Optimization and Control in Procedural Modeling

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Vojtech Krs

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2019

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. Bedřich Beneš, Chair

Computer Graphics Technology, Purdue Polytechnic Institute

Dr. James L. Mohler

Computer Graphics Technology, Purdue Polytechnic Institute

Dr. Tim McGraw

Computer Graphics Technology, Purdue Polytechnic Institute

Dr. Radomír Měch

Adobe Research, Adobe Inc.

Approved by:

Dr. Kathryne A. Newton

Associate Dean for Graduate Programs, Purdue Polytechnic Institute

ACKNOWLEDGMENTS

I would like to thank my dissertation committee members; Dr. Bedřich Beneš for his guidance in research and life in general and his ever-present optimism;
Dr. Radomír Měch for the opportunity to work on interesting problems;
Dr. Tim McGraw for his valuable comments and Dr. James Mohler for his unique perspective.

Furthermore, I would like to thank my colleagues in the High Performance Computer Graphics Lab, Adobe Research, Samsung Research America, and collaborators elsewhere for the opportunity to learn from them and better myself.

Finally, huge thanks to my family and friends for supporting me throughout this long journey, it would impossible without you, děkuji!

TABLE OF CONTENTS

Page

Page

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

Figure Page

Figure Page

Figure Page

Figure Page

ABBREVIATIONS

GPU	Graphics Processing Unit
GAN	Generative Adversarial Networks
CGAN	Conditional Generative Adversarial Networks
PM	Procedural Modeling
CSG	Constructive Solid Geometry
MCMC	Monte Carlo Markov Chain
SOSMC	Stochastically-ordered Sequential Monte Carlo
SPH	Smoothed-particle Hydrodynamics
PICO	Procedural Iterative Constrained Optimizer

ABSTRACT

Krs, Vojtech Ph.D., Purdue University, August 2019. Optimization and Control in Procedural Modeling. Major Professor: Bedrich Benes.

Procedural modeling is a powerful technique used in computer graphics to create geometric models. Instead of manual geometry definition, models are generated implicitly from a set of rules and parameters. Procedural systems have found widespread use in generating content for games, film, and simulation of natural phenomena. Their strength comes from the ability to automatically generate large amount of varied geometry. One of their drawbacks is lack of control because a small change in input parameters often causes large changes in the generated model. In this work we present three novel procedural systems, investigate different forms of control, namely simulation and optimization, and discuss them in terms of general procedural modeling workflow. First we show modeling of 3D objects with arbitrary topology via erosion and deposition simulation controlled by Smoothed Particle Hydrodynamics. Next, we present an algorithm for generating 3D curves using 2D sketches and contextual geometry. Finally, we propose a novel procedural system capable of generating arbitrary type of geometry with respect to user-defined constraints. We show that these systems can be controlled via several means and identify common preconditions that facilitate control: maximizing interactivity and amount of structured information input, minimizing unexpected behaviour, and local control akin to traditional modeling.

CHAPTER 1. INTRODUCTION

The research in computer graphics has yielded impressive results over the past decades. The heart of computer graphics lies in observing the real world and attempting to simulate it visually by using a computer, with the ultimate goal being a simulation indistinguishable from reality. There are many pieces of the puzzle to discover before this goal can be reached, but some are already in our grasp. For instance, we have long known the *rendering equation* (Kajiya, 1986) which describes how light travels through space and interacts with the environment, enabling us to render photorealistic images of virtual worlds. The required amount of computation has been gradually decreasing using various optimizations (Akenine-Moller, Haines, & Hoffman, 2018) and the hardware capabilities to do said computation have been increasing exponentially (R. Smith, 2014). Currently, photorealistic rendering is common in offline rendering (Pixar, 2018) and is currently making its way to real-time applications as well (NVIDIA Corporation, 2018). Similarly, significant advances have been made in the field of data-acquisition. *Imaging* physical objects from the atomic level (Haugstad, 2012), through human bodies (Suetens, 2002) to distant galaxies (Gendler & GaBany, 2015) has become increasingly precise and cost-effective, thanks to the advances in many fields including sensor technologies, computer vision, and artificial intelligence.

There is another piece of the puzzle that has received considerable attention in the computer graphics community: *modeling of virtual worlds*. Modeling is the process of creating a computer representation of physical or virtual objects. An important feature of modeling is the freedom it offers. This freedom enables artists, designers or scientists to model not only existing objects, but to come up with new creative shapes for solving a particular problem, simulating a particular part of a real system, or telling a visual story. Consequently, the applications of modeling span many areas, from physics simulation, through product design, to entertainment, and they affect the everyday lives of most people.

Modeling of 3D objects, however, continues to pose many challenges. The major challenge is how to express user's intent by simple means. Other challenges include creating geometry of quality, that is with sufficient detail for visual fidelity or simulation accuracy, but also of quantity, to richly fill virtual environments.

Traditional modeling by hand is a time-consuming task requiring expertise, not unlike traditional drawing on a paper or clay sculpting. In traditional modeling, the modeler manually and iteratively modifies an object until a desired shape is reached. *Procedural Modeling* (PM) is an alternative approach that generates models algorithmically and is able to produce high quality models in a fraction of time compared to modeling by hand (Ebert, Musgrave, Peachey, Perlin, & Worley, 1998). PM systems have enjoyed a widespread use, particularly in movie and game industries. The procedural approach is not only automatic, but uses a model representation that uses less memory than traditional models. However, one of the biggest disadvantages of PM systems is that they are difficult to control and therefore often require expert knowledge to use. Furthermore, each system is

typically tailored to a specific type of a physical object, such as plants, terrains or urban layouts.

In this dissertation, we explore the issue of control of procedural systems. There is a wide range of different types of procedural systems and their representations. Therefore, instead of focusing on control in general systems, we look at control from three specific angles. First, control via simulation, specifically modeling via erosion and deposition simulation. Second, we focus on fine control in 3D curve modeling. Finally, we present a novel framework able to model variety of objects via optimization with respect to user-defined constraints. The rest of this chapter will detail current modeling methods and their challenges, our proposed solutions, their significance and associated research questions.

1.1 Traditional Methods

In traditional modeling, artists typically start with individual primitives, such as vertices, lines, surfaces or volumes. Through an iterative process of modifying geometry and topology the user gradually improves the model until the desired form is reached. The gradual changes are made by a set of operations, such as applying transformations or sculpting brushes (Figure 1.1). Some of the popular software packages currently in use include Autodesk Maya (Autodesk, 2018b), 3DS Max(Autodesk, 2018a), ZBrush (Pixologic, 2018) and SolidWorks (Dassault Systèmes, 2018).

The operations and tools available to create and modify 3D models depend on the internal representation of the model and on the intended use of the model. There two categories of representations: *solid* and *boundary*.

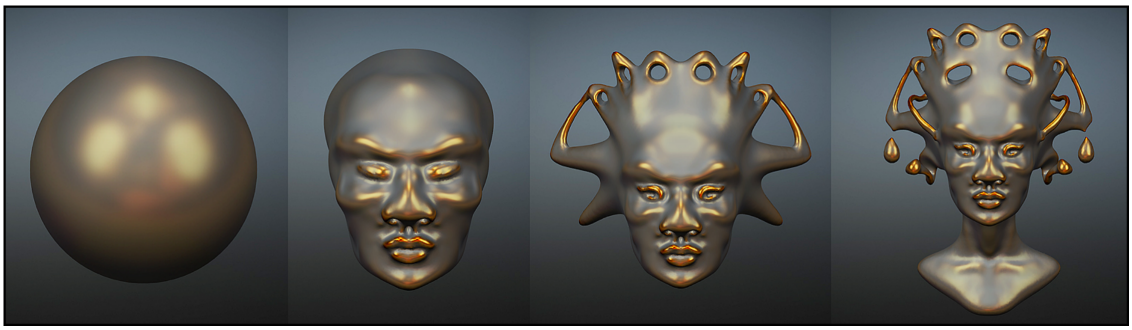


Figure 1.1. Sculpting workflow in the work of Stanculescu et al. (2011)

Solid model representations hold information about what volume in space is occupied, typically with additional properties of density or color, and can therefore provide information about the internal structure of the object. This is important, for instance, in physical simulations or for representing fluids. Typical representations include collection of volumetric primitives, *e.g.,* tetrahedrons, or a space-occupancy data-structure, such as a voxel grid or an octree (Samet, 1990). Constructive Solid Geometry (CSG) is one example of a modeling workflow leveraging solid representation. The modeler places parametrized primitives, for example spheres or cylinders, and combines them into a hierarchy using Boolean operations of union, subtraction and intersection to generate the final model. Solid representations are also sometimes used in sculpting applications because of their native support for topology changes, *e.g.,* for creating or bridging holes in an object.

Boundary representations store information about the surface of the object and are therefore more memory efficient than solid representations, while sacrificing the information about the internal structure. These representations are common for applications where the main focus is on outside appearance or spatial delimitation of the object. Boundary representations are typically used for purely virtual

objects, for example for movies or games, but they can be used to create manufacturable objects for 3D printing as well. The typical boundary representation is a triangular mesh, popular for its simplicity and efficient rasterization (Preparata & Shamos, 1985). Other representations use quadrilaterals or other polygons, curves, or freeform surfaces (Farin, 1996). A typical workflow is polygonal modeling, where the modeler subdivides, inserts and extrudes polygons to add detail while adjusting positions of individual vertices. Similarly, curve or freeform surface modeling consists of manipulating individual curves or surfaces, respectively, by placing control vertices that approximate the surface of the modeled object. Sculpting can be performed on surface representation as well, despite some drawbacks, such as the need for remeshing to preserve surface quality and handling of topology changes.

While being able to customize every atomic part of the model provides the user with great degree of control over the result, modeling complex and detailed scenes with many objects soon becomes infeasible without any automation. Some degree of automation in traditional modeling is often afforded using tools that perform high-level operations on the model, however most of the work still has to be done by hand.

1.2 Procedural Methods

Procedural modeling is a powerful technique used in Computer Graphics to create geometric models or textures. Instead of having the user define every detail explicitly, *e.g.,* individual vertices or pixels, a procedural system generates a model or an image implicitly from a set of rules and parameters. In this work, we focus on

systems generating 3D geometry. These systems have found widespread use in generating content for virtual scenes used in games (Hendrikx, Meijer, Van Der Velden, & Iosup, 2013), film, and recently even in reinforcement learning (Justesen et al., 2018). Procedural systems are particularly useful for modeling natural phenomena, such as terrains, trees, plants (Figure 1.2), or other fractal-like structures. Man-made objects that are frequently generated procedurally include buildings, cities, and road networks (Smelik, Tutenel, Bidarra, & Benes, 2014).

Figure 1.2. Procedural system for interactive plant modeling (Hädrich et al., 2017)

A popular way of representing procedural systems are as formal generative grammars (Chomsky, 1956). The grammars start with some initial symbol, called axiom, and through a series of rewriting rules the axiom is transformed into other symbols. These symbols are themselves, recursively, transformed until a terminal symbol is reached. The symbols in these grammars represent geometry, for example, line segments. The two most common grammars used are L-systems (Lindenmayer, 1968a), operating on strings, and Shape Grammars (Stiny & Gips, 1972), operating on shapes. Other representations of procedural systems include fractals for terrain generation (Mandelbrot, 1982), cellular automata (Von Neumann, Burks, et al., 1966; Wolfram, 2002) for generating game levels (Johnson, Yannakakis, & Togelius, 2010), and noise functions (Perlin, 1985) for modeling terrains, water bodies, or clouds.

Figure 1.3 shows an example of a simple L-system generating a binary fractal tree. The grammar in this example is defined as follows: $\{0, 1, [,]\}$ is a set of possible symbols, with 0 being the axiom. There are two rewriting rules: $1 \rightarrow 11$ and $0 \rightarrow 1[0]0$. The geometric interpretation of individual symbols is defined as follows: $0 \rightarrow$ draw a leaf line segment, $1 \rightarrow$ draw a line segment, $[\rightarrow$ push position and angle to a stack and turn left by 45° degrees, and $] \rightarrow$ pop position and angle from a stack and turn right by 45° degrees. The symbols in a string are interpreted in order and drawn by pen with a variable position and angle. In each iteration shown in the figure, all symbols are rewritten using these two rules in parallel. The figure shows the geometric interpretation of the intermediate strings.

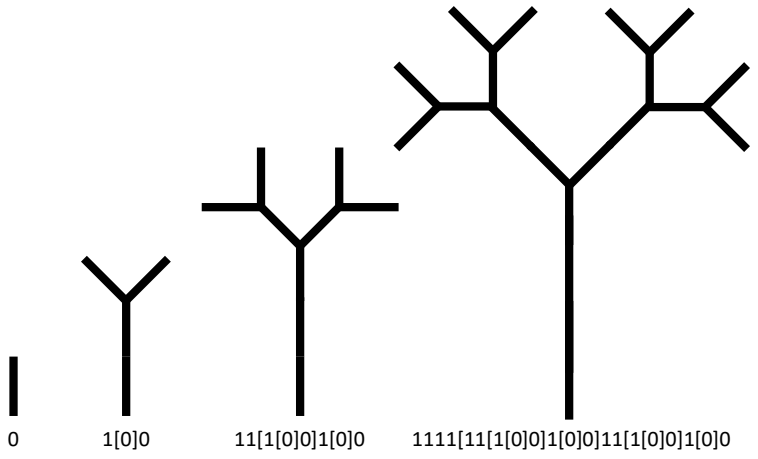


Figure 1.3. Binary fractal tree generated by a simple L-system.

Many of the procedural systems start from an initial state and evolve that state through time until a desired result is reached. This can be achieved using a grammar derivation as shown above, or an actual physics-based simulation. Various physics-based simulation methods have been used to model terrains via erosion (Beneš, Těšínskỳ, Hornyš, & Bhatia, 2006) or plant growth (Hädrich et al.,

2017). Furthermore, particle systems are typically modeled using simulation (Reeves, 1983).

One of the main strengths of procedural systems is their *emergence* property (Mitchell, 2009a). By applying very few simple rules repeatedly, a complex model emerges through the interaction between the parts of the system. This results in *data amplification*, where a relatively small number of parameters or rules given to the system yield wide variety of outputs (A. R. Smith, 1984). Stochastic procedural systems amplify the number of outputs even further by introducing randomness in to the generative process. Another important property of procedural system is their compact representation, which effectively *compresses* models that would otherwise require much more memory storage. This fact has been exploited, for example, by the *demoscene* community focusing on generating visual art with small self-contained programs (Burger, Paulovic, & Hasan, 2002). Finally, the practical benefit over traditional methods is the fact that procedural systems can generate content *automatically*, often in real time and at a controllable level of detail (Greuter, Parker, Stewart, & Leach, 2003).

1.3 Problem Statement

There are several open problems in procedural modeling, which we will discuss in the context of the recent survey by Smelik et al. (2014).

One of the issues of procedural modeling is the lack of **control**. In this work, we define control as follows: *The ability of the user to modify the procedural system such that it generates a model matching user's imagination, to the degree that is within solution space of the system.* While the essence of procedural modeling is to

generate geometry automatically, it is desirable to have a degree of control over the generative process and the final result. Ideally, we seek control of both the *global* characteristics of the model and the *local* geometry anywhere on the model. For example, when modeling a tree, we would like to have control of the overall branching angle while at the same time being able to modify individual branches. There are several reason why controlling procedural systems is difficult. One of the reasons being their *nonlinear* nature: a small change in input parameters often causes disproportionate changes in the generated model (Mitchell, 2009a). Furthermore, the system's parameters form high-dimensional spaces which are not trivial to visualize or restrict to achieve desired result (Talton, Gibson, Yang, Hanrahan, & Koltun, 2009). Finally, the sheer amount of parts generated, for example hundreds of branches when generating a tree, can be overwhelming to manage.

Fundamentally, there are two methods of control: forward and inverse. Forward control constitutes of modifying initial conditions of the generative system, for example its parameters and rules. This leads to an iterative modeling process where user adjusts parameters while observing their effect on the final geometry. While this process is typically tedious due to the nonlinear response to a change of parameters, the use of simulation has been successful in simplifying this process. An example of control via simulation can be found in the work of Beneš et al. (2006), where hydraulic erosion has been used for generating terrains. On the other hand, inverse control is performed by specifying requirements of the final result, while the system's parameters are automatically modified to generate result that fulfill those requirements. These requirements can range from sketches (Ijiri et al., 2006), painting brushes (De Carpentier & Bidarra, 2009) or volume and image

matching (Talton, Lou, Lesser, Duke, Měch, & Koltun, 2011). This leads to an optimization problem where the system's parameters are optimized such that the system generates a model respecting user specified requirements. While this decreases the nonlinearity of the system exposed to the user, it shifts the burden to the optimizer which has to navigate the nonlinear space. Therefore, finding the optimal solution is difficult and is hard to maintain interactivity of system. This parameter search problem can be often narrowed by leveraging domain knowledge, as was done, for example, in the work of (Ijiri et al., 2006) on sketching trees.

Another facet of control is the modification of the final generated geometry. Procedural systems often generate a hierarchical representation of the geometry which is conducive to manual editing after separating it from the system. However, in cases where integrated editing is desired, *i.e.,* when changes of the final geometry should propagate back to the system and **maintain consistency**, there seems to be no generic and straightforward solution. For example, when editing a generated tree geometry, should a local change of a branch propagate to instances of the same branch elsewhere on the model? What if that violates some other requirement imposed on the model, such as avoiding self-collisions? The difficulty of answering these questions is reflected in the literature, where this feature is seldom implemented. For example, Lipp, Scherzer, Wonka, and Wimmer (2011) demonstrated editing of procedurally generated city layouts where the consistency is maintained by *freezing* a small part of the urban network in a separate layer and then merging it with other, newly generated, layers. Similarly, Smelik, Tutenel, de Kraker, and Bidarra (2011) use merging of layers to maintain consistency of a system generating combination of landscapes, urban layouts and vegetation. Finally, the recent work by Vimont, Rohmer, Begault, and Cani (2017) introduced

deformation grammars that can be edited while maintaining constraints and consistency.

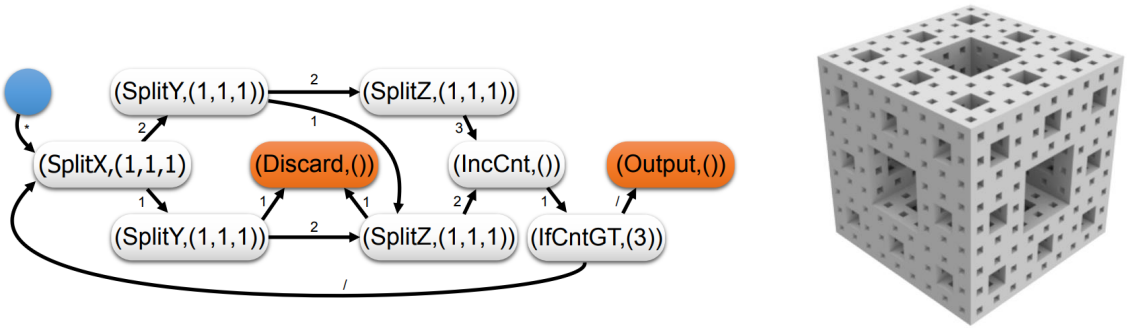


Figure 1.4. Operator graph generating the Menger sponge (Boechat et al., 2016)

Another issue in procedural modeling is that the systems are typically **specific** to a certain type of model, for example trees, terrains or buildings. While the common representations, such as L-systems (Lindenmayer, 1968a) and Shape Grammars (Stiny & Gips, 1972), can be extended and specialized to model a variety of objects, there has been little work on developing further generalizations until the recent years. For example, the work of Boechat et al. (2016) presented an operator graph representation able to represent a general geometric procedural system (Figure 1.4). Another interesting trend of research is treating procedural systems as general programs and amortizing their optimization cost using neural networks (Ellis, Ritchie, Solar-Lezama, & Tenenbaum, 2018; Sharma et al., 2018).

1.4 Research Questions

The overarching theme of this dissertation is to explore and improve control of procedural systems. The main research question being:

Is is possible to control procedural systems?

There is a wide variety of procedural modeling methods, and answering this broad question cannot be done in this work alone. Therefore, we focus on three procedural systems with different forms and degrees of control afforded to the user. We formulate a research question for each of them in attempt to understand and answer, at least in part, the overarching question. We explore control via *simulation* and via *optimization* in two different systems with a specific type of geometry. Finally, we introduce a more general system controllable via optimization.

In the first part of the dissertation we focus on indirect control via simulation. As demonstrated by the literature (Beneš et al., 2006; Krištof, Beneš, Křivánek, & Šťava, 2009; Musgrave, Kolb, & Mace, 1989; Šťava, Beneš, Brisbin, & Křivánek, 2008), it is a viable form of procedural modeling that produces natural looking results. We seek to extend this concept by applying interactive erosion and deposition simulation to modeling of arbitrary 3D objects, as opposed to terrains only. Thus, our first research question is:

1. *Can erosion and deposition simulation be controlled to model objects of arbitrary topology?*

Second, we look at control via optimization in the specific context of modeling 3D curves. Traditionally, modeling 3D curves is a painstaking process involving manual placement and modification of control vertices from multiple viewpoints. An alternative approach is to simply sketch the curve and infer its 3D shape

automatically. Unfortunately, there are infinite number of possible 3D shapes for a given sketch from a single view. We hypothesize that combining context of existing geometry to restrict the space of possible 3D curves and applying optimization with respect to curve's fairness can enable sketching from a single view. Our second research question is therefore:

2. *Is it possible to provide fine control of 3D curve generation, given a 2D sketch from single view and existing geometry as context?*

Finally, we focus on developing a generic system controllable by optimization. This system presents a novel procedural system representation and its optimization via evolutionary algorithm with respect to user-defined constraints. While most of the existing procedural systems are aimed at generating one type of object, we design this system with the ability to generate multiple types of models, such as man-made objects, trees and terrains. Additionally, the system is intended to be interactive, *i.e.,* the user can iterate and modify the system on-the-fly. The final research question is therefore:

3. *Is it possible to control procedural system generating arbitrary geometry using user-defined constraints?*

1.5 Scope

1.5.1 Control via Erosion and Deposition Simulation

The first part of this dissertation (Chapter 3) focuses on exploring control of procedural systems via *simulation*. In particular, interactively modeling shapes

using hydraulic erosion and deposition simulated using Smoothed Particle Hydrodynamics (SPH) (Monaghan & Gingold, 1977). Hydraulic erosion and deposition are natural phenomena that contribute to formation of terrains and other geological features. The eroding agent, such as river water, rain, or wind, exerts pressure and shear stress on object's surface, which results into removal and advection of material. Subsequently, deposition occurs when the advected material has lost most of its kinetic energy and gradually builds up layers of sediment. SPH is a Lagrangian method of simulating fluids and gases: the substance is represented as a set of particles, each having certain mass, velocity, and other properties. The properties of the substance at any point in space are then calculated by averaging (smoothing) the properties of nearby particles. Most of current systems for hydraulic erosion modeling focus on large scale terrain generation without local control and are limited to 2D data structures (i.e., heightmaps), which limits the achievable topology. In our approach, we utilize a layered volumetric data structure to be able to model arbitrary topology, such as overhangs (Benes & Forsbach, 2001). Finally, we focus on local control of erosion by utilizing fluid emitters which can be placed and modified interactively, much like a sculpting brush.

1.5.2 3D Curve Control via Optimization

Chapter 4 focuses on control via optimization in the specific case of modeling 3D curves. Traditionally, 3D curves are modeled by placing a set of control vertices in the 3D space that are interpolated or approximated to form a curve. Although this manual placement provides flexibility of the shape, a typical workflow consists of many iterations and observations from different points of view. We propose a system that is able to take a sketch from a single view and find the corresponding

3D curve, reducing the workflow to a single stroke. To achieve that, we utilize context of existing geometry to constraint the possible space of curves. Through an optimization process we then find the final 3D shape of the curve that respects the contextual geometry and maximizes its smoothness. Finally, we provide editing operations that enable easy modification of the sketched curve, even from different points of view.

1.5.3 General Control via Optimization

Chapter 5 presents a novel procedural system and an optimization framework. The representation is inspired by the operator graph of Boechat et al. (2016) and models the generative process as a dataflow graph where geometry flows through discrete operations. This gives the ability to generate various objects with arbitrary underlying geometric representation. The control of the system is achieved by user-specified constraints, which can range from high-level constraints, such as stability or ability to spin, to more local, for example, volume matching or collision avoidance. To generate objects that match the user given constraints, we cast the problem as multi-objective optimization problem and we attempt to solve it using an evolutionary algorithm. Specifically, we use an approach based on NeuroEvolution of Augmenting Topologies (Stanley & Miikkulainen, 2002) to optimize the graph representation using custom operations of mutation and crossover.

1.6 Significance

While the open problems in procedural modeling are not trivial, any advancements in solving them translate in an easier and faster modeling process by minimizing repetitive manual work. Particularly, solving the issue of control enables users to be more creative and have more expressive power while costing them less time and requiring less expertise. In fact, both individuals and companies can benefit from a more efficient content creation pipeline, as the expectations for realism and detail in virtual scenes increase.

The first part of this dissertation, focused on modeling object by erosion and deposition, provides a novel way to use simulation to model objects of arbitrary topology. While physics-based methods have been used in the past, our method method alleviates the problems of tweaking initial conditions by providing interactive control of the eroding agent. This significantly *speeds up the modeling process and enables modeling variety of objects with arbitrary topology.*

The second part, aimed at sketching 3D curves, provides a unique way of modeling 3D curves. Curves are ubiquitous in computer graphics and their use includes animation control (e.g., camera or character movement) and modeling freeform surfaces or tubular objects. By being able to use a single stroke from a single view, the *user can model 3D curves rapidly and conveniently.* Furthermore, an easy iterative editing process is offered to refine the curve, even from different views.

In the final chapter, a generic framework for generating procedural models via user-defined requirements is presented. This work is a *step toward generalizing procedural systems optimization*, addressing the problem of the *ad-hoc* nature of

most of procedural systems. Furthermore it offers an *interactive method of control for rapid ideation and design space exploration.*

1.7 Summary

This chapter provided the background to modeling in computer graphics and detailed the traditional and procedural modeling paradigms. We have discussed several open problems in procedural modeling, particularly the lack of control, and introduced three systems with different approaches on how to address these problems. The next chapter provides an in-depth overview of current literature on the topic of control and optimization in procedural modeling.

CHAPTER 2. REVIEW OF RELEVANT LITERATURE

Procedural modeling has been an active area of research for several decades. The earliest formalisms of procedural systems include *L-systems* and *shape grammars*.

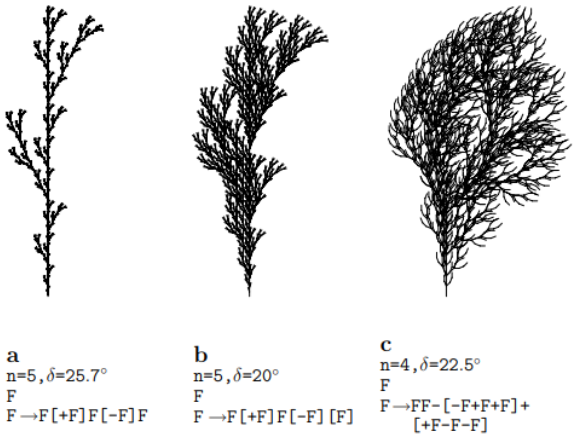


Figure 2.1. Parametric L-systems generating trees from Prusinkiewicz and Lindenmayer (1997).

L-systems were originally developed for simulating the development of multicellular organisms by Lindenmayer (1968a) and gained popularity in nature inspired procedural systems such as plant growth (Prusinkiewicz & Lindenmayer, 1997). An L-system is a formal grammar featuring parallel rewriting system in which all rules are applied in parallel as opposed to sequential rewriting of traditional grammars (Figure 2.1). It operates over *bracketed strings* of symbols. In the context of computer graphics, each symbol stands for a command to a LOGO-style

turtle that moves around a space and draws or generates geometry. A pair of opening and closing brackets in the strings represent branching, allowing L-systems to easily model tree-like phenomena. L-systems has seen number of extensions, notably parametric L-systems (Lindenmayer, 1974), which allow symbols to contain real valued parameters and stochastic L-systems (Eichhorst & Savitch, 1980) that assign each rewriting rule a certain probability. A stochastic L-system can therefore generate wide variety of models with a single ruleset.

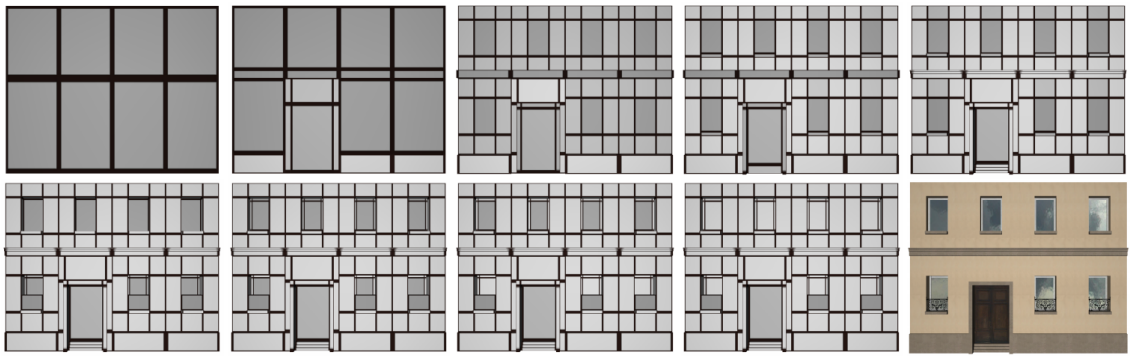


Figure 2.2. Example split grammar derivation of a facade from Wonka et al. (2003).

Shape grammars are another grammar based formalization of procedural systems. Introduced by Stiny and Gips (1972) with the goal of formalizing the generative process of paintings and sculptures, a shape grammar is an extension of formal grammar where the terminal, non-terminal and initial symbols are defined as a set of N-dimensional shapes. The grammar rewriting rules then correspond to adding a certain shape with a geometric transformation to a *canvas* based on already generated shapes. Shape grammars have become a popular tool for generation of architectural procedural models and have been extended by Wonka et al. (2003)

who introduced *split grammars*. Split grammars are a specialization of shape grammars that split individual shapes into smaller parts.

Both of the mentioned grammar systems are able to approximate many natural phenomena, especially those exhibiting fractal behavior such as self-similarity on multiple scales. Fractals themselves have been used extensively to generate landscapes, organic structures or textures as first shown by Mandelbrot (1982) and later by Musgrave et al. (1989). A related concept, while commonly not used for geometry generation, are cellular automata (Von Neumann et al., 1966; Wolfram, 2002). Cellular automata operate on a N-dimensional grid of cells where individual cells change their state based on the state of their neighbors. They share the emergence property of other procedural systems and are able to generate complex patterns using very simple rules.

2.1 Control and Guidance in Procedural Modeling

The essence of procedural systems is to generate geometry automatically, while the idea of user control is to create geometry manually. Because of this opposition, control in procedural modeling is a difficult problem that has been studied for decades but remains unsolved. Following are select works that improved the level of user control in procedural systems. Majority of past work has leveraged domain and representation specific knowledge to provide the user ways to control and guide the procedural model generation beyond mere rule and parameter adjustment. Three of the most studied domains of procedural modeling are discussed: tree, terrain and urban model generation.

2.1.1 L-systems and Trees

Trees are a crucial part of modeling realistic outdoor environments, futhermore, simulating virtual trees has been an important tool in biology and horticulture. L-systems is the formalism of choice when it comes to modeling virtual trees. Many researchers have been inspired by the biological processes and recognized that the tree geometry is heavily influenced not only by its genetics (i.e., internal parameters of the model) but also by its environment (i.e., light & shadow, nutrients). Thus, both can be leveraged to provide direct and indirect control over a generated tree model.

Měch and Prusinkiewicz (1996) extended the L-system to simulate the interaction of a growing plant and its environment, namely to grow plants into predefined shapes using pruning. Boudon, Prusinkiewicz, Federl, Godin, and Karwowski (2003) developed a tree modeling framework that allows the user to specify parameters using hierarchical decomposition of the generated L-system tree, including control of the resulting tree silhouette. They demonstrate their system's ability to finely control branches on several bonsai trees. Ijiri et al. (2006) introduced a tree modeling framework where the user can control the global shape of a tree using a single sketch. They automatically infer branching direction and level of recursion based on the sketch (Figure 2.3). Palubicki et al. (2009) developed a more advanced system utilizing the self-organizing process emerging from competition of tree buds for space and light. Their method not only produces highly realistic trees by using biology inspired parameters such as tropism and resource distribution ratios, but allows the user to control the tree shape using procedural brushes and offers other editing operations including branch pruning and bending in real-time (Figure 2.4). Longay, Runions, Boudon, and

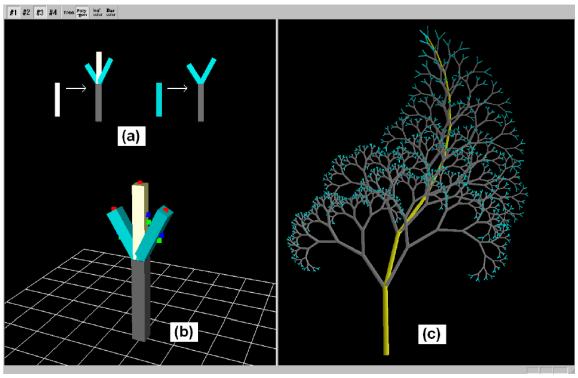


Figure 2.3. Sketch based interface for controlling L-system trees by Ijiri et al. (2006).

Prusinkiewicz (2012) later developed a sketch-based system based on the work of Palubicki et al. (2009) with a multi-touch tablet interface specifically aimed to provide simple interactive way of modeling trees. Beneš, Št'ava, Měch, and Miller

Figure 2.4. Custom shapes of plants from Palubicki et al. (2009)

(2011) improved on Palubicki et al. (2009) by using sketched regions that communicate via message passing, giving the user the ability to control procedural rules in each region and the interactions between them. This gives user a degree of both local and global control. They demonstrated the capabilities of their system on examples including not only trees, but also urban layouts and bridges.

Recently, Hädrich et al. (2017) introduced a system for modeling climbing plants that focused on providing user control over the growth process. The authors used strings of anisotropic particles to model the plants, instead of a set of line segments or generalized cylinders as previous works. This allowed them to simulate plausible physical effects such as bending and breaking of branches, and interaction with the wind (Pirk, Niese, Hädrich, Benes, & Deussen, 2014).

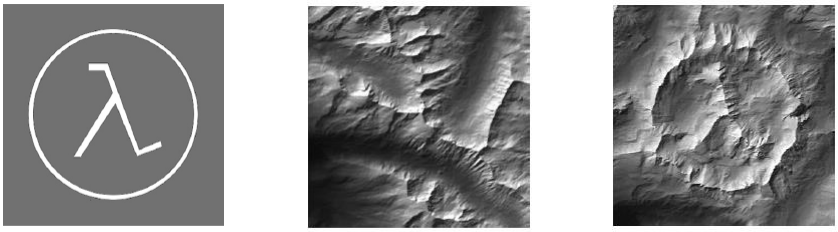
2.1.2 Terrains

Generation of terrain has a long history in computer graphics. Starting with Mandelbrot (1982) who noticed the similarity between the one-dimensional Brownian motion and the shape of mountain peaks, early methods used fractals (Peitgen, Jürgens, & Saupe, 1991), midpoint displacement (Fournier, Fussell, & Carpenter, 1982; Miller, 1986) and noise generators (Ebert et al., 1998; Musgrave et al., 1989), particularly Perlin noise (Perlin, 1985). Majority of current methods use a combination of noise functions to generate heightfields, a grayscale 2D image representing elevation at a given position. Other representations that are able to model overhangs and caves have been proposed (Benes & Forsbach, 2001; Gamito & Musgrave, 2001; Peytavie, Galin, Grosjean, & Mérillou, 2009). The early methods of terrain generation provided very little control to the user and were limited to choice of non-intuitive initial global parameters of the model, providing no local control of the result. Since then, multitude of methods have been developed that aim to provide more control over the terrain generation process.

Several simulation based methods have been proposed that leverage natural processes involved in creation of terrains in the real world, namely erosion and

weathering. Musgrave et al. (1989) developed a system that combines noise synthesis with local fractal dimension control with simple 2D simulation of hydraulic and thermal erosion implemented as cellular automata. A more sophisticated system able to erode 3D terrains was introduced by Beneš et al. (2006) that erodes and deposits material in a voxel grid using Eulerian fluid simulation. Wojtan, Carlson, Mucha, and Turk (2007) introduced a framework for simulating erosion and corrosion, including sedimentation. Šťava et al. (2008) introduced an interactive GPU implementation of a combination of hydraulic erosion algorithms on layered terrains. Finally, Krištof et al. (2009) developed a hydraulic erosion system using smoothed particle hydrodynamics (SPH) to sculpt large terrains efficiently. These systems provide an indirect control over the final terrain using fluid simulation as a proxy. This can create realistic terrains, but the control of the fluid itself may be difficult and produce non-obvious results.

A more direct control of procedural terrain generation can be afforded by sketching. Zhou et al. (2007) implemented a system that takes a line drawing corresponding to ridges and valleys from the user and produces a terrain by leveraging real or synthetic elevation data. This is accomplished by extracting features from the elevation data, matching them to the user's drawing and transferring and stitching corresponding patches into the final result (Figure 2.5). Belhadj (2007) also used line drawings to guide a terrain generator. The method he used was midpoint displacement, that is, repeated subdivision of a planar mesh, which produces fractal patterns. By constraining the subdivision process using the user provided drawings, the generated terrains remain random enough to look natural, but adhere to the user provided specification. Gain et al. (2009) used a sketch from an arbitrary perspective, as opposed to top-down view of previous methods (Figure



(a) User sketch (b) Mount Jackson (c) Synthesis Result

Figure 2.5. Sketch controlled heightmap generation by Zhou et al. (2007)

2.6). Using their method, the user defines a silhouette of ridges and the system automatically synthesizes a matching terrain. Furthermore, regions of the terrain can be filled with noise that is matching the frequency scribbled by the user. Apart from a forward synthesization, the system offers over-sketching and deletion capabilities, making the system applicable to a typical artist's workflow.
De Carpentier and Bidarra (2009) implemented a interactive GPU-based terrain

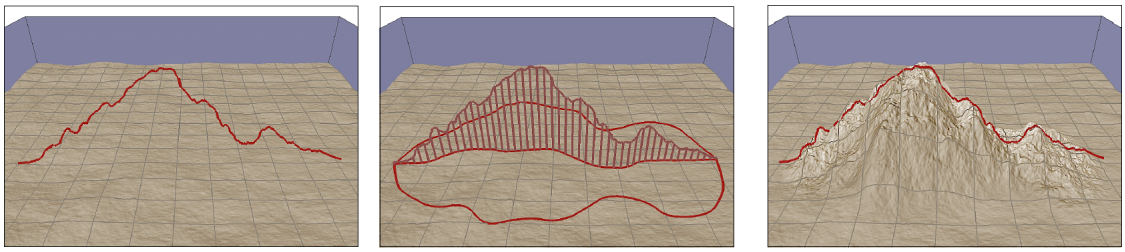


Figure 2.6. Terrain generation controlled by a drawing from perspective by Gain et al. (2009)

editing framework using procedural brushes. The user is able to paint on the terrain using a brush which in turn generates noise-based perturbations in elevation. Several different noises are available and can be seamlessly blended together during the painting process. Hnaidi, Guérin, Akkouche, Peytavie, and

Galin (2010) allow users to specify 3D curves denoting terrain features such as ridges or valleys and specify constraints on elevation, slope, angle and roughness. Their system then interactively generates the terrain using a simulated diffusion process that propagates the constraints to the entire working domain. Smelik, Tutenel, de Kraker, and Bidarra (2010) and Smelik et al. (2011) discuss a declarative approach to modeling terrains. Their framework, SketchaWorld, offers users to draw features of terrains, including cities, road networks and rivers. By combining multiple previous methods, SketchaWorld generates the terrain that includes all the features declared by the user. Gain, Merry, and Marais (2015) presented a comprehensive framework that offers high degree of local control when generating large scale terrains. They provide users with intuitive constraints and editing tools, including painting and transformation. Their framework generates the terrains using a constrained pixel-based texture synthesis from heightfield exemplars. Génevaux et al. (2015) introduced a system that is able model the terrains from a hierarchy of high-level parametrized primitives such as rivers and mountains. Similar to a CSG representation, they blend together individual terrain features to produce a final model of the terrain. Recently, Guérin et al. (2017a) presented an interactive system for sketching terrains using a conditional generative adversarial network (CGAN). By training this neural network on real and synthetic terrains conditioned on a sketch, the network can produce novel terrains for unseen sketches with high-fidelity (Figure 2.7). Cordonnier et al. (2017b) introduced a comprehensive framework inspired by real ecosystems that combines several natural phenomena to control the resulting terrain, including rainfall, vegetation, temperature, fire, and others.

