

Improving Usability in Procedural Modeling

Dissertation

zur

Erlangung des Doktorgrades (Dr. rer. nat.)

der

Mathematisch-Naturwissenschaftlichen Fakultät

der

Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Dipl.-Inf. Björn Ganster

aus

Düsseldorf

Bonn, August 2012

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen
Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn

1. Gutachter: Prof. Dr. Reinhard Klein
2. Gutachter: Prof. Dr. Andreas Weber

Tag der Promotion: 5.2.2013
Erscheinungsjahr: 2013

Abstract

This work presents new approaches and algorithms for procedural modeling geared towards user convenience and improving usability, in order to increase artists' productivity. Procedural models create geometry for 3D models from sets of rules. Existing approaches that allow to model trees, buildings, and terrain are reviewed and possible improvements are discussed. A new visual programming language for procedural modeling is discussed, where the user connects operators to visual programs called model graphs. These operators create geometry with textures, assign or evaluate variables or control the sequence of operations. When the user moves control points using the mouse in 3D space, the model graph is executed to change the geometry interactively. Thus, model graphs combine the creativity of freehand modeling with the power of programmed modeling while displaying the program structure more clearly than text-based approaches. Usability is increased as a result of these advantages.

Also, an interactive editor for botanical trees is demonstrated. In contrast to previous tree modeling systems, we propose linking rules, parameters and geometry to semantic entities. This has the advantage that problems of associating parameters and instances are completely avoided. When an entity is clicked in the viewport, its parameters are displayed immediately, changes are applied to selected entities, and viewport editing operations are reflected in the parameter set. Furthermore, we store the entities in a hierarchical data structure and allow the user to activate recursive traversal via selection options for all editing operations. The user may choose to apply viewport or parameter changes to a single entity or many entities at once, and only the geometry for the affected entities needs to be updated. The proposed user interface simplifies the modeling process and increases productivity.

Interactive editing approaches for 3D models often allow more precise control over a model than a global set of parameters that is used to generate a shape. However, usually scripted procedural modeling generates shapes directly from a fixed set of parameters, and interactive editing mostly uses a fixed set of tools. We propose to use scripts not only to generate models, but also for manipulating the models. A base script would set up the state of an object, and tool scripts would modify that state. The base script and the tool scripts generate geometry when necessary. Together, such a collection of scripts forms a template, and templates can be created for various types of objects. We examine how templates simplify the procedural modeling workflow by allowing for editing operations that are context-sensitive, flexible and powerful at the same time.

Many algorithms have been published that produce geometry for fictional landscapes. There are algorithms which produce terrain with minimal setup time, allowing to adapt the *level of detail* as the user zooms into the landscape. However, these approaches lack plausible river networks, and algorithms that create eroded terrain with river networks require a user to supervise creation and minutes or hours of computation. In contrast to that, this work demonstrates an algorithm that creates terrain with plausible river networks and adaptive *level of detail* with no more than a few seconds of preprocessing. While the system can be configured using parameters, this text focuses on the algorithm that produces the rivers. However, integrating more tools for user-controlled editing of terrain would be possible.

Zusammenfassung

Ziel der vorliegenden Arbeit ist es, prozedurale Modellierung durch neue neue Ansätze und Algorithmen einfacher, bequemer und anwendungsfreundlicher zu machen, und damit die Produktivität der Künstler zu erhöhen. Diese Anforderungen werden häufig unter dem Stichwort Usability zusammengefasst. Prozedurale Modelle spezifizieren 3D-Modelle über Regeln. Existierende Ansätze für Bäume, Gebäude und Terrain werden untersucht und es werden mögliche Verbesserungen diskutiert. Eine neue visuelle Programmiersprache für prozedurale Modelle wird vorgestellt, bei der Operatoren zu Modellgraphen verschaltet werden. Die Operatoren erzeugen texturierte Geometrie, weisen Variablen zu und werten sie aus, oder sie steuern den Ablauf der Operationen. Wenn der Benutzer Kontrollpunkte im Viewport mit der Maus verschiebt, wird der Modellgraph ausgeführt, um interaktiv neue Geometrie für das Modell zu erzeugen. Modellgraphen kombinieren die kreativen Möglichkeiten des freihändigen Editierens mit der Mächtigkeit der prozeduralen Modellierung. Darüber hinaus sind Modellgraphen eine visuelle Programmiersprache und stellen die Struktur der Algorithmen deutlicher dar als textbasierte Programmiersprachen. Als Resultat dieser Verbesserungen erhöht sich die Usability.

Ein interaktiver Editor für botanische Bäume wird ebenfalls vorgestellt. Im Gegensatz zu früheren Ansätzen schlagen wir vor, Regeln, Parameter und Geometrie zu semantischen Entitäten zu verschmelzen. Auf diese Weise werden Zuordnungsprobleme zwischen Parametern und deren Instanzen komplett vermieden. Wenn im Viewport eine Instanz angeklickt wird, werden sofort ihre Parameter angezeigt, alle Änderungen wirken sich direkt auf die betroffenen Instanzen aus, und Änderungen im Viewport werden sofort in den Parametern reflektiert. Darüber hinaus werden die Entitäten in einer hierarchischen Datenstruktur gespeichert und alle Änderungen können rekursiv auf der Hierarchie ausgeführt werden. Dem Benutzer werden Selektionsoptionen zur Verfügung gestellt, über die er Änderungen an den Parametern oder Änderungen im Viewport an einzelnen oder vielen Instanzen gleichzeitig vornehmen kann. Anschließend muss das System nur die Geometrie der betroffenen Instanzen aktualisieren. Auch hier ist das Ziel, das User Interface möglichst an den Bedürfnissen des Benutzers auszurichten, um Vereinfachungen und eine Erhöhung der Produktivität zu erreichen.

Interaktive Editieransätze für 3D-Modelle erlauben häufig eine präzisere Kontrolle über ein Modell als ein globaler Parametersatz, der für die Erzeugung des Modells genutzt wird. Trotzdem erzeugen prozedurale Modellerskripte ihre Modelle meist direkt aus einem festen Parametersatz, während interaktive Tools meist mit hartkodierten Operationen arbeiten. Wir schlagen vor, Skripte nicht nur zur Erzeugung der Modelle zu verwenden, sondern auch um die erzeugten Modelle zu editieren. Ein Basisskript soll die Statusinformationen eines Objekts anlegen, während weitere Skripte diesen Status verändern und passende Geometrie erzeugen. Diese Skripte bilden dann ein Template zum Erzeugen einer Klasse von Objekten. Verschiedene Objekttypen können jeweils ihr eigenes Template haben. Wir zeigen, wie Templates den Workflow mit prozeduralen Modellen vereinfachen können, indem Operationen geschaffen werden, die gleichzeitig kontext-sensitiv, mächtig und flexibel sind.

Es existiert eine Reihe von Verfahren, um Geometrie für synthetische Landschaften zu erzeugen. Ein Teil der Algorithmen erzeugt Geometrie mit minimaler Vorberechnung und erlaubt es, den Detailgrad der Landschaft interaktiv an die Perspektive anzupassen. Leider fehlen den so erzeugten Landschaften plausible Flussnetze. Algorithmen, die erodiertes Terrain mit Flussnetzen erzeugen, müssen aufwendig vom Benutzer überwacht werden und brauchen Minuten oder Stunden Rechenzeit. Im Gegensatz dazu stellen wir einen Algorithmus vor, der plausible Flussnetze erzeugt, während sich der Betrachter interaktiv durch die Szene bewegt. Das System kann über Parameter gesteuert werden, aber der Fokus liegt auf dem Algorithmus zur Erzeugung der Flüsse. Dennoch wäre es möglich, Tools zum benutzergesteuerten Editieren von Terrain zu integrieren.

Acknowledgements

Although a work such as this bears the name of a single author, it draws from discussions, advice and ideas developed with the aid of many persons. I would like to thank my advisor, Prof. Dr. Reinhard Klein, for many hours of creative and productive discussion. You gave me advice, ideas, helped me transform ideas into knowledge and always encouraged me to go on.

I would like to thank Mr Klein's present and past members of the computer graphics group at the University of Bonn for their discussions and advice. Stefan Hartmann, Michael Weinmann, Elena Trunz, and Heinz Christian Steinhausen helped proofreading the text. I would like to thank Evgenij Derzopf and Michael Guthe for their contributions to our paper on modeling terrain. The idea of model graphs was developed following a discussion with PD Dr. Frank Hanisch.

My parents deserve thanks for developing and cultivating my scientific interest from my earliest days. My brother Erik helped prepare leaf textures used for some of the trees. My wife's parents frequently helped keep an eye on our children, to give me more time to write. I would like to thank my wife for her general support of my dissertation. This work is dedicated to our sons Florian and Lasse.

Contents

1	Introduction	1
2	Basics	4
2.1	O Notation	5
2.2	Basic Data Structures and Algorithms	5
2.2.1	Meshes	5
2.2.2	Textures	6
2.2.3	Scene Graphs	6
2.3	Botany	7
3	Related Work	9
3.1	Visual Dataflow Pipelines	9
3.2	Procedural Modeling	11
3.2.1	Properties and Use Cases of Procedural Models	11
3.2.2	Extrusion	12
3.2.3	Generative Modeling	13
3.2.4	Programming and Scripting for Procedural Modeling	14
3.2.5	Fractals	14
3.2.6	Particle Systems	15
3.2.7	L-Systems	15
3.2.8	Procedural Modeling of Trees	17
3.2.9	Xfrog	19
3.2.10	Procedural Modeling of Cities	20
3.2.11	Procedural Modeling of Terrain	24
3.2.12	Erosion Simulation	27
3.2.13	Procedural Textures	28
3.3	Terrain Rendering and Level of Detail	29
3.4	Further Modeling and Editing Approaches	30
3.4.1	Terrain Editing	30
3.4.2	Sketch-Based Modeling	30
3.4.3	Image-Based Modeling	31
3.4.4	Modeling by Example	31
3.5	3D Modeling in Practice	33
3.5.1	3D Modeling Suites	33
3.5.2	Developing Plugins	34
3.5.3	Procedural Modeling in Computer Games	34

4	A New Visual Paradigm for Procedural Modeling	37
4.1	Introducing Model Graphs	40
4.1.1	Language Definition	40
4.1.2	Basic Node Types	44
4.1.3	Scene Graph Management	46
4.2	Introductory Example	51
4.3	Additional Language Features	55
4.3.1	Control Operators	55
4.3.2	Variable Types	56
4.3.3	Records	56
4.3.4	Updating variables	57
4.3.5	Calling Other Model Graphs and Recursion	58
4.3.6	Parallelization	61
4.3.7	Texture Generators	62
4.3.8	Geometry Creation	64
4.3.9	Animation	67
4.4	User Interface	68
4.4.1	Object Picking	68
4.4.2	Control Points	69
4.5	Implementation	70
4.5.1	General Optimizations	70
4.5.2	Rendering Optimizations	71
4.5.3	File Formats	72
4.6	Examples	73
4.6.1	Creating Plants	73
4.6.2	Creating Buildings	74
4.6.3	Creating Terrain	76
4.6.4	Smoothing Terrain	78
4.6.5	Midpoint Displacement	79
4.6.6	Erosion Simulation	80
4.6.7	Fractals	83
4.6.8	Animation	85
4.6.9	Model Graphs that Emulate Turing Machines	86
4.6.10	Putting It All Together	87
4.7	Discussion	88
4.8	Conclusion	90

5	Increasing Usability of Procedural Modeling with an Editing Approach	91
5.1	Using 1-2-tree	93
5.1.1	Selection	93
5.1.2	Viewport Tools	94
5.1.3	Parameters	94
5.2	Implementation	96
5.2.1	Selection in the Viewport	96
5.2.2	Parameter Updates	96
5.2.3	Viewport Tools	97
5.2.4	Geometry Creation and Rendering	98
5.2.5	Persistence	98
5.2.6	File Formats	99
5.3	Results	100
5.4	Discussion	103
5.5	Conclusion	105
6	Editing 3D models using templates	106
6.1	Design Considerations	106
6.2	Implementation	108
6.2.1	User Interface	108
6.2.2	Data Exchange	109
6.3	Results	111
6.3.1	Architecture	111
6.3.2	Terrain	111
6.3.3	Trees and plants	111
6.4	Discussion	113
7	Procedural Terrain with River Networks	114
7.1	Analysis	115
7.2	Overview	117
7.3	Planet Creation	118
7.3.1	Create Base Shape	119
7.3.2	Define Continents	120
7.3.3	Produce Initial River Networks	120
7.3.4	Assign Altitudes	122
7.3.5	Compute Water Levels	123
7.3.6	Assign Vertex Colors	123

7.4	Adaptive Level of Detail	125
7.4.1	Refinement	125
7.4.2	Reversing and Reapplying Refinements	127
7.4.3	Reproducibility	128
7.4.4	Create Geometry for Lakes and Rivers	129
7.5	CPU Implementation	131
7.6	GPU Implementation	132
7.7	Discussion	133
8	Systems Comparison	135
9	Conclusion	139
10	Future Work	141
10.1	Terrain Modeling	141
10.2	Crust Movements	141
10.3	User-Controlled Modeling of Planets	141
10.4	Transportation Networks	142
10.5	Placing Lakes and Endorheic Basins	144
10.6	Compiled model graphs	144
10.7	Parallel Execution of Model Graphs in Case of Several Root Nodes . .	145
10.8	Templates and Scene Graph Hierarchies	145
10.9	Further Improvements to Templates	145
A	Control Operators in plab	147
B	Operators in plab	149
C	Texture Generators	155
D	Components	157
E	Functions	161
F	Hints for Programming in plab	162
G	Parameters for Terrain Modeling	163
H	Modeling Software	164

1 Introduction

There are many ways of modeling 3D objects on a computer. Usually, artists strive to reproduce real or imagined objects as exactly as possible, in a reasonable amount of time. In other cases, only a vague description of an object can be given, and the computer is supposed to create a model which fits the description. Procedural models are created from sets of rules and algorithms. While all modeling methods on a computer rely on procedures that manipulate data, in procedural modeling there is a clear emphasis on the procedures that control the modeling process.

Procedural modeling can be used to create scenes automatically. This is useful for computer games and simulations, which create completely new environments whenever the game is played, or for creating scenery for films. Procedural modeling can also be used to add detail to user-modeled scenes, such as adding vegetation or buildings to a landscape. In some cases, editors for modeling objects mix in procedural elements. Interactive modeling systems usually allow the user only to model using primitives, whereas procedural modeling can take place on a higher level of abstraction: Instead of dealing with specific primitives, the user specifies rules for placing primitives.

Before we demonstrate how to improve procedural modeling, this work reviews previous work in procedural modeling in Section 3. Despite all the previous work, procedural graphics have often been criticized as being difficult to use. Many approaches have focused on adding new features to the procedural modeling tool set, but there has been little research on making it easier to use. This work intends to fill some of the existing gaps in making procedural graphics more accessible and to add some new algorithms.

Procedural graphics offers solutions for many problems that arise when dealing with large scenes. But the ideas are often accessible only as prototypes, integrated solutions are missing, and the tools require a lot of effort to learn. As such, for many users working in computer graphics, these solutions are unusable. Users need to think too much about how something could be done, rather than focusing on the creative aspects of their work. Usability in modeling 3D objects implies several things. First off, the interfaces should be easy to use and understand. At the same time, procedural graphics needs powerful tools to fulfill the promise of creating objects from rules. Traditionally, modeling 3D objects procedurally requires knowledge in geometry and a basic understanding of computer scientific topics such as loops and variables. For tools that specialize on modeling certain types of objects, the full complexity of procedural modeling is not required, and can be safely hidden from the user. While this may mean a loss of generality and power of expression, the goal is to improve ease of use and productivity. However, for users that know how to use it, the full complexity of procedural modeling should still be accessible.

