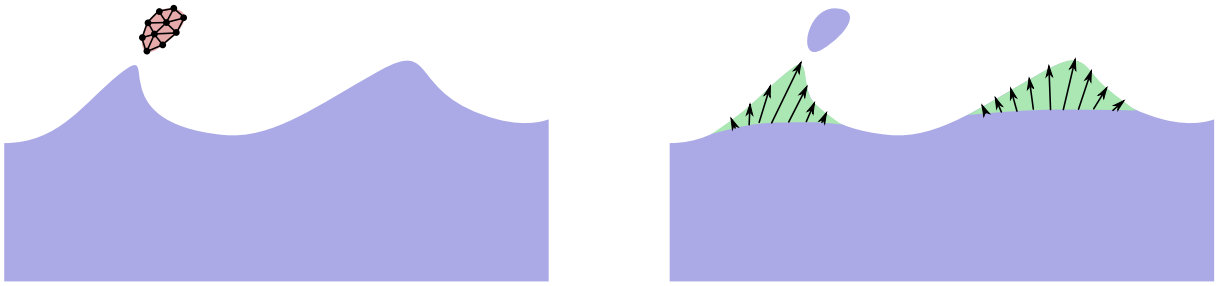
We detail our vertex-disjoint path cover process in Algorithm 3 using the following notation:

— G, V, and E represents respectively a graph, a set of vertices, and a set of edges;
— in and out subscripts refer to input and output elements;
— v_0^e and v_1^e refer to the endpoints of edge e in both G_{in} and G_{out} (since $V_{in} = V_{out}$);
— $deg(v)$ is the degree of vertex v in G_{out};
— $sort(E)$ is the in-place sort of the edges of E in the ascending cost order.

Algorithm 3 Vertex-disjoint path cover computation

$G_{in} = (V_{in}, E_{in})$
$G_{out} = (V_{out}, E_{out})$
$V_{out} = V_{in}$
$E_{out} = \varnothing$
$sort(E_{in})$
for all $e \in E_{in}$ **do**
  if $deg(v_0^e) < 2$ **and** $deg(v_1^e) < 2$ **then**
    $E_{out} \leftarrow e$
  end if
end for

At the end E_{out} represents independent paths, as illustrated in Figure 4.25b, which are optimal in the sense that the algorithm greedily minimizes our edge cost metric. These



(a) Mesh representation (b) Differential representation

Figure 4.27 – Depending on whether the frame feature has closed boundaries or not, it is stored either as a mesh (a) or as a displacement field (b).

paths describe the *space-time features*.

4.3.4 Feature representation

Space-time features can be seen as a simple set of vertices belonging to M. This representation is, however, inconvenient for direct manipulation as it strongly depends on the input animation and therefore cannot be transferred from one animation to another. To be able to copy, edit and paste space-time features in different animations, we propose to build a representation of a space-time feature which is independent from M.

In the remainder of the section, we will note a space-time feature representation $F_i = (F_i^t)_{t^s(F_i) \leq t \leq t^e(F_i)}$ where $t^{s/e}(F_i)$ are the starting/ending frame index of F_i and F_i^t is the frame feature representation of F_i at the frame t. Also, we denote by $S(F_i^t)$ the mesh part of M^t corresponding to F_i^t.

We distinguish two representations depending on whether the frame feature has boundaries or not (see Figure 4.27). In the first case, we use a *mesh representation*, noted $M(F_i^t)$ and composed of a simple 3D mesh. It is used to represent a connected component of the liquid, such as droplet or a larger body of water. In the second case, we use a *differential representation*, noted $(\tau_d(F_i^t), \tau_n(F_i^t))$, and composed of a pair of textures representing a displacement map and a normal map. It is used for frame features representing a local sub-part of a larger body of water, such as a single wave on the surface of an ocean.

A space-time feature can be composed of frame features from both categories. A typical case of mixed representation is an isolated drop falling into a larger body of water and becoming a detail of this larger surface.

In the following of this section, we detail the computation of both representations and how they can be inserted back into a different animation. This will be useful later for copying and pasting features.

4.3.4.1 Computation

Building the mesh representation of a frame feature F_i^t simply consists of transforming $S(F_i^t)$ into an independent mesh $M(F_i^t)$.

Building the differential representation of a frame feature is slightly more complex. The process is described in Figure 4.26a, and consists of three steps: Starting from the initial frame feature surface $S(F_i^t)$ we compute a smooth version $S'(F_i^t)$ using Laplacian

smoothing on the inner part of the surface $S(F_i^t) \setminus \partial S(F_i^t)$. We note $pos(n, S)$ the position of vertex n on surface S; note that $S(F_i^t)$ and $S'(F_i^t)$ describes the same vertices, but with different positions. Then we compute the displacement of each vertex $n \in N$ from $S'(F_i^t)$ to $S(F_i^t)$:

$$disp(n, S'(F_i^t), S(F_i^t)) = pos(n, S(F_i^t)) - pos(n, S'(F_i^t)) \tag{4.23}$$

Finally, we map for every vertex n, $disp(n, S'(F_i^t), S(F_i^t))$ onto $S'(F_i^t)$, and sample the linearly interpolated values into the texture $\tau_d(F_i^t)$. We similarly sample the normals of $S'(F_i^t)$ into $\tau_n(F_i^t)$. The samplings are performed on the GPU using the standard off-screen rasterization pipeline.

4.3.4.2 Insertion

Mesh representations are trivially inserted by copying $M(F_i^{t'})$ into M^t.

To insert a feature representation $(\tau_d(F_i^{t'}), \tau_n(F_i^{t'}))$ into M^t at location p, we first need to identify the part $\Pi \subset M^t$ which will be deformed. Starting from $N_\Pi = \{n_0\}$ where

$$n_0 = \underset{n \in N_{M^t}}{\operatorname{argmin}}(\|pos(n, M^t) - p\|) \tag{4.24}$$

we progressively dilate Π until it fills the bounding box of size $L(F_i^{t'})$ centered in p.

Once Π has been computed, we compute its smooth version Π' on which we project $\tau_d(F_i^{t'})$ and $\tau_n(F_i^{t'})$, yielding two attributes for each vertex $n \in N_{\Pi'}$, a displacement $disp_0(n, \Pi')$ and a normal $norm_0(n, \Pi')$. We define a new vertex attribute for each vertex n as the rotation matrix issued from the two vectors $norm_0(n, \Pi')$ and $norm(n, \Pi')$:

$$rot(n, \Pi') = rot(norm_0(n, \Pi'), norm(n, \Pi'))$$

Each vertex $n \in N_\Pi$ is displaced of $rot(n, \Pi') \times disp_0(n, \Pi')$, yielding the deformed surface Π''. These operations allow to counter the effects of low-resolution shapes of both $S(F_i^t)$ and Π. Figure 4.26b illustrates these steps.

4.3.5 Sculpting Tools

Once space-time features representations have been computed, they can either be manipulated by the user to modify the current liquid animation, or they can be extracted and re-used in another liquid animation to enrich it. This section described the set of tools we propose; they are essentially the space-time analogue of common tools used for sculpting static geometry such as those presented by Ferley et al. (2000), Schmidt and Singh (2010), or Takayama et al. (2011).

Selection The first thing one might want from an interaction system is to specify which of the multiple entities of the scene are to be interacted with. This is usually performed through object selection. In our case, objects are space-time features, and they can be selected and grouped by clicking on their shape at a given frame.

Copy and cut The copy operation consists of creating the representation of the selected features, as explained in Section 4.3.4.1. The cut operation is similar to the copy operation, except that the representation of the feature is removed from the animation after it has

been computed. Once a feature or a feature group has been copied or cut, its representation becomes the current input data of further tools. It is later designated as "the current feature."

Export and import The current feature can be exported into a dedicated binary file format which stores its representation at each frame. This allows it to be imported back later to the same animation, or into a different one. Once imported, a feature becomes the current feature.

Paste The pasting operation allows a user to insert the current feature into a target animation, as explained in Section 4.3.4.2.

Space-time Deformation The user might want to use the feature in a different spatial and temporal configuration from the one in which it was extracted, so we propose adapted deformation tools. The position, orientation, and spatial scale of the current feature can be controlled with the mouse, and a real-time visual feed-back allows the user to set the feature in the configuration they require. By navigating in the animation, the user can also choose the initial frame of the current feature and set a time scale. This leads to a speed-up or slowdown of the feature animation.

Fade in and out When pasting a wave, the user can specify a fade in and a fade out interval. This means that that the feature will not immediately appear, but instead it smoothly grows in the beginning of its lifetime and smoothly disappears before the last frame of its lifetime. We achieve this effect by linearly blending the pasted displacement field over time with weights varying between 0 and 1.