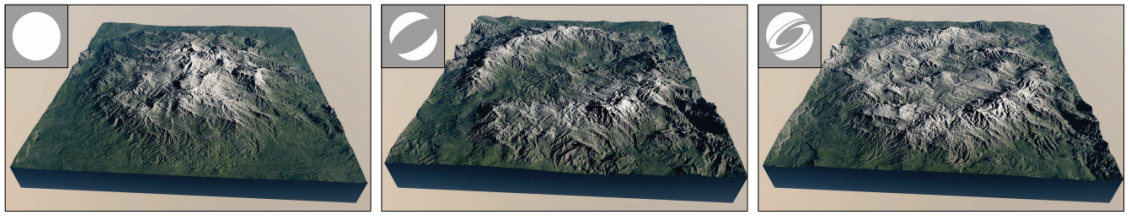


Figure 2.7. Example of the method of Guérin et al. (2017a). Realistic terrains can be generated based on a constraint image (insets).

2.1.3 Urban Modeling

Procedural modeling of urban scenes has been an another active area of research in recent years. Several systems have been proposed that can generate anything from room layouts, through facades and buildings, to entire cities. *Shape grammars* (Stiny & Gips, 1972), *split grammars* (Wonka et al., 2003) and *CGA grammars* (P. Müller, Wonka, Haegler, Ulmer, & Van Gool, 2006) have been particularly popular for generating facades and buildings, while *L-systems* have been employed in generating street layouts (Parish & Müller, 2001). The topic of urban modeling is quite wide, with focus not only on procedural generation but also on building and facade reconstruction (Nishida, Bousseau, & Aliaga, 2018; Wu, Yan, Dong, Zhang, & Wonka, 2014), proceduralization (Demir & Aliaga, 2018) and city-layout compression (Fiser et al., 2016). Following works were selected based on how they handle the control and guidance of the modeling process.

Several methods utilizing a simulation for creating cities have been proposed, Lechner, Watson, and Wilensky (2003) and Vanegas, Aliaga, Benes, and Waddell (2009) used simulated agents and let the city layout emerge from their interactions. Weber, Müller, Wonka, and Gross (2009) proposed a similar method but simulating

plausible city growth in time. The disadvantage of these methods is that they provide only global control over the general layout.

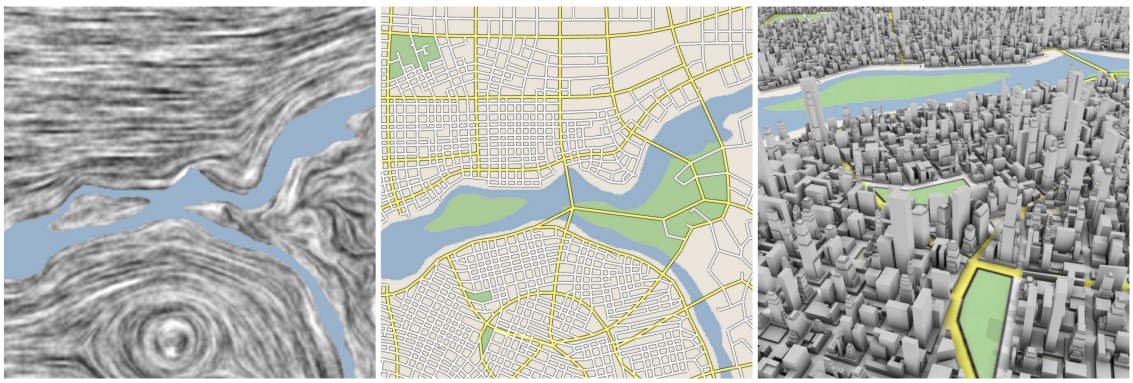


Figure 2.8. In G. Chen et al. (2008), a tensor field (left) is used to generate a street layout (middle) and a final city geometry is generated (right).

In the method of de Villiers and Naicker (2006), the user can create and modify city layouts using a simple sketch and gesture based interface. The user is able to sketch the road networks and subsequently modify properties of the city blocks using simple gestures. A similar system was developed by Kelly and McCabe (2007) who allow users to fill discrete regions with procedural patterns, for example a grid or organic like networks. An example-based method was presented by Aliaga, Vanegas, and Benes (2008) that uses both structure and image based synthesis to generate new city layouts. In their method, the user is able to manually stitch and expand parts of the city without having to connect individual roads. G. Chen et al. (2008) drive the city layout generation using a tensor field (Figure 2.8). Their system provides several global and local editing operations, including brush strokes, that modify the underlying tensor field and provide intuitive control of the flow of the streets. A system developed by Lipp et al. (2011)

offers even more local and precise control of urban layout that preserves validity of the layout and provide tools that allow users to make certain parts of the layout persistent even if the whole city is regenerated. Recently, Nishida et al. (2016b) presented a framework that allows for easy generation of urban road networks even by novice users based on exemplars. First, they start with a sketch of an area and existing road network, and then proceed to grow the network while allowing the user to blend and warp specific portions of the network at will.

Procedural building generation has not received the same amount of focus on control as the other types of models. Lipp, Wonka, and Wimmer (2008) attempted to address this issue by developing a visual editor for shape grammars. Although providing high degree of control unlike any previous building modeling system, the user still has to be familiar with the internal rules of the grammar. Patow (2012) and Silva, Müller, Bidarra, and Coelho (2013) later developed graph based editors to help visualize and easily edit the shape grammars. These systems however only provide access to the procedural rules and parameters and do not offer local control of the geometry.

To provide a more direct control of the resulting building model, Nishida et al. (2016b) presented a neural network approach that allows the user the sketch a building and a procedural system generates the geometry best explaining the sketch. They maintain a database of simple grammars capable of generating parts of a building, such as roofs or windows. Then they train a neural network to predict parts of a building based on a user sketch by utilizing a non-photorealistic rendered dataset of pre-generated building parts. Their system is able to quickly model buildings true to to the sketch, however this approach is limited by the amount of hand-crafted data available for training of the neural network.

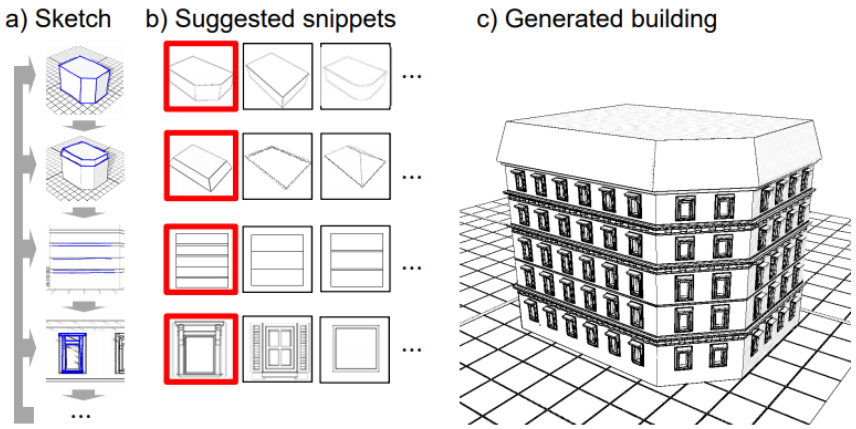


Figure 2.9. Pipeline of generating procedural building from a sketch proposed by Nishida et al. (2016b).

2.1.4 Conclusions

Effectively and efficiently controlling procedural system remains an open problem, although impressive progress has been made to date. To conclude, several points can be made based on the surveyed literature:

- Sketching and painting is a popular method to guide various procedural systems and is accessible to novice users.
- Iterative modification of the model, especially modification that does not break consistency of the model, *i.e.,* the model still follows the procedural system's rules, is desirable but seldom implemented (Lipp et al., 2011, 2008; Smelik et al., 2011; Vimont et al., 2017).
- Control of procedural models through indirect use of simulation provides natural looking results, however lacks local control of the model.

- Graph based and hierarchical interfaces have been seldom employed in the scientific literature, in contrast to commercial packages where they are relatively common (Autodesk, 2018b; Epic Games, 2018; Keeter, 2015; Unity Technologies, 2018).
- Majority of the contributions to control of procedural modeling focus on specific classes of procedural system, with a few exceptions that target general procedural modeling (Beneš et al., 2011; Vimont et al., 2017).

2.2 Optimizing Procedural Systems

Procedural systems can be viewed not only as a grammar, but as a specific subclass of general programs: a set of instructions that perform a task, where the task is to generate a virtual representation of an object based on some input. Optimizing and evolving general programs has been a topic of research for several decades and the idea has been even mentioned by Alan Turing as early as 1950 (Turing, 1950). Particularly genetic programming (Koza, 1992), the simulated evolution of programs, has been successful in finding optimal programs for various tasks. For an overview of the latest literature on the topic of genetic programming, and genetic program improvement in particular, see Petke et al. (2018). This section discusses literature related to optimizing procedural systems, either in the context of parameter or program search.

2.2.1 Parameter search

One way to optimize programs is to attempt to find optimal set of parameters that allow the program complete a particular task. In the context of procedural modeling there are several areas where this approach has been utilized.

In *constraint based procedural modeling* the goal is: given a set of constraints and a procedural system, find parameters of the system that generate a model adhering to the constraints. The constraints are typically modeled as a single objective function that is supposed to be minimized.

Another recent area of research is *inverse procedural modeling*, which looks at the problem from a different angle: given an object and a procedural system, find parameters of the system that generates the object. Again, the problem typically reduces to minimizing an objective function, in this case a measure of similarity between the given object and procedurally generated one. Therefore, both inverse and constraint based modeling can be reduced to a parameter search that minimizes an objective function.

Recently, several *neural network* based methods have been presented. In these methods, the neural network is used to amortize the cost of optimization. The network is typically trained to map constraints to parameters of the procedural system.

Finally, a related area of *design space exploration* focuses on searching through the parameter space and aims to provide the user with a curated collection of procedurally generated models.

Constraint based procedural modeling. Whiting, Ochsendorf, and Durand (2009) optimized parameters of a CGA grammar (P. Müller et al., 2006) to

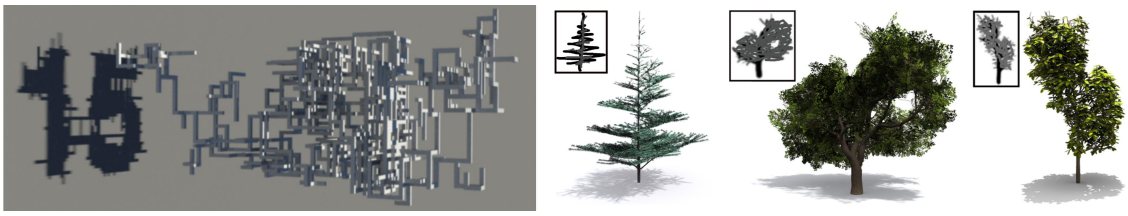


Figure 2.10. Procedural models optimized to adhere to a constraint. Shadow matching using SOSMC by Ritchie, Thomas, et al. (2016) (left). Sketch matching using MCMC by Talton, Lou, Lesser, Duke, Měch, and Koltun (2011) (right)

generate stable buildings. In their approach they use a measure of infeasibility as a energy function that is minimized using gradient descent. Talton, Lou, Lesser, Duke, Měch, and Koltun (2011) optimized grammar-based procedural systems, including L-systems and shape grammars, to follow constraints using the Monte Carlo Markov Chain method (MCMC). Some of the constraints they used are 2D and 3D shape matching and sketch matching (Figure 2.10). Their method works by sampling the production space of a given grammar to find parameters of the procedural system that best match the given constraints. Although this method can achieve impressive results, the time it takes to converge is on the order of minutes or hours, making it impractical for interactive modeling. Ritchie, Mildenhall, Goodman, and Hanrahan (2015) presented an improvement to the MCMC method by using Stochastically-Ordered Sequential Monte Carlo (SOSMC). The key idea is to gather information during the procedural generation and inform the choices of generating next piece. This is in contrast to Talton, Lou, Lesser, Duke, Měch, and Koltun (2011) where a quality of the model is determined after the whole model has been generated. They showed notable improvement in generation times over MCMC, however their method can have problems converging when choosing

suboptimal choices in the beginning of the process, as noted by Haubenwallner et al. (2017). Boechat et al. (2016) further improved on the optimization speed of Talton, Lou, Lesser, Duke, Měch, and Koltun (2011) using a GPU implementation and the operator graph representation of the procedural system.

One of the first works that investigated **inverse procedural modeling** was Aliaga, Rosen, and Bekins (2007). They presented an algorithm capable of inferring a split grammar from an existing subdivided model of a building. Their system was applied to interactive building completion. Šťava, Beneš, Měch, Aliaga, and Krištof (2010) developed a similar algorithm capable of finding parameters of 2D L-systems. Both of these methods look for repeated elements, symmetries and spatial relationships to encode the studied object into a grammar. Vanegas, Garcia-Dorado, Aliaga, Benes, and Waddell (2012) used a MCMC approach to model cities. In their system they define several *indicators* that locally evaluate the model, such as floor-to-area ratio and sun exposure. The user is able to choose the values of the indicators and a city model is generated to match them. Another method that uses MCMC is the work of Šťava et al. (2014) who presented a new parametric tree-generating model along with an efficient similarity metric of two trees. Their method can therefore find the parameters of wide variety of trees in the matter of minutes. Emilien, Vimont, Cani, Poulin, and Benes (2015) used inverse procedural modeling to analyze patches of objects positioned on terrain to learn their spatial pattern, which can then be reproduced elsewhere in a scene with a brush interface. Demir and Aliaga (2018) presented a grammar extraction framework for architectural models that provides the user with a degree of control to guide the process, allowing easy extraction of a procedural model from an existing architectural model.

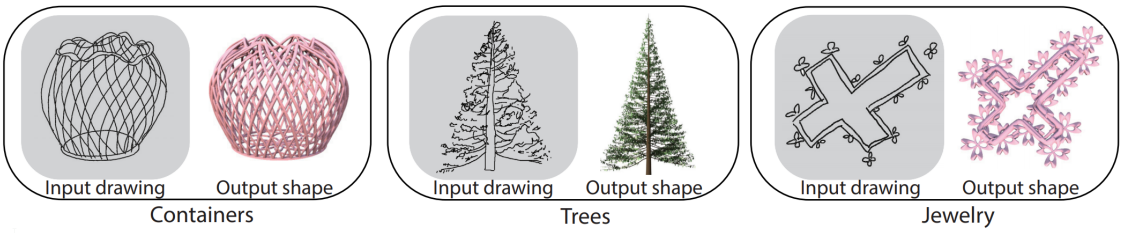


Figure 2.11. Sketch and generated model pairs from Huang et al. (2016)

Neural based amortization. Ritchie, Thomas, et al. (2016) trained a neural network that decides parameters of the procedural system during model growth. They demonstrated their system on growing a vine into a predefined 2D shape in a fraction of time of previous methods. However, this method requires application specific pre-training of the neural network, effectively amortizing the cost of optimization. Further work on amortizing inference in general probabilistic programs is discussed in Ritchie, Horsfall, and Goodman (2016). Another neural network approach has been presented by Huang et al. (2016) who proposed a system capable of predicting procedural system parameters based on a 2D sketch. They showed impressive results generating containers, trees and ornamental objects (Figure 2.11). Their system is trained on a number of pre-generated parameter-model pairs. Their neural architecture consists of two subnetworks, one trained for regression of continuous parameters, the other on classification of discrete parameters. Aforementioned Nishida et al. (2018) presented a similar approach that generates procedural buildings based on a 2D sketch and Guérin et al. (2017a) proposed a terrain generator utilizing *Generative Adversarial Networks*.

Due to the nonlinearity inherent to procedural systems, tweaking parameters may change the resulting models drastically. **Design space exploration** focuses on

providing the user with representative sample of the parameter space and tools to explore it intuitively. Talton et al. (2009) developed such a system for exploring the parameter space. They collected a number models created in their software by thousands of users. From this data they learned the desirable regions of the parametric space using *kernel density estimation*. Their method then samples the parametric space and generates novel models. Lienhard, Specht, Neubert, Pauly, and Müller (2014) also sample the parametric space and combine it with *desirable view estimation*. Their method selects a set of visually varied models that the user can choose to generate. They demonstrate their approach in an application where the user is shown a gallery of thumbnails of available models. Yumer et al. (2015)

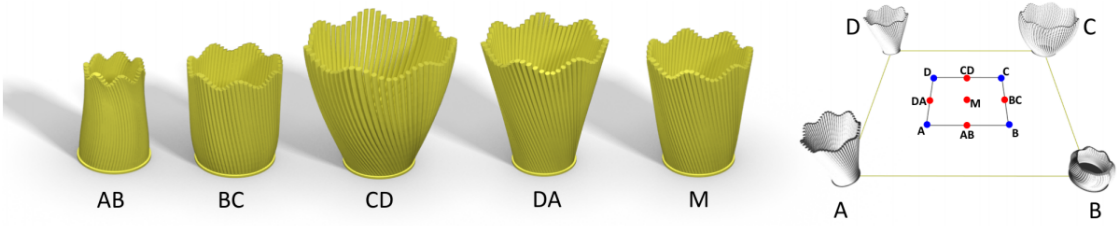


Figure 2.12. Yumer et al. (2015) achieved multi-dimensional linear interpolation of procedurally generated models using an autoencoder neural network.

used an *autoencoder* neural network to encode procedural model. The autoencoder reduces the dimension of the parameter space and simplifies navigation of this space. By learning a latent representation, they are able to smoothly interpolate between any model in a linear fashion, which would be impossible in the original parameter space. To ensure that the latent space interpolation is continuous in shape space, they augment the input and output of the autoencoder with shape descriptors.

The disadvantages of parameter search methods is that the generating system is not allowed to vary. This is useful when we have a well-defined description of the problem and we are only missing parameters for the particular model.

2.2.2 Program search

Program search is a more general approach that focuses on optimizing the program itself, which in context of procedural systems means the set of rules. While parameters can usually be mapped into some N-dimensional vector, program representations tend to be represented by a graph or list of instructions. When it comes to optimization, there are two main issues. First, programs are typically discrete and non-differentiable, making any gradient based methods impractical. Second, the dimensionality of the program can vary widely. The first issue is typically solved by employing other types of optimization processes, such as genetic programming. The second can be alleviated by different strategies, such as Reversible Jump MCMC used by Talton, Lou, Lesser, Duke, Měch, and Koltun (2011).

Many algorithms have been inspired by *evolution by means of natural selection*, for example the seminal work of Karl Sims on evolving virtual creatures (Sims, 1994b). The **genetic programming** (GP) is a technique that evolves computer programs (Koza, 1992). GP has seen remarkable success and amount of research in past decades, as illustrated by Koza's survey of GP methods that are competitive with humans on difficult tasks like image classification and circuit design (Koza, 2010). Genetic programming evolves programs by evaluating a population of individual programs based on their fitness and applies mutation and crossover

operations to create new individuals. The programs with the best fitness are kept in the population and the rest is discarded. By repeating this process, the algorithm iteratively maximizes fitness while having the important property of avoiding local minima.

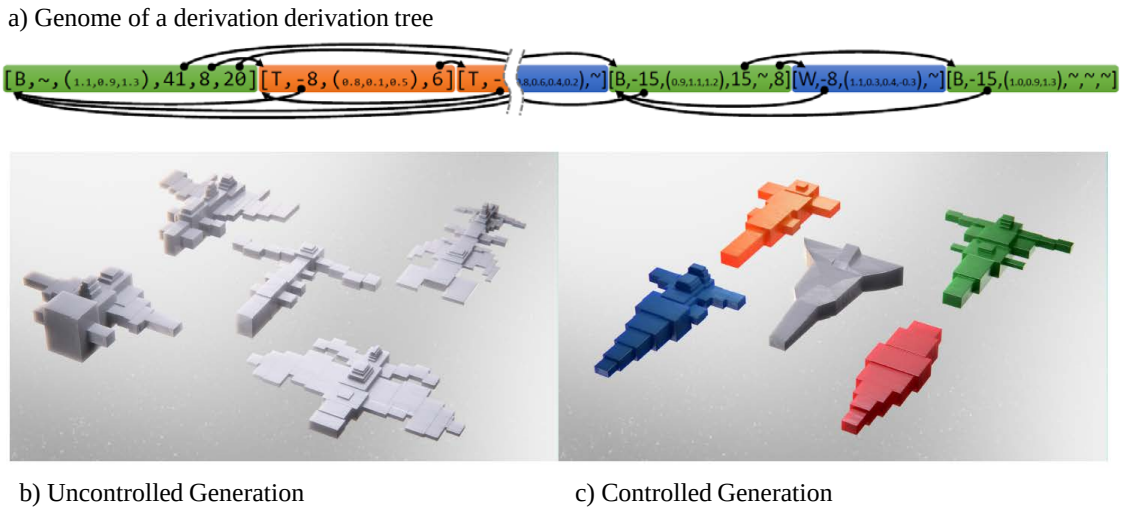


Figure 2.13. Genetic algorithm applied to procedural model optimization (Haubenwallner et al., 2017). The derivation tree of a given grammar is encoded into a linear genome (a). One of the demonstrated examples shows a shape controlled generation (c) versus random derivation of the grammar (b).

There is little work in the literature that applied genetic programming to problems in procedural modeling. McDermott (2012) presented a method for evolving graph grammars that produce 3D networks using *grammatical evolution* (O'neill, Ryan, Keijzer, & Cattolico, 2003). The fitness function they have employed measures only the complexity of the model. Therefore, there is little control in what geometry is generated.

Haubenwallner et al. (2017) uses a *genetic algorithm* to evolve procedural models. They encode the derivation of a given grammar into a genome (Figure 2.13a) and perform mutation and crossover operations that retain validity of the grammar derivation. They showed that their approach converges faster and more often than previous methods of Talton, Lou, Lesser, Duke, Měch, and Koltun (2011) and Ritchie et al. (2015), demonstrating that evolutionary optimization is a valid and effective way to optimize procedural models. However, this approach only finds a particular derivation of a procedural grammar.

Neural based methods have been recently applied to program optimization as well. Beltramelli (2017) used deep learning to translate screenshots of graphical user interfaces into a code able to generate it, such as HTML/CSS or XML. One of the most recent examples is the work of Ellis et al. (2018). They proposed a method that is able to take simple hand-drawn images and translate them into a graphic programs able to generate LaTeX-style figures. The programs follow a simple grammar that include simple primitive drawing, loops and conditional statements.

Similarly, Sharma et al. (2018) showed a neural approach that infers a simple program, equivalent to a CSG hierarchy, that constructs a given 2D or 3D shape (Figure 2.14). Their method uses reinforcement learning and encoder-decoder architecture, where the input is an image of an object, the output is a program that generates an object, and the reward is the minimization of difference of the two in image space.

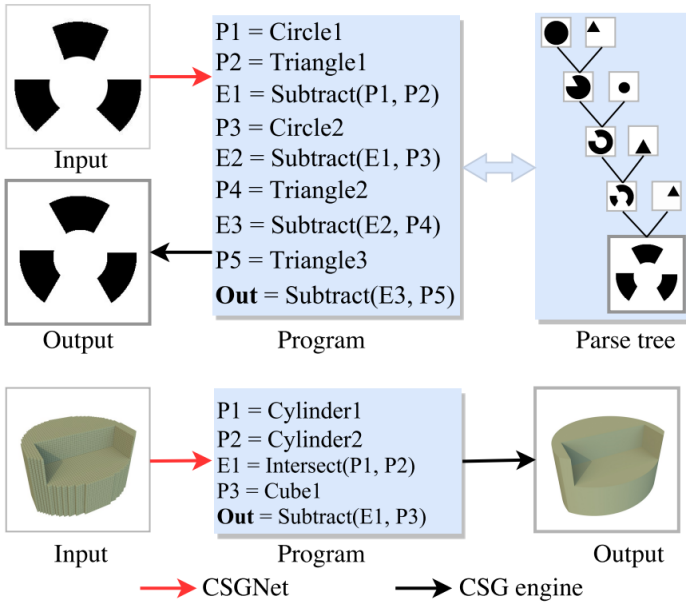


Figure 2.14. Method of Sharma et al. (2018) converts a 2D or 3D object to a program and an equivalent CSG representation.

2.2.3 Conclusions

Optimization of procedural systems is a currently active research area with many avenues for further research. Following points can be made to summarize the current literature:

- Optimization has been proven to be a great way to control and guide procedural modeling. This can be expected to improve even further with the increase of computational resources.
- Majority of the work in the domain of procedural system optimization has focused on optimizing over parameter space.

- Genetic Programming, although powerful at optimizing programs (which procedural systems are a subset of), has not been applied in this context in any significant way.
- Neural networks and deep learning have been shown to be useful in amortizing the cost of optimization (Guérin et al., 2017a; Nishida et al., 2018; Ritchie, Thomas, et al., 2016). While not as flexible as general optimization methods, they are extremely useful tool for applications where a predefined training dataset is available.
- Neural networks excel at learning representations, as shown in Yumer et al. (2015). There is opportunity for further research that can leverage these learned representations.

2.3 Summary

This chapter provided an overview of the literature related to procedural modeling, specifically their control and optimization. Furthermore it identified common themes, open problems and opportunities for future work. The next chapters will discuss the three proposed procedural systems.

CHAPTER 3. EROSION AND DEPOSITION SIMULATION

This chapter is a result of collaboration with Torsten Hädrich[1], Alejandro Guayaquil[2], Oliver Deussen[1], Sören Pirk[1], and Bedrich Benes[2]. It has been submitted to *SCA 2019: Symposium on Computer Animation*. Author's contribution include implementation of the erosion and deposition system, rendering pipeline, result generation, measurement, and manuscript writing.

[1]University of Konstanz, [2]Purdue University

3.1 Abstract

A novel user-assisted method for physics-inspired modeling of geomorphological features on polygonal meshes using material erosion and deposition as the driving mechanisms is introduced. Multiple polygonal meshes that define an input scene are converted into a volumetric data structure that allows to efficiently track mass and boundary of the resulting morphological changes. Our approach allows simulation of multiple materials and different media. We use Smoothed Particle Hydrodynamics (SPH) to simulate fluids and to track the eroded material. Erosion and deposition happen on the material boundary. The eroded material is converted to material particles and naturally deposits in locations such as sinks or corners. Once deposited, we convert material particles back into the volumetric representation. This limits the number of required particles and enables capturing a

large variety of natural terrain features, such as mountain ridges, valleys, overhangs, and arches. By using graphics hardware, our method runs at interactive rates, with the ability to process moderately complex scenes with interactive feedback.

Figure 3.1. An example of sculpting by using wind. The dragon embedded in the block of stone reveals itself after being interactively eroded out by the user. Our system allows to interchangeably represent material as particles and volumes and thereby enables the efficient modeling of erosion and deposition of complex scenes.

3.2 Introduction

Defining and controlling natural shape morphology is still a challenging problem in computer graphics. Real world objects undergo a large variety of morphological changes that can be modeled by a human, but this is often a tedious manual process. Many areas of computer graphics have been inspired by Nature, and many approaches exist that allow for user-controlled creation of real-world features. A good example is rendering, where physics-based algorithms are preferred over *ad hoc* approaches. However, geometric modeling still heavily relies on approaches, where the user defines an object shape by manual carving and adding material.

This may be tedious, requires experienced users, and does not scale for large models or large quantity of elements.

Physics-based modeling has been successfully used in computer animation, fluids, and material appearance but it is not commonly used for shape modeling. In nature, one of the most important morphogenetic agents are erosion and deposition. Among the geomorphological processes that consistently shape natural objects, interaction with wind and water are probably the most important ones. Although these methods have been previously used in computer graphics, they are still not common in production probably because of their low controllability. Many parameters are required to control the simulation, and the effects are often difficult to predict and handle. Furthermore, existing approaches mostly focus on modeling large scale objects, such as terrains, and cannot capture subtle structures and fine details. The usual procedure is that the simulation is executed and if the results are not desirable, the user needs to restart the simulation with different input parameters.

We introduce wind erosion as a new method for physically-based modeling of geomorphological processes that uses erosion and deposition. Our focus is on the simplicity of use and interactivity. We start from a set of polygonal meshes representing input objects and their inner structure. These meshes are first converted to a volumetric representation. The user controls erosion and deposition by interactively manipulating fluid emitters that affect the objects. Our framework combines two kinds of particles in one system, one that represents fluids and another one for the eroded material. During the simulation, each surface voxel is examined and its stress is calculated. Depending on the material properties, it can be eroded and the eroded mass is converted to material particles. The released

particles are advected by the fluid particles and they naturally sink to the ground. If a material particle did not move for a certain amount of time, it is deposited and converted back to a voxel. The dual particle representation allows for a simple implementation and it also allows to treat both phenomena independently. In theory, any fluid simulation that provides stress on the boundary could be used. Besides the automatic modeling of geomorphological changes in large scenes, such as terrains, our system supports interactive shaping of different kinds of objects. Controlling the fluid emitters allows the user to precisely release material at certain parts of the input geometry. Similar to a brush tool this provides new means for modeling geometry. Figure 3.1 shows an example of erosion and deposition based modeling. The dragon embedded in the material reveals itself after the outer material is interactively eroded out by a user.

We present the following contributions:

- the introduction of a dual SPH simulation for erosion and transportation of materials,
- a novel erosion and deposition model that employs particles for fluids and eroded materials and a volumetric data structure for input meshes,
- the simulation of multiple materials to approximate layered objects, and
- we introduce means for interactively shaping and accentuating different objects.

3.3 Related Work

Modeling shapes with appearance similar to natural objects has been the focus of computer graphics research for many years. While early approaches were inspired by fractals Fournier et al. (1982); Mandelbrot (1982) it has been quickly noticed that a convenient way to simulate natural phenomena is by mimicking natural processes such as erosion Musgrave et al. (1989).

Appearance modeling by simulated aging adds complex natural phenomena to computer graphics S. Merillou and Ghazanfarpour (2008) and increases the visual realism of objects and landscapes. One of the main challenges lies in efficiently tracking morphological changes of the input geometry. The approach of Bremer et al. (Bremer, Porumbescu, Kuester, Joy, & Hamann, 2001) attempts to overcome this limitation by presenting adaptive distance fields. Dorsey et al. (Dorsey, Edelman, Jensen, Legakis, & Pedersen, 1999) introduced a slab-based data structure that supports a weathering model for simulating the flow of moisture and recrystallization of minerals in stones. Dorsey and Hanrahan (Dorsey & Hanrahan, 1996) as well as Merillou et al. (N. Merillou, Merillou, Galin, & Ghazanfarpour, 2012) model the rendering of metallic patinas and the effects of salt decay to enhance the appearance of buildings. Gagnon et al. (Gagnon & Paquette, 2011) and Cutler et al. (Cutler, Dorsey, McMillan, Müller, & Jagnow, 2002) employ procedural techniques for modeling and authoring solid models. Wang et al. (Wang et al., 2006) propose a simulation technique for changing the appearance of materials over time and thus to model different degrees of weathering for man-made objects and even plants. A key problem with these methods is their large number of control parameters that make it difficult for users to express their intent, in addition to low interactivity.

Many approaches focus on approximating physical processes to simulate complex surfaces and geomorphological behavior. One of the early approaches by Blinn (Blinn, 1982) introduces reflection of light on dusty surfaces and clouds. Various approaches address aspects of modeling such natural phenomena, such as corrosion S. Merillou, Dischler, and Ghazanfarpour (2001a), peeling of coatings Gobron and Chiba (2001b); Paquette, Poulin, and Drettakis (2002) and patination Chang and Shih (2000), cracking patterns Gobron and Chiba (2001a); Hirota, Tanoue, and Kaneko (1998); Iben and O'Brien (2006), surface scratches Bosch, Pueyo, Merillou, and Ghazanfarpour (2004); S. Merillou, Dischler, and Ghazanfarpour (2001b), fractures Busaryev, Dey, and Wang (2013); M. Müller, Chentanez, and Kim (2013); Pauly et al. (2005), rust Chang and Shih (2003), and drying Lu, Georghiades, Rushmeier, Dorsey, and Xu (2005), or a geometric simulation of woodification Kratt et al. (2015). Many of those algorithms are common in production, however, most of them focus on appearance changes. In our work, we address the geometrical changes of the model.

While the previous methods focus on small-scale objects, the synthesis of large scale objects is a very difficult problem. Among large objects, terrains have been an open problem for almost three decades. The first works focused on using fractals Musgrave et al. (1989), other methods use predefined maps Miller (1986), splines combined with fractals Szeliski and Terzopoulos (1989), or noise-based procedural approaches Ebert et al. (1998). Peytavie et al. (Peytavie et al., 2009) propose a method for modeling complex geomorphological phenomena such as stone arcs or caves with a high degree of realism by creating arches, overhangs, and stones by aperiodic tiling. Other methods focus on interactive modeling and authoring of terrain and its corresponding features Jones, Farley, Butler, and

Beardall (2010); Schneider, Boldte, and Westermann (2006); Šťava et al. (2008); Vanek, Benes, Herout, and Stava (2011). Emilien et al. (Emilien, Poulin, Cani, & Vimont, 2014) amends this by designing waterfall scenes. In Emilien et al. (2015) localized inverse procedural modeling is used that learned from existing models to allow the user interactively modeling object distribution in virtual worlds. Deep learning has been used to generate terrains from sketches by using cGANs in Guérin et al. (2017b). Just recently a novel approach has been introduced Cordonnier, Cani, Benes, Braun, and Galin (2018) that uses subsurface geology to simulate large-scale terrains.

Even erosion and deposition have been used to simulate realistic materials. One of the early works of Kelly et al. (Kelley, Malin, & Nielson, 1988) and Hsu et al. (Hsu & Wong, 1995) simulate eroded terrains. Other methods focus on expressiveness Nagashima (1998), layered data structures Benes and Forsbach (2001) or on interactive modeling mechanisms Neidhold, Wacker, and Deussen (2005) to efficiently model terrain erosion. Advances in hardware technology allowed for more complex and realistic methods for computing hydraulic erosion Beneš et al. (2006) even at interactive rates Anh, Sourin, and Aswani (2007). More recently, Wojtan et al. (Wojtan et al., 2007) proposed a discretized approach for simulating physical and chemical behavior based on finite differences and level sets. Šťava et al. (Šťava et al., 2008) propose a method that supports a variety of local and global editing operations by integrating different erosion approaches. Just recently, ecosystems were combined with terrain erosion in Cordonnier et al. (2017a).

Close to our work is the approach of Kristof et al. (Krištof et al., 2009) who also coupled a physically-based erosion model with smoothed particle hydrodynamics.

In their approach, particles carry eroded material that is implicitly and explicitly advected due to particle motion. In contrast to them we introduce a dual particle approach that supports the interactive erosion, transport and deposition of different materials. To efficiently track morphological changes we use a layered data structure Benes and Forsbach (2001). Additionally, we couple a fluid simulation based on SPH Monaghan (2005) with solid granular particles (Alduán & Otaduy, 2011; Narain, Golas, & Lin, 2010) to model released eroded material. These granular particles are transported by the fluid system and naturally deposit material in characteristic locations such as sinks and corners. Our system supports various materials such as rock, sand stone, soil and dust as well as different media such as wind and water.

3.4 Overview

An overview of our method is given in Figure 3.2. Input is a set of polygonal surface meshes that represent embedded volumetric objects, the fluid sources, and parameters that control erosion simulation. In the simplest configuration the entire system runs completely automatically. The user can also interact with the model by moving the fluid emitters and obstacles in the scene and by changing system properties such as the strength of the particles or their density.

Our system first converts the input mesh into a layered slab-based data structure introduced in Benes and Forsbach (2001). This representation stores detailed information about the interface of the material and the outer environment. It also allows us to track the geomorphological changes typical for an erosion process.

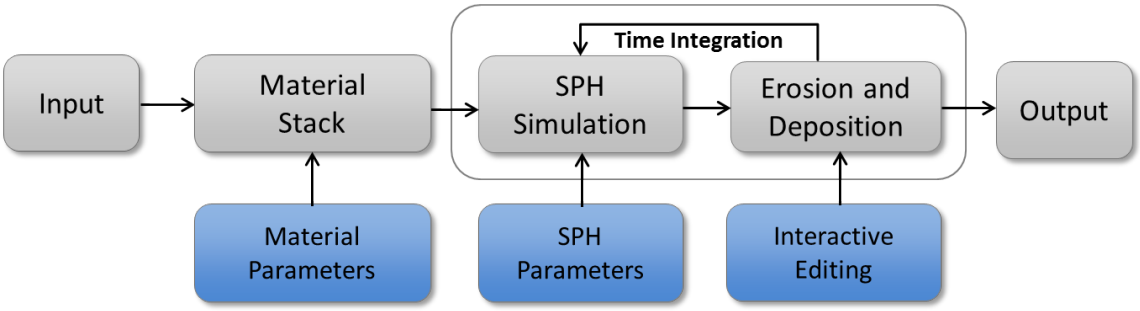


Figure 3.2. Overview: given is a set of input meshes representing the volumetric object, a set of parameters for material properties, and the erosion simulation. The mesh is converted into a volumetric representation. Erosion is simulated by converting object material at the outermost layer of the object into material particles and by tracing and depositing those particles using SPH.

Each slab represents a volume with different material properties at the surface of the material and we will call it *a material stack* (see Figure 3.3).

We simulate erosion with a Lagrangian fluid model that uses particles as discrete quantities – Smoothed Particle Hydrodynamics (SPH) Monaghan (1992, 2005). Such systems are adaptive and allow for quick integration and interaction with a given geometry. They provide complex fluid phenomena such as turbulence, fusion, and separation. The SPH simulation runs entirely independently of the material erosion/deposition and it just traces particles in the scene. We use the shear stress from the SPH to erode the material and the material particles carry the material.

To speed up the calculation, we use an additional data structure to store voxels that represent the outermost material layer of our model on the interface between the geometry and the fluid. We calculate the shear stress caused by the SPH simulation on the surface of the object. If the stress exceeds a material critical value, the material erodes. Erosion is simulated by removing the material from the

affected voxel and by converting it to material particles that are advected by the fluid. Gravity causes them to sink. If the material particles do not move for some time they are deposited to the object and integrated into its material stack.

This approach provides interactivity by computing the simulation on the graphics hardware. The user manipulates the erosion by controlling fluid emitters in the scene that allows for an efficient shaping (carving) and editing of geometry and is similar to the various brush tools available in image processors.

3.5 Data Structure

Initially, we implemented volumetric slabs that extrude triangles of the input mesh that were introduced by Dorsey et al. (Dorsey et al., 1999) for stone weathering. This data structure is highly suitable for small and localized surface changes, but it requires expensive re-triangulation and slab reconstruction when large changes occur that deemed ineffective in our context. Moreover, this data structure does not handle topological changes of the mesh caused by erosion and deposition.

Our data structure is an extension of a hybrid layered representation introduced by Benes and Forsbach (Benes & Forsbach, 2001). This representation compresses materials as layers of run-length encoded voxels that we call *stacks*. The main advantage of this representation is its high compression and the ability to represent overhangs and caves. Moreover, it can also efficiently track the mass and the boundary of morphological changes. Peytavie et al. (Peytavie et al., 2009) extended this representation with an implicit model that allows for constructing a smooth surface suitable for SPH collision detection and for the conversion back to a triangular mesh.

The erosion and deposition processes occur on the boundary of the object. Therefore, we have further extended the previous work by explicitly representing the list of boundary voxels, *i.e., active voxels* that participate in the erosion and deposition process. When a new voxel is exposed it is added to the list of active voxels and when a voxel is deposited, the potentially covered voxels are tested for removal from the list.

Figure 3.3 shows a 3D canyon scene with caves and overhangs encoded as volumetric representation (top) and its rendering as an implicit surface (bottom).

3.5.1 Stacks Generation

The input to our method is a set of polygonal meshes where each mesh represents a different material. The conversion from a triangular mesh to our data structure is performed by the scanline algorithm. It initially converts a mesh into an evenly spaced volumetric grid – each space between two meshes is one material.

The voxels corresponding to the area outside the mesh are marked as layers of air. Additionally, a unit sized ground stack with zero mass is created to allow for deposition of particles that fall on the ground. Finally, voxels are converted into a material stack if they are on top of each other, are not exposed, and share the same material type.