This work approaches the design of user interfaces for procedural modeling from several perspectives. It starts by examining how an interface can be created that makes procedural modeling as easy as possible without losing the expressional power offered by this type of system. We propose a visual programming language that is useful for procedural modeling in general, and examine how some types of models can be constructed. Our visual procedural modeling language produces textures and geometry

using graphs, which are called model graphs. The graphs contain graphical nodes that are used to define operations. A module called SceneFactory executes model graphs to obtain a scene graph to display. In contrast to other visual programming languages, data is stored in variables, rather than being transported by pipelines. Instead, edges between the nodes define the order of execution. Some nodes may alter the order of execution, while others define textures, geometry, variables or transformations. Model graphs may define control points that can be used to edit the models in the viewport. Section 4 discusses this visual programming language in detail and expands on the author's previous paper [50].

Next, the text analyzes how a balanced approach between expressional power and ease of use can be achieved. In many approaches to procedural modeling, the user starts by defining the parameters of the model, then an algorithm creates geometry, which is finally displayed to the user. If the user needs to adapt the parameters, the process repeats. This has a number of drawbacks. In each iteration of editing parameters, recreating geometry, and analyzing the effects of the editing operations, the geometry is recreated from scratch. This may lead to flickering. Even slight changes to the parameters may lead to vastly different results, because random values are applied in different local order, even when the changes address only a small part of the model. Recreating the entire model may also take longer than necessary if only part of the model is supposed to be changed. In many procedural algorithms, parameters are shared between many instances, preventing changes to individual instances. These challenges can be addressed by storing entities in a scene graph, where each instance has its own parameter set. We demonstrate these ideas with a system that is geared towards making trees as easy to model as possible. The user may choose to apply edits to a single branch, branches on the same level of recursion, all branches on the same level of recursion and below that, or all branches. Branches may further be tagged, and operations can be applied to branches that have the specified tags. Changes are applied only to selected entities, which reduces flickering, enhances persistence of edits, and decreases the time spent on the updates. Section 5 illustrates this with a convenient and powerful user interface for editing trees. The chapter is based on a paper previously published by the author [51].

The parameter sets of procedural models are often kept small, because large parameter sets are difficult and unintuitive to use. Interactive editing is often preferred, where users can change aspects of a model using intuitive tools rather than manipulating a parameter set that controls only global aspects of a model. In Section 6 we analyze how modeling with model graphs can be further improved in this regard. The goal is a user interface that allows users to harness the full expressional power of procedural graphics while providing an interface that hides its complexity. This can be achieved by allowing some users to create scripts that govern the creation and editing of objects. While users that create the scripts still have to deal with the full complexity of procedural modeling, this complexity can be hidden from users who simply want to use the scripts. A single model graph can be used to set up the state of an object, and further model graphs would be used to manipulate this state information and generate geometry from the state information. Operations that alter state include replacing a window tile, or generating additional elements, such as adding a layer of branches to a tree. Together, these model graphs can be seen as a template that is used to create and edit instances of a class of objects. A Cinema 4D plugin was created to demonstrate how templates can

simplify editing objects. Templates are demonstrated for editing buildings and terrain, but most existing model graphs can be used as base model graphs, even when they lack tool model graphs, and more templates can be created when needed.

Maximizing user convenience in procedural modeling means that the user does not have to model anything himself. Instead, the user mostly gives up control over the outcome, and the system produces the model automatically. A set of parameters can be displayed, and the user either applies the defaults, or may change the parameters if desired. As an example for such an algorithm, we demonstrate a system that lets users explore procedural planets. Existing algorithms generate landscapes or entire planets immediately with view-dependent resolution and without preprocessing. Unfortunately, landscapes generated by such algorithms lack river networks and therefore appear unnatural. Algorithms that integrate plausible river networks are computationally expensive and cannot be used to generate a locally adapting high resolution landscape during a fly-through. Section 7 discusses the author's contributions to a novel algorithm to overcome this problem [39]. The algorithm creates complete planets and landscapes with plausible river networks within seconds. It starts with a coarse base geometry of a planet without further preprocessing and user intervention. It uses graphics hardware to generate adaptively refined landscape geometry during fly-throughs.

This work concludes by comparing the presented approaches in terms of usability and power of expression in Section 8. While Turing-complete languages are most powerful and most difficult to master, less powerful systems are often easier to use and are sometimes powerful enough.

As a convention, *italicized* words can be looked up in the index.

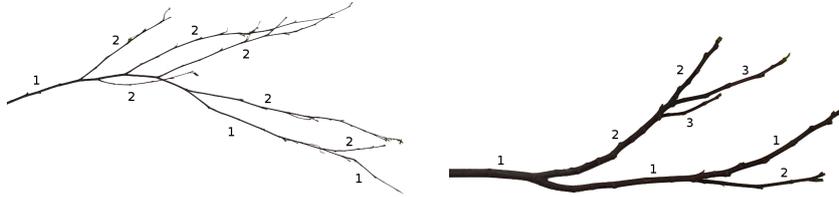


Figure 1: Branches from a birch tree (left) and a fig tree (right) in Gravelius numbering scheme

2 Basics

Many systems can be described as graphs, including branches on a tree, streets, river systems, or the human circulatory system. A *graph* is defined as an ordered pair $G = (V, E)$, where V is a set of *vertices*, and E is a set of *edges* connecting the vertices. Vertices are often also called *nodes*. Edges may be *directed* or *undirected*. If an edge $e \in E$ from $a \in V$ to $b \in V$ is directed, it cannot be followed in reverse direction, but there may be an opposite edge from b to a . Undirected edges can be followed in both directions. Directed edges can be defined as a subset $E \subseteq \{(A, B) | A, B \in V\} = V \times V$. Undirected edges can be defined as a subset $E \subseteq \{\{A, B\} | A, B \in V\}$. Edge (u, v) is an *outgoing* edge for u and an *incoming* edge for v . The edges can also be defined in a manner that allows several distinct edges between pairs of vertices, which are called *multi edges*. A graph that consists of only directed edges is a directed graph, whereas a graph that contains only undirected edges is an undirected graph. A *path* is an ordered set of vertices v_1, \dots, v_n , where v_i and v_{i+1} are connected by edges. For graphs with directed edges, a path must not follow edges in reversed direction. A path that starts and ends in the same vertex $v_1 = v_n$ is called a *cycle*. If a graph does not contain a cycle it is called *acyclic*. A graph is said to be *connected* if for each pair of vertices, there is a path that connects them. In graph theory, a rooted *tree* is a connected acyclic graph with a designated *root* vertex.

An edge e of a graph G is said to be *incident* to a vertex v if v is an end point of e . In this case, we also say that v is incident to e . Two edges e and f that are incident to a shared vertex v are said to be *adjacent* [29, 42]. Two vertices are said to be *adjacent* if there is an edge between them.

Numbering schemes have been proposed to number branches in trees in graph theory. The *Gravelius numbering scheme* was proposed for rivers and streams [56]. The scheme assigns 1 to the root of the tree. At every branching point, one edge is assigned the number of the parent branch, while the other branch is assigned a value one order higher. The *Weibull numbering scheme* was proposed for branching structures of lungs [160]. At each branching point, all child branches are assigned a value one order higher than their parent branch. Examples are given in Figure 1 and Figure 2.

The *Four Color Theorem* states that four colors suffice to color regions in a graph so that all regions that share an edge have different color. All currently accepted proofs rely on computers [4].

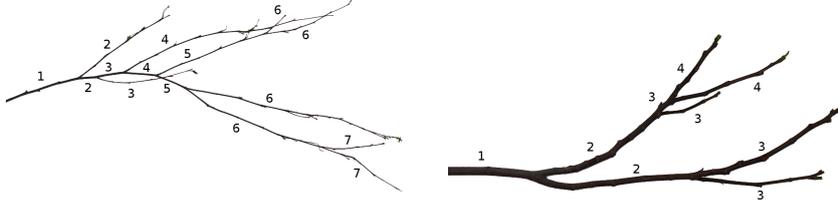


Figure 2: Branches from a birch tree (left) and a fig tree (right) in Weibull numbering scheme

2.1 O Notation

An important criterium for selecting algorithms is asymptotic runtime. For sufficiently large problems, the algorithm with best asymptotic runtime is usually the fastest. However, for small problems, simpler algorithms may be faster. The following definitions are used to characterize the asymptotic runtime of algorithms:

- $O(f)$ is the set of functions that do not grow faster than $f(n)$ asymptotically:

$$O(f) = \{g : \mathbb{N} \rightarrow \mathbb{R} \mid \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{R} \forall n > n_0 : g(n) \leq cf(n)\}. \quad (1)$$

- $\Omega(f)$ is the set of functions that grow at least as fast as $f(n)$ asymptotically:

$$\Omega(f) = \{g : \mathbb{N} \rightarrow \mathbb{R} \mid \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{R} \forall n > n_0 : g(n) \geq cf(n)\}. \quad (2)$$

- $\Theta(f)$ is the set of exact boundaries:

$$\Theta(f) = O(f) \cap \Omega(f). \quad (3)$$

2.2 Basic Data Structures and Algorithms

In this work, *adjacency lists* are used to store graphs. In an adjacency list, vertex objects store any information associated with the vertices. Edges are not stored explicitly, instead every vertex stores a list of adjacent vertices and optionally edge properties. For directed graphs, the lists for incoming and outgoing edges are separate. Alternatively, vertices would store pointers to edge objects, which would also store the edge properties. This data structure is called an *incidence list*. Certain types of operations may be executed faster if the graphs are stored in matrices. Here, (a_{ij}) stores the cost of travelling from v_i to v_j .

2.2.1 Meshes

In computer graphics, object surfaces are stored as graphs. These graphs are called *meshes*. A vertex in a mesh represents a salient point on the object's surface. *Polygons* are represented by cycles in the mesh. The number of vertices, edges and polygons is often kept small to reduce hardware requirements. A mesh is closed if each edge belongs to two faces. The *indexed faceset* is a very important data structure for meshes,

because it allows for very efficient rendering in OpenGL. Here, vertices are stored as an array of coordinates, and polygons are stored as vectors of indices into the vertex buffer. The user decides how much information is stored for the vertices. Traditionally, x and y coordinates are required, and optionally z and w coordinates can be stated. Furthermore, a vertex may have up to four texture coordinates, a surface normal, and color information. The surface normal is used for lighting the surface surrounding the vertex. Vertex and pixel shaders further increase flexibility of describing surface and material properties. The full definition of how OpenGL supports indexed facesets, including data formats and API functions, can be found in the OpenGL reference documentation [137]. More precise definitions are given from a practical perspective in Section 4.1.1 and Section 4.3.8.

2.2.2 Textures

A proven strategy for optimizing algorithms is to favor calling a function that contains a loop instead of calling a function from a loop, because the performance penalty of calls multiplies in a loop. For this reason, graphics APIs moved from setting state with many tiny function calls to fewer calls that pass masses of data. A prominent example is the shift from the `glBegin/glVertex/glEnd` paradigm to functions such as `glDrawElements`. *State sorting* is used to further minimize the number of calls. This also affects texture management, because `glDrawElements` does not support individual textures for every polygon. Instead, textures are compiled into *texture atlases* [24], and the u, v coordinates of the vertices must be adapted accordingly. The process of assigning u, v coordinates is called *UV unwrapping* or *parameterization* [38], and it is a non-trivial process for several reasons. Polygons that share edges in 3D space may be placed right next to each other in the atlas, but often it is not possible to pack all textures tightly into an atlas without distorting the textures. When mipmapping and similar techniques are used, additional measures may be necessary. Polygons cannot be packed tightly when their edges do not fit, because this may cause seam artifacts at the polygon edges. This can be prevented by leaving space between different textures unused, and filling the unused texels with border texels from the surrounding texture patches.

High-quality renderings for films are often created using *raytracing*. For raytracing, the penalty of switching between textures is far less severe than in OpenGL, and often higher quality graphics are the goal when using raytracing, so it is practical to use one or more textures for a single polygon, instead of compiling an atlas [22]. With raytracing, UV unwrapping becomes unnecessary, as every polygon can use its own texture. Hand-painted textures are preferred, and procedural textures are stored rather than evaluated on the fly, to prevent filtering problems.

2.2.3 Scene Graphs

While indexed facesets may be well suited for rendering, users often prefer data structures for modeling that reflect the hierarchical nature of objects, where each object may consist of several parts. For example, car tires follow the general movement of a car, but require additional transformations when the steering wheel is turned. Another example is a robot, whose limbs all require additional transformations compared to the

torso or hip. Also, planets follow the movement of their star, but in addition, they orbit their star and rotate. In these cases, users want to model such relationships with a hierarchy. Hierarchical data structures for scenes store objects and rendering state, and are called *scene graphs*. Rendering state for the objects includes camera settings, light sources, material information, and textures, among other things.

By allowing scene graph nodes to have several parents, objects may be placed in the scene at several locations, but the geometry for the object needs to be stored only once. This is called *instancing*, and it is the reason why a scene graph is not a tree. While in object-oriented programming, instantiating a class refers to the process of creating data that conforms with its class description, in computer graphics, instancing refers to the process of representing one object with another by storing one object as a reference to another object. Hart and Defanti describe fractals as object hierarchies that may contain cycles [63]. For rendering scene graphs with cycles, creating geometry should be aborted when the details become smaller than a pixel, but most implementations restrict scene graphs to directed acyclic graphs. Scene graphs were introduced by Strauss and Carey [146].

Organic characters, including humans and animals, are often more difficult to animate than objects that consist of discrete parts, where each part has its own transformation. Organic objects are commonly animated by assigning each vertex to one or more vertices of a skeleton. The vertices then follow the animation of the skeleton.

2.3 Botany

As some of the algorithms in this work deal with modeling trees and plants, basic principles of botany are covered in this section. A tree strives to gain access to sun light, nutrients and water with as little material as possible, and tries to minimize the routes of transport. At the same time, trees have to resist forces of wind, gravity, and rain. In part, tree growth is controlled by stress hormones that are emitted as a result of these forces. A tree displays *gravitropism* if its branches are pulled down by gravity. Negative gravitropism would cause branches to bend upward despite gravity. *Phototropism* lets plants grow towards light. This is also called *heliotropism* if referring specifically to the sun.

Each branch begins in a *bud*. A bud starts in a node, and space between two nodes on a branch is called an *internode*. There are different types of layouts for buds. In *distichy*, buds grow from opposite directions. In *dispersion*, buds are placed at an angle of $135^\circ - 144^\circ$, approximating the *Golden angle*. In *decussation*, n buds grow from a single node, at an angle of $360^\circ/n$, alternating at $180^\circ/n$.

In *Monopodial branching*, the side branches are shorter and weaker than the main branch, whereas in *sympodial branching*, the side branches are stronger and longer than the main branch. In a *dichasium*, pairs of buds are growing from one node in opposite direction. In a *monochasium*, one side of the branches dominates. The dominating branching side may appear to continue the main branch, and may be difficult to tell from monopodial branching.

A more detailed look at the principles of plant forms from a perspective of computer graphics is given in Deussen's book [41].

3 Related Work

In this chapter, relevant previous work is reviewed to provide the reader with background for the later chapters.

3.1 Visual Dataflow Pipelines

Visual dataflow pipelines (*VDFPs*) are used for many tasks in computer graphics, e.g. they control the dataflow and object dependencies in Maya [55] and they can be used to write shaders [54]. In many editors, e.g. text editors, editing operations can be undone and redone, but editing after undoing operations deletes the undone operations, and they may need to be reentered. In Maya, editing operations are stored in a VDFFP called the Dependency Graph. This allows to change aspects of certain editing operations without losing other changes made later.

Visual dataflow pipelines are founded on dataflow languages. Ackerman lists the following properties for dataflow languages [1]:

1. Freedom from side effects: Functions calculate values without modifying state,
2. Locality of effects: Unnecessary data dependencies should be avoided,
3. Equivalence of instruction scheduling constraints with data dependencies,
4. Single assignment convention: each variable may be assigned only once,
5. As a consequence of properties (1) and (4), loops require an unusual notation,
6. Procedures have no history sensitivity.