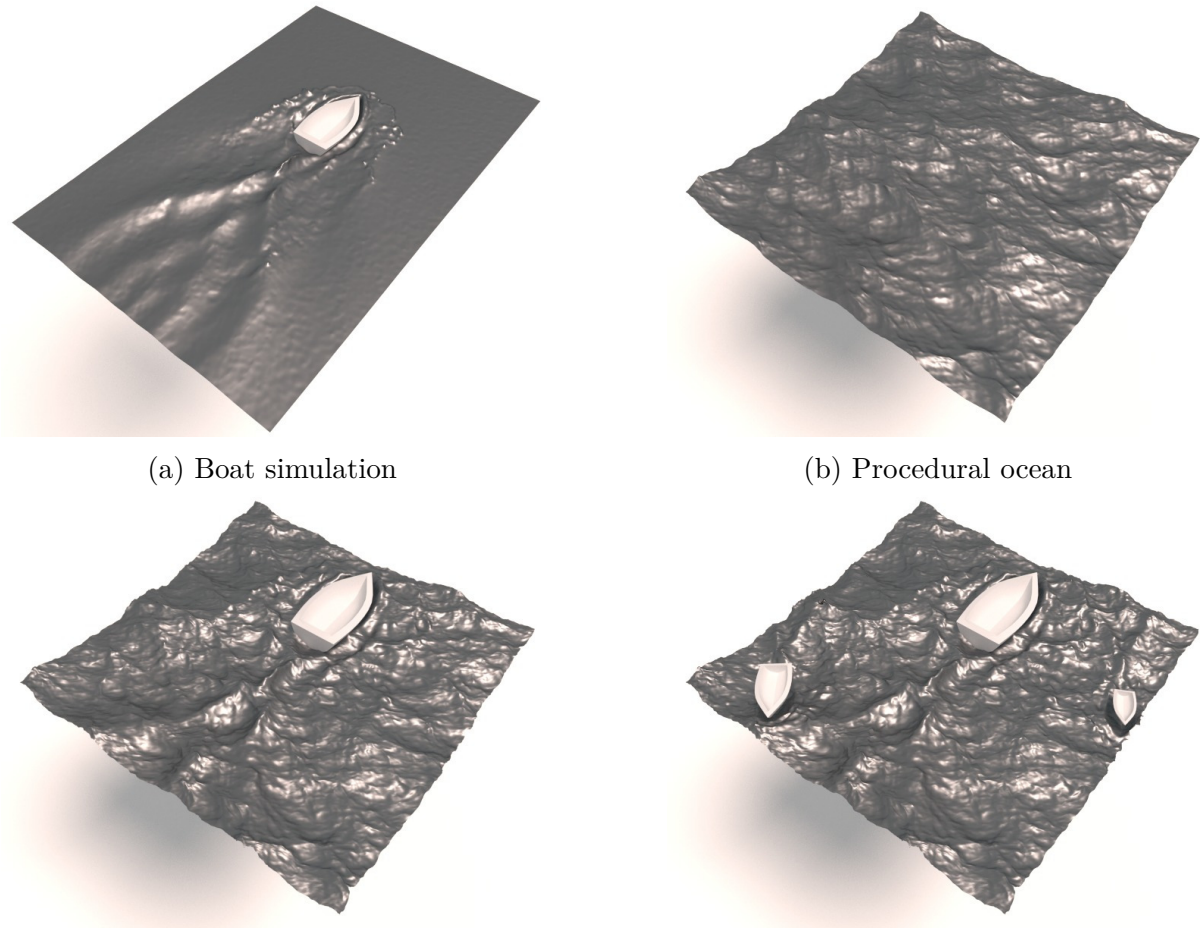
Trajectory editing Space-time deformations influence all frames at once, whereas the user might want to control each frame individually. Per-frame spatial feature manipulation is achieved through a dedicated feature trajectory edit tool. This tool allows a user to displace the representation of a feature at a given frame while visualizing the positions of the feature at all the frames.

4.3.6 Results

In this section, we detail results achieved using our sculpting system. They illustrate the different tools described in the Section 4.3.5 and alternative usage of our method that we found interesting.

Boat wake In Figure 4.28, we illustrate the capability of our method to extract *space-time features* from arbitrary inputs (e.g. Eulerian or Lagrangian simulation, spectral methods, shallow water, real liquid surface acquisition) and combine them to create a plausible animation. We start from two animations: The first one (Figure 4.28a) was computed using the FLIP simulation method of Zhu and Bridson (2005) and represents a boat traversing a fluid tank and forming a wake. The second one (Figure 4.28b) is a procedural animation of ocean computed using the method of Tessendorf (2004) and exhibits numerous small scale details. Then we extract the boat wake and paste it on

the ocean animation at three different positions with different scales and orientations (Figures 4.28c and 4.28d).



(a) Boat simulation

(b) Procedural ocean

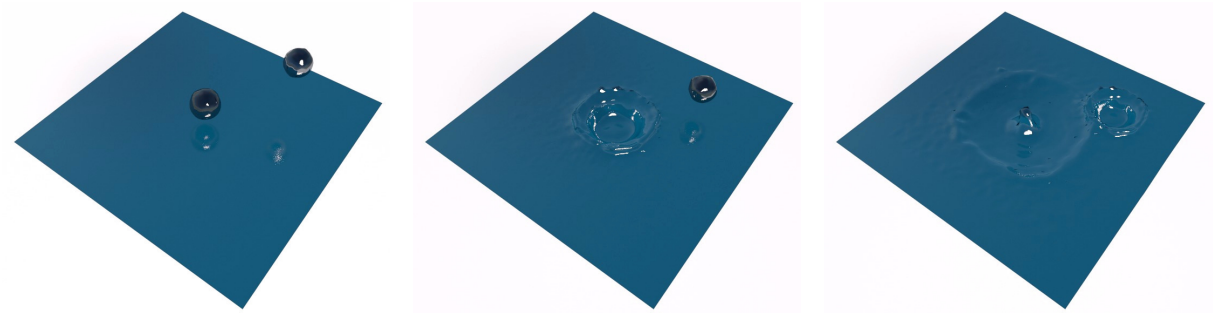
(c) Pasting of the wake

(d) Multiple pasting

Figure 4.28 – From the animation of boat generated using a FLIP simulation (left), our sculpting system allows us to extract the wake of the boat in a single *space-time feature*. Then we can manipulate this feature and paste it into an ocean animation generated procedurally (middle). Editing the feature and pasting it multiple times allows us to interactively model a complex scene (right) without re-simulating.

Trajectory editing In Figure 4.29, we applied several edits to a *space-time feature* capturing a crown splash. First, we temporally remapped the feature to slow it down. Second, we pasted it twice on a static plane at different locations. Third, we edited the trajectory of the droplets to change the height of their fall. Finally, we used a fading out to obtain a smooth transition with the initial plane.

Animation enrichment An interesting aspect of our method is that it can be used to enrich static objects or non-fluid objects with a fluid-like behavior. In Figure 4.30, we enriched a static object with a splash extracted from a liquid animation. More generally, our method allows us to combines results obtained with very different methods such as procedural animation, eulerian and lagrangian simulations, shallow water simulation, or artist-created animations.



(a) $t = t_0$ (b) $t = t_1$ (c) $t = t_2$

Figure 4.29 – From an existing liquid animation we extracted a complete crown splash into a single *space-time feature*. The feature combines both the fall of a drop and the resulting splash. We edit and paste this feature twice at different locations and modify the height of the droplets. Here, we show different frames of the final animation.

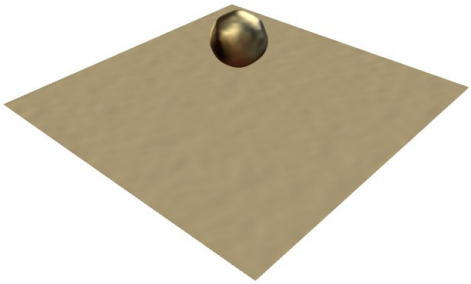
4.3.7 Discussion

Our method is not without limitations, and we suggest several directions for future work.

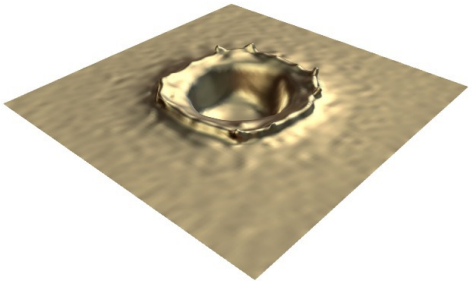
Physical consistency Even though the space-time features selected by the user capture realistic behavior, the way they are edited and inserted may spoil the realism of the resulting animation. As we do not check for physical consistency, the plausibility of the result depends on the user's artistic skill. An extension of our method would be to adapt the destination surface so that it matches the input features under physical constraints such as volume preservation. To incorporate further physical constraints such as momentum conservation, using mesh sequences as input would not be sufficient anymore and additional information such as velocity would be required. Designing an interactive editing method given these constraints may be difficult to achieve.

Resolution issues Geometrical details may be lost when pasting a feature if the resolution of the target mesh sequence is too coarse. To remedy this limitation, we could add an automatic mesh refinement scheme such that the resolution of the target mesh always locally matches the resolution of details in the pasted feature.

Aggregation robustness The aggregation of regions of interest into space-time features is a key component of our approach. However, as it is based on geometrical similarities between two consecutive frames, it might fail if the time step between two frames is too large or if parts of the water body are moving too fast, such as in the case of dynamic splashes with lots of fast moving droplets. Even if it has not been an issue for the results of this paper, we would like to enforce the robustness of the aggregation step by adding a new metric which would measure the physical coherency between two regions of interest. This metric would take into account some inferred velocity for the region. It could also incorporate some cause and effect relationships; for example, a falling drop will cause waves.



(a) Initial animation at $t = t_0$



(b) Initial animation at $t = t_1$

(c) Final animation at $t = t_0$

(d) Final animation at $t = t_1$

Figure 4.30 – From an initial simulation of a falling drop (a, b) space-time features can be extracted and pasted back into an other mesh, generating a new animation (c, d).

Memory consumption For our results, we worked with short sequence of liquid animations but when editing a large sequence of high resolution meshes and extracting potentially large space-time features, memory consumption may become a problem. A classical solution would be to use a multi-resolution approach, as shown by Ponchio and Hormann (2008). The user would manipulate a low resolution version of the animation which would ensure interactivity. Then, the user's edits choice would be transferred to the high resolution version of the animation as an off-line post-process.

Feature editing We proposed basic tools for the space-time edition of features and there are several avenues for future work. Firstly, our copy/paste method is only able to deal with simple deformations of a surface. By using the work of Takahashi et al. (2003) to extract and insert displacement fields, we could handle much more complex cases. Secondly, we would like to propose a space-time sculpting tool close to space deformers such as constant volume tools of Angelidis et al. (2006) or von Funck et al. (2006) and topology modifiers as in Stanculescu et al. (2011). The idea would be to let the user sculpt a specific frame and to interpolate the deformation over time. Finally, we think it would be useful to let any edited parameter (scale, rotation, etc.) to be key-framed in order to make time-varying effects more easily controllable.

4.3.8 Conclusion

This section introduces the first method for interactively editing existing transient fluid animations. Our method is based on an intuitive sculpting metaphor where the user can select, copy, edit and paste coherent *space-time features*. This approach allows a user to quickly design new liquid animations. In the future, we think that our representation for *space-time features* could be extended and used to manipulate animations at a higher level, similarly to a story-board. This kind of preservative structure would require to be built upon semantic knowledge about liquids.