3.5.2 Stacks Boundary – Active Voxels

As mentioned above, instead of evaluating the SPH and model interaction in every voxel, we use an additional list of voxels that represents the layers lying on the

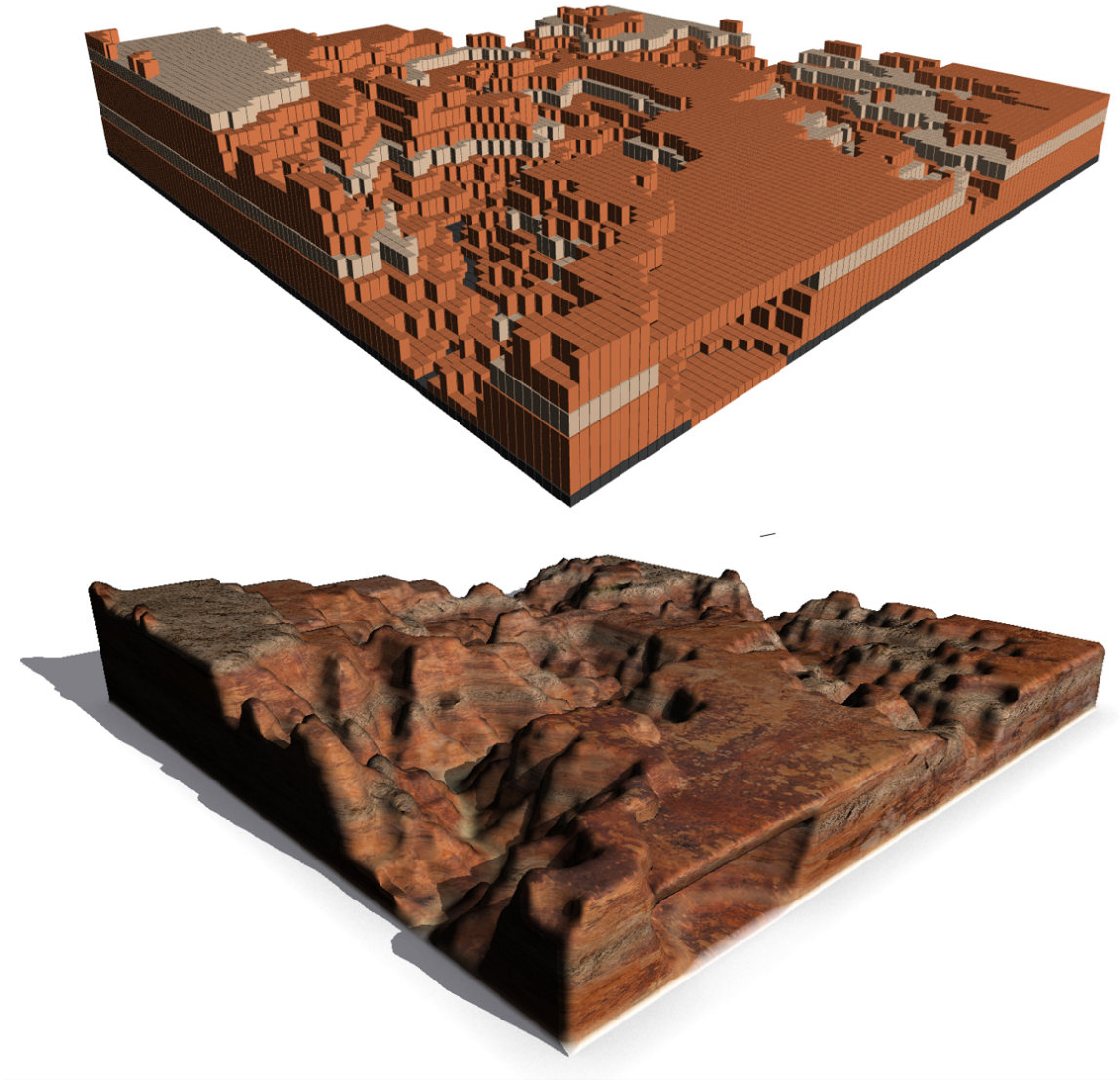


Figure 3.3. Input meshes are converted into a layered volumetric structure, the material stack (top), that is rendered as implicit surfaces (bottom).

erodeable boundary of the model. This significantly speeds up the erosion and deposition simulation.

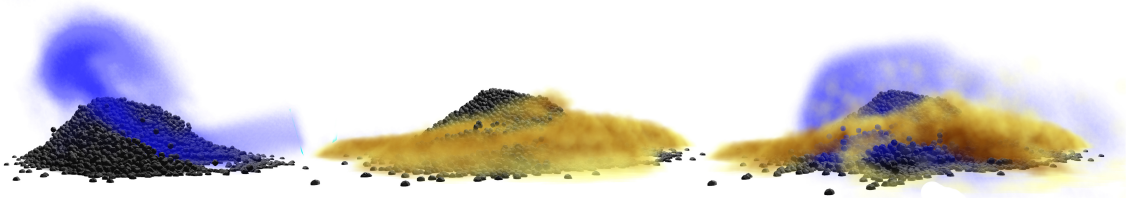


Figure 3.4. Interaction of different particle types. Granular particles interacting with air (left); a layer of dust on top of a pile of a granular material (middle), and all three particle types interacting with each other (right).

The exposed voxel structure represents a volume on the surface, it carries the information about the amount of material in its volume, which is implemented as a single scalar value in the range of $(0, 1]$. Note that the active voxels can be also inside the structure if there are caves present.

Additionally, the active voxels data structure also contains surface vertices, *i.e.,* the points where the implicit surface intersects the voxel, and the normal vectors in them. Finally, the interconnection between exposed voxels and their respective layers is achieved by both structures, which contain array indices that point to one another. The boundary voxels need to be updated after erosion, when some voxels may disappear, and deposition, when new voxels may be created.

3.5.3 Isosurfaces

During the erosion and deposition simulation the SPH particles collide with the surface. In order to calculate collision response, we need to know the location of

the surface and the local gradient. Therefore, we evaluate the isovalue f of the implicit model at a given point $\mathbf{p}$ as

$$f(\mathbf{p}) = \frac{1}{4\sigma^3}\left(\sum_{m\in M} V_m(\mathbf{p}) - 1\right), \tag{3.1}$$

where V_m is the volume of the material of type m inside a cube kernel, whose sides are 2σ. The set M includes material types except air.

Our algorithm calculates the material volume, traverses the active voxels and stacks in the neighborhood of the point, and sums volume contributions from layers in these stacks. If a slab is located on the surface, its volume contribution is multiplied with the amount of material that is present in it. This allows for a smooth transition of the surface shape as the material erodes away or deposits.

3.6 Particle Simulation

A particle system that couples two different materials is introduced: one is the fluid that corresponds to the erosion agent and the other are particles that represent eroded material. We simulate the fluid and the eroded materials that are advected by using Lagrangian particle integration. In particular, we use SPH according to Monaghan (Monaghan, 1992) to sample continuous behavior of the fluid flow. Particles are trackable in 3D space and they allow for easy and scalable simulation that can be ported on the GPU. Moreover, they are implicitly adaptive since they move in the areas where the erosion is happening and the calculation needs to be executed.

SPH have been used for erosion simulation in the work by Kristof et al. (Krištof et al., 2009). In contrast to the previous work, we use a dual particle system and we

do not use the fluid particles to carry material. Also, we do not use an explicit boundary representation by converting the boundary to particles, but we employ a volumetric representation that makes our approach fully 3D and allows for easier solution of topological changes. Finally, we aim at interactive erosion and deposition simulation. We convert volumes of material into particles and couple them with fluid particles similar to the approach of Lenaerts and Dutré (Lenaerts & Dutre, 2009). Material and fluid particles are integrated by solving for pressure gradients in the continuum domain. Compared to other granular material methods this blends well with SPH and allows for an efficient processing at interactive rates.

3.6.1 Smoothed Particle Hydrodynamics

SPH solves the movement of the fluid by representing it as a set of independent particles that carry physical quantities, *e.g.,* such as pressure and mass. The acceleration a_i of the $i-$th particle is computed as

$$\mathbf{a}_i = \frac{d\mathbf{v}_i}{dt} = (-\nabla p + \mu\nabla^2\mathbf{v} + \mathbf{f}_{ext})\frac{1}{\rho}, \tag{3.2}$$

where $-\nabla p$ is pressure, $\mu\nabla^2\mathbf{v}$ is the viscosity, external forces are denoted by $\mathbf{f}_{ext}$, and density by ρ. Fluid quantities $A(\mathbf{x})$ at a certain location $\mathbf{x}$ (different from the location of the particles) are computed as a weighted sum of neighboring particles j

$$A(\mathbf{x}) = \sum_{j=1}^{N} V_j A_j \; W(\mathbf{x} - \mathbf{x}_j, h), \tag{3.3}$$

where V_j is the volume and W a smoothing kernel with a compact support radius h.

The fluid moves over time by applying Eulerian integration.

3.6.2 Eroded Material

Material particles are transported by the fluid that is simulated as a second SPH system that is embedded in the fluid. The particles of the material can be thought of as granular material carried by the fluid.

The Mohr-Coulomb yield criterion is employed to determine material yielding. It states that plastic deformation occurs if the friction stress exceeds the yield strength. Otherwise the particles remain undeformed.

Zhu and Bridson (Zhu & Bridson, 2005) treat particles in non-flowing regions as rigid particle clusters, other approaches compute granular forces in an iterative predictive-corrective fashion in order to dissipate the strain rate Ihmsen, Wahl, and Teschner (2012). To avoid computational overhead of such methods in our simulation, particles in non-flowing regions are marked as static, that also simplifies the material deposition (Section 3.8).

The governing equation for conservation of momentum of granular material can be written as

$$\rho \frac{d\mathbf{v}_i}{dt} = \mathbf{f}_{pressure} + \mathbf{f}_{shear} + \mathbf{f}_{ext}, \tag{3.4}$$

where $\mathbf{f}_{shear}$ represents friction between particles. A shear stress tensor $\hat{s}$ is computed as

$$\hat{s} = 2\mu(D - \frac{1}{3}|D|\delta), \tag{3.5}$$

where μ is the shear modulus, D is the strain rate tensor

$$D = \frac{1}{2}(\nabla \mathbf{v} + \nabla \mathbf{v}^T),$$

moreover, $|D|$ is the trace of the strain rate tensor, and δ is the Kronecker's delta tensor. The shear force is computed by the SPH approximation

$$\mathbf{f}_{shear} = \sum_j \frac{m_j}{\rho_j} \hat{s}_j \nabla W(\mathbf{x} - \mathbf{x}_j, h) \tag{3.6}$$

Figure 3.4 illustrates the interaction of the different particle types in our framework. Granular particles form a characteristic pile of material. Wind interacts with the granular particles and sets them in motion (left). Dust particles form a layer of material on top of the pile (middle) and also react with the wind (right).

3.7 Erosion

Erosion occurs on the boundary of the object and it is caused by an erosion agent such as wind or water. We model erosion by coupling a physically-plausible fluid simulation with the layer-based data structure described in Section 3.5. In contrast to previous approaches, we couple the fluid simulation with granular materials. Both particle types are modeled based on an continuum approach described in Lenaerts and Dutré (Lenaerts & Dutre, 2009). Due to the Lagrangian setting we can detect local interactions efficiently while particles individually interact with the surface to cause different erosion behavior.

We capture the interaction between fluid and the object by accumulating the fluid shear stress τ for each surface voxel. The occurrence of the stress triggers the erosion process that resolves a stack into granular and dust particles. These particles are released at the location of the stack and immediately interact with the particles of the fluid simulation.

3.7.1 Erosion Rate

The shear stress τ is calculated by using the power law Wojtan et al. (2007):

$$\tau = \theta^{0.5},$$

where θ is the shear rate expressed in terms of a velocity of the fluid $\theta = v_r/l$, where v_r is the relative velocity of the fluid and l is the distance over which the shear is applied Krištof et al. (2009). The erosion rate ϵ depends on the shear stress and the critical shear stress τ_c Partheniades (1965): $\epsilon = \kappa(\tau - \tau_c)$, where κ is the proportionality constant. The amount of removed material in each surface voxel is then calculated as

$$\frac{dm}{dt} = \epsilon, \tag{3.7}$$

Note that we only need one material property τ_c that characterizes the material.

3.7.2 Particle Emission

The eroded material from Eqn. (3.7) is removed from the active voxel and converted to material particles that are advected by the fluid. The amount of material dm is converted to M particles. We assume each particle has a mass m_p so $M = dm/m_p$. The particles are emitted in disc-like clusters oriented in the direction of the reflected fluid assuming non-slip boundary.

The position $\mathbf{p}$ of the emitted particle is calculated as

$$\mathbf{p} = \frac{1}{n}\sum_{v \in V} v + \delta\frac{\mathbf{V_R}}{\|\mathbf{V_R}\|} + \gamma_1 e_1 + \gamma_2 e_2, \tag{3.8}$$

where V is the set of n vertices on the intersection of the surface stack and the implicit surface, $\mathbf{V_R}$ is the velocity vector reflected over the averaged normal of the

stack vertices, and δ is a user-defined scalar value that controls how far from the surface the particles should be placed. The values of e_1 and e_2 are unit sized vectors orthogonal to $\mathbf{V_R}$ and each other and γ_1 and γ_2 are two scalars generated from a normal distribution. The last define the spread of the particles that are emitted in a cone-like shape. Finally, the velocity $\mathbf{v}$ of the emitted particle is

$$v_{\parallel} = \cos(\alpha)\,|\mathbf{V_R}| \qquad (3.9)$$
$$v_{\perp} = \sin(\alpha)\,|\mathbf{V_R}|$$

The parameter α is a user-specified value that allows to approximate the behavior of different material types.

3.7.3 Surface Voxel Update

After the material removal from active voxels and particle emission, the amount of material in a voxel may reach zero. To track the morphological changes of the modeled object we dynamically adapt the data structure. First, we search the neighborhood of the fully eroded voxels and mark the stacks and height intervals that will become exposed to air layers when these eroded voxels are removed. We then remove the voxels, and subdivide all previously marked stacks.

The subdivision is an important step, as it makes sure that all the layers that are on the surface have unit height. Using these small layers on the boundary allows us to track changes in shape with greater detail and also emit and deposit eroded particles with better precision. The subdivision takes a stack of layers and height intervals to subdivide as input. It first divides the stack in unit sized layers and then fuses the layers of same material that were not marked for subdivision. For

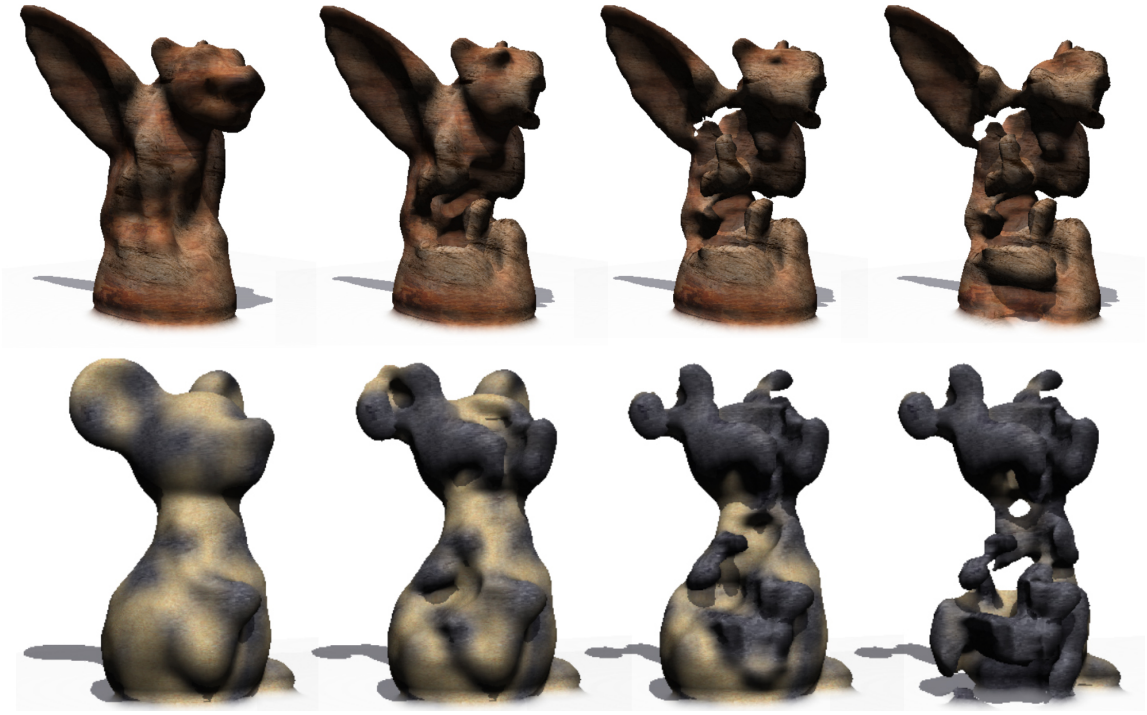


Figure 3.5. The erosion of a gargoyle and a mouse statue. SPH particles interact with the surface and slowly carry away material. We handle different material types and their distribution is defined by a noise function. The soft material erodes faster than the hard material.

each newly exposed layer, an exposed voxel structure is created to keep track of the amount of material.

Figure 3.5 shows the erosion of two statues. SPH particles interact with the surface and slowly carry away material. In our system we can handle different material types. In these examples we used a noise function for the distribution of a soft and a hard material. As can be seen, soft material erodes faster and the hard material remains.

3.8 Deposition

A complementary process to erosion is deposition that converts the freely moving material back to a solid volumetric object. The moving particles of material are advected by the fluid as described in Section 3.6. The material particles have their mass and if the movement of the fluid ceases, they sink down to the ground and naturally accumulate in characteristic locations (Figure 3.6).

Figure 3.6. Erosion and Deposition: the erosion of an input mesh (left) releases material particles that are transported by the fluid. The released material is represented by particles that deposit in the scene (right).

3.8.1 Particle Conversion to Surface Voxels

We simulate the material deposition of the freely moving material by voxelizing the particles that do not move for a certain period of time. In this way a new layer is deposited on the object surface. The newly deposited layer has material properties of an easily corrodible material.

A fixed particle is not allowed to move freely, but it requires a certain amount of stress to be released again. Over time a resting particle requires increased stress to be released. This is similar to natural deposition behavior; volatile sediment particles are transported, sink down, and slowly become more rigid. An example in Figure 3.7 show a complex form that has been modeled by a simple process.

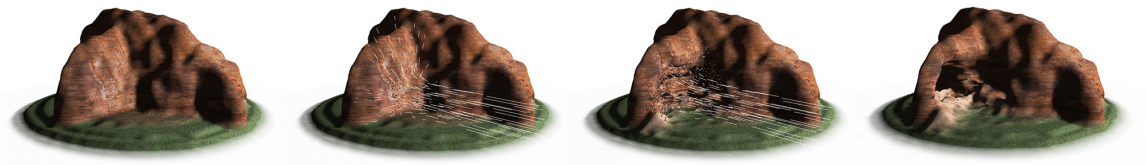


Figure 3.7. The particle-based modeling allows to dynamically track precise collision of particles with objects. Fluid particles (visualized as white stream lines) collide with the object and erode it. The material is transported by the fluid and deposits elsewhere.

3.8.2 Stack Update

Before a fixed particle can be deposited, we first check whether it has come in to contact with the surface. We calculate the local normal vector of the contact point and check whether it points in the direction opposite to the gravity in order to avoid deposition on faces pointing down. If the angle between the normal and

negative gravity direction is less than 45° the particle is marked as being ready for deposition.

During the deposition step, the voxel neighborhood of every surface is checked for fixed particles that are ready for deposition. The amount of material represented by these particle is added to the corresponding stack and the particles are removed from the fluid simulation. If the amount of material in an individual voxel reaches its capacity, we create a new voxel on its top and update its amount of material. This voxel is added to the active voxels. Finally, we update the corresponding stack and the list of surface particles. This provides us with a fine-grained level of control and lowers the number of required particles to model these phenomena.

Figure 3.6 exemplifies this process. The material is eroded from the surface of the object, the erosion process releases material particles that are carried away by the fluid. These particles have different characteristics than SPH particles and over time will deposit somewhere in the scene.

3.9 Implementation and Results

We have implemented our system in C++; all experiments were conducted on a desktop computer with an Intel i7 processor at 3.4 GHz and 16 GB RAM. Results were rendered with an Nvidia Geforce 780 GPU in our own graphics framework. We used OpenGL 4.2 for rendering and employed the compute capabilities of modern GPUs based on CUDA 7.0 for modeling.

We **render** the model interactively as a set of 3D boxes to get a rough idea of the model shape. Moreover, we have implemented a CUDA-based raytracer with ambient occlusion and shadows that allows to visualize the implicit surface with

textured materials. We trace the isosurface by sending rays from the camera and detecting a sign change in the isovalue (see Eqn (3.1) in Section 3.5.3). We then refine the surface point by a few iterations of Newton's method. The normals are calculated as the gradient of the isovalues.

The texturing is done similarly to Peytavie et al. (2009) by weighting the material contributions at a given point and blending their respective textures. The uv-coordinates for texturing are then calculated as tri-planar projection from the three world axes. The Oren-Nayar reflectance model has been used in all figures.

The particles are visualized as sprites by using alpha blending. Granular particles are visualized as polygonal objects using instanced rendering. Air particles are rendered as sprites with alpha blending. Trajectory lines of the air particles are rendered to capture their motion. Volumetric lighting for the dust particles is performed using the half-angle slicing method Fernando, Haines, and Sweeney (2001).

User Interaction: Even though our system can run completely automatically, it was designed with interactivity in mind and thus provides several levels of interaction. Although the implementation of fluids and granular materials depends on a number of parameters, it can still be used intuitively. The main parameters are the material properties of the input objects, each of those is specified once per polygonal mesh. During interaction, the user can adjust the intensity of the fluid (and its pressure) as well as viscosity, mass, rest density, and friction. Most of the parameters are defined by the system, the user only needs to define the material properties and to interact with the fluid emitters (see the accompanying video). Different particles are either created automatically, *e.g.,* dust particles are created when wind erodes an object, or directly infused by the user.

The fast response of our system makes it possible to use for interactive and physics-based sculpting of models. Figure 3.1 illustrates these capabilities. The user starts with a block of material and interactively carves out the dragon model while the material deposits automatically. It is important to note that the user can also handle the deposited material, as the deposition is not immediate. This is similar to blowing away a sand pile. Another example of virtual sculpting by using wind is shown in Figure 3.9. The user interacts with the framework to shape a lion statue. The input mesh is converted to our data-structure (a), that allows to efficiently track volumetric changes of the model (c) - (d). The system runs at interactive rates and thereby allows to immediately see the result of the modeling process (e).

Figure 3.8. Deposition without erosion is possible when the user only sprays the scene with material particles that deposit on the surface of the existing objects and convert into them.

Although we aim for physically-based modeling, our method also allows for erosion without deposition. Here the material is simply being removed from the system as soon as it is eroded. Figure 3.8 shows the opposite process, where the user blows the material into the virtual scene. The material dusts on the surface of the objects and eventually deposits and converts back to a solid object.

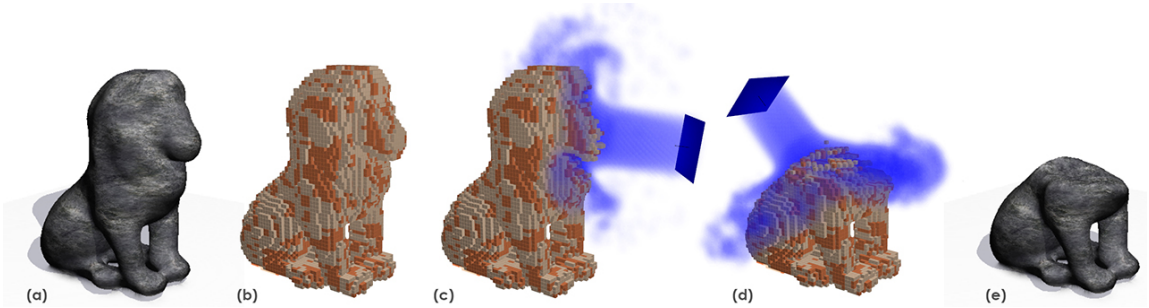


Figure 3.9. A lion statue interactively shaped with our system. The input model (a) is converted into our volumetric representation (b). The user interacts with the system by placing the emitter and by changing material properties and wind strength ((c)-(d)). Our system supports interactive rates allowing to immediately see the result of the modeling process (e).

Automatic mode: Our system can perform physically-based erosion and deposition in a fully automated mode with no interaction. In Figure 3.10 we show a large scene that has been eroded by a wind prevailing from one direction. The simulation forms patterns and objects commonly found in the nature. No user interaction was used in this experiment.

Performance: Table 3.1 shows computation times for the results shown in the paper using our non-optimized code. Depending on the model complexity we reach between $0.1 - 10$ fps, for larger scenes, with the raytracing-based rendering being the largest bottleneck.

Table 3.1.
Modeling and rendering times for the figures shown in the paper. Rendering times are based on a resolution of 1270x720px. For some figures we did not enable the deposition or erosion.

	Specification		Updates			
Figure	**Layers** (k)	**Particles** (k)	**Particles** (ms)	**Erosion** (ms)	**Deposition** (ms)	**Render** (ms)
3.1	18-28	30-65	17-29	5	0.46	860
3.3	40	-	-	-	-	827
3.5 (top)	76-81	20	18-21	5	-	519
3.5 (bottom)	72-80	25	22-25	7	-	525
3.7	101-110	50	14-16	6	0.47	414
3.9	82-88	20	22	8	-	726
3.8	27-40	20	21	-	0.5	834
3.10	14	80	75	2	0.49	1270

3.10 Conclusion

We have introduced a novel approach for shaping polygonal meshes by an interactive erosion and deposition modeling. The overall goal was to integrate erosion and deposition into creative frameworks by simplifying the physics and by providing interactive frame rates. Our system supports full 3D volumetric objects with multiple materials and enables modeling complex geomorphological phenomena, such as overhangs, arches, dust, and sand piles. We employed a dual SPH simulation that couples fluids and granular materials in a Lagrangian setup.

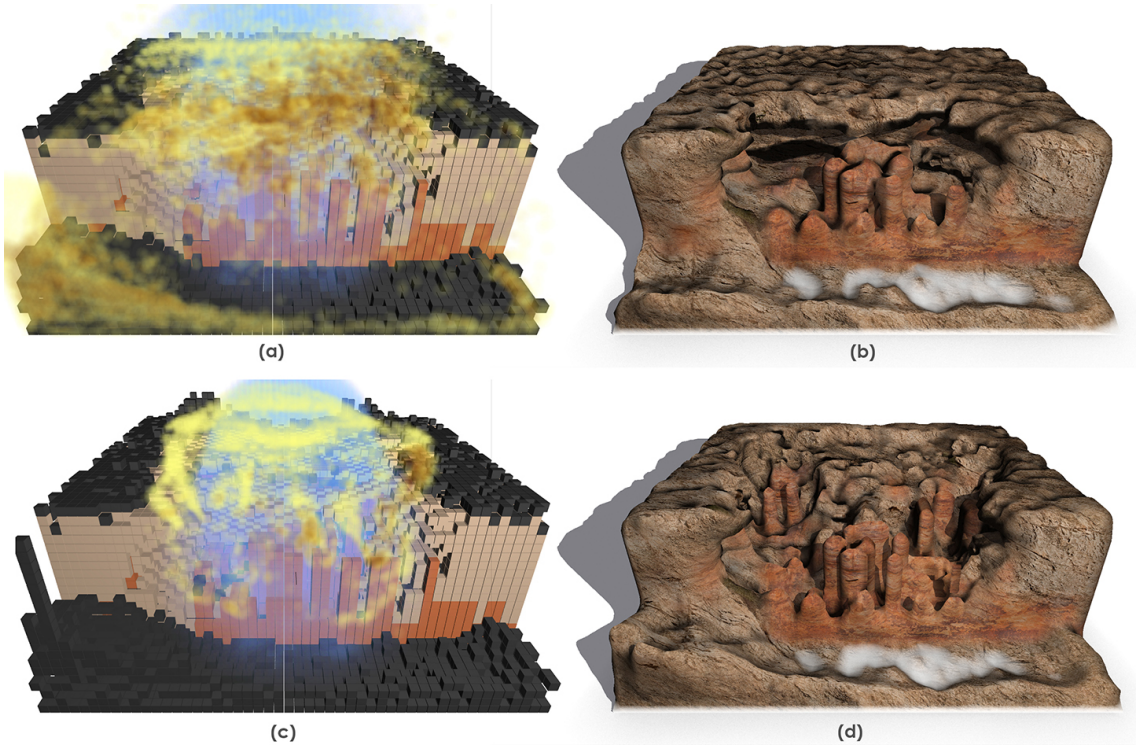


Figure 3.10. An automatically generated canyon scene. Granular, dust, and air particles interact with the layer-based data structure (a, c). The interaction of particles with individual voxels models the erosion and deposition of material that allows to create realistic scenes with a high level of complexity (b, d).

The proposed erosion and deposition model resolves material from a layer-based representation of an object into particles. Material particles are transported by the fluid and naturally deposited in the scene. Our approach represents material as voxels or particles and thereby allows for converting material from one state to another. Once particles accumulate in a certain location, we track their movement and eventually transform them back into volumes. This limits the number of particles and makes the computation efficient.

Compared to existing approaches we proposed a physically-plausible system that allows new modeling approaches. In particular, we have shown that modeling erosion and deposition of material can be used as new means for shaping and accentuating individual objects. Similar to real sculptures, the user can use fluid emitters to carve out naturally looking objects from a block of stone. Our system runs at interactive rates and thus enables users an immediate feedback loop for the modeling process. The individual parts of the system can be used independently; the user can use only the deposition or only the erosion of objects.

However, our system holds a number of limitations. First, our current data structure does not support kinematic physics and it allows to create geometry that does not follow proper constraints; *e.g.,* we can easily achieve objects hanging in space without any support that would fall in reality. Second, we only approximate the material properties and it would be important to actually provide real material measurements and implement them in our system. Although we did not go in this direction, this would certainly be possible and it would open an interesting avenue for incorporating real materials into computer graphics. Third, our current data structure is not dynamically adaptive. There is a limit of resolution and it would be interesting to increase this wherever possible and necessary. It would be also interesting to exploit adaptive volumetric structures for material representation and fluid simulation. Fourth, we did not focus on the usability of our implementation, but only on the development of the method. It would be interesting to exploit how and if users can quickly and intuitively generate their intent by using such approach. Finally, although the texturing used in our system is fast, it does not provide good visual results. A volumetric texturing would be more suitable.

CHAPTER 4. 3D CURVE SKETCHING

This chapter has been published and presented at *SIGGRAPH 2017* (Krs, Yumer, Carr, Benes, & Měch, 2017). Author's contribution include implementation and manuscript preparation.

4.1 Abstract

We introduce Skippy, a novel algorithm for 3D interactive curve modeling from a single view. While positing curves in space can be a tedious task, our rapid sketching algorithm allows users to draw curves in and around existing geometry in a controllable manner. The key insight behind our system is to automatically infer the 3D curve coordinates by enumerating a large set of potential curve trajectories. More specifically, we partition 2D strokes into continuous segments that land both *on* and *off* the geometry, duplicating segments that could be placed in front or behind, to form a directed graph. We use distance fields to estimate 3D coordinates for our curve segments and solve for an optimally smooth path that follows the curvature of the scene geometry while avoiding intersections. Using our curve design framework we present a collection of novel editing operations allowing artists to rapidly explore and refine the combinatorial space of solutions. Furthermore, we include the quick placement of transient geometry to aid in guiding the 3D curve.

Finally we demonstrate our interactive design curve system on a variety of applications including geometric modeling, and camera motion path planning.

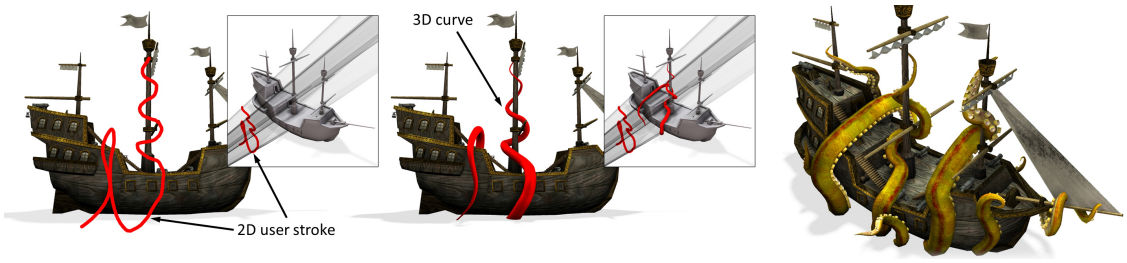


Figure 4.1. The user draws a 2D stroke in front of the 3D model (left). The 2D stroke is converted into a 3D curve (middle). A complete 3D model from multiple curves is generated within a few seconds (right).

4.2 Introduction

Computer graphics has achieved impressive results in areas such as rendering and computer animation. However, 3D modeling still poses many challenges; one of them is expressing user intent by simple means. User interaction, which is at the heart of modeling, is where most of the related complexity still exists. This is mainly due to the fact that most input and display devices currently in use are 2D, which is not intuitive to human interaction with the world, where we operate and think in 3D. To overcome the loss of depth, the user is usually forced to change the viewpoint, rotate the object, or use multiple viewports at once Bae, Balakrishnan, and Singh (2008); which can lead to a loss of efficiency.

A particularly difficult problem is drawing of 3D curves. These *space curves* are important to a variety of tasks such as planning of trajectories of dynamic objects, for example, particle systems or virtual cameras, design of curved surface patches,

such as NURBS, or swept surfaces, or generalized cylinders. The main problem of drawing 3D curves in 2D is that there is an infinite number of possible configurations of the curve in the missing dimension, and, as explored by Schmidt et al. (Schmidt, Khan, Kurtenbach, & Singh, 2009), even expert users have trouble with drawing 3D objects and curves. The foreshortening caused by perspective projection is especially difficult to get right and the resulting objects hardly match user's intent. While the use of shadows as visual depth cues has been shown to improve spatial understanding Cohen, Markosian, Zeleznik, Hughes, and Barzel (1999); specifying the shape of curves when drawing in 3D poses an additional set of challenges.

Prior work has addressed this issue by using constraints and additional sources of information to infer the desired curve's shape. One of the common problems not addressed by the previous work is drawing behind occluding surfaces or drawing curves with high curvature and torsion, such as spirals. One of the underlying mechanisms that makes this such a hard task is that our prior knowledge of the object's shape alters our perception Taylor and Mitchell (1997). As noted by Matthews and Adams (Matthews & Adams, 2008), "people seem to draw what they know rather than what they see".

One way to make drawing in 3D easier is to impose assumptions about the underlying form, such as regularity, symmetry, planarity, and orthogonality Bae et al. (2008); Kara and Shimada (2007); Schmidt, Khan, Singh, and Kurtenbach (2009); B. Xu et al. (2014). While these approaches work well in practice they are often restricted to a certain class of shapes, which may limit artistic expressiveness.

Reasoning about the 3D representation of the sketched curve using other 3D objects in the scene is a more practical and less disturbing approach from the user's

point of view since it does not require any change of viewport at the time of drawing. De Paoli and Singh (De Paoli & Singh, 2015) used this insight for modeling shapes around already existing 3D geometry. Their approach is limited to shorter and fully visible or partially visible symmetrical curve segments, which apply well to local shape modeling.

We present *Skippy*, a new method for sketching 3D curves from a single viewpoint using both existing or transient 3D geometry for shape inference. Our method enables the user to draw arbitrarily long, smooth curves that are placed at various depths between objects in the scene and that can also be partially occluded at authoring time (Figure 4.1). After being drawn the 3D curve can *skip* between different depths around the objects by clicking on the part of the curve that is visible or obstructed by an object. The user can also click on any surface and add transient guiding objects. Such temporary geometry is then also used to guide drawing of the curve without having to change the viewpoint. Furthermore, our system allows a quick intuitive way for users to guide and control the *skipping* behavior, allowing curves to be easily woven in a complex manner through the negative space surrounding any geometry. We do not place underlying assumptions about these curves (i.e., that they form surfaces, or represent regular geometric forms), and as such our curves can be used for a variety of applications from motion path planning, surface decoration, and even shape design. Finally we demonstrate a number of intuitive overdrawing mechanisms that enable iterative refinement of curve solutions interactively. Our main contributions are as follows:

- an intuitive approach for drawing 3D curves with varying depth using only 2D input by leveraging existing 3D shapes as guides,

- an efficient graph data structure that stores valid variations of the 3D curve for the input 2D stroke and enables real-time interaction with the curve, and
- a set of novel editing operations specifically for *skip* editing, re-drawing, and via anchoring through template 3D shapes.

4.3 Related Work

Positioning 3D curves using a 2D interface poses numerous challenges, a number of which have been tackled in the sketch based modeling literature. We refer the reader to Olsen, Samavati, Sousa, and Jorge (2009) and a recent survey Kazmi, You, and Zhang (2014) for a more complete overview of this domain. To highlight the most relevant work in this space, we divide related work into a set of broad categories; each relying on different sets of underlying assumptions which ease the 3D curve drawing process.

Design Curve Modeling. The early work of Cohen et al. (Cohen et al., 1999) presented a single view interface for designing 3D space curves. The novel idea of this work was to rely on shadows as additional depth cues. This conceptual idea is orthogonal to our approach and we leverage a form of shadowing (i.e., real-time screen space ambient occlusion) in our interface to improve spatial understanding. The work of Cohen et al. (Cohen et al., 1999) did not directly address the tedious nature of specifying curve shape during the drawing process.

One way to make drawing in 3D easier is to choose an angle that minimizes the foreshortening. A popular approach, investigated for drawing 3D curves, is to rely on epipolar geometry. This allows sketching from a second viewpoint to find a unique solution to the curve's shape Bae et al. (2008), or choosing two view

orthogonal directions Karpenko, Hughes, and Raskar (2004). Such approaches either require a symmetric structure to be drawn, or consistent change of viewpoint; limiting artistic expressiveness.

An important aspect that is often considered during object sketching are occlusions. Cordier and Seo (Cordier & Seo, 2007) construct self-occluding objects from free-form sketches using constrained optimization. Similarly, McCrae and Singh (McCrae & Singh, 2008, 2009) , use clothoid splines, to infer 3D road networks from sketches with self-crossings. McCann and Pollard (McCann & Pollard, 2009) order interactively 2D objects with local overlaps, which Igarashi and Mitani (Igarashi & Mitani, 2010) extended to 3D, and LayerPaint C.-W. Fu, Xia, and He (2010) allows users to paint on occluded surfaces using a multi-layer approach.

Another approach that helps with 3D modeling is leveraging existing geometry and environment. Coleman and Singh (Coleman & Singh, 2006) presented a method that adapts existing rough 3D curves to their surroundings. Turquin et al. (Turquin, Wither, Boissieux, Cani, & Hughes, 2007) allows users to sketch clothes on 3D mannequins from a single view by inferring the 3D position of sketched curves. Furthermore, 3D curves inferred from sketches have been used for hair design H. Fu, Wei, Tai, and Quan (2007); Wither, Bertails, and Cani (2007). OverCoat Schmid, Senn, Gross, and Sumner (2011) uses proxy geometry to embed brush strokes in 3D space and enables sculpting of the underlying proxy geometry with brush strokes. Perhaps most closely aligned with our method is that of De Paoli and Singh (De Paoli & Singh, 2015) who presented SecondSkin. Rather than allowing curves to be drawn directly *on* a surface, it allows artists to sketch design curves in the nearby shell offset space surrounding an object; providing many

interesting design operations. In contrast, our system allows the user to simultaneously sketch over large collections of shapes, deals with dense occlusions, specifies curves that are further from the surface, all without changing view.

Organic Surface Modeling. The ambiguity of taking 2D sketch curves and producing 3D content can be reduced by assuming that the sketch curves represent some underlying *organic* form. Both the seminal work of Teddy Igarashi, Matsuoka, and Tanaka (1999) and Fibermesh Nealen, Igarashi, Sorkine, and Alexa (2007), begin the design process by assuming the curves reside on silhouette edges of some inflated base shape. Karpenko et al. (Karpenko et al., 2004) presented SmoothSketch which extended this notion by analyzing T-intersections and cusps in the drawing to enable the creation of more complex base shapes. Initial 3D base forms can be used to anchor more complex curve sketching operations. For example curves drawn directly *on* a surface can be pulled and tugged to deform the underlying shape. These surface curves can also be used as localized regions for extrusion Igarashi et al. (1999); Nealen et al. (2007). Both these works demonstrate that the presence of an existing 3D shape can bootstrap the 3D drawing process enabling the creation of more complex form. Recently, 2D curves have been used to construct 3D cartoon canvases in Bessmeltsev, Chang, Vining, Sheffer, and Singh (2015). We take inspiration from these works, however, we focus our attention on populating the empty space between shapes allowing our designers to bootstrap the 3D curve design with the types of complex 3D models that can easily be found on the web or in shape repositories.