A statement in a dataflow language can be executed when all of its input variables have been assigned. Several assignments may be executed in parallel. Stevens et al [144] proposed to visualize the dependencies between the assignments, and to use this notation for programming. The visual representation of a dataflow program is equivalent to its textual representation. A *visual dataflow pipeline* consists of a number of *nodes* with *input* and *output ports* and *pipelines* that connect the ports, similar to a graph with edges and vertices. It can be understood as a network that guides the execution of operations. Each node wraps an operation that may have several input and output *attributes*, and each attribute is associated with an input or output port. Directed edges connect the ports to transport data between the nodes. The edges are referred to as pipelines, and the graph is called a visual dataflow pipeline. When a node receives a new set of inputs, it computes a function from the input parameters to produce output data and feeds the output through the outgoing pipelines to other nodes. In a VDFFP, the *single assignment convention* is embodied in the fact that every input may be connected to only one output, but one output may be connected to several inputs. The visual notation asserts equivalence of instruction scheduling constraints with data dependencies.

Visual languages display structural dependencies and the parameters for function calls very clearly and they are recognized as being easier to learn than textual representations of programs. While textual programs are structured by keywords such as `if`,

for, while, or the semicolon, VDFPs are structured by nodes and their connections. VDFPs do not require the user to type keywords, and are thus less prone to syntactic errors due to missing or misplaced keywords, but they may require users to connect nodes for each operator in a formula. It is often faster to enter a formula on the keyboard.

VDFPs have several issues that are not handled satisfactory. If an array is the output of a node and it is the input of two nodes, each node has to copy the entire array or it has to maintain a list of changes to the array. *Loops* have to be expressed as cycles, as recursions, or they require specialized constructs. More importantly, it is not clear how to achieve optimal performance. If each node would be run on a processor core that communicates via messages with the other processors, processors would spend most of their time waiting for messages. Therefore, more intelligent scheduling is required. Johnston et al. demonstrate improved algorithms for these problems, but they report the problems as still open [75].

The first visual programming language used for procedural modeling was ConMan [60], which uses the principle of a dataflow pipeline. Executing a VDFP to produce a 3D model has been patented [120, 121].

3.2 Procedural Modeling

The author has contributed to the Wikipedia.org article on procedural modeling. Since November 6th, 2009, the definition of procedural modeling on Wikipedia.org reads as follows (last checked on April 24th, 2012):

”*Procedural modeling* is an umbrella term for a number of techniques in computer graphics to create 3D models and textures from sets of rules. L-Systems, fractals, and generative modeling are procedural modeling techniques since they apply algorithms for producing scenes. The set of rules that control scene construction may either be embedded into the algorithm, configurable by parameters, or the set of rules is separate from the evaluation engine. The output is called procedural content, which can be used in computer games, films, be uploaded to the internet, or the user may edit the content further. Procedural models often exhibit *database amplification*, meaning that large scenes can be generated from a much smaller amount of rules [140]. If the employed algorithm produces the same output every time, the output does not need to be stored. Often, it is sufficient to start the algorithm with the same random seed to achieve this.”

”Although all modeling techniques on a computer require algorithms to manage and store data at some point, procedural modeling focuses on creating a model from a rule set, rather than editing primitives via user input. Procedural modeling is often applied when it would be too cumbersome to create a 3D model using generic 3D modelers, or when more specialized tools are required. This is often the case for plants, architecture or landscapes.”

3.2.1 Properties and Use Cases of Procedural Models

An important property of a procedural algorithm is whether the algorithm is *reproducible*, that is, whether it generates the same output from the same input every time the algorithm is run. Often, procedural models use pseudorandom numbers to produce variations. *Pseudorandom number generators* are deterministic algorithms that are initialized with a number called the *random seed* and produce seemingly random numbers. The random seed is often set using the current time stamp. Normally, the sequence of pseudorandom numbers of a program will differ from the sequence of the same program started later, because normally a different seed produces a different sequence of pseudorandom numbers. However, by choosing a specific random seed, the entire algorithm that produces a model becomes reproducible. This is often a desired property because it allows to reproduce geometry at any time from the random seed. Only a model’s parameters and random seed need to be stored, not the geometry produced by the algorithm.

While some procedural modeling implementations create or recreate the entire scene, Hart and Defanti create geometry at the required resolution when needed [63]. This was later extended to run in parallel [32]. These papers refer to lazy evaluation or greedy evaluation: *Lazy evaluation* means that data is produced only when needed. Lazy evaluation usually requires content to be reproducible, since every time the scene is rendered, the same output is expected. *Greedy evaluation* means that the procedural model is evaluated fully once, and then the procedural content is used in stored form.

Procedural models can be used to generate objects or terrain automatically. Randomized procedural models can generate a different instance every time the algorithm is run. This is useful for training simulations and games, when a new challenge is needed every time the simulation is run. *Database amplification* implies that procedural content allows artists to create 3D models with a constant amount of human work, regardless of the size or number of the models. While the generated models/scenes/terrain may become repetitive at some point, they form a good base for further manual editing. This is useful to produce large terrains used in films, simulations and games, where production of most of the terrain is controlled loosely, but some parts need to be controlled strictly. Stored procedural content is often more efficient to render than content that is computed lazily, but the lazy approach can be used if the scene is too large to be stored in full.

Procedural models can be used to deliver *static* and *dynamic* content. Both forms can be produced either lazily or greedily. Static procedural content is either produced once and used only in stored form (*Greedy evaluation*), or a procedural algorithm with a fixed parameter set is used to produce it (*Lazy evaluation*). Randomized procedural algorithms require a fixed random seed to produce static procedural content. Static procedural content can be edited manually to enhance it.

Dynamic procedural content allows to create new instances of content when necessary. This is useful when the user expects varying environments and objects, for example in certain types of computer games, which involve exploring changing surroundings. Even so, under certain conditions it is necessary to ensure that the procedural content can be reproduced. For example, for reasons of fictional continuity, the environments may be required to look exactly the same every time they are visited by a single avatar, but they may need to be very different for another avatar. This can be ensured either by storing the random seed and parameters that were used to produce the scene, or by storing the geometry. From that point, the content is used in static form, but the dynamic procedural model can be still be used to extend the artificial environment or generate more details.

Dynamic procedural content can also be produced greedily. Applied to computer games, this means that a environment or set of objects is created completely before the user starts exploring it, but the game may generate different environments and objects every time a new game is started.

3.2.2 Extrusion

By moving one curve along another curve, it is possible to define a surface. This is called *extrusion*, and can be very useful for modeling 3D objects. *Surfaces of revolution* are defined by a curve rotated around an axis. *Loft extrusions* create a surface by smoothly interpolating between several curves. In essence, the *level of detail* for a surface generated by extrusion is given by the curves defining it, although extruded surfaces can be generated at any desired resolution: for many functions, beyond a certain resolution, more polygons generate ever smoother surfaces, but no further discernible details.

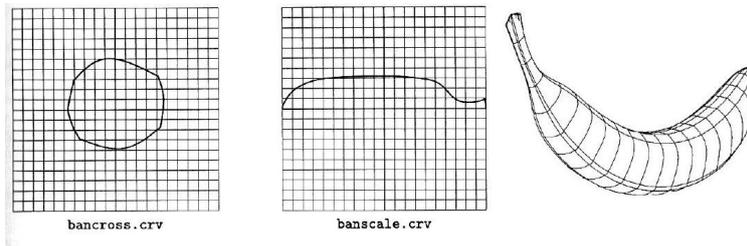


Figure 3: Two of the three curves needed to define a banana [141]

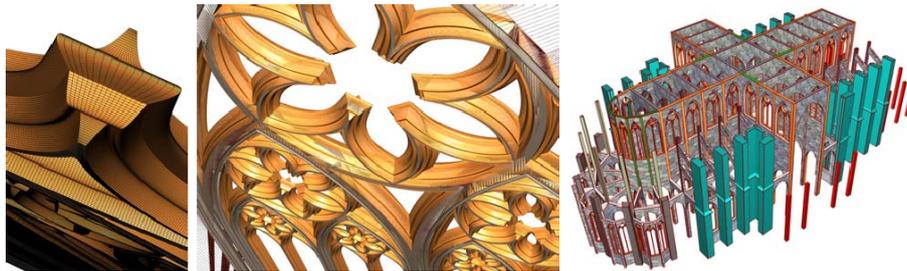


Figure 4: GML gothic architecture examples [64]

3.2.3 Generative Modeling

John Snyder generalizes the concept of extrusions by allowing for compositions of functions and calls this *generative modeling* [141]. For example, a banana can be modeled by replicating a cross section curve along another curve and by varying the radius of the cross section using a third curve (Figure 3). By introducing operators to integrate or derive functions, it is possible to calculate physical properties, or to define an object's surface to include all points where a function equals zero, similar to the *Marching Cubes Algorithm* by Lorensen et al. [90], or *level set* methods. Snyder's system was built around a C interpreter with overloaded operators.

GML (*Generative Modeling Language*, not to be confused with the *Geography Markup Language* used in Google Earth), implements some of these ideas on newer hardware [64]. It uses a notation similar to that of PostScript, which is called *postfix* or *reverse Polish notation*, where operators are placed after the operands. For example, $(3 + 2) * 4$ would be written as

3 2 add 4 mul.

Such programs are executed by pushing the literals onto the stack, then each operation fetches its operands from the stack, computes new values, and pushes the results back onto the stack, until the stack runs out of elements. This notation does not require brackets and was used on early pocket calculators because of its simplicity. GML requires human users to perform the conversion from infix to postfix notation although computers can do this faster and more reliable [2].

No solution that supports parallel generation has been presented. GML does not offer variables, instead all data is passed on the stack. At the start of a function, values may be transferred to *registers* that can be named freely, but registers are only valid inside a function. This part of a function is also called its signature.

An important feature in GML are the so-called *gizmos*, which mark salient points of an object. Gizmos work essentially the same way as control points in subdivision surfaces. Therefore, we will use the term *control points* throughout this work. When the user moves control points in the viewport using the mouse, the GML program is executed to recreate the object according to the new control point positions. Thus, GML programs define how the control points are mapped to parameters of an object. This can be duplicated in Maya by using cubes or spheres as control points, and a script that evaluates their positions to determine new parameters for a model and reconstruct the geometry. Programmable control points can be used to define the side lengths of a cube or building, the radius of a cylinder or tower, the opening angle of a piece of pie, or the hull of an object whose surface is defined procedurally. GML further features *subdivision surfaces* and *dictionary stacks*. Dictionary stacks store arrays of key/value pairs, similar to records. GML originally produced polygon lists, but was later extended to support scene graphs [53]. Figure 4 shows example models built using GML.

3.2.4 Programming and Scripting for Procedural Modeling

Ullrich et al. examine the use of JavaScript for generative modeling [153], but any Turing-complete programming language can be used to compute geometry [154]. Geometry can be written to a file and viewed using external tools. Recently, support for Python was added to several 3D modeling suites. These scripts can access the scene graph stored in the 3D suite. For example, this can be used to create objects with geometry computed by the script. Also, standard primitives like spheres can be used to mark salient points on the object, and the script can use the positions of these primitives as *control points*.

3.2.5 Fractals

Mandelbrot has characterized a number of mathematical phenomena as a new class called *fractals* [93]. The word fractal is coined after the Latin word "fractus". The corresponding Latin verb "frangere" means "to break". Mandelbrot defines a fractal as "a set for which the *Hausdorff Besicovitch dimension* strictly exceeds the topological dimension". The book by Peitgen et al. explains the Hausdorff dimension in detail [115].

The Cantor set is a fractal which is constructed as follows. Starting with the interval of $[0,1]$, each interval is split into three parts of equal length. The middle interval is removed, and the split and remove operations are repeated for the remaining intervals. After an infinite number of steps, it can be shown that the resulting set has the same cardinality as the unit interval $[0,1]$, but it is nowhere dense, which means that it does not contain any finite interval. The Cantor set has a Hausdorff Dimension of $\log 2 / \log 3$.

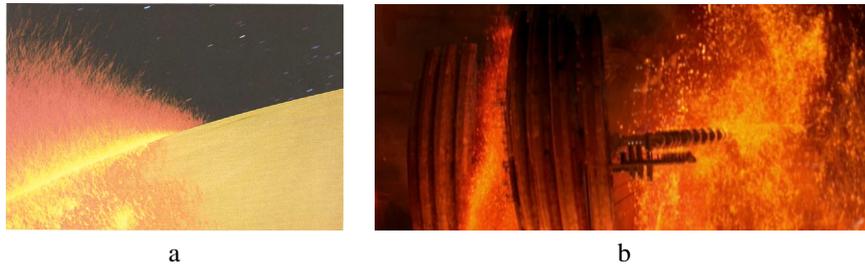


Figure 5: Particle effects. a: Star Trek II: The Wrath of Khan: Genesis effect (1982), b: Lava in Star Wars: Episode 3 (2005)

Mandelbrot has analyzed the fractal nature of many types of objects [93] we are dealing with in this work, including trees, mountains, *river networks*, and coast lines. Fractals are closely tied with procedural modeling, and many procedural algorithms can be used to create fractals. We take a closer look at some fractals in Section 4.6.7.

3.2.6 Particle Systems

Reeves introduced *particle systems* [132]. Instead of modeling surfaces with polygons or patches, particle systems model effects using clouds of animated points or imposters. *Imposters* are textured polygons that are always rotated to be perpendicular to the viewing direction. These are sometimes also called *billboards*. Particles are created and destroyed after a certain time. Often, stochastic processes govern the motion of the particles. Particle simulations may require large numbers of particles. Attributes of a particle include its position, speed, size, color, shape, transparency, age or remaining lifetime.

Particle systems can be used to simulate effects such as sparks, rain, snow, swarms, fire, explosions and similar phenomena. They were used in the film "Star Trek II: The Wrath of Khan" to visualize the *Genesis effect*, where a barren planet is made hospitable, and more recently in Star Wars, Episode 3, as demonstrated by Figure 5.

3.2.7 L-Systems

Aristid Lindenmayer [83] introduced a notation for modeling growth processes in 1968. Named after their inventor, an *L-system* applies rewriting rules to an axiom to produce a linear symbolic description of a plant, the *L-string*. An L-system requires that all matches for a rule are applied in parallel. While other formal rewriting systems keep running until only terminal symbols remain in the string, the number of rewriting iterations is an input parameter for L-systems. The next stage in the algorithm produces geometry from the L-string by interpreting the L-string as a set of commands that control movement of the so-called turtle. The turtle can move forward, rotate along its axes, store its position on a stack ("[" instruction, push), or return to the last position on the stack ("]" instruction, pop). The complete path of the turtle defines a *plant skeleton*, which is used to create geometry for the plant.

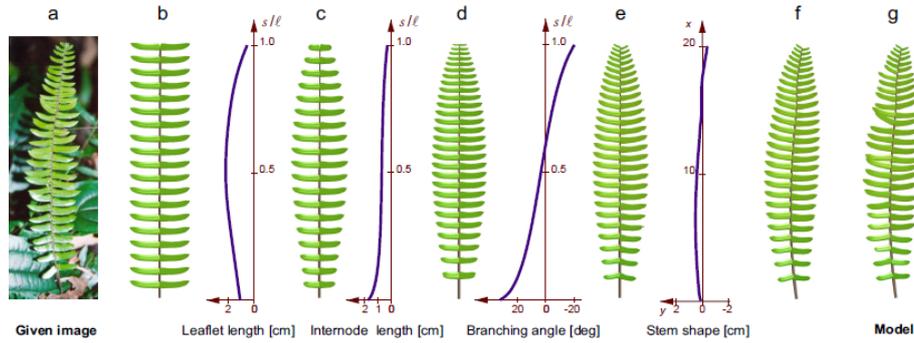


Figure 6: Using positional information to model plants

Lindenmayer and Prusinkiewicz describe this process with several extensions in their book [125]. Prusinkiewicz et al. published a series of extensions: interaction with the environment [104], an application to topiary [127], and the use of positional information [129]. *Positional information* defines how parameters vary along a branch, as demonstrated in Figure 6.

In a *deterministic L-system*, there may be only one production rule that can be applied to a single symbol in the L-string. In a *stochastic L-system*, there may be several production rules that can be applied to a single symbol in the L-string, and each possible production rule is applied with a defined probability. If each rule evaluates only one symbol in the L-string, it is *context-free*. If a rule may evaluate several symbols in the L-string, it is *context-sensitive*. Context-sensitive rules apply to the longest match. *Parameterized L-systems* allow for computation and conditional replacement. Here, the characters on the left side of ":" are replaced by the characters on the right side of "→", if the condition between ":" and "→" holds. Here is an example:

$$A(t) : t > 5 \rightarrow B(t + 1)CD(t \cdot 0.5, t - 2).$$

If $t > 5$, then the symbol $A(t)$ is replaced by $B(t + 1)CD(t \cdot 0.5, t - 2)$.