Another interesting extension would be to develop the same notion of feature-based editing on other fluid animation types such as gas. Here, one of the main challenges is to identify the structure for representing features, as well as a good metric for time consistency.

Symbol	Description
$M = (M^t)_{0 \leq t < T(M)}$	Mesh sequence
$T(.)$	Number of frames in argument
$L(.)$	Characteristic length in argument
N_X	Nodes of mesh X
P_X	Polygonal faces of mesh X
$pos(n, X)$	Position of vertex $n \in N_X$
$norm(n, X)$	Normal of vertex $n \in N_X$
$disp(n, X_1, X_2)$	$pos(n, X_2) - pos(n, X_1)$
$F_i = (F_i^t)_{t^s(F_i) \leq t \leq t^e(F_i)}$	i^{th} feature of the animation
$t^{s/e}(F_i)$	Starting/ending frame index of F_i
$S(F_i^t)$	Part of M^t corresponding to F_i^t
X'	Smoothed version of mesh or mesh part X
$M(F_i^t)$	Mesh representation of F_i^t
$(\tau_d(F_i^t), \tau_n(F_i^t))$	Differential representation of F_i^t
$C(X)$	Centre of mass of surface X
$A(X)$	Area of surface X
$V(X)$	Volume of surface X
G	Frame features adjacency graph
$V_G = \{F_j^t\}_{t,j}$	Vertices of graph G
$E_G = \{e_{ij}^t\}_{t,r(i,j)}$	Edges of graph G
ω_{ij}^t	Cost of e_{ij}^t (see Equation 4.22)
d_{ij}^t	$\|C(S(F_i^t)) - C(S(_j^{t+1}))\|$
a_{ij}^t	$\|A(S(F_i^t)) - A(S(_j^{t+1}))\|$
v_{ij}^t	$\|V(S(F_i^t)) - V(S(_j^{t+1}))\|$
ω_d	Edge cost distance weight
ω_a	Edge cost area weight
ω_v	Edge cost volume weight

Table 4.3 – Notations used throughout this article. Subscript (resp. superscript) indices are used for spatial (resp. temporal) indexing. Parenthesis (resp curly braces) are used for ordered (resp unordered) sets.

4.4 Conclusion

This chapter introduced the notions of animation temporality and preservative structure. These two concepts have been illustrated in two situations: The waterfall scenes modeling framework illustrates the static nature of preservative structures in stationary animation; This structure is used for offering dedicated editing tools as well as for formulating constraints on the object consistency. The liquid animation editing method offers to build a preservative structure allowing us to manipulate features of a transient animation.

Animation modeling is an interesting but difficult task. Considering animation temporality offers an original view point on this problem. Stationary animations have been well studied, but some intricate phenomena (of the same order than the waterfall-riverbed coupling) typical of complex objects still raise some challenges. For example, I believe that 3D fire and smoke modeling methods based on preservative structure editing would be worth investigating. On the other hand, transient phenomenon are still an area where interactive editing is far from wide-spread. In this regard, editing preservative structures seem promising.

To go further, it would be beneficial to elaborate a generic multi-scale spatio-temporal analysis framework for decomposing an animation into several constant-temporality components. For example, rivers and waves can be described as stationary or periodic phenomena at some scale and/or in specific referential. The idea would be to build a representation of these phenomena allowing to edit independently each of these components with tools dedicated to their own temporality. For instance, wave frequency and amplitude could be manipulated globally (using periodic animation editing tools), while each wave could be sculpted independently.

Chapter 5

Conclusion

Contents

5.1 Summary of this thesis 120
5.2 Future work 120
5.2.1 Continuous deformations as an animation 120
5.2.2 Deformation diffusion using preservative structures 121
5.2.3 Deformation of preservative structures 121

This, thesis introduced several contributions, some of which solve specific problems (such as the histogram interpolation method or the waterfall classification), and others are more general (such as deformation grammars and preservative structures). This chapters outlines the contributions that were presented, and discusses several axes for future work.

5.1 Summary of this thesis

In this thesis, we proposed solutions to major challenges in static and animated content modeling.

We introduced a framework defining a *complex objects* as a hierarchical structure and showed that the consistency of such objects can be decomposed hierarchically at the element level. This framework was used as a reference for presenting three main contributions regarding 3D modeling: Firstly, we presented a *sub-structure deformation method* for a part-based generation framework synthesizing complex object variations; Secondly, we developed a *distribution descriptor interpolation* algorithm within a color-painting inspired framework for distribution generation and authoring; Finally, we introduced *deformation grammars* as a generic and efficient way to interpret arbitrary deformation on complex objects.

We also introduced the notion of *temporality* of an animation as a way to *classify animations* as stationary or transient. The associated concept of *preservative structure* can be used as an efficient simplification in the case of stationary animations, as shown in a *class*-based waterfall editing framework. Transient animations can also be described by preservative structures, as illustrated in the case of general liquid animations. These structures were defined as the fluid features, and interacting with them enabled interactive editing of general liquid animations.

5.2 Future work

As shown in Chapter 2, static object deformations have been well studied, allowing these to become a new way of creation, while animation deformation is still a novel subject. In my view, many static object deformation methods could benefit from being extended to animation deformation. Some recent work already used static object generation and deformation paradigms for generating controllable animated content (see Milliez et al. (2014) and Hyun et al. (2016) for example).

5.2.1 Continuous deformations as an animation

One of the most obvious ways to use deformation methods for creating animations is to use *continuous* deformation methods. In this setup, the pre-deformation and post-deformation states of the object are considered as successive in time. The in-between states are those created by the deformation method. Time can be associated with the duration of the deformation gesture (i.e. the "real time" of the user) or to the amount of deformation. This last case frees the user from having to perform the deformation at the right speed, but imposes a pre-defined animation speed.

The challenge here is twofold.

First, animation-aware deformation tools or behaviors have to be defined. That is, for example, a deformation behavior for a character or a plant model which interprets resizing as growing. Another example would be the deformation of a forest where new trees grow in new areas. Handling such complex cases could be achieved by using deformation grammars with temporal continuity preserving rules.

Second, usable interaction tools have to be provided to the user for iteratively and efficiently editing their animated creations. One particular challenge when using the "deformation as animation" paradigm is to make the user edit the animated content while it is been played. Alternatively, a static spatio-temporal representation could be found, allowing to apply space-time deformations without synchronization issues.

5.2.2 Deformation diffusion using preservative structures

Another potential future work would be to use preservative structures to transfer static deformation methods to the animation domain: Since these structures encode correspondences between frames, it might be possible to apply some deformation method onto one frame of an animation and to extend the edit to adjacent frames using the preservative structure.

For example, consider a liquid animation on which a set of features has been computed. The users could sculpt a particular frame of the animation using their favorite deformation tool, without worrying about time continuity. The feature set could be used for diffusing the user deformation through time in the animation: This way, a given feature (e.g. a wave) having been deformed at a given frame could be seamlessly deformed in adjacent frames too. This would allow the animation to be continuous despite very time-localized edits. Pan et al. (2013) started to look in that direction with very promising results.

5.2.3 Deformation of preservative structures

Preservative structure as presented in Chapter 4 might also be considered as complex objects in the sense of Chapter 3. In that case, some complex object generation or deformation methods could be used for handling animated content. For instance, let us consider how the three methods presented in Chapter 3 could be used for deforming preservative structures.

The part-based complex object generation method presented in Section 3.2 could be re-formulated as an event-based complex animation generation method. In such a method, we could introduce an event graph, equivalent of the shape graph for animated content, corresponding to a "topological story-board" where semantic constraints would represent simultaneity or causality. This method could be used in the domain of computational story-telling for creating and editing new event chains.

Transferring the concepts of WorldBrush (see Section 3.3) to animation also seems to be worth considering. In this setup, the manipulated objects would be space-time distributions of similar events that could represent phenomena such as rain drops falling. A painting metaphor could certainly be devised, the main challenge being to define an intuitive space-time canvas on which to paint.

Finally, deformation grammars (see Section 3.4) could be used for deforming preservative structures represented as space-time complex object. This would be particularly interesting in the case where the preservative structure is multi-scale due to multiple temporality

components (as discussed in Section 4.3.7), since this structure would certainly be heavy and hard to manipulate. Such structures could be used for representing multi-scale animated phenomenon such as crowd movements, fire, or waves. Creating such a setup would require to define spatio-temporal deformations such as time stretching and compression, along with regular space deformations. In addition to the challenging creation of an efficient user interface for such deformations, another difficulty would be to define the relevant consistency properties for such objects.

Appendix A

Bibliography

Alhashim, I., Li, H., Xu, K., Cao, J., Ma, R., and Zhang, H. (2014). Topology-varying 3D shape creation via structural blending. *ACM TOG, proc. of SIGGRAPH*. 19

Alhashim, I., Zhang, H., and Liu, L. (2012). Detail-replicating shape stretching. *The Visual Computer*. 23

Angelidis, A., Cani, M.-P., Wyvill, G., and King, S. (2006). Swirling-sweepers: Constant-volume modeling. *Graph. Models*. 21, 115

Ashikhmin, M. (2001). Synthesisizing natural textures. *Proc. Symp. on Interactive 3D Graphics (I3D)*. 18

Attene, M., Falcidieno, B., and Spagnuolo, M. (2006). Hierarchical mesh segmentation based on fitting primitives. *The Visual Computer*. 33

Avidan, S. and Shamir, A. (2007). Seam carving for content-aware image resizing. *ACM Trans. Graph. (SIGGRAPH)*. 23

Barbič, J., da Silva, M., and Popović, J. (2009). Deformable object animation using reduced optimal control. *ACM Trans. Graph.* 26

Barbič, J., Sin, F., and Grinspun, E. (2012). Interactive editing of deformable simulations. *ACM Trans. Graph.* 26

Barroso, S., Besuievsky, G., and Patow, G. (2013). Visual copy and paste for procedurally modeled buildings by ruleset rewriting. *Computers and Graphics*. 16