Surface Modeling using Curve Networks. An alternative approach to aid users in creating 3D space curves is to assume the curves represent underlying man-made structure. The SKETCH interface Zeleznik, Herndon, and Hughes

(1996) starts by allowing the user to sketch box like forms which anchor additional 3D space curve sketching operations. Schmidt et al. (Schmidt, Khan, Singh, & Kurtenbach, 2009) uses sketching on an initial ground plane to build a scaffold which acts as a set of visual constraints for sketching additional 3D curves. Photographs have also been used in conjunction with sketches to disambiguate form Lau, Saul, Mitani, and Igarashi (2010). Xu et al. (B. Xu et al., 2014) infer 3D curve networks from a given 2D sketched design by assuming implied regularities, such as planarity, curvature, symmetry, and parallelism. In contrast, we target our system at freeform space curves which may or may not directly participate in defining some underlying surface.

3D Drawing using Shape Priors. By restricting the class of target shapes to conform to some underlying model (i.e. procedural or otherwise), many drawing operations can be simplified. Just recently, deep neural networks were used to automatically infer 3D architectural procedural models from 2D user single view sketches Nishida, Garcia-Dorado, Aliaga, Benes, and Bousseau (2016a). Chen et al. (X. Chen, Neubert, Xu, Deussen, & Kang, 2008) used Markov random fields with sketching to reconstruct 3D models of vegetation. Automatic character model reconstruction from 3D sketches has also been demonstrated Buchanan, Mukundan, and Doggett (2013). Drawing assistance and recommendation can also be achieved using large pre-existing 3D shape collections. For example, shadows have been used to guide the users during drawing by leveraging a database of 3D template objects Fan, Wang, Xu, Deng, and Liu (2013). While the use of strong shape priors can greatly enhance 3D drawing, they can also potentially limit artistic freedom. Our system focuses more on freeform design and does not impose such restrictions.

4.4 Method Overview

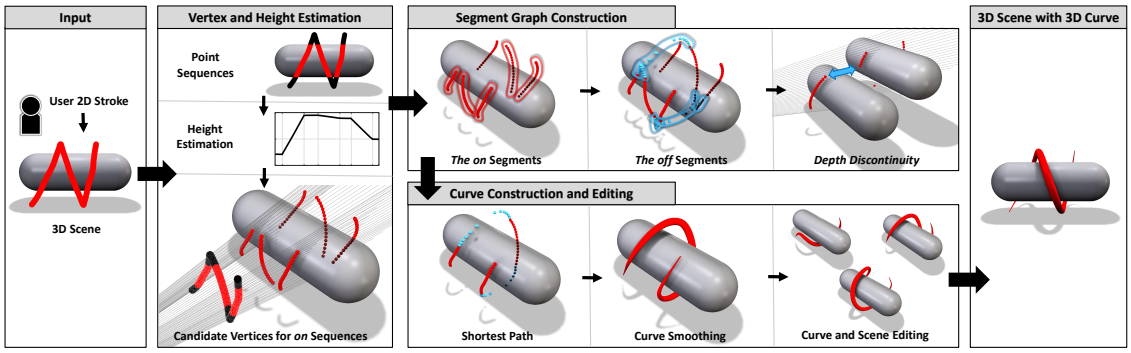


Figure 4.2. Skippy overview. A 3D scene is displayed from a single view and the user draws series of 2D points that are converted into a 3D curve that passes through the scene. First, 3D candidate vertices are found for intersecting parts of the stroke. A segment graph is built from groups of the candidate vertices and is used to store all possible valid curves. The best curve is selected and converted into a full 3D curve. The user can quickly select different curves and skip from one solution to another.

Our goal is to infer a 3D curve shape from 2D strokes so that it follows the 3D scene geometry, with optional transient guide objects that can be added by the user. The user's 2D stroke is converted to a set of equidistant points in the view plane. Given these points and distances to geometric surfaces in the scene, we estimate 3D locations. This process may generate multiple candidate 3D locations for each 2D sample. Based on a curvature criterion we then find a combination of these 3D points that makes up the 3D curve.

Figure 4.2 shows an overview of our method. The input is a 3D scene that may also contain a set of transient objects. The 3D curves are input as sequences of strokes converted into sets of equidistant points in a 2D viewing plane. The output is a 3D scene that includes both the input scene and the added objects. The only requirement for the 3D scene representation is that it must be possible to find a

distance field around it, since it is needed for fast distance calculations. After a stroke is drawn, the user can change the viewing direction and the curve is extended as the user continues to draw more strokes or modified when the user overdraws an existing part of the curve. We infer a 3D curve for each set of 2D strokes, so that the user can instantly see the modification during the sketching process.

The ***vertex and height estimation*** (Section 4.5) is the very first step of our pipeline. Note that we denote the 3D counterparts of 2D stroke points as vertices. The first step takes the input scene and the input 2D points and finds a set of candidate 3D vertices in 3D space for each 2D point and their distance from the geometry that we call the vertex height. The candidate vertices are the possible locations of the point in the 3D space. In order to create this set, we first project the set of 2D points into the 3D scene by constructing a set of rays, and classify the 2D points based on their intersection with the 3D objects as *on* and *off* points. We then use the non-intersecting rays to estimate the height for each sequence of *on* 3D vertices. Given the rays and the estimated heights, we can find a set of candidate vertex sequences for the *on* 2D points.

The ***segment graph construction*** (Section 4.6) step creates a graph of vertex segments at varying depths that is later used to find the optimal 3D curve. Segments are 3D polylines connecting given vertices. First, the segments corresponding to *on* points are constructed, forming the nodes of the segment graph. The edges of the segment graph represent the parts of the input stroke that did not intersect the geometry. These edges, which we call *off* segments, are constructed between individual *on* segments. An additional step is performed to handle depth discontinuities. This step adds nodes and edges to the segment graph that help to find a better solution.

The ***curve construction*** (Section 4.7) step, finds the best path through the segment graph and constructs a smooth cubic spline. First, both the segment graph nodes and edges are scored using a curvature criterion. Then the best path through the segment graph is found and the segments along this path are concatenated into a single curve. Finally, the curve is re-sampled and iteratively smoothed. Additionally, editing operations such as change in depth or redrawing are facilitated.

4.5 Vertex and Height Estimation

The input to our method is a 3D scene and a set of strokes that are sampled into sets of 2D points.

The first step of the pipeline takes the sequence of the input 2D points and the geometry and splits it into a sequence of *on* and *off* points that lie on the scene geometry (Figure 4.3). Afterwards, it generates the candidate vertices in 3D for the *on* sequences.

4.5.1 Point Sequences

The sequence of 2D input points is provided by the user in a single stroke. The input point sequence have varying distance, therefore we resample them so that the new sequence is equidistant. We denote the new point sequence by $P = (p_1, p_2, \ldots, p_{|P|}) \mid p_i \in \mathbb{R}^2$, where $|P|$ is the number of points.

We then divide the sequence of points into points that are *on* and *off* geometry after projection (Figure 4.3 and see also (De Paoli & Singh, 2015, page 4)). We

perform an initial projection of the 2D points p_i by casting a ray with direction r_i from the camera position c and finding intersections with the objects in the scene.

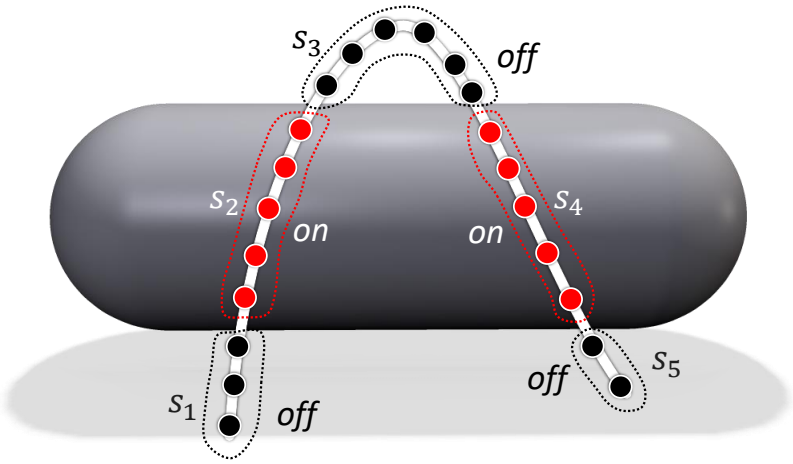


Figure 4.3. The input 2D stroke is divided into on and off sequences.

The rays that intersect the geometry will define *on* points while the non-intersection ones will determine *off* points. There is an implicit ordering of the rays r_i that is defined by the sequence of the input points p_i. Therefore, there is also an implicit ordering of the vertices v_i in the 3D space. Moreover, the points that are *off* geometry will later be used to determine the distance of the final curve from the actual object. The points are then grouped into successive sequences $S = \langle s_1, s_2 \ldots, s_{|S|} \rangle$ with a flag whether it is *on* or *off* .

A special care needs to be taken for sequences of *on* points with *depth discontinuity* such as in Figure 4.4. The situation indicates multiple obstructed surfaces and the *on* sequence needs to be split in two. In our implementation we parse all *on* sequences and we perform a check for each pair of subsequent points by comparing the distance of the ray intersections. If the intersections are far apart relative to

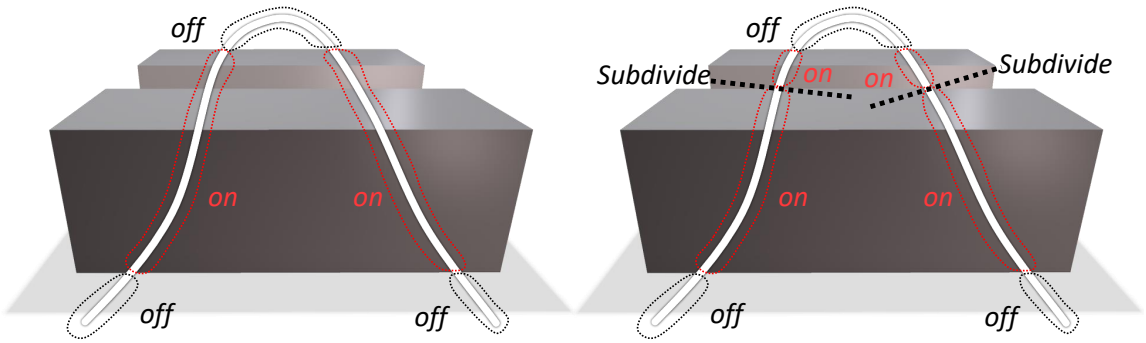


Figure 4.4. The sequence of on 2D points with depth discontinuity (left) is divided into two on sequences (right).

the distance of their screen coordinates (three times or more in our implementation, assuming unit cube workspace and unit square screen), we check whether the slope between those two intersections is continuous. We cast an additional ray in between the successive points and check if the new intersection's distance is close to the average of the distance of the involved points (40% relative depth change or less). Otherwise if the distance of the intersection of the new ray is closer to either of the two vertices, the sequence is split in two.

4.5.2 Height Estimation

Our goal is to draw the curve at a certain distance from the geometry. Although it would be possible to ask the user for an explicit distance value input, we use a more intuitive way to infer the distance from the actual stroke. In particular, the distance of the curve from the geometry is derived from the distance of the rays defining the *off* points.

We call the distance of the final curve from the geometry its *height* and we denote it as h. Height is a function that returns the value for an input point p_j or a sequence s_i. All points in an *off* sequence s_i have their height $h(s_i)$ constant. It is found as the maximum distance of the rays that define the segment from the scene geometry (Figure 4.5). The height of an *on* sequences is found by the linear interpolation of the heights of the neighboring *off* sequences (Figure 4.5). If the user starts or ends drawing on the object's surface the *on* sequence does not have two neighboring *off* sequences and we set the start or the end of the corresponding *on* sequence to zero.

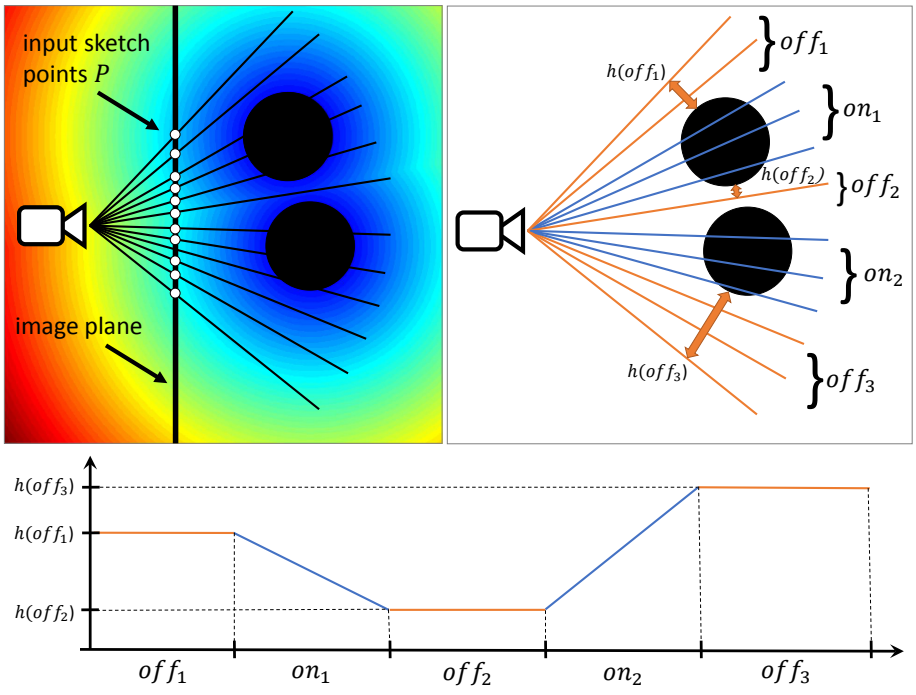


Figure 4.5. Height of the off sequences is constant and given by the color-coded distance field (left). Height of the on sequences is interpolated from the neighbors.

After this step all *off* sequences have constant height $h(s_i)$ and all *on* sequences have their height interpolated.

The result of the distance estimation step is a mapping of the input points to the distance from the scene geometry.

4.5.2.1 Candidate Vertices for *on* Sequences

Next we find candidate vertices v but only for *on* sequences; the *off* sequences are processed differently in Section 4.6. We generate a distance field *df* for the entire scene by using the L_2 norm (Figure 4.5 top left). The distance field significantly speeds up the distance calculations.

In order to generate the candidate vertices, we again cast a ray r_i for each point from each *on* sequence. The candidate vertices are found as intersections with the isosurface at distance $df(h_i)$.

This step generates a large number of candidate vertices, some of which are unnecessary and can be removed. In particular, if we encounter two volumes in a row intersecting the ray, the space in-between them will be filled by two candidate intersections: one for the "after" the first object and one for "before" the second one. However, in practical experimentation the user usually does not need to have such a small level of refinement and one point is usually enough for curve editing. We choose to discard the point that is close to the back-face in our implementation, shown as "Discarded (in-between)" in Figure 4.6. The user may not expect the multiple points in the middle that may cause jumps in depth in the 3D curve construction. Moreover, this step prunes the amount of vertices and speeds up the

computation. However, all discarded vertices are kept in the system and can be later accessed by editing operations.

The candidate vertices for *on* sequences are denoted v_i^j and are indexed in two ways. The lower index (Figure 4.6) corresponds to the index of the ray r_i that is also given by the index of the point p_i in the re-sampled input stroke. The upper index j is the ordering number of the intersection on the ray, with zero being closest to the camera. Moreover, we also assume all vertices in the first *on* sequence are not occluded i.e., the user starts drawing either off or in front of the geometry.

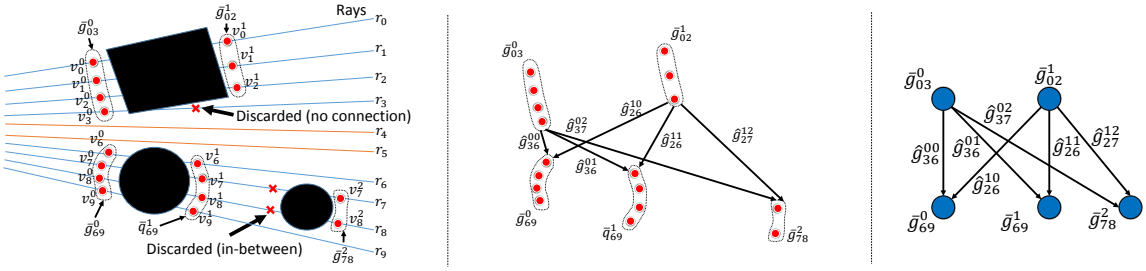


Figure 4.6. Segment creation and indexing from the candidate vertices for on segments (left), the possible connections of the on segments (middle), and the corresponding segment graph (right).

4.6 Segment Graph Construction

The previous step created candidate vertices for *on* points and identified rays that do not intersect the geometry (potential *off* points). It also created the height function for all *off* and *on* sequences.

In order to generate the curve, we could consider individual combinations of all vertices. However, a ray $r_i, i = 1, 2, \ldots, n$ can generate multiple candidate

vertices $\langle v_i^0, v_i^1, \ldots, v_i^k \rangle$ and the number of possible combinations greatly increases with each added ray and possible depths at which the candidate vertices can lie. The number of possible curves increases with $\mathcal{O}(k^n)$. Therefore it is not feasible to calculate the curve for every combination. Fortunately, many of the combinations can be trivially rejected, for example, the connecting edge should not intersect geometry or they should have a similar distance from the camera (see details in Section 4.6.1).

We group candidate vertices into *segments* denoted by g. A segment is a 3D polyline that connects candidate vertices. We further classify the segments into *on* segments $\bar{g}$ and *off* segments $\hat{g}$.

After all segments were found we construct a *segment graph* that includes all possible valid (following constraints listed in Section 4.6.1) curves in the 3D space for the given stroke. The segment graph $\mathcal{G} = \{N, E\}$ has a set of nodes N that correspond to the *on* segments and the edges E correspond to the *off* segments (Figure 4.6 right). From the segment graph we automatically offer the best segment and let the user select a different one during the scene editing, for example, to skip part of the curve to a different depth.

The inputs to the segment graph construction algorithm are the scene geometry, the *on* candidate vertices, and the *off* rays. The segment graph construction is a three step process that is described in detail below. First, we connect candidate vertices for each *on* sequence on the corresponding side of the geometry and create *on* segments. Then we connect the *on* segments by generating *off* segments. There is a special case of the depth discontinuity (Figure 4.4) which we further discuss in Section 4.6.3.

4.6.1 The *on* Segments

We create the *on* segments by connecting all candidate vertices of *on* points. Each *on* segment is denoted $\bar{g}^j_{se}$, where j is the intersection order as above, and the lower index se denotes the start and then the end vertex. For example in Figure 4.6 we have $\bar{g}^1_{69} = \langle v^1_6, v^1_7, v^1_8, v^1_9 \rangle$.

These segments will become the nodes N of the segment graph $\mathcal{G}$ (Figure 4.6 right). We create the *on* segments by connecting individual candidate vertices of consecutive *on* points and then group them into the longest segments that can be found. Individual connections of candidate vertices are tested against three criteria: 1) the connection does not intersect the geometry (example in Figure 4.6: "Discarded (no connection)"), 2) the vertices lie at similar distance from the camera, and 3) the gradient of the distance field is similar at the vertex position. We then find all segments that start at a candidate vertex with no inbound connections and end at a candidate vertex with no outgoing connections.

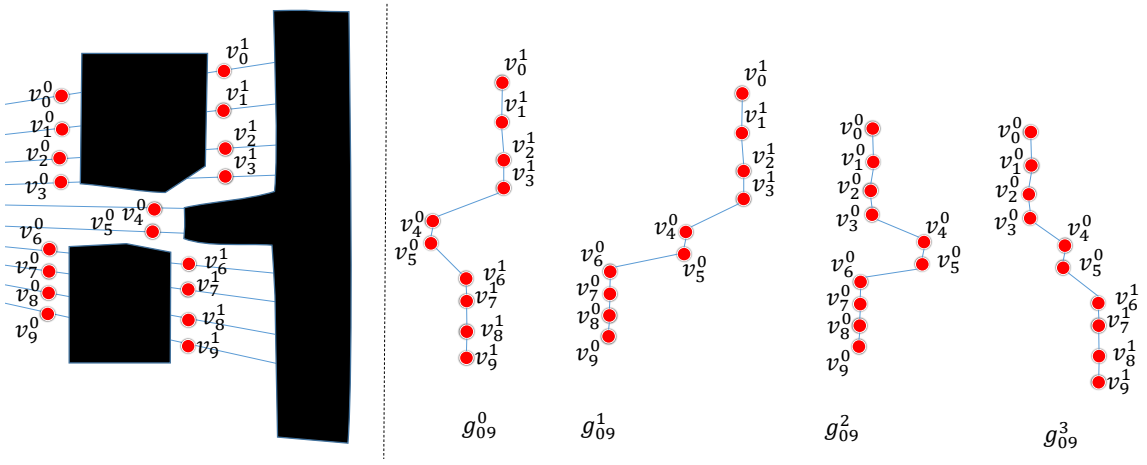


Figure 4.7. Depth discontinuity of an on segment (left) will lead to multiple on segments (right).

While this construction is simple for a short sequence of vertices on a single side of geometry, it can be more complicated in cases of depth discontinuity (Figure 4.4). Each ray can generate multiple candidate vertices at different distances (vertices with varying upper index in Figure 4.7), but the discontinuity will tend to merge and split the segments that would generate large zig-zag steps in the 3D curve. By applying the above-described construction we obtain an acyclic oriented graph (the direction of the stroke defines the orientation). From this we extract all segments that start from the candidate vertex with the lowest index and end in the candidate vertex with the highest one ($start = \{v_0^0, v_0^1\}$, $end = \{v_9^0, v_9^1\}$ in Figure 4.7). Because of the varying depth of the vertices in each segment, the upper index of the *on* segment is given by the order in which it was extracted ($g_{09}^0, g_{09}^1, \ldots, g_{09}^3$ in the example in Figure 4.7). We store *all* those segments as nodes of the segment graph, because they provide alternatives for the final curve construction. A key idea behind our approach is that we can select the subset of curve segments that lead to a desired outcome (e.g., a final curve that is smooth with a monotonic curvature).

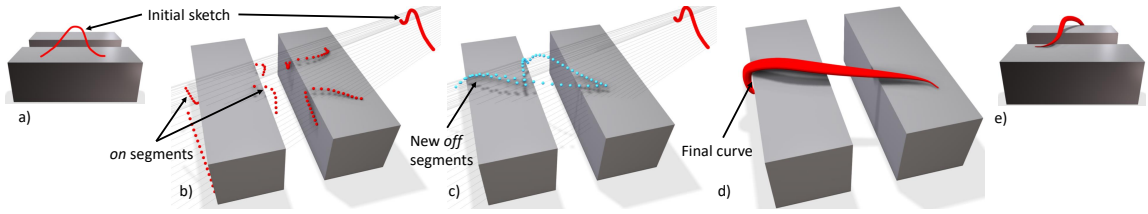
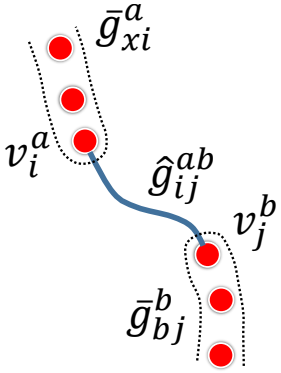


Figure 4.8. When depth discontinuity occurs (a), we break the conflicting on segments (b) and insert new off segments that bridge the discontinuity gap (c) and allow for smooth curve generation (d-e).

4.6.2 The *off* Segments

After all *on* segments are constructed we can connect them by constructing the *off* segments. Because there are multiple combinations on how the segments can be connected, we use the graph $\mathcal{G}$ where the *on* segments are its nodes and *off* segments correspond to the graph edges as shown in an example in Figure 4.6 middle. Each *on* segment (left) is connected to all segments that are accessible by skipping a sequence of *off* rays. Every such connection is an edge in the segment graph (right).



The *off* segment $\hat{g}^{ab}_{ij}$ corresponds to an edge in $\mathcal{G}$ that connects two *on* segments $\bar{g}^a_{xi}$ and $\bar{g}^b_{jy}$. More precisely, the *off* segment $\hat{g}_{ij}$ connects the last vertex v^a_i from the first *on* segment with the first vertex of the second segment v^b_j. Note that we need the upper indices for an *off* segment, because the vertices v_i and v_j can be at different depths that would cause multiple *off* edges with the same lower indices.

To find the *off* vertices of the *off* segment $\hat{g}^{ab}_{ij} = \langle w_{i+1}, \ldots, w_{j-1} \rangle$, we interpolate the depth of v^a_i and v^b_j. In other words, we interpolate between $d_i = \|c - v^a_i\|$ and $d_j = \left\|c - v^b_j\right\|$, where c is the position of the camera.

The input sketch of the *off* segment defines a surface passing the rays $\langle r_{i+1}, \ldots, r_{j-1} \rangle$ over which we interpolate the depth. Since the user is free to draw virtually any shape, such as loops, zig-zags, etc., we cannot always guarantee the *off* segment to be smooth. For most cases, we use linear interpolation of d_i and d_j to calculate the vertices (Figure 4.9 bottom). This produces a reasonably smooth

segment, especially if the input sketch is also smooth. However, the linear interpolation fails when there is a sharp corner in the input sketch. We detect the corner by using the algorithm from Ben-Haim, Harary, and Tal (2010) and use sigomoidal interpolation that achieves a smoother result that is closer to a *circular arc* (Figure 4.9 top)

$$d(t) = d_i(1 - \pi(t)) + d_j\pi(t),$$

where $\pi(t)$

$$\pi(t) = \begin{cases} -\frac{1}{2}\sqrt{1-(2t)^2} + \frac{1}{2} & \text{if } t \in [0, 0.5] \\ \frac{1}{2}\sqrt{1-(2t-2)^2} + \frac{1}{2} & \text{if } t \in (0.5, 1]\,. \end{cases} \tag{4.1}$$

Furthermore, an *off* segment can begin on the first ray or end on the last ray. In this case we do not use interpolation. For an *off* segment $\hat{g}^{xa}_{0i}$ beginning on the first ray we calculate the average change in depth $\overline{\Delta d}$ of several vertices of the following *on* segment $\bar{g}^{b}_{iy}$. (Up to four vertices were used to calculate $\overline{\Delta d}$ in our implementation.) We then extrapolate the vertices of $\hat{g}_{0b}$ as follows

$$w_q = c + (d_i + (i - q)\overline{\Delta d})r_q. \tag{4.2}$$

The case of *off* segment ending on last ray, $\hat{g}^{ay}_{i|P|}$ is analogous and the *on* segment used is the previous one $\bar{g}^{a}_{xi}$. To connect these outer *off* segments and maintain the graph structure, we add a node $\bar{g}^{0}_{00}$ or $\bar{g}^{0}_{|P||P|}$ at the beginning or end, respectively, that have zero length.

4.6.3 Depth Discontinuity

The last case is a treatment of depth discontinuity (Figure 4.8). The direct connection of the consecutive *on* segment would generate sharp corners with large

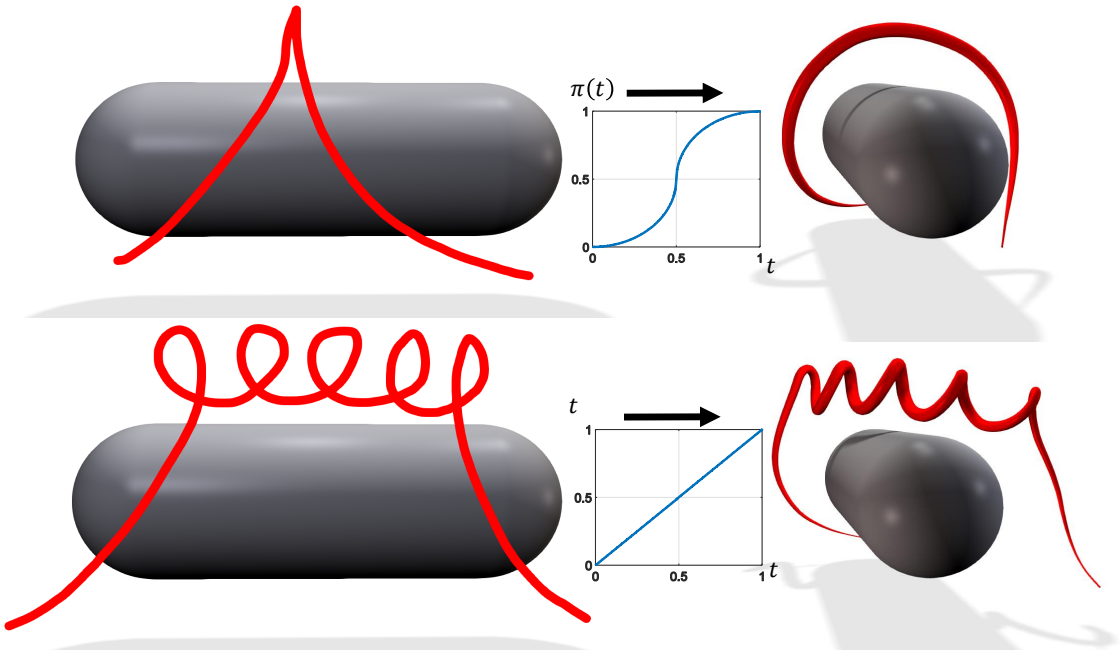


Figure 4.9. Off segment interpolation. An off segment with a single corner in initial sketch (top) is interpolated by Equation 4.1. Otherwise linear interpolation is used (bottom).

depth jumps. To avoid this situation we generate new *off* segments that bridge the *on* segments that participate in depth discontinuity as shown in Figure 4.8. The *off* segments are added between all *on* segments that are accessible by skipping an *on* sequence that includes a depth discontinuity. Note the new *off* segment is in fact parallel to the *on* segment but connects the *on* segment that is behind the second object. When multiple depth discontinuities occur, the curve generation algorithm described in Section 4.7 will select the smoothest path in the graph that corresponds to the curve passing behind the last object. This is also the typical intuitive choice of the users, but it can be overridden if needed.

4.7 Curve Construction and Editing

The previous step created a graph $\mathcal{G}$ that contains interconnected *on* and *off* segments (Figure 3.2). A number of curves can be generated by finding paths in $\mathcal{G}$ that go through all the rays of the initial sketch. However, not all curves have good visual properties. We use the intuition that the curve should be smooth and follow the curvature of the underlying geometry. Figure 4.10 shows that if the geometry is round the curve will likely go behind the object.

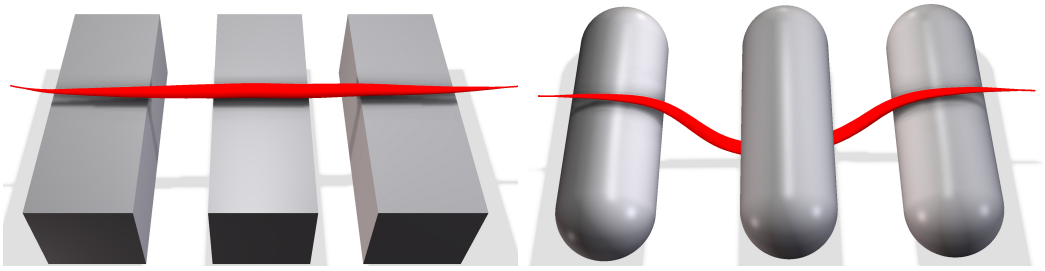


Figure 4.10. The path selection attempts to keep the curvature of the off segment constant. In this way the underlying geometry navigates the direction of the curve. The curves were sketched from top-down viewpoint.

Therefore, to select the best path, we define a weight for the nodes and edges of $\mathcal{G}$ that is based on curvature of the individual segments. When a best path through the graph is found, we connect the traversed segments into a single curve. Finally, the curve is resampled and iteratively smoothed, while making sure that the final curve does not deviate from the initial sketch and the height of the curve is preserved.

4.7.1 Optimal Path

We are not aware of any method to determine the *best* curve based on the above-described criteria. A common approach is to select a curve that does not deviate in depth but our objective is to create curves with varying depth. Another way to select the curve is by considering its fairness as explored by Levien and Séquin (Levien & Séquin, 2009), where the authors noted that one of the indicators of a fair curve is its monotonic curvature.

Since we stitch the curve together from segments, we need a criterion that can be evaluated independently for each segment and reused for different final curves. We chose the criterion to be *integral of absolute change in curvature* that we denote K:

$$K = \int |\kappa(s)'| ds, \kappa(s) = \|T'(s)\|, \tag{4.3}$$

where s is the arc length parameter of the curve and $\kappa(s)$ is the curvature defined by using the unit tangent vector T.

The minimization of this criterion favors curves with monotonic curvature. Whenever the underlying geometry causes the curve to turn, this criterion prefers the curve to keep turning, which often results in the curve wrapping around the object (Figure 4.10).

We estimate the discrete curvature for each *on* segment $\bar{g}^a_{ij}$ from its vertices $\langle v^a_i, v^a_{i+1}, \ldots, v^a_j \rangle$. The curvature of the *off* segments is calculated from its vertices but also from the last and first vertices of the connecting *on* segments (inset Figure in Section 4.6.2). This makes sure that any sharp connection between *on* and *off* segments is penalized.

The curvature is estimated differently for input strokes that form a loop. If the first p_0 and the last point $p_{|P|}$ of the input 2D stroke are within a small distance, we merge the first and last segments and the curvature is estimated for this merged segment.

The previous step assigned the weights to nodes and edges of the graph. In this step we find the path through the graph that will represent the final curve. Such a path has to start with a segment that includes the vertex for the first ray, i.e., any segment $\bar{g}^a_{0x}$, and end with a segment that includes the vertex for the last ray, i.e., any segment $\bar{g}^b_{y|P|}$. In the case of beginning or ending the stroke *off* the geometry, recall that these segments can be zero length ($\bar{g}^0_{00}$ or $\bar{g}^0_{|P||P|}$). The graph is implicitly topologically sorted, therefore we simply do a depth first traversal from the nodes starting at first ray and perform edge relaxation, noting the best predecessors at each node. To construct the curve we simply retrace the best predecessors from all nodes ending at $|P|$ and concatenate the segments. In our implementation we use Catmull-Rom splines to construct the final curve geometry.

4.7.2 Curve Smoothing

The previous steps generate a 3D curve that follows the geometry but may have some sharp turns. To improve its quality it is resampled and iteratively smoothed.

Recall that the 2D points are equidistant in 2D (Section 4.5.1). However, when projected to 3D, the distance between successive vertices of the 3D curve is not constant so we further resample the curve in 3D so that the distance between vertices is constant.

We use the active contours approach Kass, Witkin, and Terzopoulos (1988) to smooth the 3D curve with two additional constraints. First, similar to Kara and Shimada (2007), we make sure that the final curve's projection to the sketching viewpoint is similar to the sketched 2D curve. Second, we preserve curve height h that was defined in Section 4.5.2. To smooth the curve we minimize the energy of the curve E by using the gradient descent:

$$E = \int_0^1 \left(E_{internal}(s) + E_{external}(s)\right) ds. \tag{4.4}$$

The internal energy is the sum of continuity and smoothness energies:

$$E_{internal}(s) = \alpha \left\| \frac{dv(s)}{ds} \right\|^2 + \beta \left\| \frac{d^2v(s)}{ds^2} \right\|^2, \tag{4.5}$$

where $v(s)$ is the position of the curve at arc length parameter $s \in (0, 1)$. The external energy is defined as:

$$E_{external}(s) = \gamma |r(v(s)) \cdot \Gamma(s)| + \delta |d(v(s)) - h(s)|, \tag{4.6}$$

where $r\left(v(s)\right)$ is the direction of the ray from the camera to $v(s)$, $\Gamma(s)$ is the direction of the ray from the initial sketch at s, $d\left(v(s)\right)$ is the distance of $v(s)$ the geometry, and $h(s)$ is the height of the curve at s. The α, β, γ, δ are respective weights of individual terms and $\alpha + \beta + \gamma + \delta = 1$. We use $\alpha = 0.0039$, $\beta = 0.011$, $\gamma = 0.982$, and $\delta = 0.0019$ in our implementation, which prefers smoothness over equidistance of vertices and penalizes even a small deviation from the input sketch.

4.7.3 Curve and Scene Editing

Skippy offers by default a smooth curve that keeps the distance from the surface as defined by the user strokes and follows the surface geometry (please see the

accompanying video). However, this may not always be the user's preferred choice and we can easily provide alternative curves that are stored in the segment graph $\mathcal{G}$.

Stroke modifiers allow the user to change the selected curve while drawing. Using a bidirectional gesture such as a mouse wheel, the user can change the depth of the last inferred segment. The depth is only a suggestion, as a further stroke points can change the configuration of the curve. Furthermore, a modal gesture, like a key being pressed, can disable intersections. This is particularly useful in dense scenes, such as in Figure 4.14, since otherwise the system automatically assumes that any geometry intersecting the stroke will have effect on the final curve.

After the stroke has been finished, the user can **modify the curve** by redrawing its part or changing the depth of certain segments. The redrawing can be done from any viewpoint and is performed by inferring a new curve from a redraw stroke. This new curve has its end vertices fixed to match the closest vertices of the original curve and is used to replace the part of the original curve that is being redrawn. Furthermore, the depth of individual parts of the curve (the *on* segments) can be selected manually, again by using a bidirectional gesture such as the mouse wheel as shown in example in Figure 4.11 and in the accompanying video.

Figure 4.11. Once the segment graph has been calculated the curve can be modified by simply clicking on the 2D stroke or the geometry covering the stroke to change its depth.

We found that **transient geometry** is a powerful way to model the curves. Transient objects work as scaffolds for the 3D curves. We allow the user to add a transient object, defined as a mesh, by clicking on a surface of an existing one. The object is placed such that its vertical axis is perpendicular to the surface and the mouse wheel controls its scale. If there is no object in the scene, the transient object is placed to the origin. Once the transient object is not needed it can be removed by shift click. Similarly, any new 3D geometry created around the space curves generated by Skippy can act as a transient geometry and can be removed at any stage. In Figure 4.18, several transient objects were used to produce snakes that stay farther from the head. The process in Figure 4.13 is similar, except that the initial transient sphere was placed at the origin.

4.8 Implementation and Results

Our system was **implemented** in C++ with OpenGL and glm and we tested our results on an Intel-based desktop computer with Intel Xeon E5-1630 @ 3.7GHz, 12GB of DDR4 RAM and NVIDIA GeForce GTX TITAN X.

Timing of our application depends on several aspects. The most important is the number n of the resampled points p_i and the number of the triangles of the input geometry. The speed of the application also depends on the number of ray intersections, i.e., multiple occlusions. This generates multiple *on* segments that add to the number of combinations of the *off* segments (edges of the segment graph). The speed of the application also depends on various variables such as the distance field resolution (we use an octree of depth seven) and curve discretization (we use screen distance of 5-15 pixels). The distance field needs to be modified

when temporal geometry is added or deleted. To speed up the calculation we store individual distance field for each object, thus the recalculation depends only on the added geometry.

We report the timing of the individual steps of our application for all results in Table 4.1. The timing is for the last step of the model creation, when all the 2D strokes are present in the scene and the scene is most complex. In order to get to the most complex case, the user goes through a sequence of simpler scenes as shown in example in Figure 4.12 that reports timing of the design process during a model creation for two different objects. It can be seen that the number of multiple rays (holes) greatly affects the calculation time.

The *authoring* column of Table 4.1 reports the time necessary for the design of each model, when the user may undo some actions, erase wrong parts etc. Overall, the object creation was in order in seconds.

Smoothing is performed both while the user is drawing and after the stroke is finished. Only a few iterations (32) is performed while drawing. Once the stroke is finished, we smooth the curve using 32 iterations per frame until we reach no substantial change in position of the vertices or we reach a maximum limit (we typically use 512).

An example in Figure 4.13 and in the accompanying video shows usage of the **transient geometry**. A sphere is used as an object that carries the initial curve that defines the overall shape of the final object Figure 4.13 a). The sphere is deleted and the first curve becomes a part of the scene (Figure 4.13 b)).