While L-systems are powerful tools in the hands of experts, they are also cumbersome to use, and even simple models may require hours to create. However, stochastic L-systems allow to model a wide variety of plants with a single L-system. In order to prevent self-intersections, L-systems need to be able to evaluate functions. An alternative approach uses C data structures rather than L-strings [128].

An application of L-systems to animate mechanical objects has been demonstrated [74]. Applications of L-systems and similar text rewriting systems to cities and buildings are discussed in Section 3.2.10. While many early rewriting systems were limited to specific domains, like plants, streets or buildings, an approach to generalize rewriting systems has been proposed [79].

A sketch system by Beneš et al. allows the user to sketch volumes [10]. Each volume is assigned an L-system that generates geometry for the volume. The L-systems may query neighboring volumes for structural information.



Figure 7: Botanical examples. a: voxel grid by Greene[58], b: Ecosystems by Deussen et al. [40]

3.2.8 Procedural Modeling of Trees

Early attempts to model biological growth on computers use simple rules on 2D grids. These include Ulam’s cellular automata [152], Conway’s Game of Life [52], and von Neumann’s cellular automata [107]. Honda describes a procedural approach that created the first 3D *plant skeletons* [69]. It produces sympodial trees. Parameters mostly consist of *recursion factors*.

Oppenheimer proposes a recursive algorithm inspired by Mandelbrot’s fractals [111]. His system needs few parameters to describe trees because he uses recursion factors. Bloomenthal demonstrates how to produce realistic branches [16]. de Reffye et al. animate *tree growth* using procedural models [35].

Greene proposes to use *voxels* (=volume pixels) to speed up intersection tests between branches [58]. This helps prevent self-intersections when simulating plant growth and speeds up ray casting to calculate *heliotropism*. During every step of the simulation, several alternatives for growth are tested, and the most likely one is chosen. Figure 7.a shows an example.

Holton proposes the *strand model* [68]. Each branch has a positive number of strands that provide it with water and nutrients. The area of a cross section of a branch is proportional to its number of strands. At branching points, the number of strands is distributed between the new branches. Therefore, the total number of strands and the summed area of the cross sections remains constant, as required by da Vinci’s law for branches. Branches that have fewer strands are at a greater angle against their parent branch than branches with more strands.

Reeves and Blau describe an application of *particle systems* to generate landscapes with terrain and trees [133]. After generating the terrain, trees are distributed in the landscape. This can be done manually or rule-based. For rule-based placement of the trees, the region to fill, minimal tree distance and the terrain must be specified. Other parameters include the type of trees, possibly depending on altitude, amount of water and sunshine. Wind can be simulated in a grid which moves the trees or branches.

Weber and Penn propose a parameterization for trees designed for use by non-experts [159]. Instead of using recursion factors, they have parameters that apply to all branches on the same level of recursion. Therefore, parameter changes always affect all branches on the same level. It is not possible to edit individual branches. There are special parameters with a 'V' suffix which define a maximum random deviation from the base parameter. Monopodial or dichotomous branching are supported. For branching, branches can either be cloned, meaning that new branches inherit their parameters from their parent branch, or child branches are treated as a new level of recursion that uses a different set of parameters. Leaves may be placed on the last level of recursion. The paper provides many examples demonstrating that the parameters suffice for modeling a wide range of tree species, and discusses approaches to simulate pruning, wind sway, flaring at the base of the trunk, *gravitropism* and how to manage level of detail.

Deussen et al. describe a *tool pipeline* to generate landscapes that may include thousands of trees [40]. It includes an editor for terrain that can simulate erosion. Plants can be placed manually by the user or they are distributed using a half-toning algorithm that can simulate plant growth. Geometry for the plants is created using cpfg or Xfrog. cpfg implements an L-System that can simulate competition for light and space, resources in the soil, aging and death. Xfrog is discussed in the following subsection. Images are rendered using one of several methods, depending on whether the data fits into main memory. Figure 7.b shows an example.

Power et al. apply an inverse kinematics optimization technique to create a natural branch form while a user drags a branch, and branches can be bent, rotated or pruned in the viewport [123].

Since Weber and Penn published their system, computers have evolved to a point where it is possible to store parameters for individual branches, as demonstrated by Boudon et al. [19] for L-systems. Their system collects parameters in a separate hierarchical data structure called *decomposition graph*. The parameters can be inherited to child branches, similar to the inheritance mechanism used in object-oriented programming languages. The user may edit the silhouette of a branch including sub-levels, and an L-system fills it with the next level of branches and leaves.

Another area of research focuses on groups of trees whose proximity limits their growth. Beneš et al. define the shape of a plant as a surface of revolution [12]. Inside this envelope, buds are modeled as particles. An image created from the location of the light source is used to estimate incident light. Bud growth is affected by tropisms, but buds that come too close are killed. Palubicki et al. demonstrate a similar system that simulates how trees grow to fill open space [113]. The system assumes a natural distribution of the branches to reduce the number of parameters the user has to define. The user can guide growth by sketching.

Livny et al. presented a technique to model trees using lobes [88]. A lobe combines several branches and leaves to a single entity. Lobes may be used several times when rendering a tree, which reduces the amount of memory used by the individual *instances*.

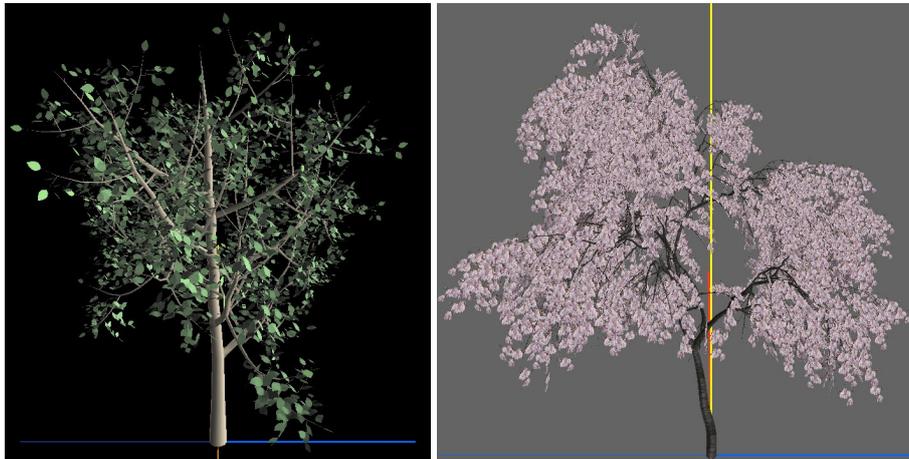


Figure 8: Examples of trees modeled using Xfrog

3.2.9 Xfrog

Xfrog is an important predecessor to the systems discussed in Section 4 and Section 5 because it is more intuitive than other previous approaches to editing trees. It is a visual program for designing plants in 3D [84] that deserves to be covered in detail. The program has a number of components that are used to create plant organs and geometry. In contrast to VDFPs, edges do not transport data but control relative placement of the organs. There are three different types of edges, with *child link* being the default. It places subsequent components relative to their parents. Another edge type is the *leaf link*. A node with a leaf link requires two child nodes. Components have a parameter that defines the number of recursions to perform on the child nodes. If a parent node requires further recursions, in case of a leaf link, the first child node is executed, but if no further recursions are required, the second child node is executed. Leaf links can be used to produce either branches using the first child node, or leaves if no further branches need to be produced. Using this edge type, leaves will only be placed on branches that have no child branches. The *branch/rib link* is the third edge type supported by Xfrog, and can be used to multiply components. Xfrog has been used to produce very convincing plants, as demonstrated by Figure 8. The following components are available:

- Simple: creates a primitive (see below),
- Horn: creates a stem or branch,
- Tree: like horn, but with starting points for new components,
- Leaf,
- Revo: creates a surface of revolution from a curve,
- Hydra: multiplies components around a point,
- PhiBall: multiplies primitives in phyllotactic order,
- Wreath: multiplies primitives on a circle,

- Attractor: deforms geometry around a ball,
- HyperPatch: deforms geometry by moving grid points - useful to simulate wind.

The components have numerous parameters that can be used to translate, rotate or scale them. Tropisms are supported using attributes. The following primitives can be used:

- None: produces nothing,
- Box,
- Sphere,
- Cone,
- Cylinder,
- Torus,
- Circle,
- Tube: creates vertices that are connected to a tube, useful for creating branches,
- Area: connects vertices from functions to an area,
- Attractor: see above,
- Triangle up: a triangle pointing up,
- Triangle down: a triangle pointing down.

Textures for leaves and bark can be photographed or scanned. Xfrog supports *keyframe animation*. In keyframe animation, the user states a complete set of parameters for each so-called *keyframe*, and the system produces an *animation* by calculating additional frames from interpolated parameter values. Growth can be animated by starting with size 0 for the organs and allowing a greater size later.

Some of the *attributes* support evaluating functions, but Xfrog is not a programming language, since variables and execution control (*if ... else, for, while, ...*) are missing. Learning Xfrog may be easier than learning to model plants using a scripting language or L-system because the visual components are very intuitive and the user does not have to type keywords. The components are strictly geared to produce plants, and the number of parameters may be overwhelming. Splitting the components into more specialized ones might increase flexibility. There appears to be no way of modeling directly in the viewport.

3.2.10 Procedural Modeling of Cities

Parish and Müller proposed a system for procedural modeling of cities [114]. City creation starts with an extended L-system for generating the *street network*. It differs from an L-system for plants in that dead ends are rare for roads, whereas tree branches rarely connect with other branches, except their parent and child branches. 2D maps can be used as input for the system, which may include elevation maps, land use,

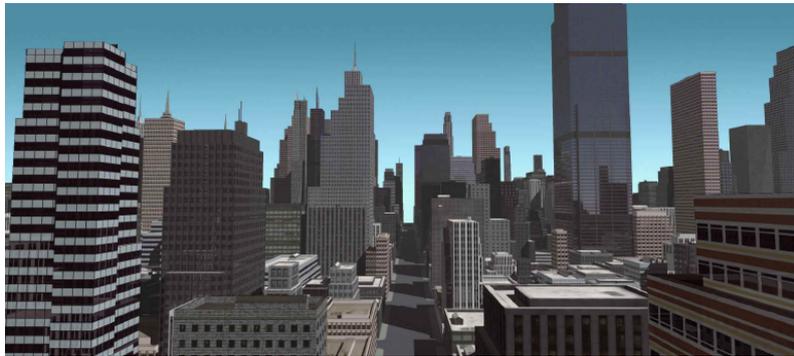


Figure 9: City example with simple mass models [114]

population density, and *street patterns*. Street patterns can be blended between New York/checkers, Paris/radial, San Francisco/least elevation, or no superimposed pattern.

Parish and Müller's L-system for road generation has very many parameters. In order to increase flexibility, a mechanism was implemented to ease changing rules. The L-system generates a generic template in each step, which is called the ideal successor, and external functions set and modify parameters in the L-system modules. Global goals are used to initialize parameter values such as street patterns and population density. The function `localConstraints` attempts to fit the parameters to local constraints, including land use, elevation, and street crossings. If no suitable parameters can be found, the module is deleted. The system by Sun et al. demonstrates how to use *Voronoi diagrams* to ease detection of road crossings [147].

Then, space between the roads is gathered to blocks, and finally buildings are placed into the blocks. The buildings are created by combining primitives using Boolean operations, controlled by a parametric, stochastic *L-system*. Textures for buildings may be extracted from photographs or can be generated procedurally. Buildings use a *texture atlas* that includes repeated elements only once. Figure 9 shows an example produced using this pipeline.

Wonka et al. [163] draw on earlier theoretical work by Stiny [145] to create a wider variety of buildings. While L-systems reflect biological growth of a plant, buildings are better described by dividing space into components. *Split grammars* define buildings using two types of rules: *Split rules* are used to split objects and *conversion rules* change the type of an object. The rules are governed by attributes, which are assigned by a control grammar. Control grammars assign a number of properties to buildings, describing material, construction rules and texture. In contrast to L-systems, execution does not end after a fixed number of steps, but rather when all symbols are terminals.

Laycock and Day present an algorithm that derives a building layout from the floor plan [82]. The system supports several types of roofs (hip roof, gable roof, mansard, gambrel and dutch hip). The corners of the roof are computed automatically. Roof types may be mixed by splitting floor plans into rectilinear polygons.

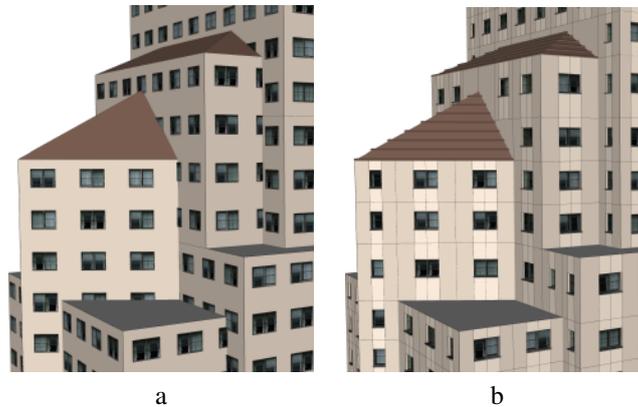


Figure 10: a: A building composed of several parts with partially overlapped windows and non-aligned floors, b: result after applying snap rules and intersection test [102]

Marvie et al. concur that the turtle metaphor used for plants is not useful for other types of shapes, e.g. buildings [94]. Therefore, they use an L-system to generate a description of urban environments, but use functions to evaluate the terminal symbols. The functions create geometry or define 3D transformations. The system also supports style information and *levels of detail*.

Müller et al. demonstrated how to integrate shape grammars with mass models [102]. Their grammar, called *CGA shape*, allows to generate models from context-sensitive rules, e.g. in order to prevent buildings from intersecting and allowing layout to adapt to nearby buildings and building parts. The buildings are organized in an *octree* to speed up *intersection tests*, and if an intersection is found, the split grammar is invoked to make changes. While L-systems use parallel rewriting to describe growth, *CGA shape* prioritizes rules and uses sequential rewriting. Typical objects include cubes, quads, cylinders, or any other 3D model. Objects may be translated, scaled or rotated. A split rule (`Subdiv`) splits an object and its bounding box along one or more axes, and sizes of the elements may be stated as absolute values or the values are relative to the size of the parent bounding box. The `Repeat` function splits objects into a number of objects that fill up the bounding volume. Objects may be split into components using the `components` function. Occlusion queries can be used to test for intersecting shapes in the condition part of the replacement syntax. The `snap` rule can be used to align parts of a building to dominant lines of other parts. By default, all faces of the mass model can be used as snap lines, as demonstrated in Figure 10.

Lipp et al. demonstrated a user interface for *CGA shape* that visualizes grammar rules as graphs [86]. Grammar rules are manipulated by editing the graph without typing on a keyboard. The system allows for semantic and geometric selection: It is possible to select and edit a single instance of an object, to select all instances with a certain value for a variable, or to select all instances below a certain node of the hierarchy. Objects are identified by specifying the names of all nodes from the scene graph root to the selected instance. Modifications are stored in an external data structure to ensure persistence.



Figure 11: Roads by Galin et al. [49]

Weber et al. simulate city growth over time [158]. The street network is stored in a graph whose edges are streets and the faces are building blocks. Each face has zoning regulations that define the type of land usage, e.g. commercial or residential. Buildings are selected based on land usage. For reasons of efficiency, parts of the simulation have been implemented to support parallel execution, and the street network does not use an L-system. Instead, the necessity for new streets is determined by testing how the street network handles trips between destinations.

The G^2 system by Krecklau et al. implements a text replacement system that can produce buildings and plants [79]. The grammar creates and manipulates objects that are stored in a scene graph. Terminal symbols may have attached non-terminal symbols that store style information, e.g. function parameters. These allow the application of further rules without multiplying rules unnecessarily. Rounded shapes may be defined using freeform deformation surfaces (FFD's).