Beardall, M., Farley, M., Ouderkirk, D., Reimschussel, C., Smith, J., Jones, M., and Egbert, P. (2007). Goblins by spheroidal weathering. *Proceedings of the Third Eurographics conference on Natural Phenomena*. 14

Beisel, Jr, R. H. (2006). *International Waterfall Classification System*. 95

Belhadj, F. and Audibert, P. (2005). Modeling landscapes with ridges and rivers: Bottom up approach. *Proceedings - GRAPHITE 2005 - 3rd International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia*. 15

Benes, B., Šťava, O., Měch, R., and Miller, G. (2011). Guided procedural modeling. *Computer Graphics Forum (Eurographics)*. 17

Benes, B., Těšínskỳ, V., Hornyš, J., and Bhatia, S. K. (2006). Hydraulic erosion. *Computer Animation and Virtual Worlds*. 15

Beneš, B., Andrysco, N., and Šťava, O. (2009). Interactive modeling of virtual ecosystems. *Proceedings of the Eurographics Workshop on Natural Phenomena (NPH)*. 15

Beneš, B. and Arriaga, X. (2005). Table mountains by virtual erosion. *Proceedings of the Eurographics Workshop on Natural Phenomena (NPH)*. 15

Beneš, B. and Forsbach, R. (2001). Layered data representation for visual simulation of terrain erosion. *Proceedings of the Spring Conference on Computer Graphics (SCCG)*. 15

Beneš, B. and Forsbach, R. (2002). Visual simulation of hydraulic erosion. *WSCG Proceedings of the International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*. 15

Beneš, B., Massih, M. A., Jarvis, P., Aliaga, D. G., and Vanegas, C. A. (2011). Urban ecosystem design. *Symposium on Interactive 3D Graphics and Games (I3D)*. 15

Bernhardt, A., Maximo, A., Velho, L., Hnaidi, H., and Cani, M.-P. (2011). Real-time terrain modeling using cpu-gpu coupled computation. *XXIV SIBGRAPI*. 18

Bernhardt, A., Pihuit, A., Cani, M.-P., and Barthe, L. (2008). Matisse : Painting 2d regions for modeling free-form shapes. *SBM'08 - Eurographics Workshop on Sketch-Based Interfaces and Modeling*. 17

Bézier, P. (1966). Définition numérique des courbes et surfaces. *Automatisme*. 2

Bhat, K. S., Seitz, S. M., Hodgins, J. K., and Khosla, P. K. (2004). Flow-based video synthesis and editing. *ACM Transactions on Graphics (SIGGRAPH)*. 28, 80

BlenderFoundation (2014). Blender. 4

Bokeloh, M., Wand, M., Koltun, V., and Seidel, H.-P. (2011). Pattern-aware shape deformation using sliding dockers. *ACM Trans. Graph. (SIGGRAPH Asia)*. 23

Bokeloh, M., Wand, M., and Seidel, H.-P. (2010). A connection between partial symmetry and inverse procedural modeling. *ACM Trans. Graph. (SIGGRAPH)*. 17

Bokeloh, M., Wand, M., Seidel, H.-P., and Koltun, V. (2012). An algebraic model for parameterized shape editing. *ACM TOG, proc. of SIGGRAPH*. 23, 35

Bonneel, N., Rabin, J., Peyré, G., and Pfister, H. (2015). Sliced and radon wasserstein barycenters of measures. *Journal of Mathematical Imaging and Vision*. 59

Bonneel, N., van de Panne, M., Paris, S., and Heidrich, W. (2011). Displacement interpolation using lagrangian mass transport. *ACM Transactions on Graphics (SIGGRAPH ASIA 2011)*. 53

Botsch, M. and Kobbelt, L. (2005). Real-time shape editing using radial basis functions. *Computer graphics forum*. 21

Botsch, M. and Sorkine, O. (2008). On linear variational surface deformation methods. *IEEE transactions on visualization and computer graphics*. 21

Boudon, F., Prusinkiewicz, P., Federl, P., Godin, C., and Karwowski, R. (2003). Interactive design of bonsai tree models. *Computer Graphics Forum, Proc. Eurographics*. 15

Bridson, R., Houriham, J., and Nordenstam, M. (2007). Curl-noise for procedural fluid flow. *ACM Transactions on Graphics (TOG)*. 25

Brosz, J., Samavati, F., and Sousa, M. (2006). Terrain synthesis by-example. *GRAPP 2006 - Proceedings of the International Conference on Computer Graphics Theory and Applications (GRAPP)*. 18

Brouet, R., Sheffer, A., Boissieux, L., and Cani, M.-P. (2012). Design preserving garment transfer. *ACM Transactions on Graphics*. 19, 35

Brousset, M., Darles, E., Meneveaux, D., Poulin, P., and Crespin, B. (2016). Simulation and control of breaking waves using an external force model. *Computers & Graphics*. 26

Bruneton, E. and Neyret, F. (2008). Real-time rendering and editing of vector-based terrains. *Computer Graphics Forum (Eurographics)*. 16, 91

Carpenter, A. and Fine, G. (2008). *Plato on knowledge and forms: selected essays*. 6

Chaudhuri, S., Kalogerakis, E., Guibas, L., and Koltun, V. (2011). Probabilistic reasoning for assembly-based 3d modeling. *ACM Transactions on Graphics (TOG)*. 19

Chen, G., Esch, G., Wonka, P., Müller, P., and Zhang, E. (2008a). Interactive procedural street modeling. *ACM Transactions on Graphics (SIGGRAPH)*. 16

Chen, L. and Meng, X. (2009). Anisotropic resizing of model with geometric textures. *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*. 22

Chen, X., Neubert, B., Xu, Y.-Q., Deussen, O., and Kang, S. B. (2008b). Sketch-based tree modeling using markov random field. *ACM Transactions on Graphics (SIGGRAPH Asia)*. 17

Chenney, S. and Forsyth, D. A. (2000). Sampling plausible solutions to multi-body constraint problems. *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. 26

Chiba, N., Muraoka, K., and Fujita, K. (1998). An erosion model based on velocity fields for the visual simulation of mountain scenery. *Journal of Visualization and Computer Animation*. 14

Cohen, J. M., Hughes, J. F., and Zeleznik, R. C. (2000). Harold: A world made of drawings. *Proceedings of the International Symposium on Non-photorealistic Animation and Rendering (NPAR)*. 17

Cook, M. T. and Agah, A. (2009). A survey of sketch-based 3-d modeling techniques. *Interacting with Computers*. 17

Cook, S. A. (1971). The complexity of theorem-proving procedures. *Proceedings of the third annual ACM symposium on Theory of computing*. 39

Cordonnier, G., Braun, J., Cani, M.-P., Benes, B., Galin, E., Peytavie, A., and Guérin, E. (2016). Large scale terrain generation from tectonic uplift and fluvial erosion. *Computer Graphics Forum (Proc. EUROGRAPHICS 2016)*. 15

Danielsson, M. and Danielsson, K. (2006). *Waterfall Lover's Guide Northern California*. 95

de Carpentier, G. J. P. and Bidarra, R. (2009). Interactive GPU-based procedural heightfield brushes. *Proceedings of the International Conference on Foundations of Digital Games (FDG)*. 18

de Reffye, P., Edelin, C., Françon, J., Jaeger, M., and Puech, C. (1988). Plant models faithful to botanical structure and development. *SIGGRAPH Comput. Graph.* 15

Dekkers, E. and Kobbelt, L. (2014). Geometry seam carving. *Computer-Aided Design*. 23

Denning, J. D., Tibaldo, V., and Pellacini, F. (2015). 3dflow: continuous summarization of mesh editing workflows. *ACM Transactions on Graphics (TOG)*. 7

Derzapf, E., Ganster, B., Guthe, M., and Klein, R. (2011). River networks for instant procedural planets. *Computer Graphics Forum (Pacific Graphics)*. 15

Desbenoit, B., Galin, E., and Akkouche, S. (2004). Simulating and modeling lichen growth. *Computer Graphics Forum (Eurographics)*. 15

Deussen, O., Hanrahan, P., Lintermann, B., Měch, R., Pharr, M., and Prusinkiewicz, P. (1998). Realistic modeling and rendering of plant ecosystems. *SIGGRAPH Comput. Graph.* 15

Dicko, A.-H., Liu, T., Gilles, B., Kavan, L., Faure, F., Palombi, O., and Cani, M.-P. (2013). Anatomy transfer. *ACM Transactions on Graphics (TOG)*. 19

Dong, W., Zhou, N., Paul, J.-C., and Zhang, X. (2009). Optimized image resizing using seam carving and scaling. *ACM Transactions on Graphics (SIGGRAPH Asia)*. 23

dos Passos, V. A. and Igarashi, T. (2013). Landsketch: A first person point-of-view example-based terrain modeling approach. *Proceedings of the International Symposium on Sketch-Based Interfaces and Modeling (SBIM)*. 18

Ebert, D. S., Musgrave, F. K., Peachey, D., Perlin, K., and Worley, S. (2002). *Texturing and Modeling: A Procedural Approach*. 13

Emilien, A., Bernhardt, A., Peytavie, A., Cani, M.-P., and Galin, E. (2012). Procedural generation of villages on arbitrary terrains. *The Visual Computer*. 16, 62

Emilien, A., Poulin, P., Cani, M.-P., and Vimont, U. (2014). Interactive procedural modelling of coherent waterfall scenes. *Computer Graphics Forum*, 34(6):22–35. 81

Emilien, A., Vimont, U., Cani, M.-P., Poulin, P., and Benes, B. (2015). Worldbrush: Interactive example-based synthesis of procedural virtual worlds. *ACM Transactions On Graphics (TOG), Proceedings of SIGGRAPH 2015*, 34(4):106. 48, 50, 69, 71

Entem, E., Barthe, L., Cani, M.-P., Cordier, F., and Van de Panne, M. (2015). Modeling 3d animals from a side-view sketch. *Computers & Graphics*. 17

Fattal, R. and Lischinski, D. (2004). Target-driven smoke animation. *ACM Trans. Graph.* 27

Ferley, E., Cani, M.-P., and Gascuel, J.-D. (2000). Practical volumetric sculpting. *Visual Computer*. 20, 102, 110