Results. Figure 4.18 shows an example of Medusa that heavily uses transient objects. The input scene is a model of the head without hair. The user starts by

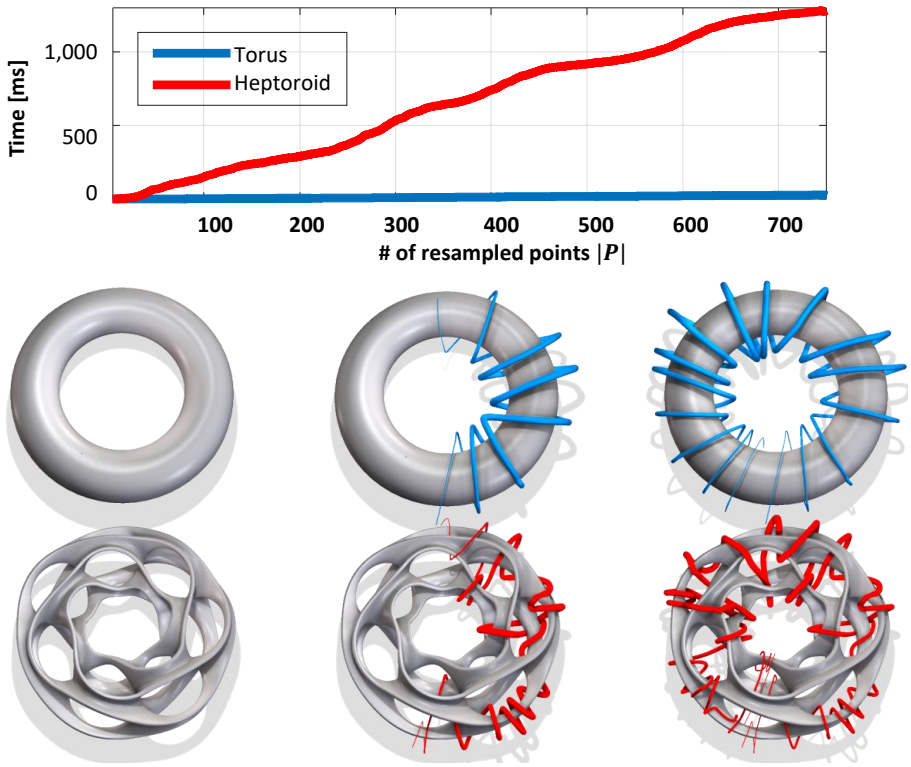


Figure 4.12. The scene generation time increases with the complexity of the scene and the number of intersections for each point.

placing several transient objects that help to position the first snakes in 3D. The snakes become a part of the scene and the user draws further snakes around them. This was a time demanding scene to complete and the overall authoring time was 16 minutes. The final scene has 36 curves with the total of $1,167$ vertices made from $1,810$ input points. The complete scene calculation from the 2D points was 2.1 seconds.

Figure 4.1 shows an example of Kraken that has been generated by multiple strokes from a single view. It is interesting to observe that the individual strokes actually

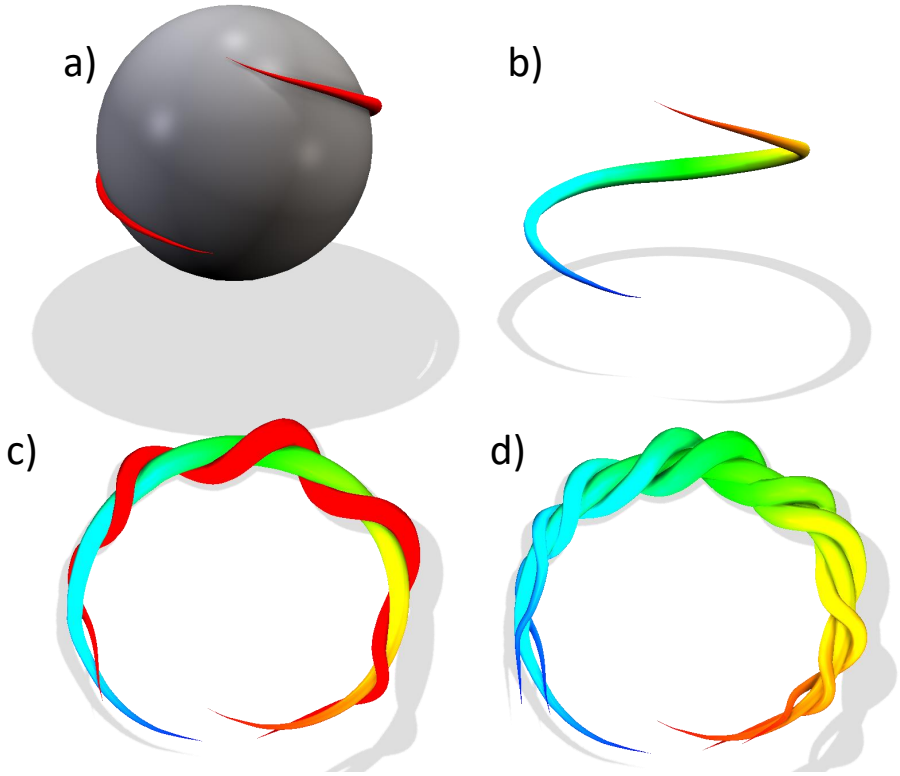


Figure 4.13. An example of usage of transient geometry. Sphere is used as a shape-defining object and after a first sketch it is deleted a). The first curve becomes part of the scene and is used to wrap several additional curves around c)-d). The overall look of the resulting geometry is defined by the initial transient sphere.

correspond to wrapping the tentacles of the Kraken around the ship. This example had 606 input points that generated 724 3D vertices. The time to generate the complete model from the 2D strokes was 98 ms and the overall authoring time was a little bit over one minute.

Figure 4.14 shows an example of a dense scene (tree) that is enhanced by cords with lights. This example includes multiple occlusions of tiny objects that is a difficult case for our framework, because it causes frequent depth skipping during

the curve drawing. This input scene had $1,273$ input points that generated $1,653$ vertices and it took nearly four seconds to generate it. Authoring of this scene took about 19 minutes.

Figure 4.14. A dense scene with tiny geometry is a difficult case because of frequent skipping during curve drawing.

Figure 4.15 shows an example of a path control in a scene of a city that could be used in an animation for camera planning. In order to provide flyover the center of the city the user created transient geometry and wrapped the curve around it. Authoring of this scene took 11 seconds and the curve generation was 16 ms.

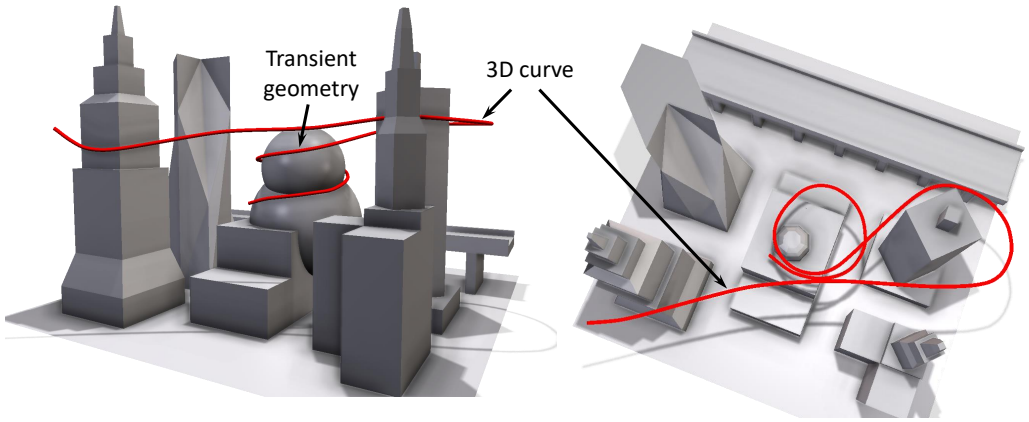


Figure 4.15. Camera path through the city can be sketched very quickly with the help of transient geometry. The curve was sketched from the viewpoint on the left.

The cup with snake in Figure 4.16 shows generation of wrapping of a 3D object with a single stroke made up of 615 points. Authoring of this scene was around 1.5 minutes and the curve was generated in 75 ms.

User interaction. We engaged novice users (see Figure 4.16) as well as professional artists to refine and test the system. Although the users had the option to change the depth of the last segment while drawing, they noted that the automatic prediction helped them to draw more efficiently.

The majority of the curves were created with a single stroke without any editing operations. Whenever a change was needed, the curve was usually removed and

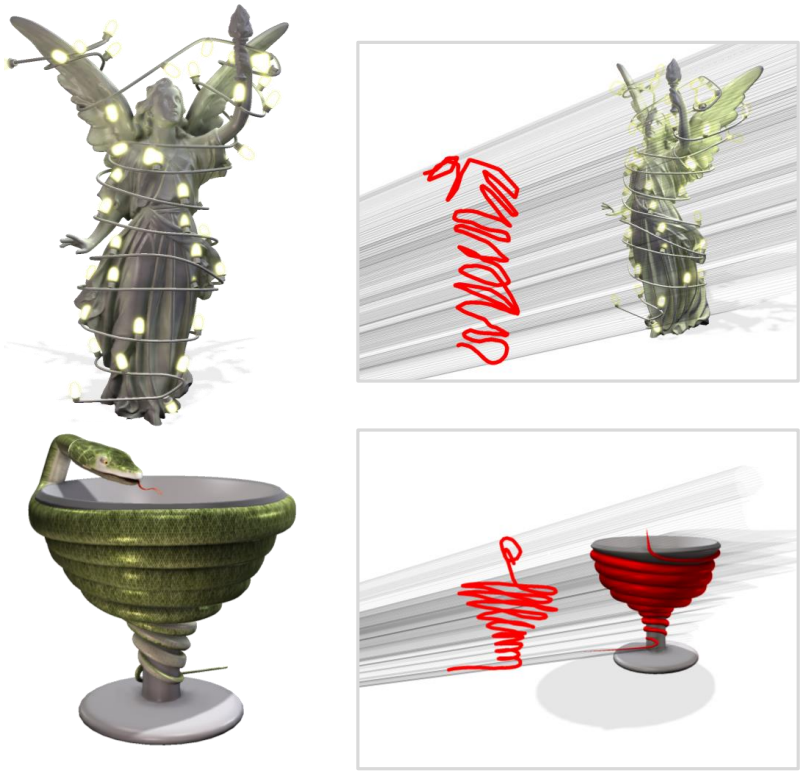


Figure 4.16. Two examples of wrapping an object by a single stroke.

sketched again from scratch. In more complicated cases, such as Figures 4.1, 4.14 and 4.18, redrawing of parts of the curve and manual depth changes were utilized more often.

Even though every curve was sketched from a single viewpoint, a change of viewpoint was not entirely eliminated. The most common perspective changes were rotating the camera to inspect the resulting curve and choosing a perspective that minimized foreshortening, since the resulting curve may differ depending on chosen viewpoint as shown in Figure 4.17.

Table 4.1.
Time Performance. Mesh triangles shown are for the final scene including meshes generated by curves. The authoring time refers to the time spent by the user, the total time is the time the system spent on generating the curve.

	Input		Time					Output	
Model	**Mesh Tri.**	**Input Points**	**Vertex Est. [ms]**	**Graph Constr. [ms]**	**Curve Const. [ms]**	**Total [ms]**	**Authoring [s]**	**# of Curves**	**# of 3D Vertices**
Kraken (Fig 4.1)	72,236	606	71	9	18	98	84	8	724
Curve art (Fig 4.13)	41,328	348	15	9	10	34	41	3	290
Medusa (Fig 4.18)	477,280	1,810	1,921	176	11	2108	984	36	1167
Tree (Fig 4.14)	81,132	3689	319	53	57	429	1140	35	851
City (Fig 4.15)	3,588	192	12	4	2	16	11	1	67
Lucy (Fig 4.16 top)	139,940	665	118	305	715	1138	210	1	417
Cup (Fig 4.16 bottom)	10,240	615	49	9	15	73	95	1	715

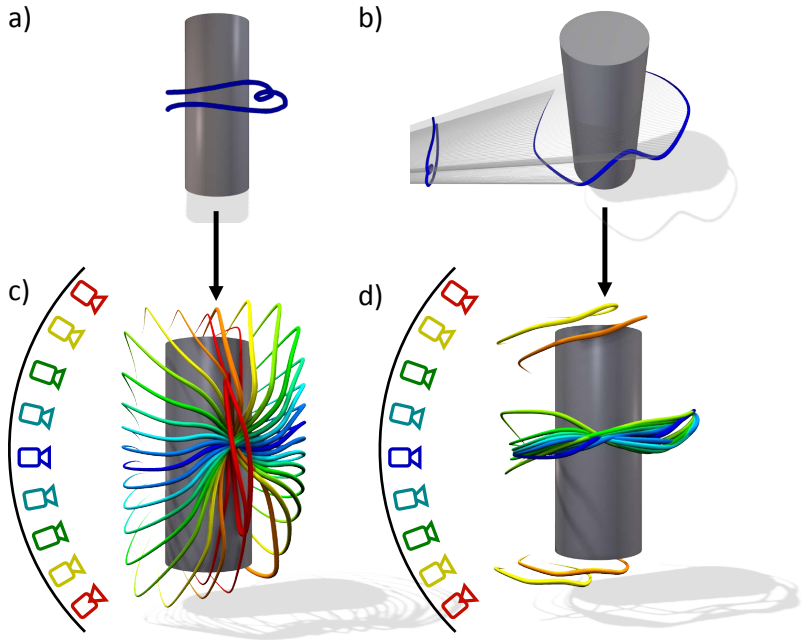


Figure 4.17. Influence of the viewpoint on the final curve. Curve defined by a stroke (a) is inferred (b). By applying the same stroke from different viewpoints (c), the shape of the curve changes depending on the orientation. In (d) we generate the curve from (b) by using its projection to a different viewpoint, which gets progressively harder due to foreshortening and for more oblique views it is hard to obtain the same curve. Blue to red transition signifies the increasing absolute elevation angle of the camera.

4.9 Conclusion

We presented Skippy, a novel algorithm for 3D curves generation from single view. The user draws a 2D stroke and the algorithm divides it into *on* and *off* sequences. The distance of the 3D curve from the object (its height) is estimated from the user stroke. The sequences are converted to *on* segments in 3D and so called segment graph encodes the *on* segments as its nodes and all possible connections of the *off* segments as segment graph edges. The optimal path is generated that follows the

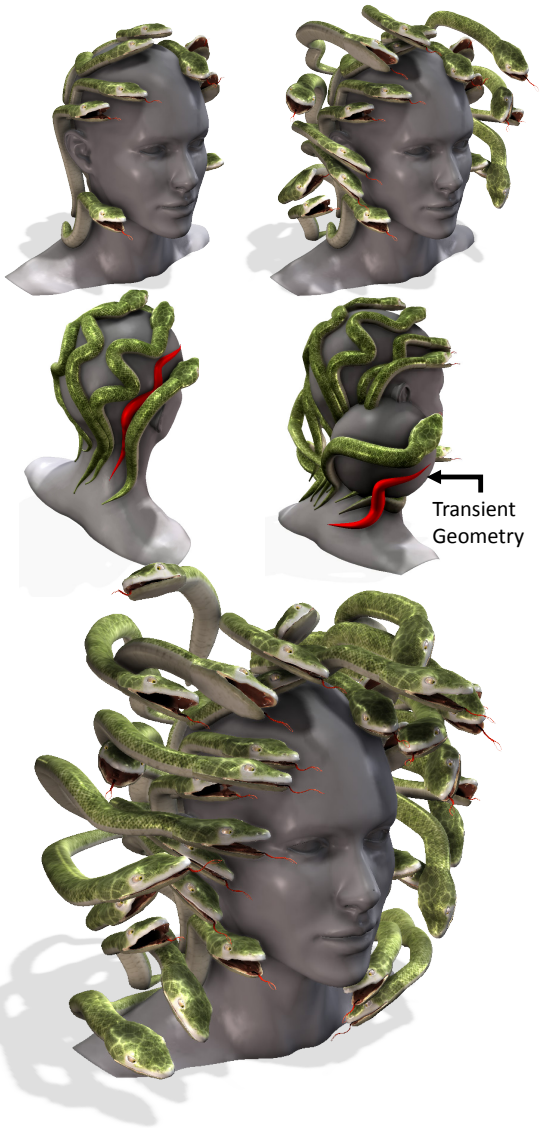


Figure 4.18. Two frames from the creation of Medusa (top left and right) show usage of transient geometry that keeps the snakes away from the head. Some snakes eventually become part of the geometry and are used as supporting objects as well.

object geometry and has monotone curvature. The user can quickly edit the curve by selecting alternative options that are all encoded in the segment graph. Moreover, we include the concept of transient geometry that is used to scaffold the curve creation. We show Skippy on a number of examples ranging from simple wrapping of curves around objects to complex scenes with intertwined geometries.

Limitations. One of the limitations arising from input geometries in the form of triangle soups, is that arbitrarily small elements can be in place. Skippy will not be able to treat these small and noisy geometries before any post-processing. Intersection with such structures cause jumping between different options. Although we solve this partially by allowing a variable threshold of rays that can be ignored, it shifts the problem to higher frequencies or causes unwanted geometries to be ignored. Also, Skippy does not eliminate the need for changing the viewpoint. In case of a complex guiding object or scene the user may have to draw the curve in several parts and reposition the view between each part to reduce the foreshortening of the guiding objects. Still, the number of necessary viewpoint changes is small compared to traditional approaches.

There are several possible avenues for **future work**. One of them is to allow branching or more complex network topologies on the generated curves. Similarly to our redrawing approach, the curve end points could be restricted, for example to lie on an existing curve structure, or we could enforce orthogonality or parallelity to existing curves. Another area of future work could address the shape of the curve. We assume that a visually plausible curve is a smooth one, but it would be interesting to allow different options, such as corners and sharp features. One of the ways to achieve this would be to use Manhattan or Chebyshev distance instead of Euclidean as the basis of the distance field or explicitly detect sharp features and

modify the curve accordingly. We have experimented with the way the curve is controlled by the object surface. So far we only consider the distance of the stroke that defines the height of the curve. This concept could generalize to considering different properties such as salient features that could attract or repulse the curve, or the texture on the surface could further control the curve shape. Finally, we believe our method has strong potential application in both Virtual and Augmented Reality (VR, AR) modeling systems, allowing artists to rapidly decorate and populate large or distant spaces (either virtual or real) that might not otherwise be easily accessible. In other words, Skippy may allow artists to draw content that extends beyond their physical reach; such a system is particularly relevant to AR where the world cannot be scaled down. For this reason we find it exciting to explore the use of Skippy combined with 6-DOF tracking for design tasks in immersive and virtual environments.

4.10 Acknowledgements

This work has been sponsored by Adobe Research and by the National Science Foundation grant #1606396 *Haptic-Based Learning Experiences as Cognitive Mediators for Conceptual Understanding and Representational Competence in Engineering Education.* We would like to thank Darius Bigbee and Suren Deepak for help with the modeling of various examples. Furthermore, we would like to thank to Alexander Spivak for providing the Medieval Ship model, kaneflame3d for providing the Female Head model, Standford 3D Scanning Repository for providing the Armadillo and Lucy models and Carlo H. Séquin for providing the Heptoroid model.

CHAPTER 5. PROCEDURAL ITERATIVE CONSTRAINED OPTIMIZER

This chapter is a result of collaboration with Mathieu Gaillard[1], Radomír Měch[2], Nathan Carr[2], and Bedrich Benes[1]. It is currently being prepared for submission to *SIGGRAPH Asia 2019*. Author's contribution include implementation and manuscript preparation.

[1]Purdue University, [2]Adobe Research

5.1 Abstract

Procedural modeling has produced amazing results, yet fundamental issues such as controllability and limited user guidance persist. We introduce a novel procedural model called PICO (Procedural Iterative Constrained Optimizer) and PICO-Graph that is the underlying procedural model designed with optimization in mind. The key novelty of PICO is that it enables the exploration of generative designs by combining both user and environmental constraints into a single framework by using optimization. The PICO-Graph procedural model consists of a set of geometry generating operations and a set of axioms connected in a directed graph with cycles. The forward generation is initiated by a set of axioms that use the connections to send coordinate systems and geometric objects through the PICO-Graph, which in turn generate more objects. This allows for the fast generation of complex and varied geometries. Moreover, we combine PICO-Graph

with efficient optimization that allows for quick exploration of the generated models and the generation of variants. The user defines the rules, the axioms, and the set of constraints; for example whether an existing object should be supported by the generated model, whether symmetries exist, whether the object should spin, etc. PICO then generates a class of geometric models and optimizes them so that they fulfill the constraints. The generation and the optimization in our implementation is interactive and can be influenced by the user while running. For example, the user can sketch the constraints and direct the generation in the desired direction. We show PICO on a variety of examples such as the generation of procedural chairs with multiple supports, generation of support structures for 3D printing, generation of spinning objects, or generation of procedural terrains matching a given input.

5.2 Introduction

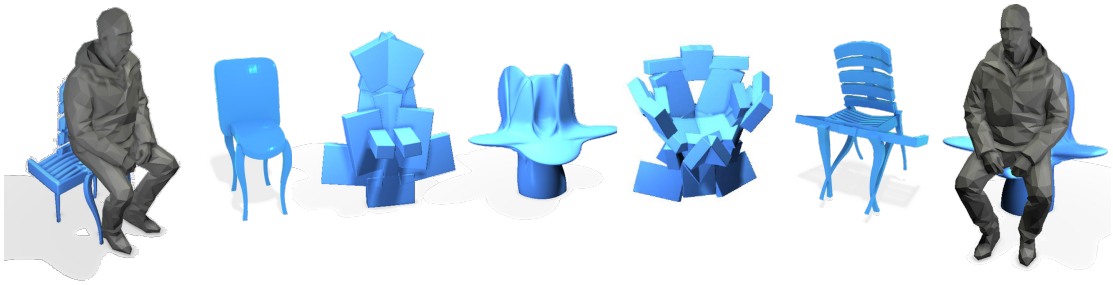


Figure 5.1. PICO automatically generated various procedural models of a chair. The user first marks the bottom of the character as an active area and PICO generates the procedural chair that supports it. All models were built while making sure the model will not tip over.

Procedural modeling has been successfully applied in a wide variety of areas such as vegetation, texturing, architecture, and decorative design. One of its most

important strengths is the ability to encapsulate a large variety of shapes into a concise formal description that can be efficiently parameterized. This, in effect, allows for the generation of variants of the structures by changing the procedural model parameters or rules.

While procedural modeling has been recognized as a strong and expressive methodology capable of solving various problems, its strength has not been fully harnessed because of its disadvantages. Probably the most important problem is the difficulty to fully comprehend the procedural shape derivation from an initial state (axiom). Procedural models exhibit complex behavior and non-linearity between input parameters and output shape Flake (1998); Mitchell (2009b) because the derivation may include various feedback that can exponentially amplify some features while diminishing others. The designer is usually left with trial-and-error experimentation. Due to the lack of controllability, practical applications of procedural models usually hide the procedural rules and show just an interface Huang, Kalogerakis, Yumer, and Mech (2017); Nishida et al. (2016b). Alternatively, they may provide sets of examples that are reused which is a common strategy adopted by many commercial products ESRI (2017); SideFx (2019).

Procedural models can be targeted to a specific goal by using optimization. User-defined constraints have been used to control procedural models as detailed in Section 5.3. These approaches attempt to find the parameters of the procedural system that generate results matching the user-specified requirements and/or adhere to the user-defined constraints. Constraint-based procedural models provide the user with the ability to define what the final result should look like, without worrying about the internals of the system. However, existing systems target narrow domains and usually focus on a single procedural model. More general

approaches capable of working with multiple procedural representations often lack interactivity due to the sheer amount of search space that must be explored and lack of any domain-specific information that could otherwise speed up the process. Furthermore, they optimize a predefined function that cannot be changed during the optimization. Finally, prior work focuses predominantly on optimizing the *derivation* of a *predefined* grammar, *e.g.,* a tree grammar to grow into a desired shape. There has been little work on optimizing the procedural rules themselves.

Several observations motivate this work. The first key observation is that a *set* of user-defined constraints can be used together to impose complex requirements on the generated objects. For example, the *function* of the object can be specified with a handful of geometric constraints, as we demonstrate by generating a variety of free-form chairs (Figure 5.1). The second key observation is that a *broader procedural system* can be created that encompasses the commonly used hand-crafted grammars. This generic system can then be optimized to produce a wider range of objects that a single hand-crafted grammar cannot express. The generated objects can match the user's intent by simply defining and controlling a set of constraints. This is again demonstrated by Figure 5.1, where no chair-specific grammar has been created, and the procedural models were evolved automatically. The last observation is that, contrary to existing off-line approaches, instant visual feedback and the ability to interactively control the optimization process significantly improves the expressiveness of procedural modeling.

We introduce PICO (Procedural Iterative Constrained Optimizer), a framework for procedural geometry optimization and interactive modeling. At the heart of the framework is a novel procedural model which we call PICO-Graph. This model uses a data-flow paradigm, where nodes represent geometry generation operations

and edges define travel paths of objects. Objects travel through this graph between source (axiom) and sink (scene output) nodes, triggering operations that generate more geometry whenever they arrive at any node in the graph. This representation supports branching, recursion, and instancing. The definition of the geometry generation operations and traveling objects is flexible and supports arbitrary 2D and 3D geometry such as user-defined meshes. The PICO framework itself consists of an interactive constraint definition system and an optimization engine that refines the PICO-Graph to match the given constraints. The optimization consists of a multi-objective evolutionary algorithm which is capable of optimizing graphs with cycles, as opposed to only derivation trees.

We demonstrate the capabilities of PICO on a variety of examples, including automatically generated geometries such as chairs, trees, 3D printing supports, and terrains. We show that many of these examples can be controlled interactively by using simple and intuitive constraints. Furthermore, we demonstrate that our optimizer coupled with the PICO-Graph representation outperforms previous work in terms of speed. We claim the following main contributions:

1. A novel procedural model, called PICO-Graph, that generates wide range of 3D and 2D geometry. A simple design with fast evaluation makes it suitable for optimization.

2. We couple PICO-Graph with a novel optimization technique that allows for interactive user-controlled structure generation.

3. We introduce a novel procedural language at a higher level of abstraction, where the user provides building blocks but the system finds their relationships to generate the desired object.

An example in Figure 5.1 shows an application of our framework. The input is a 3D mesh model of a person. The user interaction consists of marking areas that require support, specifying additional constraints, *e.g.,* stability and mass minimization, and choosing the building blocks for the model. Our optimization then evolves models that satisfy the given constraints.

5.3 Related Work

Procedural modeling is a broad topic that has been applied in a variety of contexts. For readers interested in more broad coverage of the topic, we reference a number of state-of-the-art surveys. These include the generation of procedural worlds Natali, Lidal, Parulek, Viola, and Patel (2013); Smelik et al. (2014), optimization of procedural models for games Togelius, Yannakakis, Stanley, and Browne (2011), and inverse procedural modeling Aliaga, Demir, Benes, and Wand (2016).

Early explorations in **procedural modeling** leveraged fractals, focused on generation of terrains Fournier et al. (1982); Galin et al. (2019), and vegetation Aono and Kunii (1984). Shape grammars and split grammars Stiny and Gips (1972) were successfully applied into architectural models in Wonka et al. (2003). Split grammars were extended in various directions including procedural buildings P. Müller et al. (2006) and just recently into a procedural model called `CGA++` in Schwarz and Müller (2015). Numerous examples of purely procedural models exist such as the approach of Merrell and Manocha (2011) that generates infinite architectural structures by using only procedural rules. While these systems can produce complex high quality output, they have a low level of directability and often fail to capture the user-intent of the artist or designer.

Many design tools have been designed around specific tasks such as the design specific shape classes (*e.g.,* chairs Garcia and Romão (2015)) or handling the arrangement or placement of shapes Guerrero, Jeschke, Wimmer, and Wonka (2015). By targeting the system with some level of domain specificity more compelling results can often be achieved. However, the drawback of most procedural modeling systems is the lack of artistic control. Our system is agnostic to the class of shapes allowing it to be used on a a variety of tasks. Procedural models give rise to an often exponential space of variation. While approaches have been suggested for navigating and exploring these spaces Talton et al. (2009), more direct artistic control remains challenging.

Control for Procedural Models: a lot of active research has gone into addressing control of procedural models. Ijiri et al. (2006) introduced a system that can encode a simple user sketch as L-system and Palubicki et al. (2009) used sketching of attraction particles to interactively control growth of simulated vegetation. Closely related is the work of Mitra and Pauly (2009) who optimize 3D structures so that they match user-defined shadows. Guided procedural modeling (Benes, Šťava, Měch, & Miller, 2011) generalizes the concept of environments by closing procedural models into guides that can communicate by message passing and Krecklau and Kobbelt (2011) introduced a procedural model that allows for generation of interconnected structures.

A declarative approach to procedural modeling of virtual worlds of Smelik et al. (2011) models terrains by defining constraints. Ritchie et al. suggests controlling procedural models by stochastically-ordered sequential Monte Carlo programs in Ritchie et al. (2015) and they later introduced neurally-guided procedural models in Ritchie, Thomas, et al. (2016). Recently, procedural models were coupled

with sketching and deep learning to provide a more natural interface for artists Huang et al. (2017); Nishida et al. (2016b), where deep learning recognizes the sketch and selects the procedural model and its parameters. Neural based methods have been recently applied to program optimization as well. One of the most recent examples is the work of Ellis et al. (2018) proposed a method that is able to take simple hand-drawn images and translate them into a graphic programs able to generate LaTeX-style figures. The graphics programs follow a simple grammar that include simple primitive drawing, loops and conditional statements. Similarly, Sharma et al. (2018) showed a neural approach that infers a simple program, equivalent to a CSG hierarchy, that constructs a given 2D or 3D shape. Their method uses reinforcement learning and encoder-decoder architecture, where the input is an image of an object, the output is a program that generates an object, and the reward is the difference of the two in image space. Conversely, Du et al. (2018) introduced an analytic method of synthesizing a CSG tree from existing geometry using a search of possible CSG programs.

Procedural Modeling and Optimization: Procedural models were coupled with various optimization approaches in the past. Sims used a combination of genetic algorithms along with competition for resources to evolve virtual creatures in an environment with simple physics in his seminal paper Sims (1994a). Hornby and Pollack (2001) used L-systems and evolutionary algorithms to generate various shapes and Talton, Lou, Lesser, Duke, Měch, and Koltun (2011) used L-systems to parse states of expression of a rule set to find an optimal geometry by using Metropolis Hasting variant called Reversible Jump Monte Carlo Markov Chains (MCMC). Contrary to the previous work, our approach does not require fixed set of rules and the rules and their dependencies are generated automatically during

the optimization step. Merrell, Schkufza, Li, Agrawala, and Koltun (2011) used similar approach to organize furniture in a virtual scene and MCMC was also used to layout synthesis in Yeh, Yang, Watson, Goodman, and Hanrahan (2012). Localized learning of stochastic procedural models for virtual terrains has been used in brush-like approach in Emilien et al. (2015). Although MCMC approaches provide good results, they tend to be very slow for large scenes.

Structurally sound masonry buildings were achieved via optimization in Whiting et al. (2009) and our approach shares analogy with this work in that it attempts to use functional constraints. However, our definition of function does not encompass only the structure, but also other aspects such as volume, touching, proximity, etc.

Měch and Miller (2012) introduced Deco that uses a scripting language to generate 2D or 3D patterns by guiding the growth of the procedural model to follow the user input. In our approach, we control the model indirectly by modifying the constraints and by painting on the objects. Also similar to our method is the work of McDermott (2012) who leverage graph grammars to evolve 3D shapes. However, the control of their method is low as opposed to our approach that allows using constraints to guide the procedural optimization to a desired output. Bergen and Ross (2013) used aesthetic criteria to evolve L-systems and K. Xu, Zhang, Cohen-Or, and Chen (2012) optimized shape collections of genetic algorithms by using a higher semantic representation. Finally, Haubenwallner et al. (2017) used genetic algorithms to find procedural grammar expansion to match given constraints and was an inspiration for this work and we compare in Section 5.7.

Most previous works use a fixed procedural model or provide a direct control for its definition. We were inspired by the seminal work Sims (1994a) and ours is closest in spirit to Bergen and Ross (2013); Haubenwallner et al. (2017); Jacob (1994).

Compared to Jacob (1994), we propose a new expressive class of procedural models that can create a variety of shapes, without having to use predefined shapes like flowers or leaves Jacob (1994), or use of voxel representation Bergen and Ross (2013). We also introduce a novel optimization system enabling an interactive control during the evolution process that allows for incremental updates.

Procedural model representations: Numerous representations for procedural models have been proposed, whose formalism is rooted in programming language design. These include data flow models as well as stream processing Abelson, Sussman, and Sussman (1997); Wadge and Ashcroft (1985). Here we mention only the most relevant systems from which our work takes inspiration. Lindenmayer introduced L-systems Lindenmayer (1968b) that were extended by geometric interpretation and recursion by Prusinkiewicz (1986). L-systems are linear, while our approach aims at volumetric objects and allows for geometric operations on them. In general, L-system rules are not easy to evolve directly, and as a result only a relatively simple cases of L-systems have been evolved so far Jacob (1994); McCormack (2004). Stellar grammars Velho (2003) were used to generate subdivision structures and this approach is similar to ours, except we attempt to expand each vertex. Similarly, `vv-system` allows vertex-vertex expansion to simulate subdivision surface in C. Smith and Prusinkiewicz (2004). Our procedural model is close to the operator graph representation Boechat et al. (2016) with the most important difference being that we use a mix of coordinate frames and 2D/3D primitives as the traveling objects among the rules that control the generated shape. Moreover, we also provide novel optimization approach that allows for reconnecting the rules, their mutations, and cross-over.

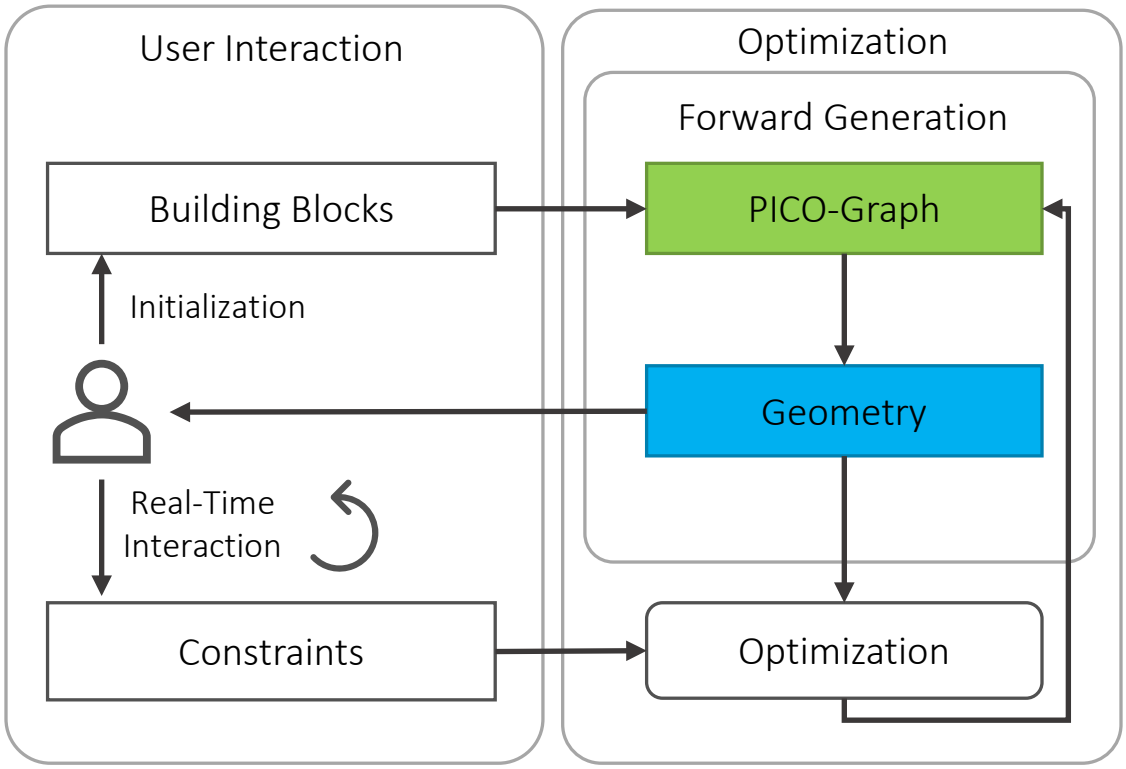


Figure 5.2. Overview of PICO. User defines the *building blocks* which represent parametrized geometry generating operations with connectivity information. During the *Forward Generation* the user also connects the operations into a *PICO-Graph* that generates the output *geometry*. Alternatively, the user can define a set of *constraints* that are *optimized* for by using an evolutionary algorithm. The constraints can be modified interactively as various geometries are generated and shown to the user.

5.4 Method Overview

The ***input*** to our method is a set of *building blocks*, *i.e.,* definition of geometry generating operations, and *constraints*, *i.e.,* requirements from the user how should the generated geometry look like. Geometry generating operations can be either simple geometric objects, such as spheres or boxes, or user-defined geometries

imported from existing meshes. These operations may be parameterized (size, orientation, recursion limit) and must contain information on how they can be connected to other building blocks. The connectivity information, in examples shown in this work, is a set of coordinate frame transformations.

The building blocks are connected into a *PICO-Graph*, which is the underlying procedural representation in our system. Although our framework supports manual definition of PICO-Graph, this may quickly become an overwhelming task when modeling complex objects. The key contribution of our work is the automatic generation of procedural models by using user-defined *constraints and evolution*. Some constraints can be specified by a simple toggle (*e.g.,* that the object should be stable), some require manual input (*e.g.,* sketching of support surfaces or image to match), and some require loading external geometry (for example for object avoidance). Each constraint has an associated importance that allows the user to control various design trade-offs. An important feature of our system is the fast evolution algorithm that allows for dynamic and interactive modifications of constraints by the user during the model generation.

PICO can be used for ***forward generation*** to generate geometry by manually defining the PICO-graph, *i.e.,* connecting individual building blocks. The PICO-graph is a dataflow graph in which objects travel from a source node (axiom) to a sink node (scene output). The objects traveling in our implementation are *coordinate frames* and *2D or 3D geometry*. The geometry generating operations are therefore defined as taking either frames or geometry as input and outputting further frames or geometries or a combination of both. The actual procedural output geometry generation starts by sending initial objects from the source nodes. The objects trigger the geometry generating operations on the nodes they travel to.

These operations generate new objects which are sent further into the graph. Finally, the objects accumulated at sink node(s) can be gathered into the final geometry (see an example of forward generation in Figure 5.3 and the accompanying video).

The ***optimization*** iteratively evaluates geometry against the user-specified constraints and modifies the PICO-Graph such that the generated geometry satisfies the constraints. Constraints that cannot be enforced directly are combined into a fitness that is maximized by solving a weighted multi-objective optimization problem by using a novel evolutionary algorithm. The algorithm maintains a population of individuals which are defined by using PICO-Graph as their genotype and the generated geometry as their phenotype. New solutions are generated using mutation and crossover operators defined over the PICO-Graph. Furthermore, we use niching, *i.e.,* we maintain different species in a population, to maintain diversity and to explore fitness landscape that may be multi-modal.

5.5 Forward Generation

PICO-Graph is the procedural model used in our system. It is built by using *building blocks*, *i.e.,* geometry generating nodes that take other geometry as input and create more geometry. The PICO-Graph defines both the geometry generation operations along with the order in which the operations should be applied to produce the final model. Figure 5.4 shows an overview of the PICO-Graph.

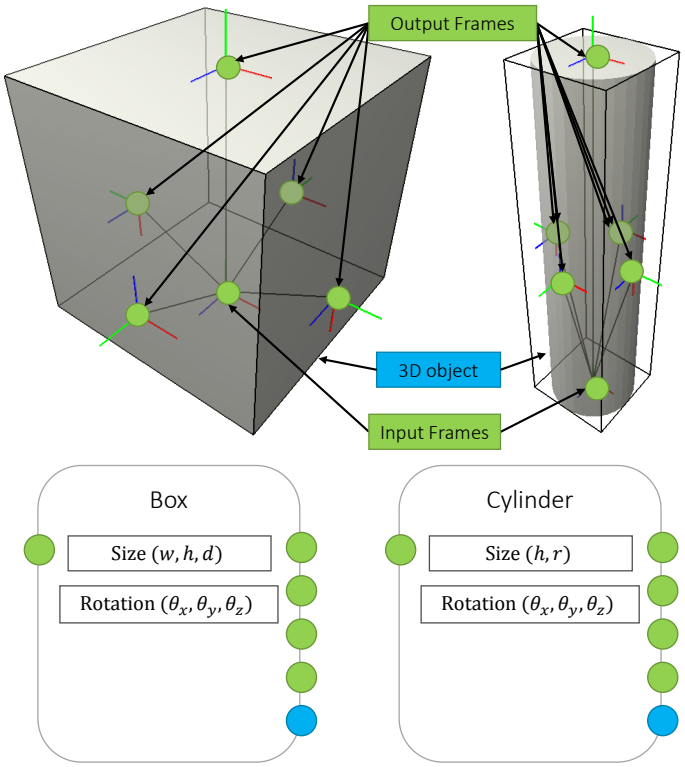


Figure 5.3. An example of two building blocks generating a box and a cylinder respectively. The input and output frames can be positioned and orientated arbitrarily and define how the primitive will connect to others. The size and orientation with respect to an incoming frame are parametrized.