Galín et al. describe algorithms that generate roads by minimizing a cost function [49]. The cost function is a line integral over several weighted functions, including the length of a road, elevation, cost of bridges and tunnels, while avoiding narrow curves and steep elevation changes. Dijkstra's algorithm is used to find the route with minimal cost through the grid. The vertices forming the road are connected to a clothoid spline [156]. The mesh surrounding the road is altered to support the road. Figure 11 shows an image of a road constructed using this system.

A new scene description language by Krecklau and Kobbelt allows to model connection points between objects [80]. The same syntax is used for deformable objects as for rigid objects. The grammar allows to connect end points of chains using inverse kinematics. Alternatively, chains may be modeled as FFDs.

Merrell et al. present a technique for generating residential buildings [101]. The algorithm starts by specifying a so-called program for the building, which is a list of rooms and their connectivity. As there are many possible rules that govern the number, size,

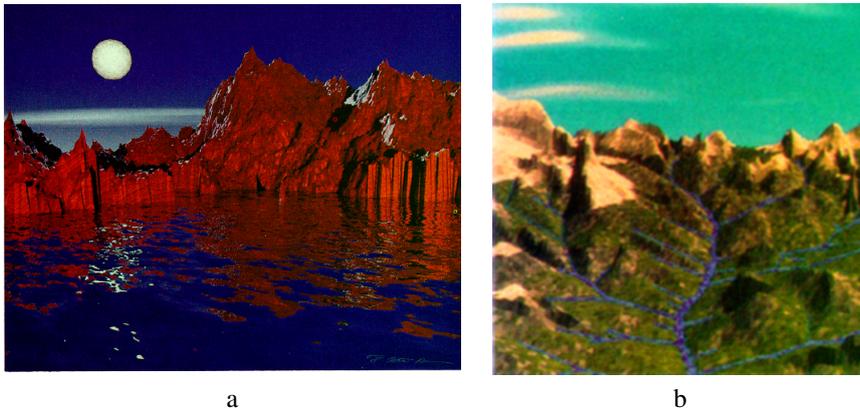


Figure 12: Terrain examples. a: Lethe by Musgrave et al. [103], b: Shreve Valley by Kelley et al. [76]

and connectivity of the involved rooms, a Bayesian Network was trained using 120 programs. From the generated program, the rooms are derived using the Metropolis algorithm [124]. The Metropolis algorithm shifts walls between the rooms until all conditions in the program are satisfied or a certain amount of time has passed.

A user interface by Lipp et al. supports cut-and-paste style editing for street networks [85]. The user selects part of a street network to cut, paste, move or edit, and a merging algorithm fits the existing and selected parts together. The merging algorithm seeks to fulfil a number of constraints that ensure local consistency and that the operations are reversible. An alternative algorithm is used in case that the pasted map piece is not allowed to change at all.

Kelly and Wonka describe a procedural model that expands on Laycock and Day's work [77, 82]. In addition to the floor plan, a vertical cross section of the building is given, the profile. By sweeping the profile along the floor plan, a 3D model is created. Different parts of the building may use different profiles.

3.2.11 Procedural Modeling of Terrain

Geomorphology studies the processes that shape the relief of Earth. Among these are crust movements, vulcanism and erosion. Erosion can be caused by temperature changes (thermal erosion), water (fluvial erosion), glaciers (glacial erosion), wind (eolian erosion), and other effects. An overview of the fractal algorithms and texturing approaches can be gained by reading the books by Ebert et al. [45] and Peitgen et al. [115].

Fournier et al. demonstrated how to produce terrain or entire planets using *midpoint displacement* [47]. Their algorithm starts with a polygon or sphere and recursively inserts new vertices which split polygons into several new polygons. A new vertex's altitude is the average altitude of the surrounding vertices, plus or minus a random value depending on the length of the edges. Midpoint displacement can be used to

globally increase the resolution of terrain, or to locally adapt the *level of detail* of a terrain to the camera position. For terrain created using midpoint displacement, it is possible to select an altitude that represents sea level to obtain mountain ranges near the sea. Erosion processes tend to create rough mountains and round valleys. Real terrain has *river networks* in the valleys that usually run only at small slope, whereas mountain ridges decline steeply. These effects can be modeled in midpoint displacement by varying the fractal dimension [103]. However, the algorithm is unsuitable to increase the resolution of eroded terrain. As the algorithm does not check which areas are part of river networks, it might introduce mountain peaks into rivers. Bokeloh and Wand [17] propose a GPU-based implementation of the midpoint displacement algorithm with view-dependent refinements. According to their research, this increases performance by a factor of ten. Figure 12.a shows an example image, and Section 4.6.5 presents an implementation of the algorithm.

Kelley et al. proposed an algorithm for generating procedural terrain with river networks [76]. Their algorithm uses a mesh data structure and the observation that water flows along edges that are lower than the surrounding landscape. Their algorithm marks edges in the mesh as rivers to form river networks, then it calculates river vertex altitudes and finally assigns higher altitudes to the surrounding mountains. All tributaries in the terrain lead into a single, main river. Figure 12.b shows an example terrain created using this technique.

Mandelbrot issued a call for an algorithm to create fractal terrain with river networks [115]. Bardeen proposed such an algorithm, but its details remain unpublished, so we cannot compare this approach with others [7]. However, a compiled version can be obtained as a plugin for the software *MojoWorld*. Prusinkiewicz and Hammel integrated midpoint displacement and the creation of a single river into a text rewriting system [126]. They report that the river flows at a constant altitude, valleys are asymmetric, one valley side may be very steep, and that the river has no tributaries. Belhadj and Audibert demonstrated how to use a *particle system* to create mountain ridges and river networks on a grid [9].

A number of approaches generate terrain that tries to satisfy constraints. Szeliski and Terzopoulos generate terrain by minimizing an energy function to fit user-defined points [148]. Pouderoux et al. describe an algorithm that recursively subdivides terrain into several domains [122]. Each domain is approximated by radial basis functions that are combined into a function that describes the entire terrain. Stachniak and Stuerzlinger use a Gaussian kernel as a deformation operator to find terrains that are within ϵ from the given constraints [142]. An algorithm by Belhadj uses midpoint displacement for reconstructing terrain under local constraints with similar results [8], but their algorithm is reported to be more than 150 times faster than the algorithm by Pouderoux et al. [122]. These algorithms could also be used for procedural modeling by procedurally defining the constraints.

Hultquist et al. demonstrate a system that lets the user describe a scene with adjectives. The procedural system matches the adjectives to a parameter space for generating a scene [71].

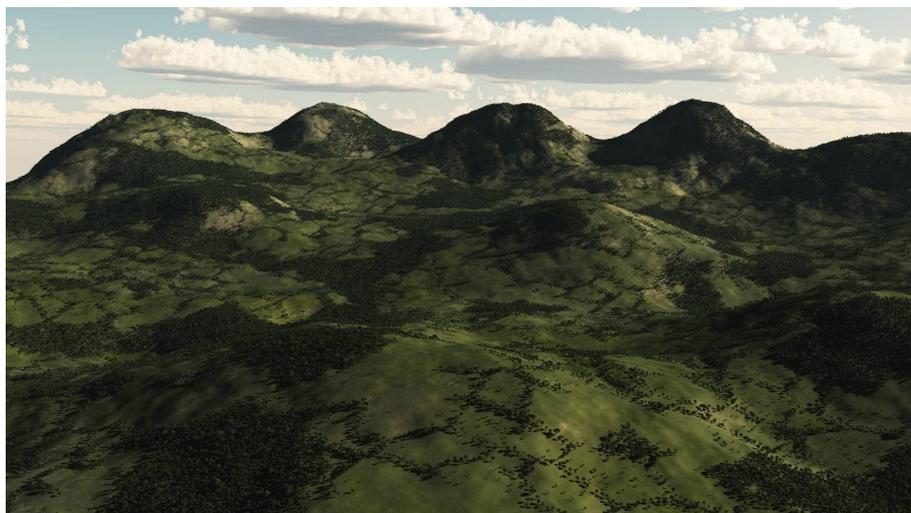


Figure 13: A terrain edited using tools proposed by Hnaidi et al. [66]

Several algorithms that generate new terrain by transplanting detail from existing terrain have been proposed. Brosz et al. start from a given base terrain, giving a rough estimate of the target terrain to generate [20]. Salient points of the base terrain are mapped to the target terrain using multi-resolution analysis. The mapping can be defined by the user or it is deduced automatically [46]. In the algorithm of Zhou et al., the user sketches the terrain [164]. The algorithm searches for suitable regions from an example height field that can be used in place of the sketched terrain.

Bruneton and Neyret store terrain in a quad tree [21]. The user may move parts of the terrain, and the system ensures that objects at the terrain surface remain at the surface. Roads, rivers and railways are treated as splines. Details can be created procedurally or manually.

Smelik et al. combine procedural generation with sketching for editing terrain [139]. This technique is called procedural sketching. Ecotopes are areas of homogenous terrain and features, and the user splits terrain into ecotopes by sketching. Rivers, streets and cities are also provided as sketches. Editing operations may affect objects in the neighborhood of the sketched object, and the user adds detail using several tools.

Hnaidi et al. proposed a method that defines terrain from user-defined control curves that include ridge lines, river beds, hills, cracks and cliffs [66]. Two-dimensional piecewise *Bezier* cubic splines are used to define the control curves. The terrain is computed from a set of partial differential equations which are solved on the GPU [112]. While the height field can be generated at arbitrary resolution from the curves, the *level of detail* is limited by the curves that define the terrain. At high polygon counts, additional polygons make features rounder but do not add further detail. Figure 13 shows a terrain created using their algorithm.

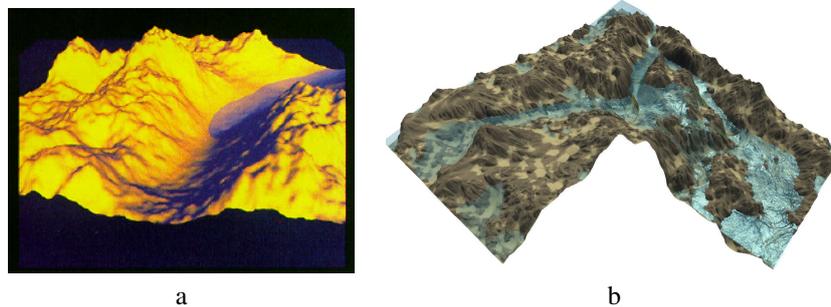


Figure 14: Terrain examples. a: Erosion simulation by Musgrave et al. [103], b: Erosion simulation running on current graphics hardware [11]

3.2.12 Erosion Simulation

Musgrave et al. proposed to simulate *fluvial* and *thermal erosion* on fractal terrain [103]. During every step of *erosion simulation*, rain is dropped onto the surface and gathered in lakes, until it leaks at the lake's lowest border, where glaciofluvial erosion forms a new river bed. As the simulation proceeds, many initial lakes are converted to river beds in this manner, and a *river network* is created. Terrain near rivers is usually exposed to fluvial erosion, so the fractal dimension is decreased at lower altitudes or near rivers. Figure 14.a demonstrates the idea of the algorithm.

Numerous extensions to erosion simulation have been proposed. Chiba et al. simulate erosion by particles with a focus on effects caused by their velocity [27]. Beneš and Forsbach proposed a data structure that stores several layers of material in a grid. For each layer, material information and thickness are stored [13]. In order to find the material at a given point in 3D space, it may be necessary to add the thicknesses of several layers of material, but this representation is more memory efficient than e.g. voxels [119]. They also showed that if the terrain is split into strips, erosion simulation can run in parallel for each strip, only the boundary areas need to be treated separately [14], and they demonstrated how to integrate evaporation into erosion simulations [15].

Neidhold et al. use a fluid solver to speed up erosion simulation, allowing to obtain an eroded landscape in a matter of minutes [105]. Their algorithm allows the user to change the settings for erosion simulation while the simulation is running. Kristof et al. used smoothed particle hydrodynamics for erosion simulation, which is based on a particle simulation to approximate the Navier-Stokes equations [81].

An optimized GPU implementation of erosion simulation has been proposed by Št'ava et al. [143]. The simulation achieves 20 frames per second on a grid of 2048x1024 with four layers of material. The user may adjust parameters interactively. The system simulates forces and pressure fields for rapidly moving water [95], and slowly moving water erodes terrain through dissolution [11]. Figure 14.b shows a terrain eroded using this implementation.

3.2.13 Procedural Textures

The idea of midpoint displacement can be applied to creating textures as well. Instead of an altitude, this yields a color value for each pixel. Perlin proposed to use a hash of the coordinates for the new texel in place of a random function [116, 117].

For procedural modeling, textures often need to be created in an automated manner. Creating the texture for a 3D object by evaluating a 2D function for each texel may result in several problems. The problems of creating texture atlases pointed out in Section 2.2.2 also play a role here. Firstly, the textures for two polygons likely will not fit at seams of the UV mesh. Secondly, if the polygons in 3D space are not proportional to their size in the UV mesh, the texture may look distorted. Thirdly, if the texture function is anisotropic, this will carry over to the 3D model, meaning that polygons that were rotated in the 3D mesh may deviate from the uniform look.

These problems can be solved by using 3D textures instead [116]. These are also called *solid textures*. When using solid textures, the 3D coordinates are transformed into the volume data, so there is no UV mesh and there can be no seams. Textures use a fixed number of texels per unit surface area, and anisotropic materials are rendered correctly. Of course, a 3D texture may require far more memory than a 2D texture, and for opaque objects, only a fraction of the pixels may be visible. Therefore, algorithms were developed to create 2D textures from 3D textures that avoid the problems described above.

The practice of creating a texture from a number of effects is called *texture baking*. For example, it is common practice to bake lighting simulations such as ambient occlusion into textures. If a UV mesh exists, 3D textures can be baked into a 2D texture in the following manner. The algorithm iterates over all polygons. For each polygon, the vertices v_1, \dots, v_n are retrieved. Every texel that belongs to the polygon is transformed into 3D space by interpolating the vertex coordinates with the u, v values associated with the texel. The resulting 3D coordinates are used to evaluate the 3D texture function, whose value is copied into the texel of the 2D texture at the u, v coordinates.

This solves texture inconsistencies at the polygon seams, and anisotropisms present in 3D space carry over correctly to the 2D texture. When using a *texture atlas* [24] with *mipmapping*, seam artifacts can be prevented by obeying a few simple rules. Firstly, the polygons must be placed so that several texels separate each pair of polygons, otherwise, texels from other textures may bleed into the seams. Secondly, it is advantageous to place polygons that use a single material in a different section of the UV mesh than those of other material, and to leave a wider margin between polygons that use different material. Thirdly, the texels between the polygons must be filled with the nearest texel belonging to a polygon, otherwise, uninitialized texels could bleed into the seams.

Cutler et al. demonstrate a scripting language to define the internal structure of an object [33]. This can be used to produce solid textures for objects that are cut or broken. Different basis functions for solid texturing have been proposed [45]. A number of effects can be modeled using 3D *Voronoi diagrams*, noise, and other volumetric procedural effects. If layers of material are modeled correctly, textures can be created when it becomes necessary to display cross sections, for example, when cutting through food.

3.3 Terrain Rendering and Level of Detail

Editing terrain usually requires rendering the terrain, so terrain rendering algorithms are reviewed in the following. For rendering terrain data, it is common practice to locally adapt the resolution of the geometry to the distance of the viewer. Duchaineau et al. store triangles in a binary tree, where each level stores the geometry for a single *level of detail* [43]. In each frame, the level of detail is checked for each triangle and may be adjusted using splits or merges. Losasso and Hoppe introduce the *geometry clipmap*, which stores geometry for quadratic regions centered around the user, similar to *mipmapping* [91].

Dachsbacher and Stamminger use geometry images as proposed by Gu et al. to render terrain [34, 59]. Geometry images encode a mesh as a 2D image by mapping the x , y , z coordinates to the red, green and blue color channels of an image. Neighboring pixels in the image are assumed to form quads, but the connectivity does not have to be stored. Instead, the mesh is rendered as quads whose vertices are read from the image in a vertex shader. While this data structure allows to vary the level of detail locally without having to adapt the mesh, culling occluded areas has not been demonstrated for this algorithm. The level of detail in the input height field can be increased using sum of noise functions. For texturing, material information is assigned to every texel, such as snow, rock, or grass, depending on the altitude, elevation and orientation.