Ferstl, F., Ando, R., Wojtan, C., Westermann, R., and Thuerey, N. (2016). Narrow band flip for liquid simulations. *Computer Graphics Forum*. 26

Foster, N. and Metaxas, D. (1997). Controlling fluid animation. *Computer Graphics International, 1997. Proceedings*. 26

Funkhouser, T., Kazhdan, M., Shilane, P., Min, P., Kiefer, W., Tal, A., Rusinkiewicz, S., and Dobkin, D. (2004). Modeling by example. *ACM Transactions on Graphics (TOG)*. 19

Gain, J., Marais, P., and Neeser, R. (2014). City sketching. 17

Gain, J., Marais, P., and Strasser, W. (2009). Terrain sketching. *Proc. Symp. on Interactive 3D Graphics and Games (I3D)*. 18

Gain, J., Merry, B., and Marais, P. (2015). Parallel, realistic and controllable terrain synthesis. *Computer Graphics Forum*. 18

Gal, R., Sorkine, O., Mitra, N. J., and Cohen-Or, D. (2009). iwires: an analyze-and-edit approach to shape manipulation. *ACM TOG, proc. of SIGGRAPH*. 23

Gal, R., Sorkine, O., Popa, T., Sheffer, A., and Cohen-Or, D. (2007). 3D collage: Expressive non-realistic modeling. *NPAR: Proceedings of the 5th International Symposium on Non-Photorealistic Animation and Rendering*. 18

Galin, E., Peytavie, A., Guérin, E., and Beneš, B. (2011). Authoring hierarchical road networks. *Computer Graphics Forum (Pacific Graphics)*. 16

Galin, E., Peytavie, A., MarÃ©chal, N., and GuÃ©rin, E. (2010). Procedural generation of roads. *Computer Graphics Forum (Eurographics)*. 16, 100

Gamito, M. N. and Musgrave, F. K. (2001). Procedural landscapes with overhangs. *10th Portuguese Computer Graphics Meeting*. 14, 93, 100

Geiss, R. (2007). Generating complex procedural terrains using the gpu. *GPU Gems*. 14

Génevaux, J.-D., Galin, E., Guérin, E., Peytavie, A., and Beneš, B. (2013). Terrain generation using procedural models based on hydrology. *ACM Trans. Graphics (SIGGRAPH)*. 14, 15

Génevaux, J.-D., Galin, E., Peytavie, A., Guérin, E., Briquet, C., Grosbellet, F., and Benes, B. (2015). Terrain modelling from feature primitives. *Computer Graphics Forum*. 14, 100

Geyer, C. J. and Møller, J. (1994). Simulation procedures and likelihood inference for spatial point processes. *Scandinavian journal of statistics*. 50

Gonzàlez, F., Paradinas, T., Coll, N., and Patow, G. (2013). *cages: A multi-level, multi-cage based system for mesh deformation. *ACM Transactions on Graphics*. 22

Gregson, J., Ihrke, I., Thuerey, N., and Heidrich, W. (2014). From capture to simulation: Connecting forward and inverse problems in fluids. *ACM Trans. Graph.* 28

Guay, M., Cani, M.-P., and Ronfard, R. (2013). The line of action: an intuitive interface for expressive character posing. *ACM Transactions on Graphics*. 25

Guerrero, P., Jeschke, S., Wimmer, M., and Wonka, P. (2014). Edit propagation using geometric relationship functions. *ACM TOG, proc. of SIGGRAPH*. 24

Guerrero, P., Jeschke, S., Wimmer, M., and Wonka, P. (2015). Learning shape placements by example. *Tog, Siggraph*. 19

Hahn, F., Martin, S., Thomaszewski, B., Sumner, R., Coros, S., and Gross, M. (2012). Rig-space physics. *ACM transactions on graphics (TOG)*. 7

Hinsinger, D., Neyret, F., and Cani, M.-P. (2002). Interactive animation of ocean waves. *ACM-SIGGRAPH - EG Symposium on Computer Animation (SCA'02)*. 25

Hnaidi, H., Guérin, É., Akkouche, S., Peytavie, A., and Galin, É. (2010). Feature based terrain generation using diffusion equation. *Computer Graphics Forum (Pacific Graphics)*. 18, 93

Hong, J.-M. and Kim, C.-H. (2004). Controlling fluid animation with geometric potential: Research articles. *Computer Animation and Virtual Worlds*. 27

Horvath, C. and Geiger, W. (2009). Directable, high-resolution simulation of fire on the gpu. *ACM Transactions on Graphics (TOG)*. 28

Horvath, C. J. (2015). Empirical directional wave spectra for computer graphics. *Proceedings of the 2015 Symposium on Digital Production*. 25

Huang, R. and Keyser, J. (2013). Automated sampling and control of gaseous simulations. *The Visual Computer*. 27

Huang, R., Melek, Z., and Keyser, J. (2011). Preview-based sampling for controlling gaseous simulations. *Proceedings of the 2011 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. 27

Huang, Z., Gong, G., and Han, L. (2015). Physically-based smoke simulation for computer graphics: a survey. *Multimedia Tools and Applications*. 26

Hurtut, T. (2010). 2d artistic images analysis, a content-based survey. 17

Hurtut, T., Landes, P.-E., Thollot, J., Gousseau, Y., Drouillhet, R., and Coeurjolly, J.-F. (2009). Appearance-guided synthesis of element arrangements by example. *Proc. Symp. on Non-Photorealistic Animation and Rendering (NPAR)*. 19, 50, 55, 58

Hyun, K., Lee, K., and Lee, J. (2016). Motion grammars for character animation. *Computer Graphics Forum*. 120

Igarashi, T., Matsuoka, S., and Tanaka, H. (1999). Teddy: a sketching interface for 3d freeform design. *ACM TOG , Proc. of Siggraph*. 17

Igarashi, T., Moscovich, T., and Hughes, J. F. (2005). As-rigid-as-possible shape manipulation. *ACM Transactions on Graphics (SIGGRAPH)*. 21

Ihmsen, M., Akinci, N., Akinci, G., and Teschner, M. (2012). Unified spray, foam and air bubbles for particle-based fluids. *Vis. Comput.* 105

Ihmsen, M., Orthmann, J., Solenthaler, B., Kolb, A., and Teschner, M. (2014). Sph fluids in computer graphics. 26

Ijiri, T., r Měch, R., Igarashi, T., and Miller, G. (2008). An example-based procedural system for element arrangement. *Computer Graphics Forum (Eurographics)*. 19

Ilcik, M., Musialski, P., Auzinger, T., and Wimmer, M. (2015). Layer-based procedural design of facades. *Computer Graphics Forum*. 16

Jacobson, A., Deng, Z., Kavan, L., and Lewis, J. (2014). Skinning: real-time shape deformation. *ACM SIGGRAPH*. 22

Jain, A., Thormählen, T., Ritschel, T., and Seidel, H.-P. (2012). Exploring shape variations by 3d-model decomposition and part-based recombination. *Computer Graphics Forum*. 19

Jenny, B., Hutzler, E., and Hurni, L. (2010). Point pattern synthesis. *The Cartographic Journal*. 19

Jeschke, S. and Wojtan, C. (2015). Water wave animation via wavefront parameter interpolation. *ACM Transactions on Graphics (TOG)*. 25

Jordao, K., Pettré, J., Christie, M., and Cani, M.-P. (2014). Crowd sculpting: A space-time sculpting method for populating virtual environments. *CGF, proc. of Eurographics*. 80

Kalogerakis, E., Chaudhuri, S., Koller, D., and Koltun, V. (2012). A probabilistic model for component-based shape synthesis. *ACM Transactions on Graphics (TOG)*. 19

Karpenko, O. A. and Hughes, J. F. (2006). Smoothsketch: 3D free-form shapes from complex sketches. *ACM Transactions on Graphics (SIGGRAPH)*. 17

Kazi, R. H., Igarashi, T., Zhao, S., and Davis, R. (2012). Vignette: Interactive texture design and manipulation with freeform gestures for pen-and-ink illustration. *Proc. SIGCHI Conference on Human Factors in Computing Systems (CHI)*. 17

Kelley, A. D., Malin, M. C., and Nielson, G. M. (1988). Terrain simulation using a model of stream erosion. *SIGGRAPH Comput. Graph.* 14

Kelly, T. and Wonka, P. (2011). Interactive architectural modeling with procedural extrusions. *ACM Transactions on Graphics*. 16

Kho, Y. and Garland, M. (2007). Sketching mesh deformations. *Acm siggraph 2007 courses*. 24

Kilian, M., Mitra, N. J., and Pottmann, H. (2007). Geometric modeling in shape space. *ACM Transactions on Graphics (TOG)*. 6

Kim, T., Tessendorf, J., and Thürey, N. (2013). Closest point turbulence for liquid surfaces. *ACM Transactions on Graphics (TOG)*. 28

Kim, Y., Machiraju, R., and David, T. (2006). Path-based control of smoke simulations. *Proceedings of the 2006 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. 28

Kraevoy, V., Julius, D., and Sheffer, A. (2007). Model composition from interchangeable components. *Computer Graphics and Applications, 2007. PG'07. 15th Pacific Conference on*. 19

Kraevoy, V., Sheffer, A., Shamir, A., and Cohen-Or, D. (2008). Non-homogeneous resizing of complex models. *ACM Transactions on Graphics (SIGGRAPH Asia)*. 21, 23

Krištof, P., Beneš, B., Krivánek, J., and Šťava, O. (2009). Hydraulic erosion using smoothed particle hydrodynamics. *Computer Graphics Forum (Eurographics)*. 15

Kruskal, J. B. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*. 107

Landes, P.-E., Galerne, B., and Hurtut, T. (2013). A shape-aware model for discrete texture synthesis. *Computer Graphics Forum (EGSR)*. 19

Larive, M. and Gaildrat, V. (2006). Wall grammar for building generation. *Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia*. 16

Lars Krecklau, L. K. (2011). Procedural modeling of interconnected structures. *CGF, proc. of Eurographics*. 15

Leblanc, L., Houle, J., and Poulin, P. (2011). Component-based modeling of complete buildings. *Proceedings of Graphics Interface*. 16