5.5.1 Building Blocks

The building blocks are geometry generating operations Op that take in a spatial object S_{in} (triangle meshes, 3D coordinate frames, Gaussians, and Constructive Solid Geometry (CSG) trees in our implementation). The operation Op generates a

new set of spatial objects $S_i, i \in (0, n)$ (which can be of different types), subject to the operation's parameters $p_j, j \in (0, k)$:

$$Op : (S_{in}, p_0, p_1, \cdots, p_k) \rightarrow (S_0, S_1, \cdots, S_n). \tag{5.1}$$

Figure 5.4 (bottom) shows a graphical representation of this general operation. We use two common forms of spatial objects in our implementation: *coordinate frames* and *2D/3D objects*. The coordinate frames F describe a linear transformation as a 4×4 matrix. The 2D/3D objects are either defined parametrically, for simple primitives such as spheres or cuboids, or using data, *e.g.,* a mesh or a signed distance field.

Figure 5.3 shows two examples of the building blocks, one generating a box and the other a cylinder. Both take coordinate frames as input and output a 3D object and four more coordinate frames. The new coordinate frames can then be used to generate further objects. The operation generating a box is written as:

$$\begin{aligned}
Box : (F_{in}, w, h, d, \theta_x, \theta_y, \theta_z) \rightarrow \\
(F_{front}, F_{top}, F_{left}, F_{bottom}, F_{right}, O_{box}), \\
F_{front} = T(0, h, 0) R(\theta_x, \theta_y, \theta_z) F_{in} \\
F_{top} = T(0, d/2, 0) R(-\pi/2, 0, 0) R(\theta_x, \theta_y, \theta_z) F_{in} \\
F_{left} = T(0, w/2, 0) R(0, 0, \pi/2) R(\theta_x, \theta_y, \theta_z) F_{in} \\
F_{bottom} = T(0, d/2, 0) R(\pi/2, 0, 0) R(\theta_x, \theta_y, \theta_z) F_{in} \\
F_{right} = T(0, w/2, 0) R(0, 0, -\pi/2) R(\theta_x, \theta_y, \theta_z) F_{in},
\end{aligned} \tag{5.2}$$

where T and R are translation and rotation matrices, respectively. The generated object O_{box} is a box of size (w, h, d) at the origin, transformed by $R(\theta_x, \theta_y, \theta_z) F_{in}$; in our implementation, we use θ to adjust the frame of every generated geometry.

Furthermore, consistent in the notation in L-systems, the y axis is direction of procedural generation (growth) and it corresponds to the frame F_{front}.

Each of the building block's parameters $p_j \in P_j$ has an associated domain P_j; for example, the rotation angle parameters can be restricted to a certain range, *e.g.,* $-\pi/4 \leq P_{\theta_x} \leq \pi/4$. This equips the user with a degree of control over the general style of the generated geometry during the optimization (Section 5.6) .

5.5.2 PICO-Graph

The PICO-Graph is a data-flow graph that allows geometrical objects (coordinate frames and 2D/3D geometry) to flow through the graph. The PICO-Graph is a directed multi-graph G consisting of nodes $v_i \in V$ and directed edges $e_i \in E$:

$$G = (V, E). \tag{5.3}$$

Each node has a set of inputs I_{v_i} and outputs O_{v_i}, corresponding to I and O in Eqn(5.1). Edges connect individual outputs to individual inputs, providing one-to-one mapping:

$$E : \{O_{v_i} \ \forall v_i \in V\} \rightarrowtail \{I_{v_i} \ \forall v_i \in V\}. \tag{5.4}$$

Note that this mapping allows multiple outputs connected to a single input. The set of all nodes V consists of three subsets: set of source (axiom) nodes, set of building block (geometry generating) nodes, and a set of sink nodes. Figure 5.4 shows a diagram of the graph, as well as a general building block node (inset). The source nodes (axioms) have no inputs. The sink nodes have no outputs, and they collect objects that traveled to them.

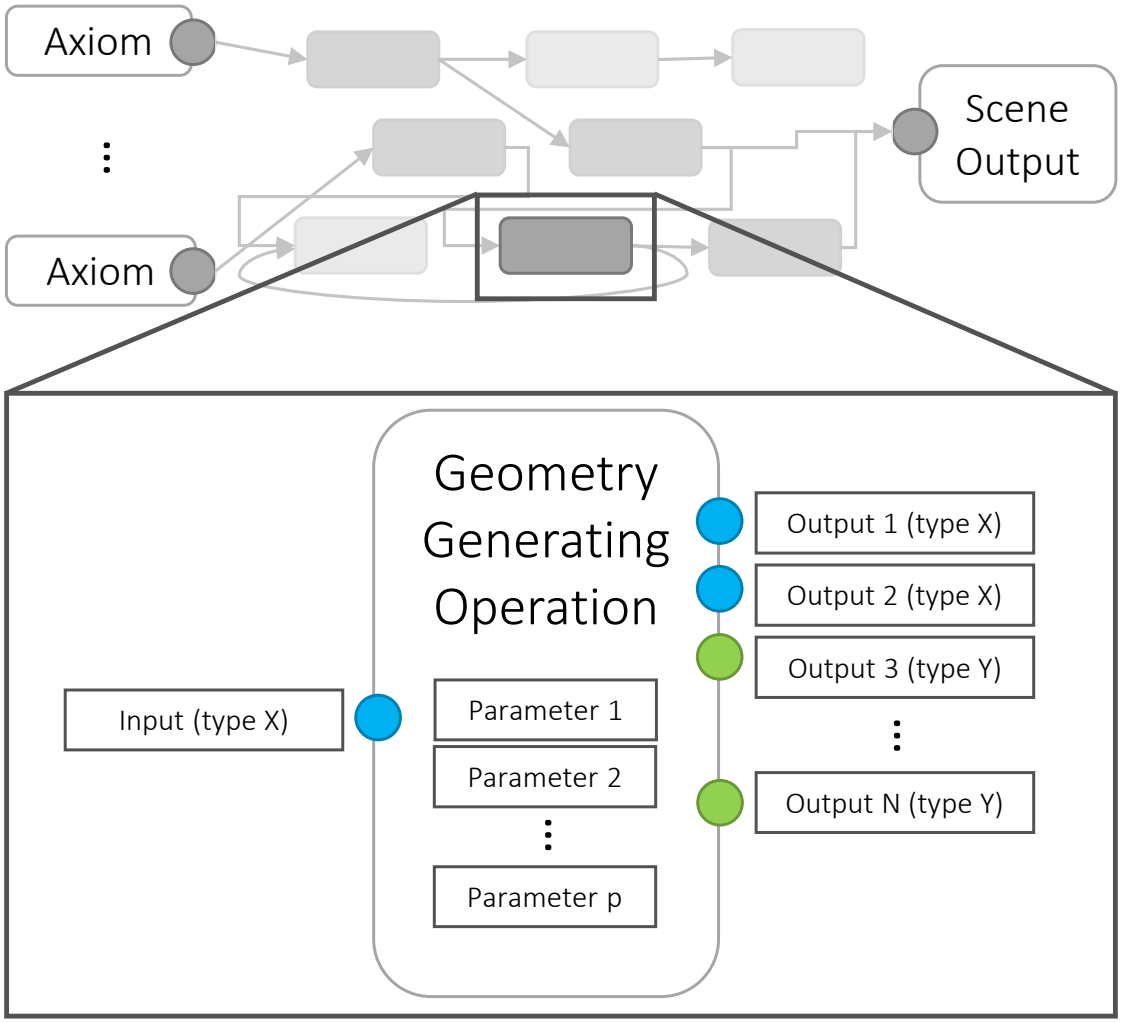


Figure 5.4. Schematic of the PICO-Graph (top). The graph includes source (axiom) nodes, geometry generating nodes, and sink nodes (scene output). Objects travel through this graph from sources to sinks, invoking geometry generating operations, which create and send more objects down the graph. A general template for a geometry generating operation is shown in the zoomed portion. Each operation has a single input and multiple outputs (shown in different colors) and it has multiple parameters that influence the objects generation.

The graph is **executed** by first initializing objects in the axiom nodes. These objects are sent into the next nodes, as prescribed by E. Whenever an object arrives at a node, the operation associated with that node is executed and new objects are created that are sent further down the graph. The execution ends when there are no traveling objects left and the generated geometry is accumulated in the sink nodes. In our implementation, we use Constructive Solid Geometry (CSG) hierarchy to accumulate the 3D objects, and an array for 2D objects. We denote the generated geometry as $\mathcal{G}$.

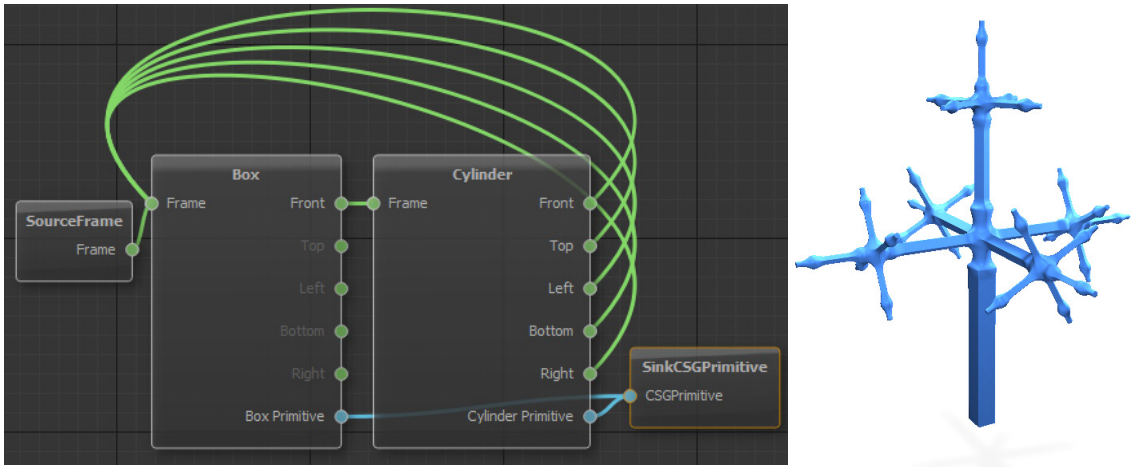


Figure 5.5. An example of a PICO-Graph with cycles (left) and the generated recursive structure (right). Blue edges represent Constructive Solid Geometry (CSG) primitives and green edges 3D coordinate frames. The sink operation blends the incoming primitives and outputs them to the scene.

The PICO-Graph may contain directed cycles (Figure 5.5) leading to a **recursive** generation. The recursion is tracked by counting each time an object, or its descendants, visit a given node. A recursion limit is enforced to stop further execution of an object (1-3 in our experiments).

If a node has multiple incoming edges, the node is executed for each object that travels through it. Similarly to recursion, this produces an **instance** of the same geometry (at a different position with a different frame), but contrary to recursion, it only happens once (unless the node is part of a cycle as well).

5.6 Optimization

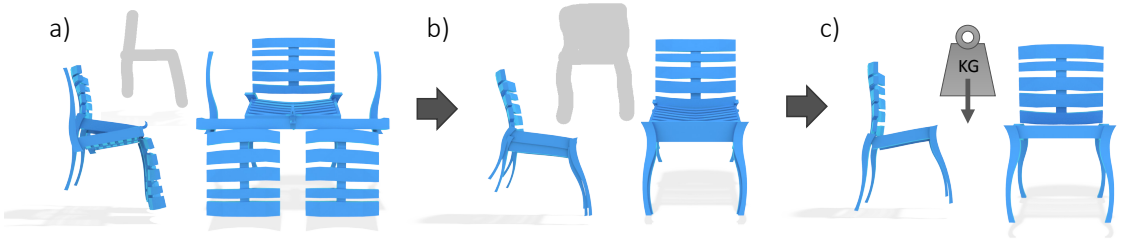


Figure 5.6. Modeling chair using several constraints. First, the user sketches a side view (a). However, the model is free to grow in the direction away or to the user. Therefore, a second sketch may be needed from another view (b). Finally, to remove unnecessary parts, a mass minimizing constraint is applied (c).

PICO is well-suited for the forward generation of procedural geometry. Furthermore, our implementation provides immediate visual feedback of the output (see the accompanying video). However, manual definition of the procedural model is a known difficult problem that is exacerbated if both constraints and the environment are considered.

We have designed PICO so that the PICO-Graph can be generated and efficiently optimized automatically by using an evolutionary approach. To guide the optimization we use user-defined constraints. Some constraints can be enforced

directly, for example symmetry, while others have to be quantified as objective functions that are minimized.

5.6.1 Hard Constraints

Hard constraints must always be met and they can specified and enforced directly by modifying the PICO-Graph. Our current implementation supports a number of hard constraints including symmetry, spin and parameter spaces.

Parameter spaces: P_j for parameters p_j are defined by the user and the optimization is constrained to sample values from these spaces. Each space is defined by specifying minimum and maximum values. The optimization samples these spaces uniformly for initialization and perturbs them by a value sampled from a normal distribution.

Plane and axis **symmetry** can be set by the user interactively. If the building blocks contain two symmetrical frames F_0 and F_1, we modify the graph such that the outputs $O^0_{v_i}$ and $O^1_{v_i}$ of a node v_i that correspond to these frames are routed to the same input I_{v_j} of a node v_j. Because the objects output from node v_i are oriented according to the symmetric frames F_0 and F_1, the two sets of objects created further down the graph (in v_j and further) will be symmetric as well.

Spinning objects (Figure 5.13) have their center of mass aligned to the spinning axis and the spinning axis itself should be parallel to the maximal axis of inertia (Bächer, Whiting, Bickel, & Sorkine-Hornung, 2014). We transform the geometry to have its center of mass at the origin and we rotate it by using rotation Q that is computed by using the eigen-decomposition $Q\Lambda Q^T = I$ where I is the inertia tensor.

The 3D printing supports in Figure 5.12 are constrained to have a maximum angle (45° in our example) and the overhang points are automatically connected to the nearest geometry. If there's no geometry in the cone specified by the above maximum angle, the overhang is connected directly to the ground to ensure printability.

5.6.2 Soft Constraints

In addition to hard constraints our system also supports the modeling of soft constraints. We model each soft constraint by an associated objective function. The optimization then minimizes all of the objective functions to find a Pareto optimal solution, subject to the hard constraints outlined above. The user can modify the importance of each constraint to further control the optimization.

We categorize the objective functions into two types: environmental and intrinsic. That are normalized.

The **environmental** objective functions encompass extrinsic properties of the model including 3D protected volumes P_i, the scene bounding box Ω, ground plane G, and the points from the interacting surfaces from the input geometry Q_i.

The protected volumes are input by the user as 3D objects and they indicate 3D space that the generated geometry should avoid. The scene bounding box Ω limits the operational space of the generated geometry by defining its extent and making sure that the object does not become unreasonably large. The ground plane G makes sure the generated model touches the ground and is also used to optimize for stability of the objects that should not tip over.

Furthermore, the user can add user-defined objects to the scene and mark target areas by painting manually on their surfaces. We refer to them as *interacting surfaces* and they specify locations to which the generated geometry should grow. If the interaction surfaces are present, the goal of the optimization is to expand the procedural geometry so that it approximates the shape of the interacting surfaces, for example by generating a chair that follows the shape a person that sits on it. The interacting surfaces are sampled into a set of 3D points denoted by Q and the objective function attempts to minimize the distance between Q and the generated procedural geometry $\mathcal{G}$. If the goal is to generate an object that touches all points in Q, the objective function is

$$\frac{1}{|Q|}\sum_{q\in\mathcal{Q}}\frac{d(q,\mathcal{G})}{|\Omega_{diag}|}, \tag{5.5}$$

where $|\Omega_{diag}|$ denotes the length of the diagonal of the domain's bounding box, *i.e.,* the largest possible distance and $d(p,\mathcal{G})$ is the distance between a point p and $\mathcal{G}$.

If the goal is to only *touch* the interaction surface, for example the ground plane G, the function is

$$\min_{q\in\mathcal{Q}}\frac{d(q,\mathcal{G})}{|\Omega_{diag}|}. \tag{5.6}$$

Protected volumes specify regions into which the generated objects should not grow. We chose to model this constraint as soft, as it facilitates intermediate solutions that eventually lead to a solution without any collisions. The objective function is defined as

$$\frac{V(P\cap\mathcal{G})}{V(P)} \tag{5.7}$$

where P is the protected volume and V denotes the user-defined volume that should not be entered.

Sketching: To control the shape of the generated geometry more finely, we introduce a sketch matching constraint. The sketch is defined as a binary mask $\mathcal{I}_s$ that is either sketched or downloaded and it is compared to a perspective projection of the generated geometry $\mathcal{I}_g$. The objective function is defined as

$$smoothstep(N_0, 0, N_g) - smoothstep(N_1, 0, N_g), \tag{5.8}$$

where N_g is the number of set pixels in $\mathcal{I}_g$, N_0 is the number set in $\mathcal{I}_g$ but not in $\mathcal{I}_s$, and N_1 is the number set in both $\mathcal{I}_g$ and $\mathcal{I}_s$.

The *stability* of the generated geometry $\mathcal{G}$ is also optimized. For an object to be stable the following equation must hold:

$$m' \in \text{Conv}(\mathcal{G} \cap G), \tag{5.9}$$

where m' is the center of mass m projected along the gravity vector to the ground plane G, and Conv denotes a convex hull. The objective function that maximizes stability is:

$$\frac{|m' - \text{Conv}(\mathcal{G} \cap G)_{centroid}|}{|\Omega_{diag}|}. \tag{5.10}$$

Note that we assume constant density throughout the object to compute its center of mass that is satisfied in 3D printing.

The *spinnability* of the object can be guaranteed by the hard constraints outlined above, but the quality of the spin can be further improved by minimizing the ratio of its moments of inertia (Bächer et al., 2014, Eqn (3)).

The **intrinsic** members of the objective function consider various properties of the generated structure $\mathcal{G}$. Intrinsic members are the volume of the bounding box of $\mathcal{G}$ its mass, number of generated geometric primitives, and the total length of the graph induced by the tokens passed around in the PICO graph.

We control the size of the object by minimizing its bounding volume using the following objective function

$$\frac{V(\mathcal{G}_{BB})}{V(\Omega)}. \tag{5.11}$$

Furthermore, to avoid bulky objects that contain unnecessary parts (with respect to other objectives) we minimize mass using

$$\frac{\rho V(\mathcal{G})}{V(\Omega)}, \tag{5.12}$$

where ρ is the density of the structure. We keep $\rho = 1$ in our implementation.

Figure 5.6 shows the effect of applying several constraints in the modeling process. The user is free to apply them at once or subsequently as needed, as is shown in the figure. First a side sketch is created and then one from the front, which determines the desired shape of the object. Finally, a mass minimizing constraint is used to simplify the generated model.

5.6.3 Evolutionary Algorithm

The evolutionary approach optimizes the set of objective functions given by the user-defined constraints. The main steps of the algorithm are population *initialization*, *speciation*, *evaluation*, *selection*, and *reproduction*. Our overall algorithm shares commonalities with Genetic Algorithms, *i.e.,* we define a genotype and a phenotype, and Genetic Programming (Koza, 1992), *i.e.,* we evolve graphs that can be conceptualized as programs. Furthermore, we adapted techniques from Neuroevolution of Augmenting Topologies (NEAT) Stanley and Miikkulainen (2002) that allow us to measure compatibility of individuals for reproduction versus keeping a separate species.

The population is **initialized** with a set of random individuals, each representing the minimal working PICO-Graph, *i.e.,* one axiom, one geometry generating node, and one sink, each with randomized parameters. The individual consist of a genotype and a phenotype. The genotype is a description of a single PICO-Graph G. The genotype includes a list of nodes, along with their parameters, and a list of edges, along with information whether they are enabled or disabled. We keep an *innovation number* associated with every edge, which tracks new topological changes within the broader population and assist in the crossover operator and speciation. An edge between nodes is considered to be a *gene*. The phenotype is defined as the generated geometry $\mathcal{G}$ and is used for evaluation.

The **evaluation** consists of computing the fitness $F(I)$ for each individual I. Because the objective functions may have different ranges and distributions, we use the *sum of weighted global ratios* (Bentley & Wakefield, 1998) to compute the fitness:

$$F(I) = \frac{1}{\sum_{i=0}^{N-1} w_i} \sum_{i=0}^{N-1} w_i \frac{f_i(I) - f_i^{min}}{f_i^{max} - f_i^{min}}, \tag{5.13}$$

where f_i is the i-th objective function out of N, f_i^{min} and f_i^{max} are the minimum and maximum values of f_i for the entire population throughout all past generations, and w_i is the user-defined *importance* of a member function f_i.

Speciation is a process of dividing the population into multiple distinct species based on a similarity metric, called *compatibility*, such that genotypically similar individuals are grouped together and reproduce only within the species. This ensures diversity in the population and helps explore multi-modal fitness landscapes. We use a modified definition of compatibility from Stanley and

Miikkulainen (2002) which differs in the term quantifying identical genes. Our compatibility between two genotypes g_a and g_b is defined as:

$$\delta(g_a, g_b) = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \overline{W}, \tag{5.14}$$

where N is the total number of genes (edges in the graph) and D and E is the number of disjoint and excess genes respectively (appearing in only one of the genotypes). The term $\overline{W}$ is computed as a distance between parameters p_j of the nodes $v \in V$ in the graph. Thanks to the innovation numbers, we can track nodes that occupy the same position in the graph topology, but are parameterized differently in different individuals. Therefore we sum over the all differences in parameters of nodes that are connected by the genes that appear in both genotypes. The difference between two parameters p_a and p_b are calculated by using an L_2-norm. The coefficients c_1, c_2 and c_3 are used to weight the contributions of genes in compatibility ($c_1 = 2$, $c_2 = 2$, and $c_3 = 1$ in our implementation). Finally, to decide if two individuals belong to same species, we use a threshold t_δ. If $\delta > t_\delta$, individuals do not belong to the same species and a new species is created, unless there exists an individual within an existing species whose compatibility is below the threshold. We vary t_δ during the optimization process to keep the number of species constant, in our case $3 - 5$ species for population size of 150. Finally, we employ fitness sharing within the species.

We **select** individuals for reproduction from the top $5 - 15\%$ individuals in each species. The **reproduction** uses two operators, *mutation* and *crossover*, to produce children from selected individuals. We either use mutation only (5% of the time), crossover only (85% of the time), or crossover with subsequent mutation.

We use five distinct types of **mutations**:

1. add a node (after an existing node)
2. insert node (between connected nodes)
3. mutate parameter
4. add an edge
5. toggle edge

Add node finds an open output in the graph and adds a randomly initialized node, causing the geometry to grow outward. *Insert node* finds an existing edge and replaces it with a new node and two new edges, again causing the model the grow by prolongation. *Mutate parameter* randomly perturbs parameters of the nodes, with low probability (5%) but by a large amount ($\sigma = 80\%$ of parameter space P, using a normal distribution). *Add an edge* connects two nodes that were not previously connected. Note that if this mutation is not performed, the graph will remain cycle-free and will resemble a grammar derivation tree, similar to the work of Haubenwallner et al. (2017). Finally, *toggle edge* randomly disables and enables edges in the graph, allowing for pruning of unnecessary parts of the graph or reactivating parts that may be relevant to the current state of the optimization.

Figure 5.7 shows an example of two generated structures resembling biological trees (top) that were combined into four by two separate crossover operations.

We adapted the **crossover** operator from Stanley and Miikkulainen (2002), with the main difference being that we need to transfer node parameters from parents to child. Two parent genotypes are first aligned using their innovation numbers, same as in the compatibility calculation (Eqn 5.14). Genes (*i.e.,* edges in the graph) that occur in both parents are randomly chosen from one parent to transfer to the child.

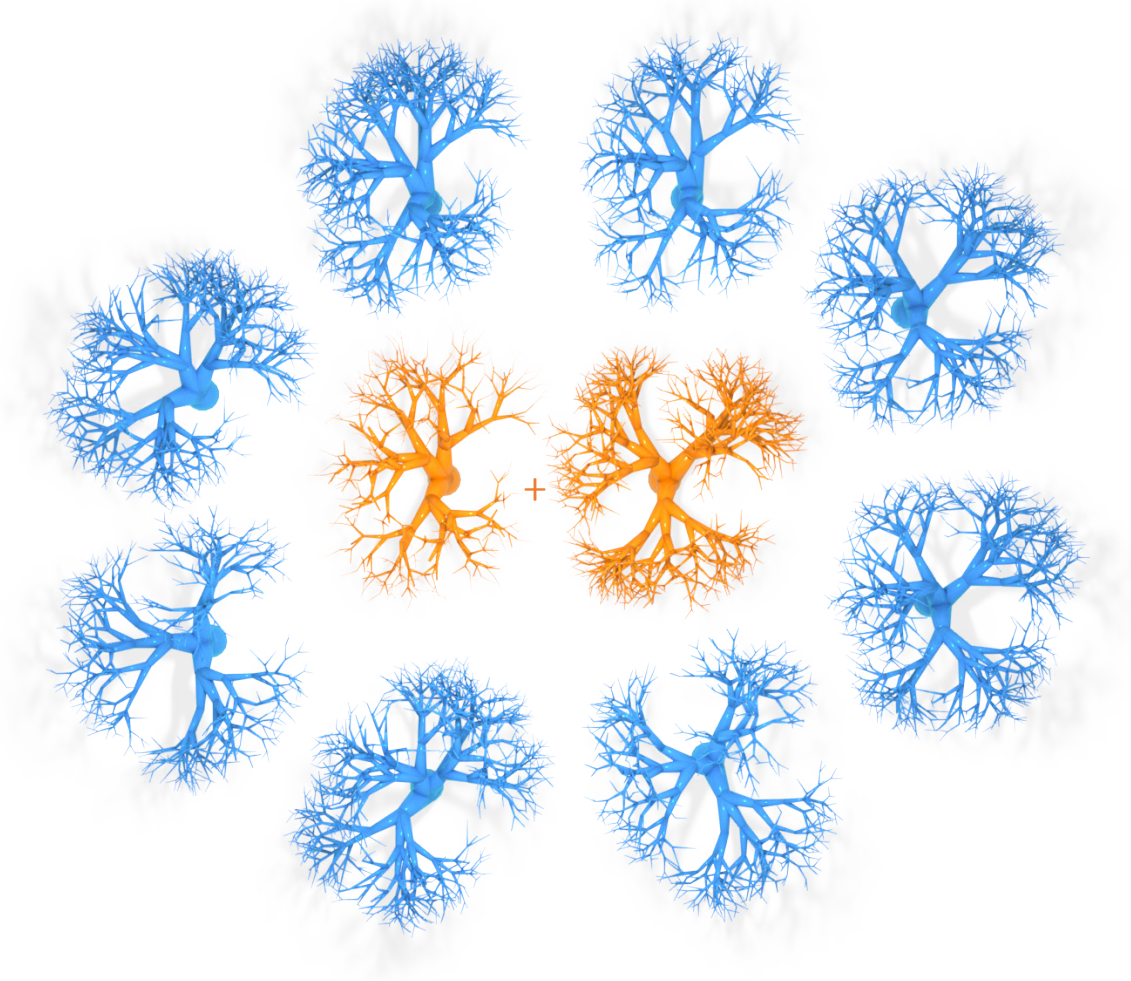


Figure 5.7. Crossover on PICO-Graphs generating trees. The two middle trees are parents (orange) that generate offsprings (blue) by using the crossover operator.

Excess and disjoint genes are copied from the fitter parent. Finally, for whichever edge transfers to the child, the parameters associated with the nodes that the edge connects are transferred to the child as well. An example of the result of the crossover operator is shown in Figure 5.7.

The children replace all the parents after the reproduction step and form a new generation. However, We keep the best individual for each species, ensuring that the best to-date solution survives. The new generation is divided into the species again and the process is repeated until a stopping criterion has been met. In our implementation, we stop if after 100 generations there is no improvement in the fitness, or, in interactive sessions, whenever user decides to stop the optimization.

Our algorithm starts from a minimal graph and progressively increases the number of nodes and edges in the graph, which increases the complexity of the generated model. The reproduction operators need to generate new solutions that would, ideally, be fitter than previous generation. However, in practice, the mutation and crossover operators often worsen the solution. We have observed that the rate of improvement gradually slows down with the increased complexity, particularly because there are a lot of mutations performed on parts of the graph that do not need to be mutated. For example, in case of tree growth in Figure 4.14, mutations to nodes generating the root and initial branches do not need to be changed after first few hundred of generations. For that reason we use *gene freezing*. We track whenever a gene mutation contributed to improvement in fitness. If there has been no improvement in a certain number of generations (50 in our implementation), the gene is frozen and cannot be mutated by parameter mutations and node insertion mutations. We randomly unfreeze frozen genes with a probability of 0.5%. Finally, we unfreeze all genes if any of the constraints have changed, so that the system can adapt to the new environment.

5.6.4 Convergence

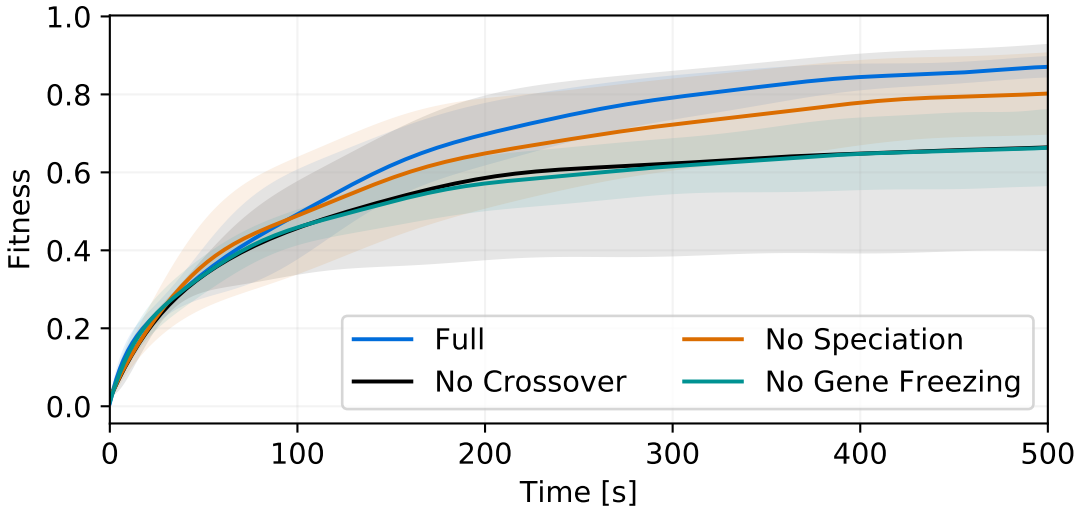


Figure 5.8. Average fitness and its standard deviation through time for the tree sketch example (Figure 4.14). Individual curves show convergence for variations of the algorithm without crossover, speciation, or gene freezing. Values are an average of five runs.

Individual parts of the algorithm influence the overall convergence of the algorithm. Figure 5.8 shows an ablation experiment where we disabled different parts of the algorithm. We use the structures from Figure 4.14, where we try to grow a tree model that matches a sketch. Besides the full algorithm, we ran a variation without crossover (*i.e.,* asexual reproduction through mutation), speciation, and gene freezing. The fitness improves best over time if all parts are used and we conclude that all of these parts contribute to better convergence.

Figure 5.9 shows the influence of the population size on the convergence and time. The results are aligned with common behavior of genetic algorithms Davis (1991); Haupt and Haupt (2004): increasing the size of the population improves the

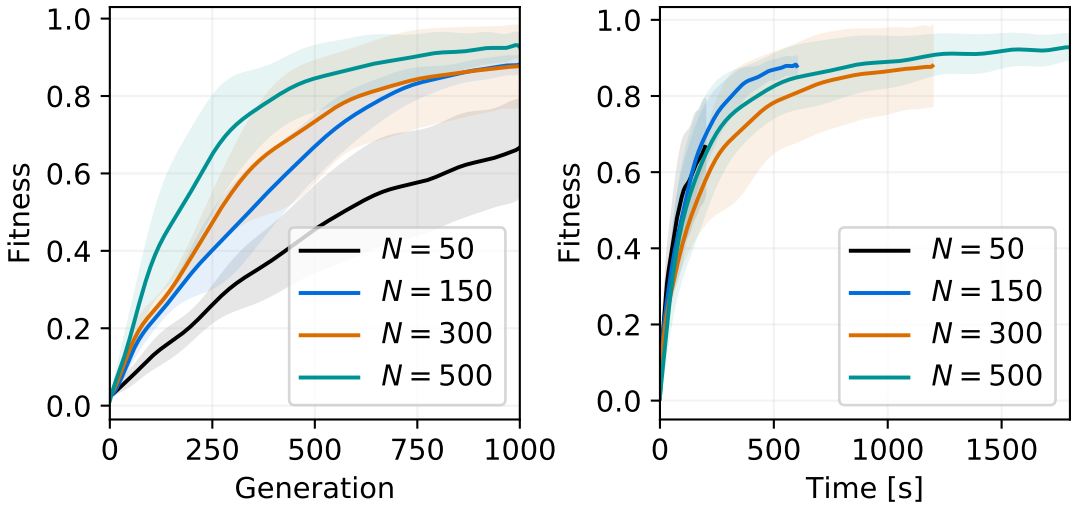


Figure 5.9. Effect of population size N on the average fitness and its standard deviation through time for the tree sketch example (Figure 4.14). The experiment was run for 1,000 generations, and five runs per curve.

convergence significantly (left). However, at a cost of increased computation time (right). We chose the population size of 150 for our examples, because it gave us a good middle ground between speed and convergence.

5.7 Implementation and Results

5.7.1 Implementation

We have implemented PICO in $\mathcal{C}$++ with support of OpenGL, GLSL and CUDA for rendering. Results were generated on a desktop computer with an Intel i7 processor clocked at 4.0 Ghz, 16 GB of RAM, and an NVIDIA Titan Xp graphics card.

We represent the 3D objects by a constructive solid geometry (CSG) tree, *i.e.,* a tree with set operations as inner nodes and geometric primitives as leafs. Because many of our objective functions require distance estimation, we represent geometric primitives by an analytic signed distance function or a signed distance field. Set operations are then performed on the signed distance d. For example, the union operation between two primitives a and b is defined as $min(d_a(p), d_b(p))$ for a point p.

We render objects by using ray-marching on the GPU implemented in CUDA, where tree traversals are expensive. Therefore we convert the CSG tree into a custom program representation using the Sethi-Ullman (Sethi & Ullman, 1970) algorithm. The resulting program is then uploaded to the CUDA constant memory and evaluated on the GPU, or evaluated directly on the CPU.

In order to quickly detect collisions, we convert all meshes into a signed distance field (SDF) representation by first voxelizing using ray-casting and then using the Fast Marching Method (Sethian, 1996). The collision volume is then calculated as the volume of the intersection of the SDF of the mesh and SDF of the CSG tree. We do this calculation recursively, by subdividing the domain's axis aligned box

(AABB), evaluating the SDF at the box's center. If the absolute distance is greater than half the diagonal of the AABB, the entire AABB is either completely inside or outside of the volume, depending on the sign of the distance. Otherwise we subdivide further until we reach a certain depth ($6 - 8$ in our implementation). The same algorithm is used to compute the mass, volume, and moment of inertia tensor of the generated object.

5.7.2 Results

Table 5.1 shows statistics of generation of the results. The input includes the number of different constraints and number of different building blocks. The optimization consists of the generation time [ms], evaluation of fitness in [ms], reproduction time [ms], number of generations in the evolutionary optimization, and the total optimization time in seconds. The output includes number of generated coordinate frames, geometric objects (*e.g.,* primitives, meshes, Gaussians) and the total number of used geometry generating operations. The most expensive fitness calculation was for the Spinning objects in Figure 5.13, which includes calculating the moment of inertia tensor, and took on average 1,746.3 [ms] for the entire population. Concerning terrains, the generation operation takes the most time due to the cost of evaluating Perlin noise, which is the bottleneck for this application.

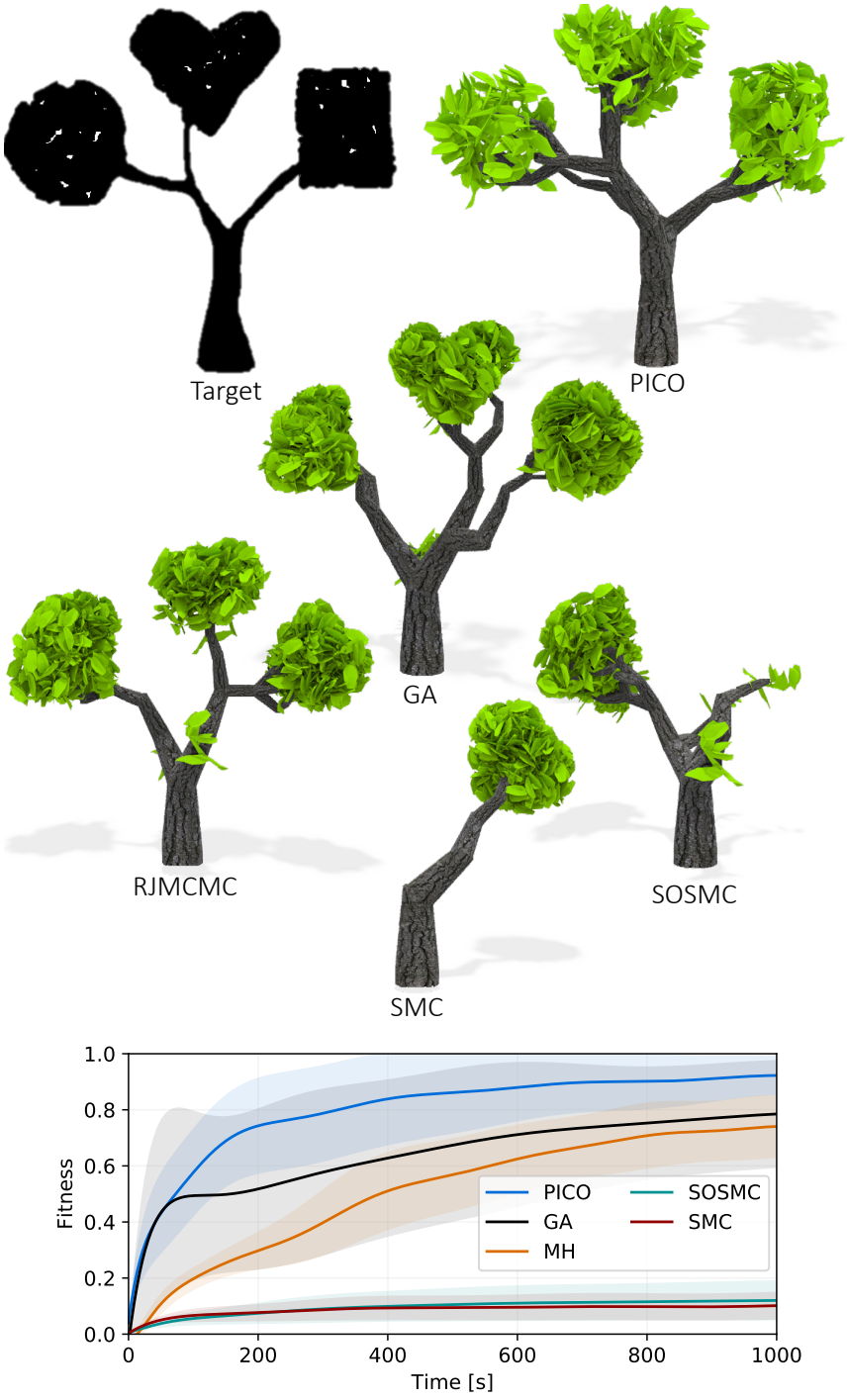


Figure 5.10. Evaluation of tree sketching from ShapeGenetics (Haubenwallner et al., 2017). Target image specifies areas where the tree should grow. Shown is a comparison of generated models from our system and models generated using ShapeGenetics implementation of various algorithms. We also show that our system achieves higher fitness faster (bottom). Curves correspond to are average fitness over 10 runs and bands show the standard deviation.

The first example in Figure 5.1 shows an array of generated 3D structures. The bottom and back part of the person are marked and a 3D chair is fully automatically generated by optimizing for touching the marked areas, stability, and small mass of the entire structure. We have building blocks defined from actual chair meshes to make the result visually plausible.

Figure 5.11. A procedural hat hanger is automatically expanded every time a new hat is added.

To evaluate our approach against existing methods, we chose to recreate the tree grammar from ShapeGenetics (Haubenwallner et al., 2017, Figure 8a). We used a single type of building block that generated branch geometry and branched three-fold, or generated leaf geometry if none of its outputs were being used. The constraint in this experiment was matching a sketch shown in Figure 5.10, and we used identical fitness and experiment setup to ShapeGenetics. We ran PICO and the implementation of Genetic Algorithm (GA) (Haubenwallner et al., 2017), Reversible Jump Monte Carlo Markov Chain (RJMCMC) (Talton, Lou, Lesser, Duke, Měch, & Koltun, 2011), Sequential Markov Chain (SMC) (Doucet, Godsill, & Andrieu, 2000) and Stochastically Ordered Markov Chain (SOSMC) (Ritchie et al., 2015). The result geometry is shown in Figure5.10 (top). Figure 5.10 (bottom) shows the mean fitness and its standard deviation as a function of time for individual methods. The SMC and SOSMC methods have issues converging from

the start and are unable to cover the entire space of the sketch. The RJMCMC and GA methods converge to satisfactory results. Our method outperforms them, especially in the first half of the optimization process.