Google Earth is a 3D landscape viewer created by Google. Input data is taken from satellites and aerial photography and hosted on Google servers. When zooming into the landscapes, texture detail pops into view as data is loaded from the servers, but it is possible to zoom from orbit down to individual houses. Users may add 3D models to enhance the views.

Rogge et al. demonstrate a system that allows to visualize continental drift [135]. The system uses a database that includes information on climate and vegetation. The user has to assist the system to select continents and model their motion.

3.4 Further Modeling and Editing Approaches

The approaches and algorithms discussed in this subsection are not directly related to procedural modeling, but can be useful under similar circumstances or in combination with procedural modeling.

3.4.1 Terrain Editing

As some of the algorithms in this work deal with editing terrain, we need to review previous approaches for these problems. Peytavie et al. propose several tools for editing terrain [119]. Canyons, tunnels and cliffs can be carved from the terrain by moving user-defined curves along other user-defined curves. The surface of a terrain is defined implicitly: $S = \{p \in \mathbb{R}^3 | f(p) = 0\}$. They use the same data structure as Beneš and Forsbach [13]. Material is transported between neighboring stacks in the grid if their slope exceeds a certain angle. The user may add or remove material using a spherical tool. An additional tool is used to carve canyons, tunnels and cliffs from terrain, by moving user-defined curves along other user-defined curves. For rendering, the implicit representation of the surface is converted to triangles.

Another algorithm by Peytavie et al. fills volumes using rock piles that are created by dividing a volume into *Voronoi* cells [118]. The cells are eroded except at chosen points where the cells touch to stabilize the rock pile. The method does not guarantee a stable rock pile, but it produces visually plausible results. The results could be improved by simulating rock movements until they stabilize. For generating larger areas of rock piles, it is possible to generate tiles of rocks. The tiles must be placed in an aperiodical fashion to avoid visible repetitions.

Vanek et al. present a system that allows to edit terrain at several *levels of detail* [155]. Level of detail is based on tiles and mipmapping, but cracks between the tiles are not handled by the algorithm. The input can be loaded from a file or it is created manually. Physics-based effects such as thermal weathering and hydraulic erosion are calculated on the GPU.

3.4.2 Sketch-Based Modeling

Sketch-based modeling is an important input technique that should be considered when designing convenient and intuitive procedural systems. It allows a user to create a 2D sketch from which the computer derives a 3D model.

Gain et al. propose a sketch-based technique for terrain. The user sketches the silhouette and optionally shadow and horizontal boundaries, and the algorithm generates terrain to match the sketch [48].

Sketch interfaces have been demonstrated for trees, flowers [73] and phyllotactic arrangement of plant organs [3]. Okabe et al. presented a system that deduces a 3D model of a tree from a sketch, by assuming that trees spread branches so that the distances between the branches are maximized [110]. The system integrates three example-based

interaction techniques that can be used to alter the model after the geometry has been created. The Sketch L-system by Ijiri et al. allows the user to edit production rules in the 3D viewport [72]. The system applies the production rules interactively while the user sketches the trunk.

In the system by Chen et al., the user sketches the branching structure and optionally the tree crown [26]. The system consults a database to find a fitting tree template, which is used to find suitable parameters for branches and leaves. The database contains 20 exemplars, which were created using the technique by Neubert et al. [106].

In the system of Wither et al., the user specifies trees by sketching the convex hull for a single branch [162]. The system derives parameters from the sketch and applies them to other instances.

3.4.3 Image-Based Modeling

Image-based modeling strives to understand and reproduce shapes from images. This could help to generate models that use aspects of existing objects. Early works on *reconstructing buildings* from aerial photos and photos of façades include [65, 36]. Since digital cameras became widespread, research on reproducing plants and buildings from images has received more attention [131, 136, 138, 150]. The user interface by Quan et al. allows the user to draw and move curves, to edit the radius of a branch, or to place leaves in the 3D viewport [130]. The system by Neubert et al. [106] requires only few input images and permits sketching. The user provides only approximate positions for the images, and no registration is needed.

Livny et al. present a technique to automatically reconstruct tree skeletal structures from point clouds [89]. The method supports multiple overlapping trees without segmentation. The branch structure graph, also called the tree skeleton, is constructed and optimized using an orientation field [106]. Geometry is created in the last step.

Ma et al. present a technique that creates new images by duplicating parts of existing images [92]. The algorithm minimizes an energy function and matches samples based on their position and size. The paper demonstrates the technique with images of various topics.

3.4.4 Modeling by Example

Merrell et al. present several algorithms that extend a given input object while keeping its characteristics. For a base mesh composed of several predefined shapes [97], rules are derived that describe alternative arrangements of the parts composing the shapes. New shapes can be generated from the predefined shapes and extracted rules. Possible further rules include symmetries and limitations to user-defined volumes. The idea is extended to meshes using a function [98]

$$E(x) = \begin{cases} 1, & \text{if } x \text{ is inside the polyeder of the input mesh } E \\ 0, & \text{otherwise.} \end{cases} \quad (4)$$

The output mesh is also considered as a function $M(x)$. $M(x)$ is constructed so that all points in $M(x)$ are locally similar to a point in $E(x)$, but $M(x)$ may differ globally from $E(x)$. The downside of this algorithm is that the user cannot steer the creation of the output mesh. A further paper by the authors adds constraints that allow users to control the size of the objects, or to define algebraic constraints, as well as connectivity [99]. The authors describe a similar algorithm for synthesizing new curves from input curves [100]. Here, the output curves consist of segments of the input curves.

The system by Talton et al. takes an L-system and a target geometry, and tries to grow a model from the L-system that matches the target geometry [149]. The algorithm uses Markov chain Monte Carlo optimization methods to find suitable productions of the L-system [57]. Context-free grammars are used to generate objects, and the technique is limited to shapes that can be generated using the input L-system.

3.5 3D Modeling in Practice

Computer graphics research is very practical in nature. Computer graphics papers normally have to prove that the presented ideas work by providing a prototype implementation which produces agreeable pictures. In movie and computer game productions, tools and workflows are required that work reliably and mostly without development and experimentation, which sometimes leads to different solutions. Knowing these approaches, including their strengths and shortcomings, can be an important inspiration to research problems that have practical relevance.

3.5.1 3D Modeling Suites

3D modeling suites are relevant for this work for several reasons. They demonstrate possible solutions for workflows, where a lot of design effort was invested to find intuitive solutions. Also, 3D artists receive a lot of training to enable them to use these tools productively, and will learn similar designs more easily than completely new designs. In order to maximize convenience for these users, new solutions should be integrated as plugins into these modeling suites. The 3D modeling suites are furthermore important because they define a common knowledge base for all people who work in 3D graphics.

Programs like *3ds Max*, *Maya*, and *Cinema 4D* integrate modeling, animation, *raytracing*, and many more features into 3D modeling suites that can be used to create virtual 3D environments. Although these programs differ in details, an artist's overall working approach is more or less the same, so they are categorized as 3D modeling suites in this work. The modeling process usually focuses on creating and manipulating primitives in a scene graph. For organic shapes, the polygon mesh can be refined using subdivision surfaces. The packages also contain controls to deform objects. Textures play an important role in enhancing the look of objects. Textures can be created procedurally, by painting directly on the object similar to the way an airbrush works, or they are created using external 2D paint tools.

The suites offer powerful functions to animate objects. Movement can be defined using *keyframe animation* or mathematical formulas. Poses can be defined by placing each joint at a certain angle (*forward kinematics*), by pulling the ends of the extremities (*inverse kinematics*), or by mixing both. Motion blending is used to control changes between different types of animation, such as walking and running. Mass effects like rain, bubbles, snow, explosions can be simulated using *particle effects*. 3D modeling suites support creating fixed animations. However, for use with procedural graphics, these animations have to be adjusted to the walking styles of different persons, different types of ground, and slope. On treacherous ground, a character will walk carefully, to prevent skidding and falling, and the animations have to be blended smoothly.

3D modeling suites usually can be programmed using scripting languages, class libraries, or visual programming languages. These languages could be used either directly or with minor additions for procedural modeling. Abilities to govern the creation and placement of objects by rules are usually limited and focused on specific goals, such as specialized solutions for hair and fur, or spreading objects over a surface.

3.5.2 Developing Plugins

For this work, three systems have been evaluated for plugin development: *Cinema 4D*, *Maya* and *Verse*. An SDK and examples accompany each installation of *Cinema 4D*. Documentation can be downloaded from <http://www.plugincafe.com> and there is a forum at <http://www.plugincafe.com/forum>. A *Cinema 4D* scene is stored in an object named `BaseDocument`. Triangles and quads are stored in `PolygonObject`, which is derived from class `PointObject`. Polygons with more than 4 vertices are stored in `NGons`. Dialogs are described by `.res` resource files. Python was integrated as an additional scripting language. *Cinema 4D* supports several types of plugins. In this work, the most important plugin type is the node plugin, which can be used to create additional nodes in the scene graph. *Cinema 4D* plugins are dynamically linked libraries (DLLs) under Windows, or shared libraries under MacOS.

Plugins for *Maya* can be implemented as additional dependency graph nodes. In *Maya*, control points are called locators, and shape nodes hold the geometry. Development for *Maya* requires the *Maya Development Kit*. Gould has additional information on this [55, Vol. I, page 344, and Vol. II, page 113]. *Maya* GUI elements are created by MEL commands, Python or QT. C++ objects in *Maya* are split into storage objects, referred to as objects, and algorithms manipulating these objects are implemented in separate objects, called functors.

The uni-verse project was working on a protocol for interactive data exchange between different modeling applications via network [67]. Plugins would be used to connect to all modeling suites supported by the uni-verse project, which currently include 3ds Max, Blender, and GIMP on Windows, MacOS and Linux. Using the plugins, these systems can reference files that may be hosted locally or on a server, and the plugin keeps the scene up-to-date automatically without manual reloading. Thus, several users can simultaneously edit the same scene.

3.5.3 Procedural Modeling in Computer Games

Computer games use procedural content in both static and dynamic form. *Static procedural content* may be used to generate large environments, which can be edited manually before shipment. The benefit is that game designers can tailor the worlds to the specific needs of the story and the capabilities of the hardware. The drawback is that the player learns to master specific situations rather than adapting to new challenges and gets bored quickly after finishing the game once. *Dynamic procedural content* allows for environments that change every time the game is played.

The *Sentinel* (1986) is among the first games that rendered 3D landscapes. At that time, consumer-grade hardware was not capable of storing the amounts of data needed, so there was no alternative to generating the terrains on demand using procedural methods. The game consists of 10,000 levels. Since rendering a landscape on 8-bit computers took about 3 s, movement was limited.



Figure 15: In the computer game Spore, creatures are assembled from predefined parts

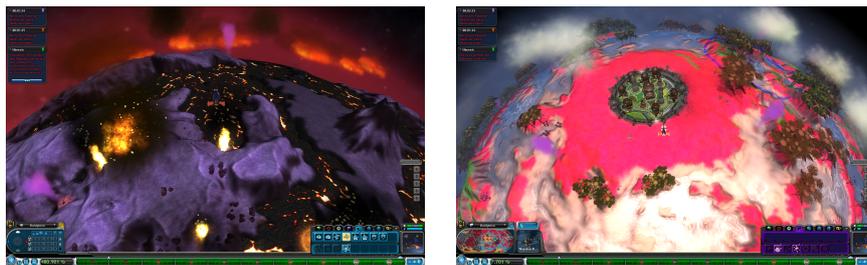


Figure 16: Spore: Inhospitable planets can be terraformed interactively

Sid Meier's Civilization (1991) constructs a procedural world and stores it in a grid. The grid has a fixed resolution, but Civilization IV (2005) uses a 3D engine to allow zooming into and out of the grid. Constructed worlds have one or more continents and simple rivers that do not branch. Prior to Civilization V (2010), a rectangular grid was used, and moving units from one square to a neighboring one always cost one movement point, even for diagonal moves, whose cost should be $\sqrt{2}$. Civilization V improves fairness of movement cost by using a hexagon field.

SimCity (1989) allows the player to build his own city while competing with AI players. The *level of detail* is fixed, but in SimCity 2000 (1994), the terrain altitudes can be increased or decreased using excavation tools, and the user may create river systems.

The computer game Spore (2008) allows the player to explore a galaxy. It uses procedural generation extensively. Creatures can be assembled from predefined parts and are animated procedurally to walk, fly or dance. Vehicles are edited in a similar fashion. Planets are created from a procedural model that allows to generate geometry at

varying resolutions, but the resolution cannot be adapted locally and it takes several seconds to generate the geometry. The terrain may include seas and rivers, but rivers do not branch. Details on the various techniques used in the computer game Spore were given in sketch presentations at SIGGRAPH 2007 [161, 37, 31, 28, 5].

CityEngine has been used for creating levels for the car racing series Need for Speed [157].

4 A New Visual Paradigm for Procedural Modeling

In comparing different approaches to procedural modeling, we find that each type of procedural modeling language has its own strengths and weaknesses. *Xfrog* is a very user-friendly solution for modeling plants, but its workflow cannot be applied for more complex procedural modeling tasks that require looped or recursive computations and it does not allow editing in the viewport. *VDFPs* are also based on nodes and offer user-defined functions, but creating and connecting the nodes for formulas may take longer than typing the formula on the keyboard, and support for loops is lacking. *L-systems* yield results comparable to those of *Xfrog*, but it takes even more time to achieve them. *Scripting languages* do not offer the ease-of-use found in *Xfrog* or *VDFPs*, unless a GUI is also implemented. GML allows to evaluate *control points*. Control points allow artists to edit objects directly in the viewport. While 3D modeling suites support this only for a few primitives, in GML this means that the generation of arbitrary objects can be steered using control points. However, this still comes at the price of mastering a language based on postfix notation.

So while we have not found an ideal procedural modeling system, we can identify traits that we would like such a system to have. These are:

- **Generality:** Our most important goal is to have a system that can create all types of models.
- **Viewport editing:** GML allows to define how a model is produced from control points in the viewport in a flexible and intuitive manner.
- **Formulas:** Procedural modeling relies heavily on mathematical formulas. Therefore, it is important that formulas can be expressed easily. Human users are used to dealing with mathematical formulas in infix notation.
- **Variables and arrays:** For supporting formulas in infix notation, data must be stored in variables. Collections of similar data can often be stored conveniently in arrays, or stored in records.
- **Loops:** We need to have a simple notation and efficient support for loops.
- **Modularization:** Rather than coding a complete model in a single module, it should be possible to reuse code from different modules. This also reduces the complexity of describing a single model.
- **Autoparallelization:** Procedural models can take a lot of CPU time to produce. Users want to take advantage of several CPU cores without bothering how it can be implemented in parallel.
- **Texturing:** Textures are very useful to increase the perceived *level of detail* without adding geometry.

Table 1 compares previous procedural modeling systems regarding the aspects discussed above. Standard 3D graphics suites include scripting languages, store their state in scene graphs, and often include visual dataflow pipelines. Table 2 describes how the features in *Maya*, *3ds Max* and *Cinema 4D* relate to Table 1.