Lee, C. H., Varshney, A., and Jacobs, D. W. (2005). Mesh saliency. *ACM transactions on graphics (TOG)*. 23

Lee, U., Yoon, S., Shim, H., Vasseur, P., and Demonceaux, C. (2014). Local path planning in a complex environment for self-driving car. *Cyber Technology in Automation, Control, and Intelligent Systems (CYBER), 2014 IEEE 4th Annual International Conference on*. 3

Li, S., Huang, J., de Goes, F., Jin, X., Bao, H., and Desbrun, M. (2014). Space-time editing of elastic motion through material optimization and reduction. *ACM Trans. Graph*. 26

Liao, Z., Joshi, N., and Hoppe, H. (2013). Automated video looping with progressive dynamism. *ACM Transactions on Graphics (TOG)*. 79

Lindenmayer, A. (1968). Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of Theoretical Biology*. 15

Lipp, M., Scherzer, D., Wonka, P., and Wimmer, M. (2011). Interactive modeling of city layouts using layers of procedural content. *Computer Graphics Forum*. 16

Lipp, M., Wonka, P., and Wimmer, M. (2008). Interactive visual editing of grammars for procedural architecture. *ACM Transactions on Graphics (SIGGRAPH)*. 16

Liu, H., Vimont, U., Wand, M., Cani, M.-P., Hahmann, S., Rohmer, D., and Mitra, N. J. (2015). Replaceable substructures for efficient part-based modeling. *Computer Graphics Forum, Proceedings of Eurographics 2015*, 34(2):503–513. 36

Longay, S., Runions, A., Boudon, F., , and Prusinkiewicz, P. (2012). Treesketch: Interactive procedural modeling of trees on a a tablet. *Symposium on Sketch-Based Interfaces and Modeling (SBIM)*. 17, 71

Ma, C., Huang, H., Sheffer, A., Kalogerakis, E., and Wang, R. (2014). Analogy-driven 3d style transfer. *Eurographics 2014*. 19

Madill, J. and Mould, D. (2013). Target particle control of smoke simulation. *Proceedings of Graphics Interface 2013*. 28

Manescu, P., Azencot, J., Ladjal, H., Beuve, M., and Shariat, B. (2013). Human liver multiphysics modeling for 4d dosimetry during hadrontherapy. *International Symposium on Biomedical Imaging*. 3

Manteau, P.-L., Vimont, U., Rohmer, D., Cani, M.-P., and Wojtan, C. (2016). Space-time sculpting of liquid animation. *To appear in the proceedings of Motion In Games 2016*.

Maya, A. (2016). Autodesk, inc. 4, 97

McCrae, J. and Singh, K. (2009). Sketch-based path design. *Proceedings of Graphics Interface (GI)*. 17

McDonnell, K. T., Qin, H., and Wlodarczyk, R. A. (2001). Virtual clay: a real-time sculpting system with haptic toolkits. *Proceedings of the 2001 symposium on Interactive 3D graphics*. 21

McNamara, A., Treuille, A., Popović, Z., and Stam, J. (2004). Fluid control using the adjoint method. *ACM Trans. Graph.* 27

Mei, X., Decaudin, P. c., and Hu, B.-G. (2007). Fast hydraulic erosion simulation and visualization on GPU. *Proceedings - Pacific Conference on Computer Graphics and Applications*. 15

Mercier, O., Beauchemin, C., Thuerey, N., Kim, T., and Nowrouzezahrai, D. (2015). Surface turbulence for particle-based liquid simulations. *Transactions on Graphics (SIGGRAPH Asia)*. 27, 28, 105

Merrell, P. and Manocha, D. (2011). Model synthesis: A general procedural modeling algorithm. *Visualization and Computer Graphics, IEEE Transactions on*. 15

Merrell, P., Schkufza, E., and Koltun, V. (2010). Computer-generated residential building layouts. *ACM Transactions on Graphics (AIGGRAPH Asia)*. 16

Merrell, P., Schkufza, E., Li, Z., Agrawala, M., and Koltun, V. (2011). Interactive furniture layout using interior design guidelines. *ACM Transactions on Graphics (TOG)*. 16

Meylan, S., Vimont, U., Incerti, S., Clairand, I., and Villagrasa, C. (2016). Geant4-dna simulations using complex dna geometries generated by the dnafabric tool. *Computer Physics Communications*, 204:159–169.

Miao, Y. and Lin, H. (2013). Visual saliency guided global and local resizing for 3d models. *Computer-Aided Design and Computer Graphics (CAD/Graphics)*. 23

Mihalef, V., Metaxas, D., and Sussman, M. (2004). Animation and control of breaking waves. *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*. 27

Milliez, A., Noris, G., Baran, I., Coros, S., Cani, M.-P., Nitti, M., Marra, A., Gross, M., and Sumner, R. W. (2014). Hierarchical motion brushes for animation instancing. *Proc. Workshop on Non-Photorealistic Animation and Rendering (NPAR)*. 120

Milliez, A., Wand, M., Cani, M.-P., and Seidel, H.-P. (2013). Mutable elastic models for sculpting structured shapes. *CGF, proc. of Eurographics*. 23, 80

Mitra, N. J., Wand, M., Zhang, H., Cohen-Or, D., and Bokeloh, M. (2014). Structure-aware shape processing. *Eurographics - State of the Art Reports*. 22, 35, 37

Müller, P., Wonka, P., Haegler, S., Ulmer, A., and Van Gool, L. (2006). Procedural modeling of buildings. *ACM Trans. Graph. (SIGGRAPH)*. 16, 62

Musgrave, F. K., Kolb, C. E., and Mace, R. S. (1989). The synthesis and rendering of eroded fractal terrains. *SIGGRAPH Comput. Graph.* 14

Měch, R. and Prusinkiewicz, P. (1996). Visual models of plants interacting with their environment. *SIGGRAPH Comput. Graph.* 15

Narain, R., Kwatra, V., Lee, H.-P., Kim, T., Carlson, M., and Lin, M. C. (2007). Feature-guided dynamic texture synthesis on continuous flows. *Proceedings of the 18th Eurographics Conference on Rendering Techniques*. 28, 105

Natali, M., Lidal, E. M., Viola, I., and Patel, D. (2013). Modeling terrains and subsurface geology. *Eurographics, State of the Art Report*. 14

Nealen, A., Sorkine, O., Alexa, M., and Cohen-Or, D. (2007). A sketch-based interface for detail-preserving mesh editing. *ACM SIGGRAPH 2007 courses*. 24

Neidhold, B., Wacker, M., and Deussen, O. (2005). Interactive physically based fluid and erosion simulation. *Proceedings of the Eurographics Workshop on Natural Phenomena (NPH)*. 15

Neobarok (2016). Lucian stanculescu. 4

Niantic (2016). Pokémon go. 3

Nielsen, M. B. and Bridson, R. (2011). Guide shapes for high resolution naturalistic liquid simulation. *ACM Trans. Graph.* 27

Nielsen, M. B. and Christensen, B. B. (2010). Improved variational guiding of smoke animations. *Computer Graphics Forum.* 27

Nielsen, M. B., Christensen, B. B., Zafar, N. B., Roble, D., and Museth, K. (2009). Guiding of smoke animations through variational coupling of simulations at different resolutions. *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation.* 27

Nielsen, M. B., Söderström, A., and Bridson, R. (2013). Synthesizing waves from animated height fields. *ACM Transactions on Graphics (TOG).* 25

Nieto, J. R. and Susín, A. (2013). Cage based deformations: a survey. *Deformation models.* 22

Okabe, M., Dobashi, Y., Anjyo, K., and Onai, R. (2015). Fluid volume modeling from sparse multi-view images by appearance transfer. *ACM Trans. Graph.* 28

Olsen, L., Samavati, F. F., Sousa, M. C., and Jorge, J. A. (2009). Sketch-based modeling: A survey. *Computers & Graphics.* 17

Ovsjanikov, M., Li, W., Guibas, L., and Mitra, N. J. (2011). Exploration of continuous variability in collections of 3d shapes. *ACM Transactions on Graphics (TOG).* 24

Owada, S., Nielsen, F., and Igarashi, T. (2006). Copy-paste synthesis of 3d geometry with repetitive patterns. *International Symposium on Smart Graphics.* 23

Öztireli, A. C. and Gross, M. (2012). Analysis and synthesis of point distributions based on pair correlation. *ACM TOG, proc. of SIGGRAPH.* 50

Pan, Z., Huang, J., Tong, Y., Zheng, C., and Bao, H. (2013). Interactive localized liquid motion editing. *ACM Trans. Graph.* 28, 121

Pang, M.-Y. b. and Zhao, R.-B. b. (2009). Algorithm for synthesizing large-scale virtual terrain from images using radially weighted blending. 18

Parish, Y. I. H. and Müller, P. (2001). Procedural modeling of cities. *SIGGRAPH Comput. Graph.* 16

Perlin, K. (1985). An image synthesizer. *SIGGRAPH Comput. Graph.* 13, 25

Peyrat, A., Terraz, O., Merillou, S., and Galin, E. (2008). Generating vast varieties of realistic leaves with parametric 2gmap l-systems. *The Visual Computer.* 15

Peytavie, A., Galin, E., Grosjean, J., and Merillou, S. (2009a). Arches: a framework for modeling complex terrains. *Computer Graphics Forum (Eurographics).* 14, 100

Peytavie, A., Galin, E., Grosjean, J., and Merillou, S. (2009b). Procedural generation of rock piles using aperiodic tiling. *Computer Graphics Forum (Pacific Graphics)*. 14

Pighin, F., Cohen, J. M., and Shah, M. (2004). Modeling and editing flows using advected radial basis functions. *Proceedings of the 2004 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. 28

Pirk, S., Stava, O., Kratt, J., Massih Said, M. A., Neubert, B., Mech, R., Benes, B., and Deussen, O. (2012). Plastic trees: interactive self-adapting botanical tree models. *ACM Transactions on Graphics*. 15