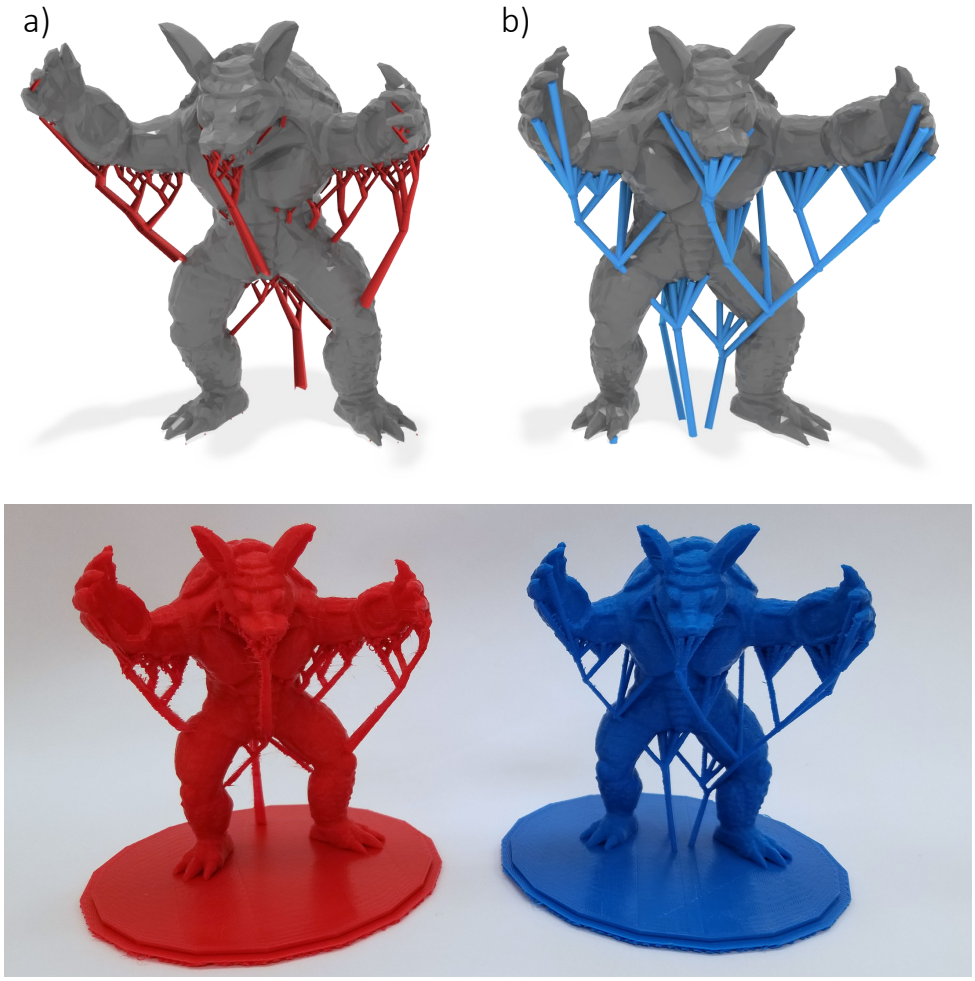


Figure 5.12. PICO can automatically generate organic supports for 3D printing. We compared our generated supports (b) to the work of (Vanek et al., 2014) (a). We used the same overhang points and same dimensions of the print and we achieved a comparable resulting weight of the used material.

An interesting application of PICO is for generating organic **support structures for 3D printing** (Figure 5.12). We compared our approach to CleverSupport (Vanek et al., 2014), a method that that grows tree-like supports from overhang points. We took the same model and used the same sampling and generated the supports using PICO. The building blocks used were simple cylinders, branching up to four-fold. There were two hard constraints used: angle with gravity vector had to be less than 45° and all the overhang points had to be connected to our generated object. The main optimization goal was minimization of the length of the structure. Note that we use multiple axiom nodes in this example to grow multiple tree-like structures at the same time. We printed the object and compared the resulting weight of used material. Ours being 85.10g, compared to 86.54g achieved by (Vanek et al., 2014). There are factors that were not considered, for example, structural strength, tips for easy removal or optimized profile of the supports. However, we show that we achieve the same task with a comparable amount of material.

We conducted a **small pilot user study** with four participants who were asked to create simple model of gazebo. PICO had all constraints set to allow for quick design and the users were asked to interact with the system by sketching constraints. After that the participants filled a small survey on a four point Likert-scale (2-strongly agree, 1-agree, -1-disagree, -2-strongly disagree). The results to our questions were: *This system is easy to use: 1.25*, *I can achieve my intent quickly: 0.5*, *I can control the design easily: 1.25*, *The response is fast: 1*, and *I need to understand procedural modeling: -4* showing that the need to understand procedural rules is not necessary in order to use the system. Moreover, the participants identified themselves as *I have previous experience in procedural modeling: 1.25*, and *I have previous experience in computer design: 0.75*.

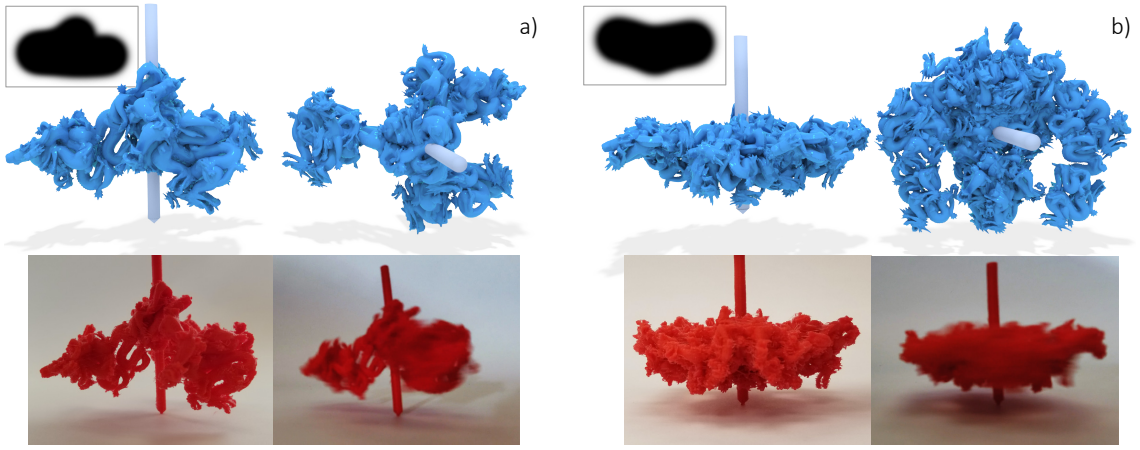


Figure 5.13. Two spinnable objects a) and b), shown from a side and top view, have been generated from Stanford dragon building blocks. The shape was guided by a sketch constraint from a single view, shown in insets and corresponding to the view on the left. The center of mass and maximal axis of inertia have been aligned using hard constraints. The system optimized the placement of building blocks to improve the quality of the spin. No symmetries were enforced but the optimization process discovered a symmetrical geometry nevertheless.

Another example shows an automatically generated structure that is able to be spun and stay balanced while **spinning** (Figure 5.13); which is an application inspired by Bächer et al. (2014). The main objective of the optimization is to achieve distribution of mass such that ratio of lateral axes of inertia to the principal axis is as small as possible. Furthermore, to control the shape of the spinning top, we sketch a rough shape from a side view. The center of mass and alignment of the principal axis of inertia with the spinning axis are enforced as hard constraints. Note that we intentionally disabled the hard symmetry constraint and we did not use symmetrical frames. Nevertheless, the system found symmetrical models automatically through optimization.

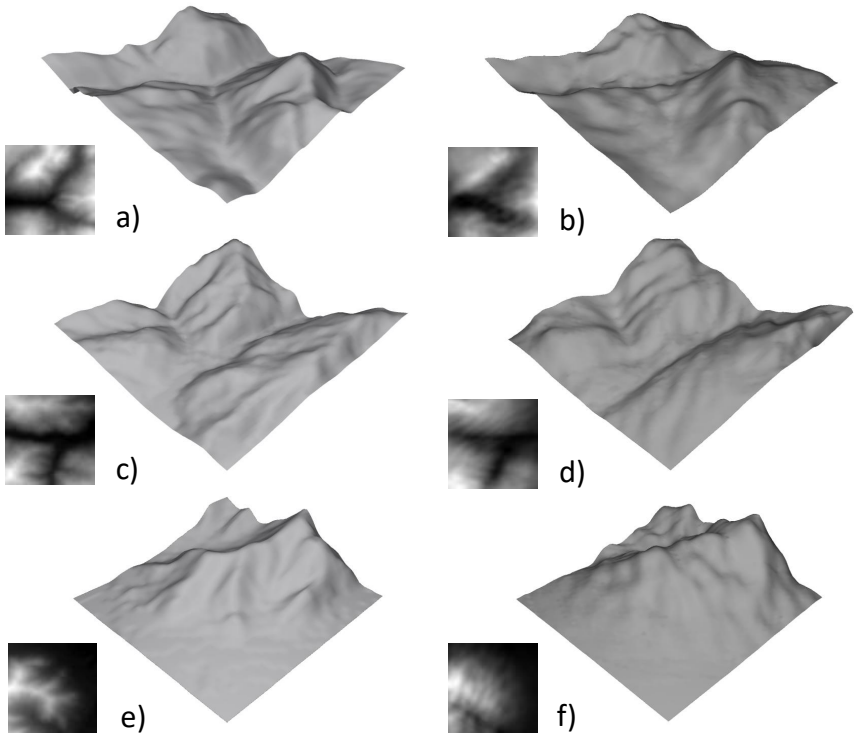


Figure 5.14. PICO matched the real terrain from the left by a set of Gaussians (right).

Although there are many methods for **procedural terrain** generation and we do not claim a contribution to this field, we wanted to show the expressiveness of our method by matching three real terrains taken as a sample of digital elevation map of Alps (resolution 64×64 pixels, 30 meters per pixel) by the PICO procedural model (Figure 5.14). The geometry generating operations were 2D Gaussians modulated by Perlin noise. The resulting height map is the sum of the contributions of all the Gaussians primitives. We used Mean Square Error (MSE) to compare the height maps. The Gaussians cannot capture well the fine details of

the terrain, but they work for the overall appearance. The optimization time was about two minutes. The accompanying video shows the optimization process.

Figure 5.11 shows an example of interactive design. A procedural **hat hanger** is automatically generated and then expanded each time a new object is added to the scene.

5.8 Conclusion

We have introduced PICO that uses PICO-Graph, which is a novel procedural model that is coupled with an evolutionary algorithm. PICO-Graph is a flexible graph representation that defines procedural generation by connecting simple geometry generating operations. Geometric objects, in our examples coordinate frames and 2D/3D geometry, are sent from axiom nodes down the graph, triggering further geometry generation in other nodes. We couple this representation with an evolutionary algorithm and we guide it using various user-defined hard and soft constraints as a means of control of the procedural generation. The evolutionary algorithm uses reproduction operators and genome compatibility defined over the PICO-Graph. Mutations are implemented as topological or parameter changes of the graph. We adapted the crossover and speciation for our procedural graph representation. The optimization allows for interactive guidance of the procedural model, but also for offline generation of complex geometry.

We have shown PICO on a variety of examples including procedural trees, automatically generated chairs, generation of supports for 3D printing, spinning objects, and even terrains. We believe that the flexibility and generality of the PICO system makes it a very powerful modeling tool for a wide range of

Table 5.1.

Statistics for the generated examples. *Input* includes number of applied constraints, and number of different types of building blocks. *Optimization* shows timing per each part: Gen. Op is the time for executing the PICO-Graph and generating geometry, Fitness evaluation, Reproduction, number of generations, and total time per iteration. The *Output* shows the number of coordinate frames passed through the graph, final geometric objects and the number of used geometry generating operations that represent the final model.

	Input		Optimization					Output		
Model	**Constraints**	**Op. Types**	**Gen. Op. [ms]**	**Fitness eval. [ms]**	**Reproduction [ms]**	**Generations**	**Total [s]**	**Frames**	**Geom. Obj.**	**Geom. Gen. Op.**
Chairs (Fig 5.1)	6	3-5	72.97	27.15	29.08	82.25	24.23	100	22	7
Armadillo supports (Fig 5.12)	4	1	248.03	728.6	195.23	207	224.67	220	429	47
Tree sketch (Fig 5.10)	1	1	134.75	678.26	138.35	2000	1747	494	495	164
Spinning objects (Fig 5.13)	2	1	9.74	1746.3	21.51	402	2216.02	533	212	38
Terrains (Fig 5.14)	1	1	248.3	0.13	14.6	169	111.5	151	86	16

applications. We have also evaluated PICO by comparing to the state-of-the art algorithms. Contrary to the existing approaches, PICO can generate existing models without the need of hand writing the underlying procedural model that is generated automatically by evolution.

Our work has a number of notable limitations. If the procedural evolution discovers an interesting pattern, it can be forgotten in next iterations or modified because of the mutations. It would be interesting to evaluate the time each structure stays in the iteration and its effect on the overall fitness. The objective function includes various criteria that can compete with each other and this can lead to a poor convergence rate in specific cases. In most cases however, PICO finds a solution very quickly that follows the user's intuition. Thanks to the interactive generation, PICO can produce results quickly and does not require any knowledge of procedural modeling as suggested by our user study. Although the constraints provide good control over the generated structure, it is not always entirely clear what the result will be. This is one of the main problems of procedural modeling and we bring a partial solution by using stochastic evolutionary algorithm with high level user guidance. Exploring finer levels of control would be beneficial.

Future work. In this work we have demonstrated PICO working with a specific set of constraints and a small set of building blocks. We think there is a potential in exploring this direction further and adapting PICO to even more domains. It would be interesting to conduct additional studies with both artists and designers to better understand workflow patterns that can enable further system refinements. While our current optimization process is efficient, we believe there still exists opportunity to improve the convergence of our method. This includes not only the

raw performance of our system, but its ability to find high quality solutions in the large search space.

CHAPTER 6. CONCLUSIONS & FUTURE WORK

The purpose of this dissertation was to explore ways to improve control in procedural modeling. Procedural systems offer a powerful way of modeling geometry. A small representation, *e.g.,* a handful a rules and parameters, can produce complicated geometry and provide ability to model not only natural phenomena but various synthetic objects as well. The automatic nature of these systems is desirable as it allows fast generation of great number and variety of models. One of the main problems in procedural modeling, however, is the lack of ability to control the complex generative process. As such, control in procedural modeling is an active area of research. Any advancements in this area benefit a wide range of people in variety of situations, from individual artists looking to produce creative works to large companies with large content production pipelines.

In this work we have focused on answering, at least in part, the question *whether procedural systems can be controlled.* We focused on three different instances of procedural systems: modeling via erosion and deposition simulation, modeling of 3D curves via 2D sketching, and modeling geometry via generic user-defined constraints. The goal of this approach was to gain insights in these specific domains that could generalize to procedural modeling as a whole.

6.1 Erosion and Deposition Simulation

To answer the question *whether erosion and deposition simulation be controlled to model objects of arbitrary topology*, we have focused on the issues that make simulations difficult to control. The two problems we looked at in terms of simulation were *interactivity* and *local control*.

Simulations are, in general, computationally expensive, which is especially true for accurate physics-based simulation of fluid. To achieve *interactivity*, we employed a SPH based simulation implemented on the GPU, which allowed us to run the simulation at interactive frame rates. Because the erosion and deposition phenomena are localized only to the surface of the objects, we explicitly kept track of the surface volumetric elements. We then computed the force equations only for these surface elements, thus eliminating unnecessary work. Finally, we employed a dual representation of the material: he static uneroded material was represented using volumetric elements, which were considered for any computation only if they were near or at the surface. The dynamic material, i.e., the advected or depositing substance, was represented using a particle in the SPH system. This allowed us to focus only on the parts of the whole system that were undergoing change and further speed up the simulation.

Another issue with simulations is that they are difficult to influence while running. Therefore, modifications are often made only to the initial and boundary conditions. To help with this issue, we designed an interactive interface that allows the user to emit the fluid from a chosen position and angle and at chosen quantity and pressure. This enables a degree of *local and direct control* and allows the user to use fluid in a similar fashion as a sculpting brush, while retaining physically plausible emergent effects on the modeled geometry.

There are several **limitations** of the proposed method. First, the simulation accuracy is limited in order to achieve interactivity. For example, we have not implemented any kinematic simulation for bulk material, only for advected material. Unphysical behavior can therefore occur, such as objects being suspended in air without falling. Furthermore, many of the material properties are only roughly approximated. Finally, the data structure and simulation domain are kept at fixed resolution. While they provide acceleration, we would like to implement an adaptive version, *e.g.,* an adaptive mesh, which refines resolution where detail is needed the most. Improving both the speed and accuracy of the simulation would enable users to model complicated objects more easily. This would enable one avenue of future work: faster than real time simulation, *i.e.,* integrating the simulation well into the future in a short period of time. Such feature would allow the user to see effects of his/hers actions further in the future, and possibly mimic the process of real erosion, which happens over millennia. Currently, any changes in the system are permanent and the user has no ability to undo any actions or simulation time. While it is possible to erode previously deposited material, it is not possible to locally add material in directions other than the gravity vector. To improve this, simulation history could be kept to easily undo simulation time. Furthermore, additional operations similar to traditional sculpting, such as adding and removing arbitrary material, could be implemented to help with the modeling process and increase local control. Many of the examples presented rely on already existing geometry with pre-defined materials. While it is possible to model via deposition only, it would be beneficial to provide users with the ability to easily assign materials to existing objects. Finally, an integration with a traditional modeling workflow for modeling rough shapes that can be eroded later could help this system to become an end-to-end modeling framework.

6.2 3D Curve Sketching

Modeling of a single 3D curve for a specific purpose, such as animation trajectory or generalized cylinder geometry, requires great deal of control. Traditionally, detailed manual specification from the artist is required. To ease this tedious process, we asked *whether it possible to provide fine control of 3D curve generation, given a 2D sketch from single view and existing geometry as context?*. First, we focused on *enforcing given constraints* as much as possible, i.e., the final curve should look like the sketch from the initial viewpoint. However, there are remaining degrees of freedom of the curve that are left unspecified. To reconcile this issue we provide the user with the ability to place existing and transient geometry in the scene as *context*, which greatly reduces the number of possible 3D curves. To find a single final solution we applied the following *assumption*: the intended curve is fair, i.e., its curvature is monotonous. Because our assumption may not match the user's intent or is affected by an imprecise stroke, we provide *editing capabilities* that allow the user to provide more information, e.g., the depth or a shape of a certain segment. In conclusion, the user has access to tools that are similar to those afforded by manual modeling, with the benefit of sketching the curve with a single stroke from a single view.

Although the presented algorithm can greatly improve the time to model 3D curves, there are several **limitations**. While it is possible to sketch only from a single view, in practice, multiple view points are necessary to visually evaluate the curve and make adjustments to any inaccuracies present. We were unable to completely eradicate these inaccuracies due to the fundamentally reduced degree of freedom of a a single 2D view and perspective foreshortening, resulting in cases when the final curve shape is not the shape that the user expected. Some of the

promising approaches to help with these problems include using 6-DOF input tracking devices, providing multiple views at the same time, or using a virtual reality headset. Another limitation is that we assume that the final curve shape is supposed to be smooth. It would be interesting to explore ways to model curves with sharp features, such as corners. We also assume that the distance of the curve to the object is determined mainly by the stroke's distance to the object in screen space. However, the distance and shape of the curve could be guided by other information, *e.g.,* salient parts of the object or surface texture. Furthermore, we only focused on single curve at a time, but an interesting avenue of further research is sketching branching curves or curve networks. Finally, a possible approach that does not rely on assumptions is to learn the expected curve shape as a function of the sketch and context via machine learning methods. The difficulty in this approach, however, is measuring accuracy, as there may be multiple correct solutions for a single sketch, and gathering enough data.

6.3 Procedural Iterative Constrained Optimizer

We investigated the question *whether it is possible to control procedural system generating arbitrary geometry using user-defined constraints.* The arbitrary geometry is achieved by a flexible operator graph representation. The main object traveling through the procedural graph is an affine transformation, applicable in most 2D or 3D modeling domains. The operations themselves can then be defined as placing any geometry to the scene according to the transformation, or modifying existing geometry. We demonstrate this by generating models using CSG-like primitives for volumetric models and Gaussian functions for terrains.

The main form of control we chose were *user-defined constraints*. These constraints reduce the number of possible geometries that can be generated by the space of all procedural graphs. Some of these constraints, i.e., contact surfaces or forbidden regions, provide *context* for the generated geometry, similar to the transient geometry in the 3D curve sketching system. Furthermore, by formulating more abstract constraints such as stability or ability to spin, the user can model the *function* of the object without worrying about the exact geometry. Finally, multiple constraints can be formulated at the same time. This leads to a multi-objective optimization problem where the individual constraints may oppose each other. Because it may be impossible to minimize all the objectives simultaneously, we provide the user with control of the *importance* of individual constraints, placing him or her in charge of the design trade-offs. A degree of global control of the generative process is provided in form of specifying parameter spaces, i.e., the distribution from which the optimization can draw when searching for optimal parameters of the geometric operations. These spaces provide the ability to influence the overall style of the model, control the symmetry and recursion depth of the model.

The proposed framework suffers from several **limitations**. A degree of *local control* is possible through direct modification of the graph, however that may be unintuitive and may produce behavior not expected by the user. To really achieve direct control over the geometry, we would have to provide control of individual generated geometry components and transfer modifications to the procedural graph. In terms of *global control*, the system has only limited control of style via defining parameter spaces. A possible avenue for future work would be to perform a style transfer or style matching from existing objects or images. We presented a

basic evolutionary design workflow, however there are many improvements that can be made: evolving and visualizing multiple populations at the same time, letting user manually assign fitness of solutions or entire populations, and reverting to previous generations whenever a solution desired by the user died or mutated excessively. While the system quickly converges to viable solutions, it prefers exploitation over exploration. Any speed improvements could allow for wider exploration and possibly even better final solutions. Finally, we would be interested in seeing this framework applied in other domains besides CSG models and terrain modeling, especially when employing accurate physical simulation or other carefully crafted objective functions.

6.4 Summary

We have discussed the limitations of the three separate procedural systems and possible future work in these three areas. While the proposed procedural systems are quite diverse, we can see several patterns emerge that may be applicable to procedural systems in general.

Let us compare traditional modeling methods, *i.e.,* manual modeling such as polygonal modeling or sculpting, and procedural methods. There are several dimensions on which we can evaluate these methods and the following list is not exhaustive. Figure 6.1 visualizes the relative ability of these two modeling paradigms in the evaluated dimensions.

Interactivity is one of the aspects that help control procedural generation and is a crucial part of traditional modeling methods. For procedural methods in particular, near real-time feedback of the system helps the user to perceive

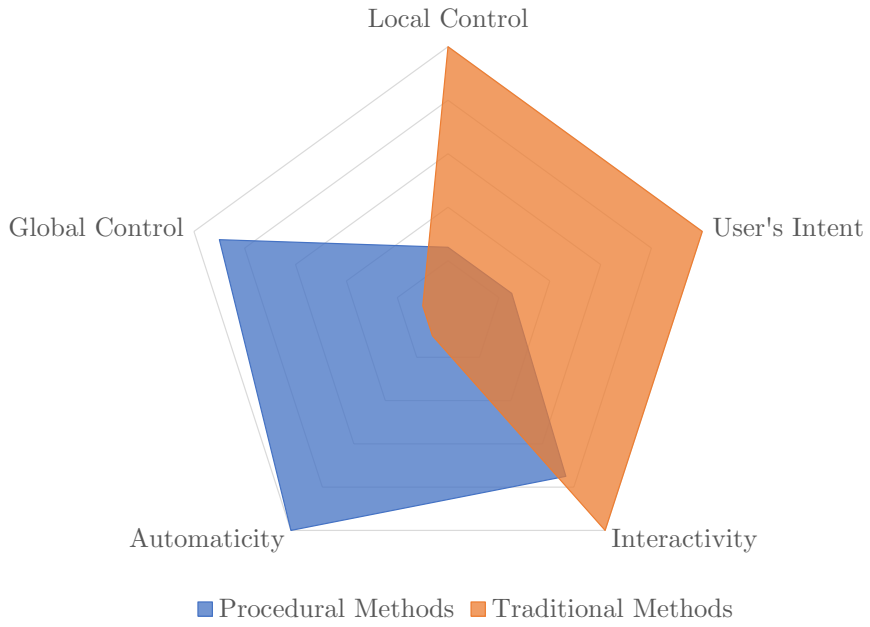


Figure 6.1. Relative comparison of the modeling paradigms across different dimensions.

and understand the underlying process of the generative process. Therefore, the ability of the user to influence the system during the generation process and see immediate consequences seems to help with learning to control the system and achieving the intended result.

User's Intent. One of the key elements of procedural systems is their non-linearity. It means, however, that the magnitude of changes in the output are disproportionate to the changes in the input. Furthermore, changes in the input may have side effects not known to the user. This results in unexpected behavior and hinders the user's ability to achieve his or hers intent. For example in the 3D curve sketching system, a curve segment may end up at a different distance from the viewpoint than the user expected, or in PICO, a

combination of user-defined constraints may render the intended shape impossible to find. On the other hand, the behavior of traditional modeling systems can be predicted because a linear relationship exists between user's input and the system's output.

Local Control. Traditional methods excel at providing users the ability to change individual parts of the modeled object. In comparison, procedural systems are often used in settings where a complicated model with many parts is desired and the emphasis is on overall appearance and not individual details. Local control seems to be an important aspect that makes procedural systems applicable in situations where small details matter as well. However, it is often not obvious how to handle local changes in relation to overall system and local control is typically afforded only indirectly, through specifying various spatial constraints.

Global Control refers to the ability of the user to change the overall apperance or style of the modeled geometry. Traditional methods suffer in this regard as they focus on local control and offer only limited global tools, for example, texturing. Procedural systems encode the global structure in their rules and parameters and therefore require less steps to change the global characteristics of the model. For example, in systems for modeling trees, parameters are readily available to change overall branching angle and other global characteristics.

Automaticity is one of the core properties of procedural systems. Traditional modeling workflows have been incorporating procedural tools to speed up the modeling process, such as grid snapping when placing objects, smart selection of geometry's elements, and automatic symmetry in sculpting. They are still,

however, predicated on the user manually interacting with the system from start to finish.

6.5 Future Work

Procedural and traditional methods differ in fundamental ways. It may be impossible to develop a system to combine the strengths and eliminate weaknesses of both. However, we believe there is room for improvement for procedural methods in terms of control. In particular, in the areas that they are lacking the most as visualized by Figure 6.1. We layout possible remedies and future avenues of research in terms of these comparative dimensions.

Interactivity has greatly improved in the recent years thanks to the availability of faster hardware and employment of more sophisticated and parallelizable methods. However, we believe it is crucial to improve the speed even further, in terms of both optimization and generation, to make the modeling process even easier and allow modeling of highly detailed geometry.

Achieving *user's intent* and reducing unexpected behavior seems to be the most difficult problem for procedural systems. A possible symptom relief is providing control of history of the generative process, *e.g.,* undo and redo operations, to revert any undesirable changes. Similarly, users can benefit from learning and understanding of how the system behaves, and therefore better anticipate the consequences of their actions. Design space exploration tools, such as the work of Talton et al. (2009), can further help with teaching the user the behavior of the system and provide useful suggestions of designs. However, it is difficult to make procedural generation produce expected results to begin with. A promising

approach is linearization (Yumer et al., 2015), a process which makes the input parameters have linear relationship with the output. Another possible avenue of future work is learning how the system should anticipate the user's intent, instead of user anticipating the behavior of the system. This approach has been employed in other complex tasks, such as text prediction (Garay-Vitoria & Abascal, 2006) and recently in traditional sculpting (Peng, Xing, & Wei, 2018). Additional information given to the system by the user should theoretically improve the system's ability to match user's intent. Our experiments with user providing *context* in form of existing geometry supports this idea. An analogy is readily available in the practice of traditional modeling - many artists choose to use *blueprints*, most often photos, to provide context, inspiration, and template for their modeling process. The act of placing a transient object in the scene, as opposed to mathematically or programmatically defining it, is a relatively easy task for the user. The benefit, especially with using complicated existing geometry, is that the amount of structured and useful information given to the system is disproportionate to the time spent on placing the object. Similarly, *sketching* is relatively straightforward task even for novice users and provides useful information that can guide the system and can be as simple as a single stroke. Therefore, we hypothesize that methods of input that transfer the most amount of useful and structured information from the user in the least amount of time are the most beneficial in the overall control of procedural systems.

In terms of improving *local control*, we can learn from traditional modeling. The methods already mentioned, providing *context* and *sketching*, are useful for specifying local constraints. However, we think that *selection and editing* strategy is important as well, as we demonstrated in case of modifying 3D curves. Selection

should allow the user to transparently specify which parts of the model should and should not be modified. Similarly, edit operations should transparently modify the model without unexpected side effects. We therefore posit that implementing traditional local control tools, *e.g.,* selection and editing, while retaining the benefits or procedural methods, *e.g.,* emergence, is a beneficial direction of further research.

To conclude, *procedural systems can be controlled* and we identified and demonstrated some of the useful strategies that may be beneficial for many existing and future procedural systems:

- maximizing interactivity
- eliminating unexpected behavior
- providing high-bandwidth structured information input methods
- providing local control methods similar to traditional modeling

As we explored only three different procedural systems, the presented techniques of control may not be applicable to all procedural systems. We suggest that further exploration in general and specific forms of control of procedural modeling is an important direction of future research.

LIST OF REFERENCES

LIST OF REFERENCES

Abelson, H., Sussman, G., & Sussman, J. (1997). *Structure and interpretation of computer programs* (2nd ed.). New York, NY, USA: McGraw-Hill, Inc.

Akenine-Moller, T., Haines, E., & Hoffman, N. (2018). *Real-time rendering*. AK Peters/CRC Press.

Alduán, I., & Otaduy, M. A. (2011). SPH granular flow with friction and cohesion. In *Proceedings of the 2011 acm siggraph/eurographics symposium on computer animation* (pp. 25–32). New York, NY, USA: ACM. doi: 10.1145/2019406.2019410

Aliaga, D. G., Demir, I., Benes, B., & Wand, M. (2016). Inverse procedural modeling of 3d models for virtual worlds. In *Acm siggraph 2016 courses* (pp. 16:1–16:316). New York, NY, USA: ACM.

Aliaga, D. G., Rosen, P. A., & Bekins, D. R. (2007). Style grammars for interactive visualization of architecture. *IEEE transactions on visualization and computer graphics*, *13*(4).

Aliaga, D. G., Vanegas, C. A., & Benes, B. (2008). Interactive example-based urban layout synthesis. In *Acm transactions on graphics (tog)* (Vol. 27, p. 160).

Anh, N. H., Sourin, A., & Aswani, P. (2007). Physically based hydraulic erosion simulation on graphics processing unit. In *Proceedings of the graphite* (pp. 257–264). New York, NY, USA: ACM. doi: 10.1145/1321261.1321308

Aono, M., & Kunii, T. L. (1984). Botanical tree image generation. *IEEE Computer Graphics and Applications*, *4(5)*, 10–34.

Autodesk. (2018a). *3ds max.* `https://www.autodesk.com/products/3ds-max/overview`. (Accessed: 2018-11-27)

Autodesk. (2018b). *Maya.* `https://www.autodesk.com/products/maya/overview`. (Accessed: 2018-11-27)

Bächer, M., Whiting, E., Bickel, B., & Sorkine-Hornung, O. (2014). Spin-it: Optimizing moment of inertia for spinnable objects. *ACM Trans. Graph.*, *33*(4), 96:1–96:10.

Bae, S.-H., Balakrishnan, R., & Singh, K. (2008). Ilovesketch: as-natural-as-possible sketching system for creating 3d curve models. In *Proc. of user interface software and technology* (pp. 151–160).

Belhadj, F. (2007). Terrain modeling: A constrained fractal model. In *Proceedings of the 5th international conference on computer graphics, virtual reality, visualisation and interaction in africa* (pp. 197–204). New York, NY, USA: ACM. doi: 10.1145/1294685.1294717

Beltramelli, T. (2017). pix2code: Generating code from a graphical user interface screenshot. *arXiv preprint arXiv:1705.07962*.

Benes, B., & Forsbach, R. (2001). Layered data representation for visual simulation of terrain erosion. In *Proceedings of the 17th spring conference on computer graphics* (pp. 80–). Washington, DC, USA: IEEE Computer Society.

Beneš, B., Šťava, O., Měch, R., & Miller, G. (2011). Guided procedural modeling. In *Computer graphics forum* (Vol. 30, pp. 325–334).

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

Benes, B., Šťava, O., Měch, R., & Miller, G. (2011, September). Guided procedural modeling. *Computer Graphics Forum*, *21*(3), 325-334.

Ben-Haim, D., Harary, G., & Tal, A. (2010). Piecewise 3d euler spirals. In *Proceedings of the 14th acm symposium on solid and physical modeling* (pp. 201–206). New York, NY, USA: ACM. doi: 10.1145/1839778.1839810

Bentley, P. J., & Wakefield, J. P. (1998). Finding acceptable solutions in the pareto-optimal range using multiobjective genetic algorithms. In *Soft computing in engineering design and manufacturing* (pp. 231–240). Springer.

Bergen, S., & Ross, B. J. (2013, Sep 01). Aesthetic 3d model evolution. *Genetic Programming and Evolvable Machines*, *14*(3), 339–367.

Bessmeltsev, M., Chang, W., Vining, N., Sheffer, A., & Singh, K. (2015, November). Modeling character canvases from cartoon drawings. *ACM Trans. Graph.*, *34*(5), 162:1–162:16. doi: 10.1145/2801134

Blinn, J. F. (1982). Light reflection functions for simulation of clouds and dusty surfaces. In *Proceedings of siggraph* (pp. 21–29). ACM. doi: 10.1145/800064.801255

Boechat, P., Dokter, M., Kenzel, M., Seidel, H.-P., Schmalstieg, D., & Steinberger, M. (2016). Representing and scheduling procedural generation using

Operator Graphs. *ACM Transactions on Graphics*, *35*(6), 1–12. doi: 10.1145/2980179.2980227

Bosch, C., Pueyo, X., Merillou, S., & Ghazanfarpour, D. (2004). Ghazanfarpour d.: A physically-based model for rendering realistic scratches. *Computer Graphics Forum*, 361–370.

Boudon, F., Prusinkiewicz, P., Federl, P., Godin, C., & Karwowski, R. (2003). Interactive design of bonsai tree models. In *Computer graphics forum* (Vol. 22, pp. 591–599).

Bremer, P. T., Porumbescu, S. D., Kuester, F., Joy, K., & Hamann, B. (2001). Virtual clay modeling using adaptive distance fields. In *Proceedings of the 2001 uc davis student workshop on computing, tr cse-2001-7.* California, Davis: University of California, Davis.

Buchanan, P., Mukundan, R., & Doggett, M. (2013). Automatic single-view character model reconstruction. In *Proceedings of the international symposium on sketch-based interfaces and modeling* (pp. 5–14).

Burger, B., Paulovic, O., & Hasan, M. (2002). Realtime visualization methods in the demoscene. In *Proceedings of the central european seminar on computer graphics* (pp. 205–218).

Busaryev, O., Dey, T. K., & Wang, H. (2013, July). Adaptive fracture simulation of multi-layered thin plates. *ACM Trans. Graph.*, *32*(4), 52:1–52:6. doi: 10.1145/2461912.2461920

Chang, Y.-X., & Shih, Z.-C. (2000). Physically-based patination for underground objects. *Computer Graphics Forum*, *19*(3), 109–117. doi: 10.1111/1467-8659.00403

Chang, Y.-X., & Shih, Z.-C. (2003). The synthesis of rust in seawater. *The Visual Computer*, *19*(1), 50-66. doi: 10.1007/s00371-002-0172-0

Chen, G., Esch, G., Wonka, P., Müller, P., & Zhang, E. (2008). Interactive procedural street modeling. In *Acm transactions on graphics (tog)* (Vol. 27, p. 103).

Chen, X., Neubert, B., Xu, Y.-Q., Deussen, O., & Kang, S. B. (2008, December). Sketch-based tree modeling using markov random field. *ACM Trans. Graph.*, *27*(5), 109:1–109:9. doi: 10.1145/1409060.1409062

Chomsky, N. (1956, Sep.). Three models for the description of language. *IRE Transactions on Information Theory*, *2*(3), 113-124. doi: 10.1109/TIT.1956.1056813

Cohen, J. M., Markosian, L., Zeleznik, R. C., Hughes, J. F., & Barzel, R. (1999). An interface for sketching 3D curves. In *Proc. of i3d* (pp. 17–21). New York, NY, USA: ACM. doi: 10.1145/300523.300655

Coleman, P., & Singh, K. (2006, May). Cords: Geometric curve primitives for modeling contact. *IEEE Comput. Graph. Appl.*, *26*(3), 72–79. doi: 10.1109/MCG.2006.54

Cordier, F., & Seo, H. (2007, January). Free-form sketching of self-occluding objects. *IEEE Comput. Graph. Appl.*, *27*(1), 50–59. doi: 10.1109/MCG.2007.8

Cordonnier, G., Cani, M.-P., Benes, B., Braun, J., & Galin, E. (2018, May). Sculpting mountains: Interactive terrain modeling based on subsurface geology. *IEEE Transactions on Visualization and Computer Graphics*, *24*(5), 1756-1769. doi: 10.1109/TVCG.2017.2689022

Cordonnier, G., Galin, E., Gain, J., Benes, B., Guérin, E., Peytavie, A., & Cani, M.-P. (2017a). Authoring landscapes by combining ecosystem and terrain erosion simulation. *ACM Transactions on Graphics (TOG)*, *36*(4), 134.

Cordonnier, G., Galin, E., Gain, J., Benes, B., Guérin, E., Peytavie, A., & Cani, M.-P. (2017b, July). Authoring landscapes by combining ecosystem and terrain erosion simulation. *ACM Trans. Graph.*, *36*(4), 134:1–134:12. doi: 10.1145/3072959.3073667

Cutler, B., Dorsey, J., McMillan, L., Müller, M., & Jagnow, R. (2002). A procedural approach to authoring solid models. In *Proceedings of siggraph* (pp. 302–311). ACM. doi: 10.1145/566570.566581

Dassault Systèmes. (2018). *Solidworks.* `https://www.solidworks.com/`. (Accessed: 2018-11-27)

Davis, L. (1991). Handbook of genetic algorithms.

De Carpentier, G. J., & Bidarra, R. (2009). Interactive gpu-based procedural heightfield brushes. In *Proceedings of the 4th international conference on foundations of digital games* (pp. 55–62).

Demir, İ., & Aliaga, D. G. (2018). Guided proceduralization: Optimizing geometry processing and grammar extraction for architectural models. *Computers & Graphics*.

De Paoli, C., & Singh, K. (2015, July). Secondskin: Sketch-based construction of layered 3D models. *ACM Trans. Graph.*, *34*(4), 126:1–126:10. doi: 10.1145/2766948

de Villiers, M., & Naicker, N. (2006). *A sketching interface for procedural city generation* (Tech. Rep.). Citeseer.

Dorsey, J., Edelman, A., Jensen, H. W., Legakis, J., & Pedersen, H. K. (1999). Modeling and rendering of weathered stone. In *Proceedings of siggraph* (pp. 225–234). ACM. doi: 10.1145/311535.311560

Dorsey, J., & Hanrahan, P. (1996). Modeling and rendering of metallic patinas. In *Proceedings of siggraph* (pp. 387–396). ACM. doi: 10.1145/237170.237278

Doucet, A., Godsill, S., & Andrieu, C. (2000). On sequential monte carlo sampling methods for bayesian filtering. *Statistics and computing*, *10*(3), 197–208.

Du, T., Inala, J. P., Pu, Y., Spielberg, A., Schulz, A., Rus, D., ... Matusik, W. (2018, December). Inversecsg: Automatic conversion of 3d models to csg trees. *ACM Trans. Graph.*, *37*(6), 213:1–213:16. Retrieved from `http://doi.acm.org/10.1145/3272127.3275006` doi: 10.1145/3272127.3275006

Ebert, D., Musgrave, K., Peachey, D., Perlin, K., & Worley, S. (1998). *Texturing and modeling: A procedural approach*. Academic Press Professional.

Eichhorst, P., & Savitch, W. J. (1980). Growth functions of stochastic Lindenmayer systems. *Information and Control*, *45*(3), 217–228. doi: 10.1016/S0019-9958(80)90593-8

Ellis, K., Ritchie, D., Solar-Lezama, A., & Tenenbaum, J. (2018). Learning to infer graphics programs from hand-drawn images. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, & R. Garnett (Eds.), *Advances in neural information processing systems 31* (pp. 6060–6069). Curran Associates, Inc.