System	Model Types	Viewport Editing	Formula Notation	Variables / Arrays	Split programs into modules	Automatic Parallelization
GenMod	any	No	infix	Yes	Yes	No
GML	any	Yes	postfix	Yes/Yes	Yes	No
L-Sys.	plants, streets	No	infix	Yes [128]	Limited [62]	No
Scripts	any	Can be impl.	infix	Yes	Yes	No
VDFPs	any	No	visual	No/Yes	Yes	Yes
Xfrog	plants	No	infix	No	No	No

Table 1: Analyzing the features of previous approaches for procedural modeling, as of April 2007

Vendor	Software Name	VDFP Name	Scripting Language
Autodesk	Maya	Dependency Graph	Maya Embedded Language (MEL), Python, QT
Autodesk	3ds Max	None	MaxScript
Maxon	Cinema 4D	XPresso	COFFEE, Python
blender.org	Blender	Nodes	Python

Table 2: Overview on visual and textual scripting languages in 3D software packages

Table 1 shows that as of April 2007, there was not a single system that met all of the goals listed above, raising the question of how such a system could be constructed. We decided to create a visual system, because it displays the structure of algorithms so clearly. There may be a big number of pipelines in a VDFP, and pipelines are unnamed. In some cases, there is no easy way to figure out the meaning of the transported data in a VDFP. This is where VDFPs fail to be easily readable. If we use named variables instead, we do not need pipelines to transport data, because all nodes draw their inputs from a central data reservoir. A VDFP defines the order of execution rather weakly, as a node is executed when its inputs have arrived. However, if we use variables, we need to define a strict order of execution. Without a strict order of execution, the values of variables and the results of assignments would not be predictable. Instead, we use edges between the nodes for defining the order of execution [63]. Furthermore, using variables allows us to state formulas in infix notation without using an excessive number of nodes. In doing so, we retain the advantage of nodes with *attributes* that wrap operations and make calling functions in visual languages so easy. The new language requires an algorithm to transform its programs to scene graphs for display. Therefore, these visual programs are sufficiently different from scene graphs and VDFPs to justify an own name. We will call them *model graphs*, as these graphs define models. Executing a model graph yields a scene graph.

The most important design goal is generality, and this can be addressed by allowing for a *Turing-complete* model of computation, and outputting polygons computed in that manner. We prove that model graphs can generate geometry that approximates any model that can be computed using finite storage and time in Section 4.6.9. While model graphs are similar to scripting languages in that they are interpreted rather than

executed natively, performance can still be improved through parallelization. The strict order of execution prevents executing nodes in parallel, but we can solve this using modularization: If one module invokes another, and the invoked module does not return data, then both modules can be executed in parallel. This is explored in more depth in Section 4.3.6. Modules that create geometry often do not need to return data. Modularization and loops are supported by allowing for special nodes in the program that call other visual programs or that execute nodes in loops.

As in *VDFPs*, each operation is reflected by a node, but model graphs differ from *VDFPs* in a number of points. First off, the edges in a *VDFP* are used to define data transfer, whereas in a model graph, edges are used to define the order of execution. Secondly, a model graph stores data in variables, whereas in a *VDFP*, data is transported between nodes without being stored in variables. A model graph may contain *cycles*, but the cycles need to be interrupted by branch operators to prevent infinite loops. Each node can have any number of predecessor and successor nodes. *Xfrog*'s nodes are specially tailored to creating plants, and they include a lot of functionality. By splitting these node types into smaller nodes, model graphs can be used more flexibly.

Since models are created from an abstract description, model graphs can output geometry at any desired polygon count, if sufficient storage is available. The predefined shapes take the number of polygons to output as a parameter, and this number can be computed from a formula, in order to meet a stated total for a complete scene.

We derive the system's name from the ability to model *plants, landscapes and buildings*, calling it *plab*.

4.1 Introducing Model Graphs

Model graphs are a node-based language. This means that programs consist of visual nodes that are connected to networks. Model graphs are intended to simplify creation of geometry and procedural modeling, so there is a specific focus on nodes that aid in these tasks. The output of a model graph is usually a scene graph. In contrast to dataflow programs, state can be stored in variables. After these variables were created, they can be accessed by all other nodes until execution of the model graph completes.

The nodes consist of attributes that specify the details of a node's operation. Each node has a specific type. The node type defines the attributes used by nodes of this type, and the operations executed by them. In general, executing a node consists of three steps: Preorder, inorder and postorder steps are executed before, between, or after visiting the child nodes. Most nodes execute their functions before visiting the child nodes. These operations can be summarized as follows:

- Previsit operations: During this phase, many nodes execute their main operation. Possible operations include:
 - Parse attribute values,
 - Create geometry or texture,
 - Create or update variables, or
 - Activate state in the scene graph.
- Visit Childs: During this phase, a node may visit child nodes.
- Post-visit operations: This is the last phase of executing a node. A node may
 - Deactivate state in the scene graph, or
 - Create or update variables.

4.1.1 Language Definition

This subsection defines the core concepts of model graphs. Firstly, we require a description of the data formats used by plab. In programming languages, the set of variables that can be accessed at any given point is usually called a *scope*. The name scope stems from telescope and is a metaphor for things that are visible in a certain context. Therefore, even if a scope in a model graph works slightly different from the usual meaning in text-oriented languages, it applies here. In plab, all nodes in a model graph can access all variables that were previously created by nodes in the same model graph. The set of variables is initially empty. Variables are not deleted during execution of a model graph, but they may be updated. The variables are deleted when execution of the model graph is complete. This means that after the creation of a variable, the scope of any variable is the entire model graph. This text will use the term "scope" to refer to the set of variables that are defined at any given time during execution of a model graph. In C++, namespaces are a similar concept to scopes.

Each plab variable has a name, type and value, and its value can be set and queried. The list of admissible variable types will be discussed in Section 4.3.2 and Section 4.3.3.

In the following, syntactic elements are described using *Backus-Naur form* [6]. Variable names use the following syntax:

```

<variable name> ::= <first char> |
                    <first char><more chars>
<first char>   ::= A | ... | Z | a | ... | z
<more chars>   ::= <more chars><more chars> |
                    <first char> | 0 | ... | 9 | _

```

As a domain specific language, the purpose of plab is to generate geometry. plab stores geometry in a scene graph. Scene graphs were defined in Section 2.2.3. Here, a *scene graph* is a directed graph where each node is a mesh. A *mesh* is also a directed graph, which is stored as an indexed faceset. An *indexed faceset* $F = (V, I)$ consists of a set of vertices $v \in V$. Each vertex is a tuple $v = (x, y, z, u, v, n_x, n_y, n_z)$, where $(x, y, z) \in \mathbb{R}^3$ are the 3D coordinates of the vertex, $u, v \in \mathbb{R}$ are the texture coordinates of the vertex, and $n = (n_x, n_y, n_z) \in \mathbb{R}^3$ is the vertex normal, $|n| = 1$. I is a set of tuples $(i_1, \dots, i_n) \in I$, $i_j \in \mathbb{N}$, $i_j < |V|$, that are interpreted as indices into V and define faces. Here, n gives the number of vertices in the polygon, e.g. $n = 3$ for a triangle. Scene graph nodes may store additional data such as transforms. Meshes were defined in Section 2.2.1.

Variable names are case sensitive. In addition, variables may be packed into records and arrays:

```

<variable identifier> ::=
    <variable name> |
    <variable identifier>.<variable identifier> |
    <variable identifier>[<formula>]

```

plab allows to use *formulas* in infix notation wherever numeric values need to be computed for node attributes:

```

<formula> ::= <pos formula> | - <pos formula>
<pos formula> ::= <number> | <function> |
    <formula> + <pos formula> |
    <formula> - <pos formula> |
    <formula> * <pos formula> |
    <formula> / <pos formula> |
    <formula> % <pos formula> |
    (<formula>) | <variable identifier>
<number> ::= <integer> | <integer>.<digits>
<integer> ::= <pos digit><digits> | <digit>
<pos digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<digit> ::= 0 | <pos digit>
<digits> ::= <digit> | <digit><digits>

```

The function names and parameters allowed for `<function>` are listed in Appendix E. While the operators `+`, `-`, `*`, `/` have their normal algebraic meaning, as limited by the data types, it should be mentioned that `%` performs modulo division. All variables that occur in formulas must exist on the scope at the time of evaluation. Otherwise, plab reports an error.

In order to define model graphs, we need the following definitions:

- An *attribute* is a key/value pair, where keys and values are zero-terminated ASCII strings.
- A *multi attribute* is an ordered list of attributes.
- A *node type* defines a list of keys of attributes and multi attributes, it declares which types of attribute values it accepts and defines the operations performed by nodes of this type.
- A node type may be derived from another node type. If node type a inherits from node type b , that means that node type a has all attributes that node type b has. Node type a may add further attributes. Attributes may contain formulas in infix notation that may reference variables in the scope.

A *model graph* is a directed graph $G = (V, E)$. Model graph vertices are called nodes. A model graph has the following properties:

- Model graph edges may only connect nodes $u \neq v, u, v \in V$.
- Each node has a single node type and provides the values for the keys in the node type to form attributes and multi attributes. A node may provide values for 0, 1, or more instances of a multi attribute.
- Outgoing edges are stored in an ordered list.
- Between two vertices u, v , only one edge may exist.

The rest of this section will discuss and explain this definition in more detail. If a path exists from node u to v , v is called a child node of u , and u is a parent node of v . Nodes without parents are called *root nodes*. The definition does not force a limit on the number of root nodes. Execution of a model graph starts in the root nodes. If a model graph does not have a root node, no code will be executed for the model graph. The empty graph and a model graph that consists only of a cycle are examples of graphs without root nodes. While these graphs can be considered as errors on the side of the programmer, this can be detected easily and the system may display a warning. If a model graph has more than one root node, the order of execution of the root nodes is not defined.

Edges are stored as lists of adjacent vertices in the vertices, thus edges are ordered. Model graphs are represented visually. In the visual representation, outgoing edges are rendered below a node, and incoming edges are rendered above the node. In order to visualize the order of the edges, the edges start from different points below a vertex, from left to right, in the same order they are stored in the node's *adjacency list*. Newly created edges are added to the end of the list of outgoing edges. The user may edit the order of the edges by deleting edges and adding them again. This must be repeated for any node that appears too early in the list of outgoing edges, because readding the nodes will place them at the end of the list. For many node types, child nodes are executed in the order they are stored in the vertices. However, some node types are required that deviate from this rule, in order to control program flow. These node types

are called *control operators*. All other node types execute their children in the order they are stored in the vertex's edge list. While a node executes child nodes, the node's state remains on the CPU stack. This allows the node to continue execution once its child nodes are executed, and results in a depth-first recursion scheme unless a control operator gets involved.

A node may have several parent nodes. Figure 49.c (page 83) gives an example of a node with several parents. A node with several parents may be executed several times even if the node is not placed in a loop.

In object-oriented programming, classes of objects with similar traits may have a common super class. The same mechanism has been implemented for node types in plab. There are abstract node types that may not be placed in a model graph, instead they define common attributes of their subclass nodes. `NodeBase` is the abstract base class of all other node types. The editor will not allow to create an instance of an abstract node type, and no code is associated with executing an abstract node type. `NodeBase` requires all subclass nodes to have a name. A *component* is used to create geometry such as spheres, cylinders or polygons. Textures for these primitives are created using *texture generators*. *Operators* are used to declare and update variables or for setting scene graph state. *Loops* and *branches* are controlled by *control operators*, which may deviate from the depth-first traversal scheme. Each node category has an *abstract node type* that subclasses `NodeBase`: `OperatorBase` is the node super class for operators and control operators, `TextureBase` for textures, and `ComponentBase` for nodes that generate geometry (components). Some of the node types are discussed in the following, but a complete and detailed description of the nodes can be found in appendices A, B, C, and D.

In a model graph, two nodes u and v may be part of a cycle. In that case, u and v are simultaneously parent and child of each other. This is not possible in a tree, because trees may not contain cycles, but most model graphs in this work are trees, therefore the designations parent and child node are used in this work. Executing nodes that form a cycle in the model graph results in a loop. These loops have to be aborted by `Comparators`, otherwise, infinite loops would result. Using `ForOperator` and `WhileOperator`, `for` and `while` loops may be specified in a more intuitive, convenient and secure manner. Therefore, cycles should be avoided in model graphs when possible.

The user programs model graphs by creating and connecting nodes and by editing the attributes of the model graph nodes. This means that the user does not have to remember keywords or the correct order of syntactic elements. Attributes are stored in correct order in the nodes, and the user does not need to bother with their order. The only exception to this are multi attributes, where the user controls how many copies of the attributes of the multi attribute can be filled in. However, the user has to use the right nodes, has to properly define the order of execution using edges between the nodes, and has to specify correct attribute values. Multi attributes are used in cases where similar information needs to be stored several times.

```

function ASSIGNMENTOPERATOR.EXECUTE
  MultiAttribute ma ← node.findAttribute("Variables")

  for i←0 → (ma.size/3)-1 do
    string varName ← ma.value[3*i]
    string varType ← ma.value[3*i+1]
    string varVal ← ma.value[3*i+2]
    variables.update(varName, varType, varVal)
  end for

  SceneGraphNode sgn ← executeChilds(node)
  updateHandles(node, sgn, sgn)
  return sgn
end function

```

Figure 17: Pseudocode for AssignmentOperator

4.1.2 Basic Node Types

While the language itself is defined above, its basic operators are introduced next. An `AssignmentOperator` is a node type that does not generate geometry. Instead, it alters the *scope* by creating and updating variables. `AssignmentOperator` has a multi attribute that contains the names and types of variables and formulas in infix notation. When an assignment is executed, the value v for each multi attribute is computed and stored. For a new variable, the type must be stated, and the variable is added to the scope. The possible variable types are presented in Section 4.3.2 and Section 4.3.3. The name of the variable must comply with the definition of `<variable name>`. If the variable already exists, its value is updated instead, and its type may be omitted. Querying a variable in a formula before the variable is created in the scope may result in an error. Figure 17 shows pseudocode for `AssignmentOperator`. Figure 20 shows pseudocode for executing child vertices. Section 4.3.4 will explain updating variables and handles (`updateHandles`) in detail.

It would be possible to allow functions that do not return values or whose return values can be ignored. In that case, the result type and variable name could remain empty for `AssignmentOperator`. However, in the visual environment of model graphs, it would be advisable to implement these functions as model graph nodes.

```

function COMPARATOR.EXECUTE
  MultiAttribute ma ← node.findAttribute("Comparisons")
  ModelGraphNode selectedChild ← NULL
  int i ← 0

  while selectedChild = NULL ∧ i < ma.size/3 ∧ i < childs.size do
    string op1 ← ma.value[3*i]
    string comp ← ma.value[3*i+1]
    string op2 ← ma.value[3*i+2]
    if test(op1, comp, op2) then
      selectedChild ← childs[i]
    else
      i += 1
    end if
  end while

  if selectedChild = NULL then
    if ma.size/3 < childs.size then
      // If no child node was selected above, and if there is one edge without
      // associated comparison, follow this edge as an "else" branch
      selectedChild ← childs[childs.size-1]
    end if
  end if

  SceneGraphNode sgn ← NULL
  if selectedChild ≠ NULL then
    sgn ← selectedChild.execute(node, i)
  end if
  updateHandles(node, sgn, sgn)
  return sgn
end function

```

Figure 18: Pseudocode for Comparator

A `Comparator` node contains a multi attribute that defines a list of comparisons. Each comparison consists of three attributes. The first attribute defines the first operand of the comparison, the second operand defines the compare operator, and the third attribute gives the second operand of the comparison. The operands can be *formulas* in infix notation. The following compare operators are allowed: < (less), <= (less or equal), = (equals), > (greater), >= (greater or equal), != (not equal, C style) and <> (not equal, Pascal style). If c is the number of the first comparison that is evaluated to `true`, then edge c from u 's ordered list of outgoing edges is followed to next vertex v to execute. This means that the first comparison that evaluates to true gives the number of the outgoing edge that is followed to the next node to execute. If there is one more edge than there are comparisons, then this additional edge is used when all comparisons failed, as an "else" branch. Figure 18 shows pseudocode for `Comparator`.

```

function POLYGONCOMPONENT.EXECUTE
  MultiAttribute ma ← node.getAttributeValue("Vertices")
  int vertexCount ← ma.size/5
  textureName ← node.getAttributeValue("Texture Name")
  Texture texture ← variables.findTexture(textureName)
  SceneGraphNode mesh ← new Mesh(texture)

  // Create vertices
  for i←0 → vertexCount-1 do
    double x ← variables.evaluate (ma.value[5*i])
    double y ← variables.evaluate (ma.value[5*i+1])
    double z ← variables.evaluate (ma.value[5*i+2])
    double u ← variables.evaluate (ma.value[5*i+3])
    double v ← variables.evaluate (ma.value[5*i+4])
    mesh.addVertex(x, y, z, u, v)
  end for

  // Connect vertices to a polygon
  Polygon poly ← mesh.addPolygon
  for i←0 → vertexCount-1 do
    poly.addVertex(i)
  end for
  poly.computeFaceNormal

  updateHandles(node, mesh, mesh)
  executeChilds(node, mesh)
  return mesh
end function

```

Figure 19: Pseudocode for PolygonComponent

PolygonComponent adds a single polygon to the scene graph. The polygon is defined by an arbitrary number of vertices whose 3D coordinates are computed from *formulas* in infix notation. These formulas are stored in a multi attribute. Figure 19 shows pseudocode for PolygonComponent.