Plumb, G. A. (2005). *Waterfall Lover's Guide Pacific Northwest*. 95

Ponchio, F. and Hormann, K. (2008). Interactive rendering of dynamic geometry. *IEEE Transaction on Visualization and Computer Graphics*. 115

Popović, J., Seitz, S. M., Erdmann, M., Popović, Z., and Witkin, A. (2000). Interactive manipulation of rigid body simulations. *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. 26

Power, J. L., Brush, A. J. B., Prusinkeiwicz, P., and Salesin, D. H. (1999). Interactive arrangement of botanical l-system models. *Symposium on Interactive 3D Graphics*. 15

Prachyabrued, M., Roden, T. E., and Benton, R. G. (2007). Procedural generation of stylized 2d maps. *Proceedings of the International Conference on Advances in Computer Entertainment Technology (ACE)*. 14

Prusinkiewicz, P. and Hammel, M. (1993). Fractal model of mountains with rivers. *Proceedings - Graphics Interface*. 15

Prusinkiewicz, P. and Lindenmayer, A. (2012). *The algorithmic beauty of plants*. 15, 33

Pytel, A. and Mann, S. (2013). Self-organized approach to modeling hydraulic erosion features. *Computers & Graphics*. 15

Raveendran, K., Thuerey, N., Wojtan, C., and Turk, G. (2012). Controlling liquids using meshes. *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. 27

Raveendran, K., Wojtan, C., Thuerey, N., and Turk, G. (2014). Blending liquids. *ACM Trans. Graph.* 29

Read, A. L. (1999). Linear interpolation of histograms. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*. 53

Rengier, F., Mehndiratta, A., von Tengg-Kobligk, H., Zechmann, C. M., Unterhinninghofen, R., Kauczor, H.-U., and Giesel, F. L. (2010). 3d printing based on imaging data: review of medical applications. *International journal of computer assisted radiology and surgery*. 3

Ritchie, D., Mildenhall, B., Goodman, N. D., and Hanrahan, P. (2015). Controlling procedural modeling programs with stochastically-ordered sequential monte carlo. 17, 74

Rohmer, D., Hahmann, S., and Cani, M.-P. (2015). Real-time continuous self-replicating details for shape deformation. *Computers & Graphics*. 23

Roudier, P., Peroche, B., and Perrin, M. (1993). Landscapes synthesis achieved through erosion and deposition process simulation. *Computer Graphics Forum (Eurographics)*. 14

Rusnell, B., Mould, D., and Eramian, M. (2009). Feature-rich distance-based terrain synthesis. *The Visual Computer*. 14

Scharl, J. (2010). A constraint based system to populate procedurally modeled cities with buildings. *Proceedings of CESCG 2010*. 16

Schmid, J., Senn, M. S., Gross, M., and Sumner, R. W. (2011). Overcoat: an implicit canvas for 3d painting. *ACM Transactions on Graphics (TOG)*. 17, 59

Schmidt, R., Khan, A., Singh, K., and Kurtenbach, G. (2009). Analytic drawing of 3d scaffolds. *ACM Transactions on Graphics (TOG)*. 17

Schmidt, R. and Singh, K. (2010). Meshmixer: an interface for rapid mesh composition. *ACM SIGGRAPH 2010 Talks*. 19, 110

Schmidt, R., Wyvill, B., Sousa, M. C., and Jorge, J. A. (2007). Shapeshop: Sketch-based solid modeling with blobtrees. *ACM SIGGRAPH 2007 courses*. 17

Schpok, J., Dwyer, W., and Ebert, D. S. (2005). Modeling and animating gases with simulation features. *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. 28

Schulz, C., von Tycowicz, C., Seidel, H.-P., and Hildebrandt, K. (2014). Animating deformable objects using sparse spacetime constraints. *ACM Trans. Graph.* 26

Schwarz, M. and Müller, P. (2015). Advanced procedural modeling of architecture. 16, 62

Serra, J. (1986). Introduction to mathematical morphology. *Computer vision, graphics, and image processing*. 104

Shi, L. and Yu, Y. (2005a). Controllable smoke animation with guiding objects. *ACM Trans. Graph.* 27

Shi, L. and Yu, Y. (2005b). Taming liquids for rapidly changing targets. *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. 27

Singh, K. and Fiume, E. (1998). Wires: A geometric deformation technique. *SIGGRAPH Comput. Graph.* 24

Smelik, R., Tutenel, T., de Kraker, K., and Bidarra, R. (2011). A declarative approach to procedural modeling of virtual worlds. *Computers & Graphics*. 16

Smelik, R. M., Tutenel, T., Bidarra, R., and Benes, B. (2014). A survey on procedural modelling for virtual worlds. *Computer Graphics Forum*. 14

Sorkine, O. and Alexa, M. (2007). As-rigid-as-possible surface modeling. *Symposium on Geometry Processing*. 21

Sorkine, O., Cohen-Or, D., Lipman, Y., Alexa, M., Rössl, C., and Seidel, H.-P. (2004). Laplacian surface editing. *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*. 21

Stam, J. (1999). Stable fluids. *ACM SIGGRAPH*. 26, 91

Stanculescu, L., Chaine, R., and Cani, M.-P. (2011). Freestyle: Sculpting meshes with self-adaptive topology. *Computers & Graphics*. 21, 70, 115

Stanculescu, L., Chaine, R., Cani, M.-P., and Singh, K. (2013). Sculpting multi-dimensional nested structures. *Computers & Graphics*. 23

Šťava, O., Beneš, B., Brisbin, M., and Křivánek, J. (2008). Interactive terrain modeling using hydraulic erosion. 15

Šťava, O., Benes, B., Měch, R., Aliaga, D. G., and Krištof, P. (2010). Inverse procedural modeling by automatic generation of l-systems. *Computer Graphics Forum (Eurographics)*. 17

Šťava, O., Pirk, S., Kratt, J., Chen, B., Měch, R., Deussen, O., and Benes, B. (2014). Inverse procedural modelling of trees. *Computer Graphics Forum*. 17

Steinberger, M., Kenzel, M., Kainz, B., Mueller, J., Wonka, P., and Schmalstieg, D. (2014a). On-the-fly generation and rendering of infinite cities on the GPU. *Computer Graphics Forum, Proc. Eurographics*. 16

Steinberger, M., Kenzel, M., Kainz, B., Müller, J., Wonka, P., and Schmalstieg, D. (2014b). Parallel generation of architecture on the GPU. *Computer Graphics Forum, Proc. Eurographics*. 16

Sumner, R. W., Zwicker, M., Gotsman, C., and Popović, J. (2005). Mesh-based inverse kinematics. *ACM transactions on graphics (TOG)*. 21

Sun, Q., Zhang, L., Zhang, M., Ying, X., Xin, S.-Q., Xia, J., and He, Y. (2013). Texture brush: An interactive surface texturing interface. *Proc. Symp. on Interactive 3D Graphics and Games (I3D)*. 17

Takahashi, T., Fujii, H., Kunimatsu, A., Hiwada, K., Saito, T., Tanaka, K., and Ueki, H. (2003). Realistic animation of fluid with splash and foam. *Computer Graphics Forum*. 105, 115

Takayama, K., Schmidt, R., Singh, K., Igarashi, T., Boubekeur, T., and Sorkine, O. (2011). Geobrush: Interactive mesh geometry cloning. *Computer Graphics Forum (proceedings of EUROGRAPHICS)*. 19, 110

Talton, J., Yang, L., Kumar, R., Lim, M., Goodman, N., and Měch, R. (2012). Learning design patterns with Bayesian grammar induction. *Proc. ACM Symp. on User Interface Software and Technology (UIST)*. 18

Talton, J. O., Lou, Y., Lesser, S., Duke, J., Měch, R., and Koltun, V. (2011). Metropolis procedural modeling. *ACM TOG, proc. of SIGGRAPH*. 14, 17

Tasse, F., Gain, J., and Marais, P. (2012). Enhanced texture-based terrain synthesis on graphics hardware. *Computer Graphics Forum*. 18

Tasse, F. P., Emilien, A., Cani, M.-P., Hahmann, S., and Dodgson, N. (2014). Feature-based terrain editing from complex sketches. *Computers & Graphics*. 18

Teoh, S. (2009). Riverland: An efficient procedural modeling system for creating realistic-looking terrains. *Advances in Visual Computing*. 15

Tessendorf, J. (2004). Simulating ocean surface. *TOG (Siggraph course notes)*. 25, 111

Thiery, J.-M., Guy, É., and Boubekeur, T. (2013). Sphere-meshes: shape approximation using spherical quadric error metrics. *ACM Transactions on Graphics (TOG)*. 22

Thürey, N. (2016). Interpolations of smoke and liquid simulations. *Transactions on Graphics (to appear)*. 27, 29

Thürey, N., Keiser, R., Pauly, M., and Rüde, U. (2006). Detail-preserving fluid control. *Proceedings of the 2006 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. 28

Tisdale, J., Kim, Z., and Hedrick, J. K. (2009). Autonomous UAV path planning and estimation. *IEEE Robotics & Automation Magazine*. 3

Tolkien, J. R. R. (1955). *The Lord of the Rings*. 55

Treuille, A., McNamara, A., Popović, Z., and Stam, J. (2003). Keyframe control of smoke simulations. *ACM Trans. Graph.* 27

Tutenel, T., Smelik, R., Lopes, R., de Kraker, K., and Bidarra, R. (2011). Generating consistent buildings: A semantic approach for integrating procedural techniques. *Computational Intelligence and AI in Games, IEEE Transactions on*. 16

Twigg, C. D. and James, D. L. (2007). Many-worlds browsing for control of multibody dynamics. *ACM Trans. Graph.* 26

Vaillant, R., Barthe, L., Guennebaud, G., Cani, M.-P., Rohmer, D., Wyvill, B., Gourmel, O., and Paulin, M. (2013). Implicit skinning: real-time skin deformation with contact modeling. *ACM Transactions on Graphics (TOG)*. 22

van Hoesel, F. (2011). Tiled directional flow. 97

Vanegas, C. A., Aliaga, D. G., and Benes, B. (2010a). Building reconstruction using manhattan-world grammars. *Computer Vision and Pattern Recognition, IEEE Conference on*. 17