Emilien, A., Poulin, P., Cani, M.-P., & Vimont, U. (2014). Interactive procedural modelling of coherent waterfall scenes. *Computer Graphics Forum*. doi: 10.1111/cgf.12515

Emilien, A., Vimont, U., Cani, M.-P., Poulin, P., & Benes, B. (2015, July). Worldbrush: Interactive example-based synthesis of procedural virtual worlds. *ACM Trans. Graph.*, *34*(4), 106:1–106:11. doi: 10.1145/2766975

Epic Games. (2018). *Unreal engine.* `https://www.unrealengine.com`. (Accessed: 2018-11-27)

ESRI. (2017, October). *City engine.* http://www.esri.com/software/cityengine.

Fan, L., Wang, R., Xu, L., Deng, J., & Liu, L. (2013). Modeling by drawing with shadow guidance. In *Comp. graph. forum* (Vol. 32, pp. 157–166).

Farin, G. E. (1996). *Curves and surfaces for computer-aided geometric design: A practical code* (4th ed.). Orlando, FL, USA: Academic Press, Inc.

Fernando, R., Haines, E., & Sweeney, T. (2001). GPU gems: programming techniques, tips, and tricks for real-time graphics. *Dimensions*, *7*(4), 816.

Fiser, M., Benes, B., Galicia, J. G., Abdul-Massih, M., Aliaga, D. G., & Krs, V. (2016). Learning geometric graph grammars. In *Proceedings of the 32nd spring conference on computer graphics* (pp. 7–15). New York, NY, USA: ACM. doi: 10.1145/2948628.2948635

Flake, G. W. (1998). *The computational beauty of nature: Computer explorations of fractals, chaos, complex systems, and adaptation*. MIT Press.

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

Fu, C.-W., Xia, J., & He, Y. (2010). Layerpaint: A multi-layer interactive 3d painting interface. In *Proceedings of the sigchi conference on human factors in computing systems* (pp. 811–820). New York, NY, USA: ACM. doi: 10.1145/1753326.1753445

Fu, H., Wei, Y., Tai, C.-L., & Quan, L. (2007). Sketching hairstyles. In *Proceedings of the 4th eurographics workshop on sketch-based interfaces and modeling* (pp. 31–36). New York, NY, USA: ACM. doi: 10.1145/1384429.1384439

Gagnon, J., & Paquette, E. (2011). Procedural and interactive icicle modeling. *The Visual Computer*, *27*(6-8), 451–461. doi: 10.1007/s00371-011-0584-9

Gain, J., Marais, P., & Strasser, W. (2009). Terrain sketching. In *Proceedings of the 2009 symposium on interactive 3d graphics and games* (pp. 31–38). New York, NY, USA: ACM. doi: 10.1145/1507149.1507155

Gain, J., Merry, B., & Marais, P. (2015, May). Parallel, realistic and controllable terrain synthesis. *Comput. Graph. Forum*, *34*(2), 105–116. doi: 10.1111/cgf.12545

Galin, E., Guérin, E., Peytavie, A., Cordonnier, G., Cani, M.-P., Benes, B., & Gain, J. (2019). A Review of Digital Terrain Modeling. *Comp. Gr. Forum*, *38*(2).

Gamito, M. N., & Musgrave, F. K. (2001). Procedural landscapes with overhangs. In *10th portuguese computer graphics meeting* (Vol. 2).

Garay-Vitoria, N., & Abascal, J. (2006, Mar 01). Text prediction systems: a survey. *Universal Access in the Information Society*, *4*(3), 188–203. doi: 10.1007/s10209-005-0005-9

Garcia, S., & Romão, L. (2015). A design tool for generic multipurpose chair design. In G. Celani, D. M. Sperling, & J. M. S. Franco (Eds.), *Computer-aided architectural design futures. the next city - new technologies and the future of the built environment* (pp. 600–619). Berlin, Heidelberg: Springer.

Gendler, R., & GaBany, R. J. (2015). The birth and evolution of astrophotography. In *Breakthrough!* (pp. 1–13). Springer.

Génevaux, J.-D., Galin, E., Peytavie, A., Guérin, E., Briquet, C., Grosbellet, F., & Benes, B. (2015). Terrain modelling from feature primitives. In *Computer graphics forum* (Vol. 34, pp. 198–210).

Gobron, S., & Chiba, N. (2001a). Crack pattern simulation based on 3d surface cellular automata. *The Visual Computer*, *17*(5), 287-309. doi: 10.1007/s003710100099

Gobron, S., & Chiba, N. (2001b). Simulation of peeling using 3d-surface cellular automata. In *Proceedings of pacific computer* (pp. 338–). IEEE Computer Society.

Greuter, S., Parker, J., Stewart, N., & Leach, G. (2003). Real-time procedural generation of 'pseudo infinite' cities. In *Proceedings of the 1st international conference on computer graphics and interactive techniques in australasia and south east asia* (pp. 87–ff). New York, NY, USA: ACM. doi: 10.1145/604471.604490

Guérin, E., Digne, J., Galin, E., Peytavie, A., Wolf, C., Benes, B., & Martinez, B. (2017a, November). Interactive example-based terrain authoring with conditional generative adversarial networks. *ACM Trans. Graph.*, *36*(6), 228:1–228:13. doi: 10.1145/3130800.3130804

Guérin, E., Digne, J., Galin, E., Peytavie, A., Wolf, C., Benes, B., & Martinez, B. (2017b, November). Interactive example-based terrain authoring with conditional generative adversarial networks. *ACM Trans. Graph.*, *36*(6), 228:1–228:13. doi: 10.1145/3130800.3130804

Guerrero, P., Jeschke, S., Wimmer, M., & Wonka, P. (2015). Learning shape placements by example. *ACM Trans. Graph.*, *34*(4), 108:1–108:13.

Hädrich, T., Benes, B., Deussen, O., & Pirk, S. (2017, May). Interactive modeling and authoring of climbing plants. *Comput. Graph. Forum*, *36*(2), 49–61. doi: 10.1111/cgf.13106

Haubenwallner, K., Seidel, H.-P., & Steinberger, M. (2017). Shapegenetics: Using genetic algorithms for procedural modeling. *Computer Graphics Forum*, *36*(2), 213–223.

Haugstad, G. (2012). *Atomic force microscopy: understanding basic modes and advanced applications*. John Wiley & Sons.

Haupt, R. L., & Haupt, S. E. (2004). *Practical genetic algorithms*. John Wiley & Sons.

Hendrikx, M., Meijer, S., Van Der Velden, J., & Iosup, A. (2013). Procedural content generation for games. *ACM Transactions on Multimedia Computing, Communications, and Applications*, *9*(1), 1–22. doi: 10.1145/2422956.2422957

Hirota, K., Tanoue, Y., & Kaneko, T. (1998). Generation of crack patterns with a physical model. *The Visual Computer*, *14*(3), 126-137. doi: 10.1007/s003710050128

Hnaidi, H., Guérin, E., Akkouche, S., Peytavie, A., & Galin, E. (2010). Feature based terrain generation using diffusion equation. In *Computer graphics forum* (Vol. 29, pp. 2179–2186).

Hornby, G. S., & Pollack, J. B. (2001). The advantages of generative grammatical encodings for physical design. In *Proceedings of the 2001 congress on evolutionary computation (ieee cat. no.01th8546)* (Vol. 1, p. 600-607 vol. 1).

Hsu, S.-c., & Wong, T.-t. (1995, January). Simulating dust accumulation. *IEEE Comput. Graph. Appl.*, *15*(1), 18–22. doi: 10.1109/38.364957

Huang, H., Kalogerakis, E., Yumer, E., & Měch, R. (2016). Shape synthesis from sketches via procedural models and convolutional networks. *IEEE Transactions on Visualization and Computer Graphics*, *2*.

Huang, H., Kalogerakis, E., Yumer, E., & Mech, R. (2017). Shape synthesis from sketches via procedural models and convolutional networks. *IEEE TVCG*, *23*(8), 2003–2013.

Iben, H. N., & O'Brien, J. F. (2006). Generating surface crack patterns. In *Proceedings of symposium on computer animation* (pp. 177–185).

Igarashi, T., Matsuoka, S., & Tanaka, H. (1999). Teddy: A sketching interface for 3d freeform design. In *Proc. of siggraph* (pp. 409–416). doi: 10.1145/311535.311602

Igarashi, T., & Mitani, J. (2010, July). Apparent layer operations for the manipulation of deformable objects. *ACM Trans. Graph.*, *29*(4), 110:1–110:7. doi: 10.1145/1778765.1778847

Ihmsen, M., Wahl, A., & Teschner, M. (2012). High-resolution simulation of granular material with sph. In *Vriphys* (pp. 53–60).

Ijiri, T., Owada, S., & Igarashi, T. (2006). The sketch l-system: Global control of tree modeling using free-form strokes. In *International symposium on smart graphics* (pp. 138–146).

Jacob, C. (1994). Genetic l-system programming. In *International conference on parallel problem solving from nature* (pp. 333–343).

Johnson, L., Yannakakis, G. N., & Togelius, J. (2010). Cellular automata for real-time generation of infinite cave levels. In *Proceedings of the 2010 workshop on procedural content generation in games* (p. 10).

Jones, M. D., Farley, M., Butler, J., & Beardall, M. (2010). Directable weathering of concave rock using curvature estimation. *IEEE TVCG*, *16*(1), 81–94. doi: 10.1109/TVCG.2009.39

Justesen, N., Torrado, R. R., Bontrager, P., Khalifa, A., Togelius, J., & Risi, S. (2018). *Illuminating generalization in deep reinforcement learning through procedural level generation.*

Kajiya, J. T. (1986). The rendering equation. In *Acm siggraph computer graphics* (Vol. 20, pp. 143–150).

Kara, L. B., & Shimada, K. (2007). Sketch-based 3d-shape creation for industrial styling design. *IEEE Comp. Graph. and Applications*, *27*(1), 60–71.

Karpenko, O., Hughes, J. F., & Raskar, R. (2004). Epipolar methods for multi-view sketching. In *Proc. on sbim* (pp. 167–173). Aire-la-Ville, Switzerland, Switzerland: Eurographics Association. doi: 10.2312/SBM/SBM04/167-173

Kass, M., Witkin, A., & Terzopoulos, D. (1988). Snakes: Active contour models. *International Journal of Computer Vision*, *1*(4), 321–331.

Kazmi, I. K., You, L., & Zhang, J. J. (2014, August). A survey of sketch based modeling systems. In *Proc. cgiv* (pp. 27–36). doi: 10.1109/CGiV.2014.27

Keeter, M. (2015). *Antimony.* `http://www.mattkeeter.com/projects/antimony/3/`. (Accessed: 2018-11-27)

Kelley, A. D., Malin, M. C., & Nielson, G. M. (1988). Terrain simulation using a model of stream erosion. In *Proceedings of siggraph* (pp. 263–268). ACM. doi: 10.1145/54852.378519

Kelly, G., & McCabe, H. (2007). Citygen: An interactive system for procedural city generation. In *Fifth international conference on game design and technology* (pp. 8–16).

Koza, J. R. (1992). *Genetic programming: On the programming of computers by means of natural selection.* Cambridge, MA, USA: MIT Press.

Koza, J. R. (2010). Human-competitive results produced by genetic programming. *Genetic Programming and Evolvable Machines*, *11*(3-4), 251–284.

Kratt, J., Spicker, M., Guayaquil, A., Fiser, M., Pirk, S., Deussen, O., ... Benes, B. (2015). Woodification: User-controlled cambial growth modeling. *Computer Graphics Forum*, *30*(6).

Krecklau, L., & Kobbelt, L. (2011). Procedural Modeling of Interconnected Structures. *Computer Graphics Forum*.

Krištof, P., Beneš, B., Křivánek, J., & Šťava, O. (2009). Hydraulic erosion using smoothed particle hydrodynamics. In *Computer graphics forum* (Vol. 28, pp. 219–228).

Krs, V., Yumer, E., Carr, N., Benes, B., & Měch, R. (2017, July). Skippy: Single view 3d curve interactive modeling. *ACM Trans. Graph.*, *36*(4), 128:1–128:12. doi: 10.1145/3072959.3073603

Lau, M., Saul, G., Mitani, J., & Igarashi, T. (2010). Modeling-in-context: user design of complementary objects with a single photo. In *Proc. of sbim* (pp. 17–24).

Lechner, T., Watson, B., & Wilensky, U. (2003). Procedural city modeling. In *1st midwestern graphics conference.*

Lenaerts, T., & Dutre, P. (2009). Mixing fluids and granular materials. *Computer Graphics Forum*, *28*(2), 213-218.

Levien, R., & Séquin, C. H. (2009). Interpolating splines: Which is the fairest of them all? *Computer-Aided Design and Applications*, *6*(1), 91–102.

Lienhard, S., Specht, M., Neubert, B., Pauly, M., & Müller, P. (2014). Thumbnail galleries for procedural models. In *Computer graphics forum* (Vol. 33, pp. 361–370).

Lindenmayer, A. (1968a). Mathematical models for cellular interaction in development. *Journal of Theoretical Biology*, *Parts I and II*(18), 280–315.

Lindenmayer, A. (1968b). Mathematical models for cellular interaction in development. *Journal of Theoretical Biology*, *Parts I and II*(18), 280–315.

Lindenmayer, A. (1974). Adding continuous components to L-systems. In G. S. A. Rozenberg (Ed.), *L systems* (pp. 53–68). Berlin: Springer-Verlag.

Lipp, M., Scherzer, D., Wonka, P., & Wimmer, M. (2011). Interactive modeling of city layouts using layers of procedural content. In *Computer graphics forum* (Vol. 30, pp. 345–354).

Lipp, M., Wonka, P., & Wimmer, M. (2008, August). Interactive visual editing of grammars for procedural architecture. *ACM Trans. Graph.*, *27*(3), 102:1–102:10. doi: 10.1145/1360612.1360701

Longay, S., Runions, A., Boudon, F., & Prusinkiewicz, P. (2012). Treesketch: Interactive procedural modeling of trees on a tablet. In *Proceedings of the international symposium on sketch-based interfaces and modeling* (pp. 107–120). Goslar Germany, Germany: Eurographics Association.

Lu, J., Georghiades, A. S., Rushmeier, H., Dorsey, J., & Xu, C. (2005). Synthesis of material drying history: Phenomenon modeling, transferring and rendering. In *Proceedings of eurographics conference on natural phenomena* (pp. 7–16). doi: 10.2312/NPH/NPH05/007-016

Mandelbrot, B. B. (1982). *The Fractal Geometry of Nature*. New York: Freeman.

Matthews, W. J., & Adams, A. (2008). Another reason why adults find it hard to draw accurately. *Perception*, *37*(4), 628–630.

McCann, J., & Pollard, N. (2009, July). Local layering. *ACM Trans. Graph.*, *28*(3), 84:1–84:7. doi: 10.1145/1531326.1531390

McCormack, J. (2004). Aesthetic evolution of l-systems revisited. In G. R. Raidl et al. (Eds.), *Applications of evolutionary computing: Evoworkshops 2004: Evobio, evocomnet, evohot, evoisap, evomusart, and evostoc, coimbra, portugal, april 5-7, 2004. proceedings* (pp. 477–488). Berlin, Heidelberg: Springer Berlin Heidelberg.

McCrae, J., & Singh, K. (2008). Sketching piecewise clothoid curves. In *Proceedings of the fifth eurographics conference on sketch-based interfaces and modeling* (pp. 1–8). Aire-la-Ville, Switzerland, Switzerland: Eurographics Association. doi: 10.2312/SBM/SBM08/001-008

McCrae, J., & Singh, K. (2009). Sketch-based path design. In *Proceedings of graphics interface 2009* (pp. 95–102). Toronto, Ont., Canada, Canada: Canadian Information Processing Society.

McDermott, J. (2012). Graph grammars as a representation for interactive evolutionary 3d design. In *International conference on evolutionary and biologically inspired music and art* (pp. 199–210).

Měch, R., & Miller, G. (2012, Dec 28). The *Deco* framework for interactive procedural modeling. *Journal of Computer Graphics Techniques (JCGT)*, *1*(1), 43–99.

Měch, R., & Prusinkiewicz, P. (1996). Visual models of plants interacting with their environment. In *Siggraph* (Vol. 96, pp. 397–410).

Merillou, N., Merillou, S., Galin, E., & Ghazanfarpour, D. (2012). Simulating how salt decay ages buildings. *IEEE Comput. Graph. Appl.*, *32*(2), 44–54. doi: 10.1109/MCG.2011.107

Merillou, S., Dischler, J., & Ghazanfarpour, D. (2001b). Surface scratches: measuring, modeling and rendering. *The Visual Computer*, *17*(1), 30-45. doi: 10.1007/s003710000093

Merillou, S., Dischler, J.-M., & Ghazanfarpour, D. (2001a). Corrosion: Simulating and rendering. In *Proceedings of graphics interface 2001* (pp. 167–174).

Merillou, S., & Ghazanfarpour, D. (2008). Technical section: A survey of aging and weathering phenomena in computer graphics. *Computers & Graphics*, *32*(2), 159–174. doi: 10.1016/j.cag.2008.01.003

Merrell, P., & Manocha, D. (2011, June). Model synthesis: A general procedural modeling algorithm. *IEEE Transactions on Visualization and Computer Graphics*, *17*(6), 715-728.

Merrell, P., Schkufza, E., Li, Z., Agrawala, M., & Koltun, V. (2011, July). Interactive furniture layout using interior design guidelines. *ACM Trans. Graph.*, *30*(4), 87:1–87:10.

Miller, G. S. P. (1986). The definition and rendering of terrain maps. In *Proceedings of siggraph* (pp. 39–48). ACM. doi: 10.1145/15922.15890

Mitchell, M. (2009a). *Complexity: A guided tour*. Oxford University Press.

Mitchell, M. (2009b). *Complexity: a guided tour*. Oxford [England] ; New York: Oxford University Press.

Mitra, N. J., & Pauly, M. (2009). Shadow art. *ACM Trans. Graph.*, *28*(5), 156:1–156:7.

Monaghan, J. J. (1992). Smoothed particle hydrodynamics. *Annual review of astronomy and astrophysics*, *30*(1), 543–574.

Monaghan, J. J. (2005). Smoothed particle hydrodynamics. *Reports on progress in physics*, *68*(8), 1703.

Monaghan, J. J., & Gingold, R. A. (1977, 12). Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society*, *181*(3), 375-389. doi: 10.1093/mnras/181.3.375

Müller, M., Chentanez, N., & Kim, T.-Y. (2013, July). Real time dynamic fracture with volumetric approximate convex decompositions. *ACM Trans. Graph.*, *32*(4), 115:1–115:10. doi: 10.1145/2461912.2461934

Müller, P., Wonka, P., Haegler, S., Ulmer, A., & Van Gool, L. (2006, July). Procedural modeling of buildings. *ACM Trans. Graph.*, *25*(3), 614–623. doi: 10.1145/1141911.1141931

Musgrave, F. K., Kolb, C. E., & Mace, R. S. (1989, July). The synthesis and rendering of eroded fractal terrains. *SIGGRAPH Comput. Graph.*, *23*(3), 41–50. doi: 10.1145/74334.74337

Nagashima, K. (1998). Computer generation of eroded valley and mountain terrains. *The Visual Computer*, *13*(9-10), 456-464. doi: 10.1007/s003710050117

Narain, R., Golas, A., & Lin, M. C. (2010). Free-flowing granular materials with two-way solid coupling. In *Acm siggraph asia 2010 papers* (pp. 173:1–173:10). New York, NY, USA: ACM. doi: 10.1145/1866158.1866195

Natali, M., Lidal, E. M., Parulek, J., Viola, I., & Patel, D. (2013). Modeling terrains and subsurface geology. In *Eurographics (stars)* (pp. 155–173).

Nealen, A., Igarashi, T., Sorkine, O., & Alexa, M. (2007). Fibermesh: Designing freeform surfaces with 3d curves. *ACM Trans. Graph.*, *26*(3). doi: 10.1145/1276377.1276429

Neidhold, B., Wacker, M., & Deussen, O. (2005). Interactive physically based fluid and erosion simulation. In *Proceedings of eurographics conference on natural phenomena* (pp. 25–33). doi: 10.2312/NPH/NPH05/025-032

Nishida, G., Bousseau, A., & Aliaga, D. G. (2018). Procedural modeling of a building from a single image. In *Computer graphics forum* (Vol. 37, pp. 415–429).

Nishida, G., Garcia-Dorado, I., Aliaga, D. G., Benes, B., & Bousseau, A. (2016a). Interactive sketching of urban procedural models. *ACM Trans. Graph.*, *35*(4), 130:1–130:11. doi: 10.1145/2897824.2925951

Nishida, G., Garcia-Dorado, I., Aliaga, D. G., Benes, B., & Bousseau, A. (2016b, July). Interactive sketching of urban procedural models. *ACM Trans. Graph.*, *35*(4), 130:1–130:11. doi: 10.1145/2897824.2925951

NVIDIA Corporation. (2018). *Geforce rtx 20 series.* `https://www.nvidia.com/en-us/geforce/20-series/`. (Accessed: 2018-11-27)

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

O'neill, M., Ryan, C., Keijzer, M., & Cattolico, M. (2003). Crossover in grammatical evolution. *Genetic programming and evolvable machines*, *4*(1), 67–93.

Palubicki, W., Horel, K., Longay, S., Runions, A., Lane, B., Měch, R., & Prusinkiewicz, P. (2009). Self-organizing tree models for image synthesis. In *Acm siggraph 2009 papers* (pp. 58:1–58:10). New York, NY, USA: ACM. doi: 10.1145/1576246.1531364

Paquette, E., Poulin, P., & Drettakis, G. (2002). The simulation of paint cracking and peeling. In *In graphics interface* (pp. 59–68).

Parish, Y. I. H., & Müller, P. (2001). Procedural modeling of cities. In *Proceedings of the 28th annual conference on computer graphics and interactive techniques* (pp. 301–308). New York, NY, USA: ACM. doi: 10.1145/383259.383292

Partheniades, E. (1965). Erosion and deposition of cohesive soils. *Journal of Hydraulics Division of the American Society of Agricultural Engineers*, 105-139.

Patow, G. (2012, March). User-friendly graph editing for procedural modeling of buildings. *IEEE Comput. Graph. Appl.*, *32*(2), 66–75. doi: 10.1109/MCG.2010.104

Pauly, M., Keiser, R., Adams, B., Dutré, P., Gross, M., & Guibas, L. J. (2005, July). Meshless animation of fracturing solids. *ACM Trans. Graph.*, *24*(3), 957–964. doi: 10.1145/1073204.1073296

Peitgen, H.-O., Jürgens, H., & Saupe, D. (1991). *Chaos and fractals: new frontiers of science.* New York: Springer-Verlag.

Peng, M., Xing, J., & Wei, L.-Y. (2018, July). Autocomplete 3d sculpting. *ACM Trans. Graph.*, *37*(4), 132:1–132:15. doi: 10.1145/3197517.3201297

Perlin, K. (1985). An image synthesizer. *ACM Siggraph Computer Graphics*, *19*(3), 287–296.

Petke, J., Haraldsson, S. O., Harman, M., Langdon, W. B., White, D. R., & Woodward, J. R. (2018). Genetic improvement of software: a comprehensive survey. *IEEE Transactions on Evolutionary Computation*, *22*(3), 415–432.

Peytavie, A., Galin, E., Grosjean, J., & Mérillou, S. (2009). Arches: a framework for modeling complex terrains. In *Computer graphics forum* (Vol. 28, pp. 457–467).

Pirk, S., Niese, T., Hädrich, T., Benes, B., & Deussen, O. (2014, November). Windy trees: Computing stress response for developmental tree models. *ACM Trans. Graph.*, *33*(6), 204:1–204:11. doi: 10.1145/2661229.2661252

Pixar. (2018). *Renderman.* `https://renderman.pixar.com/`. (Accessed: 2018-11-27)

Pixologic. (2018). *Zbrush.* `https://pixologic.com/`. (Accessed: 2018-11-27)

Preparata, F. P., & Shamos, M. I. (1985). *Computational geometry: An introduction*. Berlin, Heidelberg: Springer-Verlag.

Prusinkiewicz, P. (1986). Graphical applications of l-systems. In *Proceedings on graphics interface '86/vision interface '86* (pp. 247–253). Toronto, Ont., Canada, Canada: Canadian Information Processing Society.

Prusinkiewicz, P., & Lindenmayer, A. (1997). The algorithmic beauty of plants. *Plant Science*, *122*(1), 109–110. doi: 10.1016/S0168-9452(96)04526-8

Reeves, W. T. (1983). Particle systemsa technique for modeling a class of fuzzy objects. *ACM Transactions on Graphics (TOG)*, *2*(2), 91–108.

Ritchie, D., Horsfall, P., & Goodman, N. D. (2016). Deep amortized inference for probabilistic programs. *arXiv preprint arXiv:1610.05735*.

Ritchie, D., Mildenhall, B., Goodman, N. D., & Hanrahan, P. (2015, July). Controlling procedural modeling programs with stochastically-ordered sequential monte carlo. *ACM Trans. Graph.*, *34*(4), 105:1–105:11. doi: 10.1145/2766895

Ritchie, D., Thomas, A., Hanrahan, P., & Goodman, N. D. (2016). Neurally-guided procedural models: Amortized inference for procedural graphics programs using neural networks. In *Proceedings of the 30th international conference on neural information processing systems* (pp. 622–630). USA: Curran Associates Inc.

Samet, H. (1990). *The design and analysis of spatial data structures.* Addison-Wesley, Reading, MA.

Schmid, J., Senn, M. S., Gross, M., & Sumner, R. W. (2011, July). Overcoat: An implicit canvas for 3d painting. *ACM Trans. Graph.*, *30*(4), 28:1–28:10. doi: 10.1145/2010324.1964923

Schmidt, R., Khan, A., Kurtenbach, G., & Singh, K. (2009). On expert performance in 3D curve-drawing tasks. In *Proc. of sbim* (pp. 133–140). New York, NY, USA: ACM. doi: 10.1145/1572741.1572765

Schmidt, R., Khan, A., Singh, K., & Kurtenbach, G. (2009, December). Analytic drawing of 3d scaffolds. *ACM Trans. Graph.*, *28*(5), 149:1–149:10. doi: 10.1145/1618452.1618495

Schneider, J., Boldte, T., & Westermann, R. (2006). Real-time editing, synthesis, and rendering of infinite landscapes on GPUs. In *Vision, modeling and visualization 2006.*

Schwarz, M., & Müller, P. (2015, August). Advanced procedural modeling of architecture. *ACM Transactions on Graphics*, *34*(4 (Proceedings of SIGGRAPH 2015)), 107:1-107:12.

Sethi, R., & Ullman, J. D. (1970). The generation of optimal code for arithmetic expressions. *Journal of the ACM (JACM)*, *17*(4), 715–728.

Sethian, J. A. (1996). A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences*, *93*(4), 1591–1595.

Sharma, G., Goyal, R., Liu, D., Kalogerakis, E., & Maji, S. (2018, June). Csgnet: Neural shape parser for constructive solid geometry. In *The ieee conference on computer vision and pattern recognition (cvpr).*

SideFx. (2019). *Houdini.* https://www.sidefx.com/products/houdini/.

Silva, P. B., Müller, P., Bidarra, R., & Coelho, A. (2013). Node-based shape grammar representation and editing. In *Proceedings of the workshop on procedural content generation in games (pcg'13).*

Sims, K. (1994a). Evolving 3d morphology and behavior by competition. *Artificial life*, *1*(4), 353–372.

Sims, K. (1994b). Evolving virtual creatures. In *Proceedings of the 21st annual conference on computer graphics and interactive techniques* (pp. 15–22).

Smelik, R. M., Tutenel, T., Bidarra, R., & Benes, B. (2014, September). A survey on procedural modelling for virtual worlds. *Comput. Graph. Forum*, *33*(6), 31–50. doi: 10.1111/cgf.12276

Smelik, R. M., Tutenel, T., de Kraker, K. J., & Bidarra, R. (2010). Interactive creation of virtual worlds using procedural sketching. In *Eurographics (short papers)* (pp. 29–32).

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

Smith, A. R. (1984). Plants, fractals, and formal languages. *ACM SIGGRAPH Computer Graphics*, *18*(3), 1–10.

Smith, C., & Prusinkiewicz, P. (2004). Simulation modeling of growing tissues. In *Proceedings of the 4th international workshop on functional-structural plant models,* (p. 365-370).

Smith, R. (2014, 11 17). *Nvidia launches tesla k80, gk210 gpu.* `https://www.anandtech.com/show/8729/nvidia-launches-tesla-k80-gk210-gpu`. (Accessed: 2018-11-27)

Stanculescu, L., Chaine, R., & Cani, M.-P. (2011). Freestyle: Sculpting meshes with self-adaptive topology. *Computers & Graphics*, *35*(3), 614–622.

Stanley, K. O., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary computation*, *10*(2), 99–127.

Šťava, O., Beneš, B., Brisbin, M., & Křivánek, J. (2008). Interactive terrain modeling using hydraulic erosion. In *Proceedings of the 2008 acm siggraph/eurographics symposium on computer animation* (pp. 201–210).

Šťava, O., Beneš, B., Měch, R., Aliaga, D. G., & Krištof, P. (2010). Inverse procedural modeling by automatic generation of l-systems. In *Computer graphics forum* (Vol. 29, pp. 665–674).

Šťava, O., Pirk, S., Kratt, J., Chen, B., Měch, R., Deussen, O., & Beneš, B. (2014, September). Inverse procedural modelling of trees. *Comput. Graph. Forum*, *33*(6), 118–131. doi: 10.1111/cgf.12282

Stiny, G., & Gips, J. (1972). Shape grammars and the generative specification of painting and sculpture. In *Information Processing Proceedings of the IFIP Congress* (Vol. 2).

Suetens, P. (2002). *Fundamentals of medical imaging*. Cambridge university press.

Szeliski, R., & Terzopoulos, D. (1989). From splines to fractals. In *Proceedings of siggraph* (pp. 51–60). ACM. doi: 10.1145/74333.74338

Talton, J. O., Gibson, D., Yang, L., Hanrahan, P., & Koltun, V. (2009, December). Exploratory modeling with collaborative design spaces. *ACM Trans. Graph.*, *28*(5), 167:1–167:10. doi: 10.1145/1618452.1618513

Talton, J. O., Lou, Y., Lesser, S., Duke, J., Měch, R., & Koltun, V. (2011). Metropolis procedural modeling. *ACM Transactions on Graphics*, *30*(2), 1–14. doi: 10.1145/1944846.1944851

Talton, J. O., Lou, Y., Lesser, S., Duke, J., Měch, R., & Koltun, V. (2011, April). Metropolis procedural modeling. *ACM Trans. Graph.*, *30*, 11:1–11:14.

Taylor, L. M., & Mitchell, P. (1997). Judgments of apparent shape contaminated by knowledge of reality: Viewing circles obliquely. *British Journal of Psychology*, *88*(4), 653–670.

Togelius, J., Yannakakis, G. N., Stanley, K. O., & Browne, C. (2011, Sept). Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, *3*(3), 172-186.

Turing, A. M. (1950, January 01). Computing machinery and intelligence. *Mind*, *49*, 433–460.

Turquin, E., Wither, J., Boissieux, L., Cani, M.-P., & Hughes, J. F. (2007, January). A sketch-based interface for clothing virtual characters. *IEEE Comput. Graph. Appl.*, *27*(1), 72–81. doi: 10.1109/MCG.2007.1

Unity Technologies. (2018). *Unity.* `https://unity3d.com/`. (Accessed: 2018-11-27)

Vanegas, C. A., Aliaga, D. G., Benes, B., & Waddell, P. A. (2009, December). Interactive design of urban spaces using geometrical and behavioral modeling. *ACM Trans. Graph.*, *28*(5), 111:1–111:10. doi: 10.1145/1618452.1618457

Vanegas, C. A., Garcia-Dorado, I., Aliaga, D. G., Benes, B., & Waddell, P. (2012, November). Inverse design of urban procedural models. *ACM Trans. Graph.*, *31*(6), 168:1–168:11. doi: 10.1145/2366145.2366187

Vanek, J., Benes, B., Herout, A., & Stava, O. (2011). Large-scale physics-based terrain editing using adaptive tiles on the GPU. *IEEE Comp. Graph. and App.*, *31*(6), 35-44. doi: 10.1109/MCG.2011.66

Vanek, J., Galicia, J. A. G., & Benes, B. (2014). Clever support: Efficient support structure generation for digital fabrication. *Computer Graphics Forum*, *33*(5), 117–125.

Velho, L. (2003). Stellar subdivision grammars. In *Sgp '03* (pp. 188–199). Eurographics Association.

Vimont, U., Rohmer, D., Begault, A., & Cani, M.-P. (2017). Deformation grammars: Hierarchical constraint preservation under deformation. In *Computer graphics forum* (Vol. 36, pp. 429–443).

Von Neumann, J., Burks, A. W., et al. (1966). Theory of self-reproducing automata. *IEEE Transactions on Neural Networks*, *5*(1), 3–14.

Wadge, W. W., & Ashcroft, E. A. (1985). *Lucid, the dataflow programming language* (Vol. 303). Academic Press London.

Wang, J., Tong, X., Lin, S., Pan, M., Wang, C., Bao, H., ... Shum, H.-Y. (2006). Appearance manifolds for modeling time-variant appearance of materials. In *Acm siggraph 2006 papers* (pp. 754–761). New York, NY, USA: ACM. doi: 10.1145/1179352.1141951

Weber, B., Müller, P., Wonka, P., & Gross, M. (2009). Interactive geometric simulation of 4d cities. In *Computer graphics forum* (Vol. 28, pp. 481–492).

Whiting, E., Ochsendorf, J., & Durand, F. (2009). Procedural modeling of structurally-sound masonry buildings. In *Acm siggraph asia 2009 papers* (pp. 112:1–112:9). New York, NY, USA: ACM. doi: 10.1145/1661412.1618458

Wither, J., Bertails, F., & Cani, M.-P. (2007, June). Realistic hair from a sketch. In *Shape modeling and applications, 2007. smi '07. ieee international conference on* (p. 33-42). doi: 10.1109/SMI.2007.31

Wojtan, C., Carlson, M., Mucha, P. J., & Turk, G. (2007). Animating corrosion and erosion. In *Proceedings of the third eurographics conference on natural phenomena* (pp. 15–22). Aire-la-Ville, Switzerland, Switzerland: Eurographics Association. doi: 10.2312/NPH/NPH07/015-022

Wolfram, S. (2002). *A new kind of science* (Vol. 5). Wolfram media Champaign, IL.

Wonka, P., Wimmer, M., Sillion, F., & Ribarsky, W. (2003). Instant architecture. *ACM Transactions on Graphics*, *22*(3), 669–677. doi: 10.1145/1201775.882324

Wu, F., Yan, D.-M., Dong, W., Zhang, X., & Wonka, P. (2014, July). Inverse procedural modeling of facade layouts. *ACM Trans. Graph.*, *33*(4), 121:1–121:10. doi: 10.1145/2601097.2601162

Xu, B., Chang, W., Sheffer, A., Bousseau, A., McCrae, J., & Singh, K. (2014, July). True2form: 3d curve networks from 2d sketches via selective regularization. *ACM Trans. Graph.*, *33*(4), 131:1–131:13. doi: 10.1145/2601097.2601128

Xu, K., Zhang, H., Cohen-Or, D., & Chen, B. (2012, July). Fit and diverse: Set evolution for inspiring 3d shape galleries. *ACM Trans. Graph.*, *31*(4), 57:1–57:10.

Yeh, Y.-T., Yang, L., Watson, M., Goodman, N. D., & Hanrahan, P. (2012, July). Synthesizing open worlds with constraints using locally annealed reversible jump mcmc. *ACM Trans. Graph.*, *31*(4), 56:1–56:11.

Yumer, M. E., Asente, P., Měch, R., & Kara, L. B. (2015). Procedural modeling using autoencoder networks. In *Proceedings of the 28th annual acm symposium on user interface software & technology* (pp. 109–118). New York, NY, USA: ACM. doi: 10.1145/2807442.2807448

Zeleznik, R. C., Herndon, K. P., & Hughes, J. F. (1996). Sketch: An interface for sketching 3d scenes. In *Proc. of siggraph* (pp. 163–170). ACM. doi: 10.1145/237170.237238

Zhou, H., Sun, J., Turk, G., & Rehg, J. M. (2007, July). Terrain synthesis from digital elevation models. *IEEE Transactions on Visualization and Computer Graphics*, *13*(4), 834–848. doi: 10.1109/TVCG.2007.1027

Zhu, Y., & Bridson, R. (2005). Animating sand as a fluid. *ACM Trans. Graph.*, *24*(3), 965–972. doi: 10.1145/1073204.1073298

VITA

VITA

Vojtech Krs

Education

2015 - 2019	PhD in Computer Graphics Technology Purdue University, West Lafayette, IN, United States Dissertation title: Optimization and Control in Procedural Modeling
2011 - 2014	Bachelor of Software Engineering Czech Technical University, Prague, Czech Republic Thesis title: Sculpting in Virtual Reality

Professional and Academic Experience

2015 - 2019	Research Assistant Purdue University, West Lafayette, IN, United States
2018	Intern Samsung Research America, Mountain View, CA, United States
2017	Research Intern Adobe Systems Inc., San Jose, CA, United States

2016	Research Intern
	Adobe Systems Inc., San Jose, CA, United States

Awards and Honors

2017, 2018	Sriver Graduate Award
2014	ICT Master Thesis of the Year
2014	Deans Prize for an Exceptional Thesis,

Publications

Marek Fiser, Bedrich Benes, Jorge Garcia Galicia, Michel Abdul-Massih, Daniel G Aliaga, and Vojtěch Krs. Learning geometric graph grammars. In: *Proceedings of the 32nd Spring Conference on Computer Graphics*. SCCG 16. New York, NY, USA: ACM, 2016, pp. 715. isbn: 978-1-4503-4436-4. doi: 10.1145/2948628.2948635.

Rado Gazo, Logan Wells, Vojtěch Krs, and Bedrich Benes. Validation of automated hardwood lumber grading system. In: *Computers and Electronics in Agriculture* 155 (2018), pp. 496 500. issn: 0168-1699. doi: 10.1016/j.compag.2018.06.041.

Vojtěch Krs, Ersin Yumer, Nathan Carr, Bedrich Benes, and Radomír Měch. Skippy: Single View 3D Curve Interactive Modeling. In: *ACM Trans. Graph.* 36.4 (July 2017), 128:1128:12. issn: 0730-0301. doi: 10.1145/3072959.3073603.

Sören Pirk, Vojtěch Krs, Kaimo Hu, Suren Deepak Rajasekaran, Hao Kang, Yusuke Yoshiyasu, Bedrich Benes, and Leonidas J. Guibas. Understanding and Exploiting Object Interaction Landscapes. In: *ACM Trans. Graph.* 36.3 (June 2017), 31:1 31:14. issn: 0730-0301. doi: 10.1145/3083725.

Yoselyn Walsh, Alejandra J. Magana, Jenny P. Quintana, Vojtěch Krs, Genisson C. Coutinho, Edward J. Berger, Ida B. Ngambeki, Eddy Efendy, and Bedrich Benes. Designing Visuohaptic Simulations for Promoting Graphical Representations and Conceptual Understanding of Structural Analysis. In: *Proceedings of the IEEE-ERM 48th Annual Frontiers in Education (FIE) Conference*. San Jose, California, 2018.

Yoselyn Walsh, Alejandra J. Magana, Tugba Yuksel, Vojtěch Krs, Ida B. Ngambeki, Edward J. Berger, and Bedrich Benes. Identifying affordances of physical manipulatives tools for the design of visuo-haptic simulations. In: *ASEE 124rd Annual Conference and Exposition*. Columbus, Ohio, 2017

Logan Wells, Rado Gazo, Riccardo Del Re, Vojtěch Krs, and Bedrich Benes. Defect detection performance of automated hardwood lumber grading system. In: *Computers and Electronics in Agriculture* 155 (2018), pp. 487 495. issn: 0168-1699. doi: 10.1016/j.compag.2018.09.025.

Tugba Yuksel, Yoselyn Walsh, Vojtěch Krs, Bedrich Benes, Ida B. Ngambeki, Edward J. Berger, and Alejandra J. Magana. Exploration of affordances of visuo-haptic simulations to learn the concept of friction. In: *2017 IEEE Frontiers in Education Conference (FIE)*. 2017, pp. 19. doi: 10.1109/FIE.2017.8190471.

Funding

This research was funded in part by Adobe Resarch, National Science Foundation grant #1608762, *Inverse Procedural Material Modeling for Battery Design*, and National Science Foundation grant #1606396, *Haptic-Based Learning Experiences as Cognitive Mediators for Conceptual Understanding and Representational Competence in Engineering Education.*