Vanegas, C. A., Aliaga, D. G., Beneš, B., and Waddell, P. A. (2009). Interactive design of urban spaces using geometrical and behavioral modeling. *ACM Trans. Graph. (SIGGRAPH Asia)*. 16

Vanegas, C. A., Aliaga, D. G., Wonka, P., Müller, P., Waddell, P., and Watson, B. (2010b). Modelling the appearance and behaviour of urban spaces. *Computer Graphics Forum*. 16

Vanegas, C. A., Garcia-Dorado, I., Aliaga, D. G., Benes, B., and Waddell, P. (2012a). Inverse design of urban procedural models. *ACM Trans. Graph. (SIGGRAPH Asia)*. 17

Vanegas, C. A., Kelly, T., Weber, B., Halatsch, J., Aliaga, D. G., and Müller, P. (2012b). Procedural generation of parcels in urban modeling. *Computer Graphics Forum (Eurographics)*. 16

Vanek, J., Benes, B., Herout, A., and Šťava, O. (2011). Large-scale physics-based terrain editing using adaptive tiles on the GPU. *IEEE Computer Graphics and Applications*. 14

Vimont, U., Rohmer, D., and Cani, M.-P. (2016). Deformation grammars: Hierarchical constraint preservation under deformation. *Submitted at Computer Graphics Forum*. 32

von Funck, W., Theisel, H., and Seidel, H.-P. (2006). Vector field based shape deformations. *ACM TOG, proc. of SIGGRAPH*. 21, 115

Wang, H., Liao, M., Zhang, Q., Yang, R., and Turk, G. (2009). Physically guided liquid surface modeling from videos. *ACM Transactions on Graphics (TOG)*. 28

Watanabe, N. and Igarashi, T. (2004). A sketching interface for terrain modeling. *SIGGRAPH Posters*. 18

Wei, L.-Y., Lefebvre, S., Kwatra, V., Turk, G., et al. (2009). State of the art in example-based texture synthesis. *Eurographics, State of the Art Report*. 18

Whiting, E., Ochsendorf, J., and Durand, F. (2009). Procedural modeling of structurally-sound masonry buildings. *ACM TOG, proc. of SIGGRAPH*. 16, 35

Whitney, H. (1932). Congruent graphs and the connectivity of graphs. *American Journal of Mathematics*. 40

Wither, J., Boudon, F., Cani, M.-P., and Godin, C. (2009). Structure from silhouettes: a new paradigm for fast sketch-based design of trees. *Computer Graphics Forum (Eurographics)*. 17

Wither, J., Bouthors, A., and Cani, M.-P. (2008). Rapid sketch modeling of clouds. *Proceedings of the Eurographics Workshop on Sketch-Based Interfaces and Modeling (SBIM)*. 17

Witkin, A. and Kass, M. (1988). Spacetime constraints. *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*. 26

Wojtan, C., Mucha, P. J., and Turk, G. (2006). Keyframe control of complex particle systems using the adjoint method. 26

Wonka, P., Wimmer, M., Sillion, F., and Ribarsky, W. (2003). Instant architecture. *ACM Transactions on Graphics (SIGGRAPH)*. 16

Xu, K., Zhang, H., Cohen-Or, D., and Chen, B. (2012). Fit and diverse: set evolution for inspiring 3d shape galleries. *ACM Transactions on Graphics (TOG)*. 19

Yang, B., Liu, Y., You, L., and Jin, X. (2013). A unified smoke control method based on signed distance field. *Comput. Graph.* 28

Yeh, Y.-T., Yang, L., Watson, M., Goodman, N. D., and Hanrahan, P. (2012). Synthesizing open worlds with constraints using locally annealed reversible jump MCMC. *ACM Trans. Graph. (SIGGRAPH)*. 19

Yu, L. F., Yeung, S. K., Tang, C. K., Terzopoulos, D., Chan, T. F., and Osher, S. J. (2011). Make it home: automatic optimization of furniture arrangement. *ACM Transactions on Graphics (TOG)-Proceedings of ACM SIGGRAPH 2011, v. 30, no. 4, July 2011, article no. 86*. 19

Yuan, Z., Chen, F., and Zhao, Y. (2011). Pattern-guided smoke animation with lagrangian coherent structure. *ACM Trans. Graph.* 27

Yumer, M. E., Chaudhuri, S., Hodgins, J. K., and Kara, L. B. (2015). Semantic shape editing using deformation handles. *ACM TOG, proc. of SIGGRAPH*. 24

ZBrush (2016). Pixologic. 4

Zheng, Y., Cohen-Or, D., and Mitra, N. J. (2013). Smart variations: Functional substructures for part compatibility. *Computer Graphics Forum*. 19

Zheng, Y., Fu, H., Cohen-Or, D., Au, O. K.-C., and Tai, C.-L. (2011). Component-wise controllers for structure-preserving shape manipulation. *CGF, proc. of Eurographics*. 23

Zheng, Y., Liu, H., Dorsey, J., and Mitra, N. J. (2016). Ergonomics-inspired reshaping and exploration of collections of models. *IEEE Transactions on Visualization and Computer Graphics*. 16

Zhou, H., Sun, J., Turk, G., and Rehg, J. (2007). Terrain synthesis from digital elevation models. *IEEE Trans. Visualization and Computer Graphics*. 14, 18

Zhu, B., Iwata, M., Haraguchi, R., Ashihara, T., Umetani, N., Igarashi, T., and Nakazawa, K. (2011). Sketch-based dynamic illustration of fluid systems. *ACM Trans. on Graphics (SIGGRAPH Asia)*. 87

Zhu, Y. and Bridson, R. (2005). Animating sand as a fluid. *ACM Transactions on Graphics (TOG)*. 111

Zimmermann, J., Nealen, A., and Alexa, M. (2007). Silsketch: Automated sketch-based editing of surface meshes. *Proceedings of the Eurographics Workshop on Sketch-based Interfaces and Modeling (SBIM)*. 24

Zimmermann, J., Nealen, A., and Alexa, M. (2008). Sketching contours. *Computers & Graphics*. 21, 24

Appendix B

List of Figures

1.1 Different meshes representing the same shape . . . 6

2.1 Antagonistic aspects of modeling . . . 12
2.2 Shape generation . . . 14
2.3 Shape deformation . . . 21
2.4 Fluid animation control . . . 27

3.1 Example of complex objects . . . 33
3.2 Standard use of grammars to define hierarchical shapes. . . . 34
3.3 Hierarchy of the tree example . . . 34
3.4 Complex object deformation . . . 36
3.5 Part-based modeling: General pipeline . . . 38
3.6 Shape graph . . . 38
3.7 Part-based modeling: Topological pipeline . . . 40
3.8 Dual shap graph node ordering . . . 40
3.9 Matching cuts . . . 40
3.10 Rule table . . . 41
3.11 Boundary frame . . . 42
3.12 Results: Iterated cross-objects replacement . . . 44
3.13 Results: Multiple cross-objects replacement . . . 44
3.14 Results: In-object replacement . . . 45
3.15 Results: Structural variation . . . 45
3.16 Invalid replacements . . . 46
3.17 World element distribution descriptors . . . 49
3.18 Editing tools . . . 52
3.19 *Color* pasting and influence . . . 52
3.20 Pasting *colors* with graphs . . . 53
3.21 Tool: Brush . . . 53
3.22 Tool: Gradient . . . 54
3.23 Tool: stretch . . . 54
3.24 Tool: move . . . 55
3.25 Histogram interpolation: linear versus mass transport . . . 55
3.26 Mass transport histogram interpolation pipeline . . . 56

3.27 Histogram interpolation results . . . 56
3.28 Result: full 3D scene . . . 57
3.29 Result: map of Middle Earth . . . 57
3.30 Failure case . . . 58
3.31 PCF vs Piecewise Strauss correlation functions . . . 59
3.32 2D histogram . . . 60
3.33 Tree deformation . . . 64
3.34 Bilateral grammar tree deformation . . . 65
3.35 Result: Tree . . . 69
3.36 Result: House . . . 69
3.37 Result: Forest . . . 70
3.38 Result: Village . . . 71
3.39 Result: Volumetric distributions . . . 72
3.40 Result: Color transformation . . . 73
3.41 Result: 3D scene . . . 73
3.42 Result: Persistent editing . . . 74

4.1 Goal: Trou de Fer . . . 82
4.2 Overview of the waterfall editing framework . . . 84
4.3 Interactive visualization of each step of our pipe-line . . . 84
4.4 Minimum slope . . . 86
4.5 Branch flow repartition . . . 87
4.6 Waterfall subdivision process . . . 87
4.7 Waterfall coarse-scale geometry . . . 89
4.8 Flow influence visualization . . . 90
4.9 Integration mesh . . . 91
4.10 Procedural details visualization . . . 92
4.11 Fine-scale speed computation . . . 92
4.12 Parameter maps pipeline . . . 93
4.13 Riverbed . . . 93
4.14 Riverbed visualization . . . 94
4.15 Overhang . . . 94
4.16 Overhang visualization . . . 94
4.17 Artist representation of waterfall types . . . 95
4.18 Slope-flow diagram for waterfall classification . . . 97
4.19 Trou de fer model . . . 98
4.20 Artist-created non realistic scene . . . 99
4.21 Visualization layers . . . 99
4.22 Fluid sculpting method overview . . . 103
4.23 Feature extraction . . . 103
4.24 Curvature analysis . . . 106
4.25 Feature aggregation . . . 107
4.26 Feature displacement representation . . . 108
4.27 Representation choice . . . 109
4.28 Boat wake pasting result . . . 112
4.29 Drop pasting result . . . 113
4.30 Bunny result . . . 114

Appendix C

List of Tables

3.1 Notations relative to complex objects. 33
3.2 Notations relative to replaceable sub-structures. 39

4.1 Properties for each type of waterfall 96
4.2 Performances of our waterfall editing framework 98
4.3 FluidSculpting: notations 116

Appendix D

List of Algorithms

1 Matching cut depth first search 41
2 Metropolis-Hasting algorithm for distribution synthesis 51
3 Vertex-disjoint path cover computation 108