4.1.3 Scene Graph Management

plab supports operations on a scene graph hierarchy. Operations including transforming objects, lighting, and instancing, are covered in this section. But before these operations are explained in detail, it is explained how plab manages the scene graph, because the scene graph hierarchy defines which parts of the geometry are affected by these operations. The scene graph hierarchy is generated implicitly in plab, so normally the user does not have to actively control the creation of the scene graph. An operator that creates geometry stores the geometry in the root of a new scene graph and the geometry created by child nodes of the model graph node is stored as child nodes of the root of the model graph. Likewise, the scene graph returned by an operator may be added to the scene graphs created by its parent nodes. Nodes that do not create geometry create scene graphs using the following rules:

- If no child created a scene graph, these nodes will not create a scene graph either.
- If one child created a scene graph, this scene graph will be returned for this node as well.
- Otherwise, the child nodes created at least two scene graphs. A new scene graph is created whose root has the scene graphs created by the child nodes as its own child nodes, and this scene graph root is returned as the result of executing this node.

This is summarized in Figure 20, the variant used by node types that generate geometry can be found in Figure 21. If a scene graph was created by a node or one of its child nodes, a handle `<node name>.handle` is stored on the scope. It can be used to refer to this part of the scene graph, e.g. for instancing the part several times in the scene using `InstanceOperator`. *Instancing* refers to an operation that renders an object several times even when only a single copy of the object is stored, by referencing the object several times from the scene graph. It has an attribute `Mesh handle` that allows the user to state a previously defined handle variable. When executed, this will insert an additional reference to the geometry in the scene graph.

As all operator attributes can be looked up in the appendices, the following text concentrates on the meaning of the attributes rather than completely listing them. Without further provisions, instanced geometry would be created in the exact same position as the original position. In computer graphics, this is usually solved using transform matrices, and plab uses the same approach. In plab, transform matrices are specified using nodes. The most general transform node is `TransformOperator`. Its 16 operands define a 4×4 matrix. While operations such as shearing require this node type, usually specialized transform operators are easier to use. For scaling a vector or an object along the x, y, z axes, `ScaleOperator` can be used. Its operands `xscale`, `yscale`, `zscale` scale objects along the different axes. `TranslationOperator` is used for *translations*. Its operands `x`, `y`, `z` move child objects along the named axes by the specified distance. `RotateOperator` is used for *rotations*. Its operands state the axis around which the rotation is performed as well as the angle in degrees. The transform operators store transformations as matrices in the scene graph. These matrices affect all child nodes in the scene graph, and the transformations are disabled when rendering the node is complete.

While it would be possible to transform the geometry during its creation, this method is unsuitable for use with *instancing*, which can be achieved using `InstanceOperator`. `InstanceOperator` takes a single handle that defines a handle to geometry. The operator starts a new scene graph, it adds an additional reference to the specified object to the model graph, and adds geometry created by child vertices to the scene graph created by the node. This scene graph can then be integrated into scene graphs created by parent nodes. If a user would state the handle for geometry created by a parent model graph node, this would lead to a cycle in the model graph, and the model graph could not be rendered, so the user should be cautious when using `InstanceOperator`.

```

function EXECUTECHILDS(ModelGraphNode node)
  int childSGs ← 0
  SceneGraphNode result ← NULL
  int i ← 0

  while i < node.chlds.size ∧ result = NULL do
    ModelGraphNode child ← node.chlds[i]
    SceneGraphNode sgn ← child.execute
    if sgn ≠ NULL then
      result ← sgn
      childSGs ← 1
    end if
    ++i
  end while

  while i < node.chlds.size ∧ childSGs = 1 do
    ModelGraphNode child ← node.chlds[i]
    SceneGraphNode sgn ← child.execute
    if sgn ≠ NULL then
      newSG ← new SceneGraphNode
      newSG.addChild(result)
      newSG.addChild(sgn)
      result ← newSG
      childSGs ← 2
    end if
    ++i
  end while

  while i < node.chlds.size do
    ModelGraphNode child ← node.chlds[i]
    SceneGraphNode sgn ← child.execute
    if sgn ≠ NULL then
      result.add(sgn)
      childSGs ← childSGs + 1
    end if
    ++i
  end while
  return result
end function

```

Figure 20: Pseudocode for visiting model graph child nodes

```

function EXECUTECHILDS(ModelGraphNode node, SceneGraphNode sgn)
  for i ← 0 → node.chlds.size-1 do
    ModelGraphNode child ← node.chlds[i]
    if child ≠ NULL then
      sgn.addChild(child)
    end if
  end for
  return sgn
end function

```

Figure 21: Pseudocode for visiting model graph child nodes, for nodes that create geometry

```

function MODELGRAPHNODE.RENDER
  ActivateState() // Activate state, implemented by subclasses
  renderGeometry() // Render geometry associated with this node

  for i ← 0 → chlds.size-1 do
    SceneGraphNode child ← chlds[i]
    if child ≠ NULL then
      child.render
    end if
  end for

  DeactivateState() // implemented by subclasses
end function

```

Figure 22: Pseudocode for rendering scene graph nodes

plab supports rendering with light sources, but rendering with light sources requires surface normals and lighting parameters. `LightSourceOperator` is used to create light sources. The user specifies *formulas* for the position, color, brightness, and attenuation, and these formulas can be evaluated for several light sources during a single execution of a `LightSourceOperator`. Material properties are set in the texture nodes, including constants for diffuse and specular reflection as well as the specular exponent. Several operators allow the user to generate textures, and these operators will be described later. Operators that create geometry always have to specify a texture, but proper lighting also requires surface normals. Most operators that create geometry calculate surface normals automatically, but if the user requires more control over the process, normals can be stated using `AddPolygonWithNormalsComponent`, which will also be described later. Like the transform operators, light sources are deactivated when rendering the child nodes of `LightSourceOperator` is complete.

Other operators multiply objects, either along a vector (`MultiplyAlongVectorOperator`) or on a circle (`MultiplyOnCircleOperator`). These operators have a flag that allows the user to decide whether geometry from child nodes should be actually multiplied or only instanced. If the geometry generated by the child is always the same or very similar, it should be instanced to save memory. If the child node generates different geometry for every call, the objects have to be multiplied instead of being instanced.

After execution, the resulting scene graph can be rendered. See Figure 22 for the rendering pseudocode.

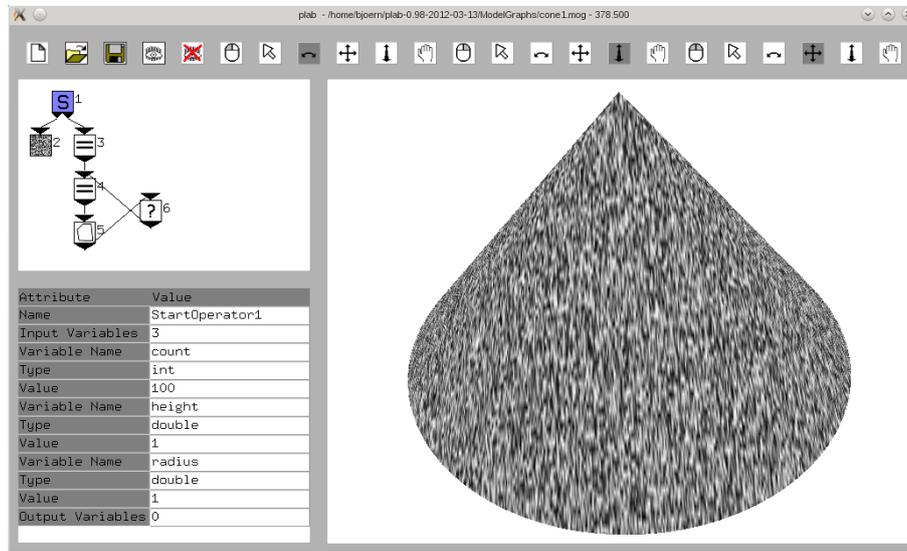


Figure 23: Model that produces a cone (top left) and result (right)

4.2 Introductory Example

Figure 23 shows a screenshot of plab. The screen is divided into three main parts: The top left shows the current model graph. The currently selected node is rendered blue and its attributes are displayed below that in the attribute editor. In the *attribute editor*, attribute keys are printed on the left, and the user may edit the attribute values on the right. On the right of the screenshot, the geometry resulting from executing the model graph is shown, a textured cone. In the following, we discuss the model graph that produces this geometry, along with several improvements.

In this case, node 1 is selected, a `StartOperator` named `StartOperator1`. `StartOperator` will be discussed in more detail later. For now, it suffices to know that a `StartOperator` specifies the input and output parameters of the model graph. For all node types, the first attribute always specifies the name of the operator. Because node 1 has no parent node, it is a root node, so execution of the model graph starts in this node. `StartOperator1` has three input parameters and no output parameters. The input parameters are stored in variables. `count` specifies the number of triangles that the cone will have. Furthermore, the cone's `height` and `radius` are stored in variables of type `double`.

Execution continues with node 2, depicted in Figure 24.a. It is a `RoughTexture-Operator` named `text`. It produces a texture of dimensions 512×128 . It specifies minimum and maximum colors, and texel colors are randomly selected between these values. The operator furthermore specifies material values that only matter when a light source is used. Since node 2 does not have a child node, execution continues with the next child node of node 1, node 3. It is an `AssignmentOperator` that initializes the counting variable `i` with zero, see Figure 24.b. Next, node 4 is executed, another `AssignmentOperator`. It is depicted in Figure 24.c and computes coordinates

Attribute	Value
Name	text
Width	512
Height	128
minRed	0
minGreen	0
minBlue	0
maxRed	1
maxGreen	1
maxBlue	1
Diffuse Reflecti	
Specular Reflect	
Specular Shinine	

a: node 2

Attribute	Value
Name	AssignmentOperator1
Variables	1
Variable Name	i
Type	int
Value	0

b: node 3

Attribute	Value
Name	AssignmentOperator2
Variables	5
Variable Name	x1
Type	double
Value	$\sin(i/\text{count}*360)$
Variable Name	y1
Type	double
Value	$\cos(i/\text{count}*360)$
Variable Name	x2
Type	double
Value	$\sin((i+1)/\text{count}*360)$
Variable Name	y2
Type	double
Value	$\cos((i+1)/\text{count}*360)$
Variable Name	i
Type	
Value	i+1

c: node 4

Attribute	Value
Name	PolygonComponent1
Texture Name	text.texture
Vertices	3
x	$x1*\text{radius}$
y	$y1*\text{radius}$
z	0
u	$(x1+1)/2$
v	$(y1+1)/2$
x	$x2*\text{radius}$
y	$y2*\text{radius}$
z	0
u	$(x2+1)/2$
v	$(y2+1)/2$
x	0
y	0
z	height
u	0.5
v	0.5

d: node 5

Attribute	Value
Name	Comparator1
Comparisons	1
Operand 1	i
Test	<
Operand 2	count

e: node 6

Figure 24: Further nodes in the model graph, cf. Figure 23

for boundary vertices of the cone and stores them in variables $x1$, $y1$, $x2$, $y2$. Also, the counting variable is increased. Node 5 is a `PolygonComponent` that uses these variables to output a polygon. Figure 24.d shows the arguments required for the polygon. `PolygonComponent` computes a normal for the polygon automatically, but requires u , v coordinates for selecting parts of the texture. Figure 23 shows that our model graph contains a cycle formed by nodes 4 to 6. Node 6 is a `Comparator` that calls node 4 as long as $i < \text{count}$, see Figure 24.e. Once i satisfies $i = \text{count}$, execution of the model graph is complete, and the results are displayed automatically, as demonstrated by the right half of Figure 23.

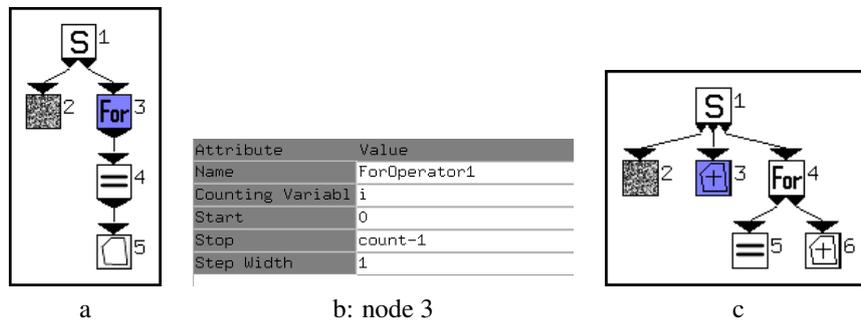


Figure 25: Variants of the model graph that creates a cone. a: Variant using ForOperator, b: node 3 in the model graph shown in a, c: Variant using AddPolygonComponent

Attribute	Value
Name	AddPolygonComponent1
Texture Name	text.texture
Container Handle	0
Index Base	0
New Points	2
x	0
y	0
z	height
u	0.5
v	0.5
Name	center
x	0
y	radius
z	0
u	0.5
v	1
Name	newv
New Polygon	0

a: node 3

Attribute	Value
Name	AddPolygonComponent2
Texture Name	text.texture
Container Handle	AddPolygonComponent1.mesh
Index Base	0
New Points	1
x	x
y	y
z	0
u	(x+1)/2
v	(y+1)/2
Name	newv
New Polygon	3
Index	0
Index	i+1
Index	i+2

b: node 6

Figure 26: AddPolygonComponent operators

While this model graph works as supposed, it can be simplified by introducing a ForOperator, see Figure 25.a. The attributes needed for this operator are shown in Figure 25.b. It automates counting variable i from $Start=0$ to $Stop=count-1=99$ in increments of $Step\ Width=1$, and calls its child nodes for any value generated in this manner. It replaces nodes 3 and 6 with a single node, and incrementing the counting variable i in node 4 is also handled by the ForOperator. This eliminates the cycle from the model graph, and the resulting model graph is easier to read.

However, the model graph can be optimized further. The center vertex of the cone is replicated for every polygon. Furthermore, going back to the assignments shown in Figure 24.c and Figure 24.d, we note that every boundary vertex is computed twice. It would be more efficient to compute vertices only once, add them to the mesh, and then recycle some of the vertices for the next polygon. In fact, after the first polygon, we only need to add a single vertex to the mesh for every new polygon. Figure 25.c shows the necessary adjustments to the model graph. The additions are two AddPolygonComponent nodes depicted in Figure 26. AddPolygonComponent1 (node 3) is shown in Figure 26.a. It creates a new mesh consisting of two vertices, the cone's center and first border vertices. AddPolygonComponent2 (node

6), shown in Figure 26.b, references the geometry generated by `AddPolygonComponent1` by stating `AddPolygonComponent1.mesh` as a container handle. This means that geometry is not added to a new mesh or scene graph, instead the geometry is added to the existing mesh created by `AddPolygonComponent1` (node 3). `AddPolygonComponent2` (node 6) adds an additional vertex and outputs a polygon every time it is executed. The polygon is specified by vertex indices. Compared with the previous versions of this model graph, for large polygon counts, this reduces the number of vertices almost to a third by reusing vertices instead of duplicating them.

```

function FOROPERATOR.EXECUTE
    string varName ← node.getAttributeValue("Counting Variable")
    double startVal ← variables.evaluateAttribute(node, "Start")
    double stepWidth ← variables.evaluateAttribute(node, "Step Width")
    double stopVal ← variables.evaluateAttribute(node, "Stop")
    Variable var ← variables.findVariable(varName, startVal)
    SceneGraphNode sgn ← NULL // created when needed later

    if var = NULL then
        var ← new DoubleVariable(varName)
        variables.add(var)
    end if

    while (stepWidth > 0 ∧ var ≤ stopVal) ∨ (stepWidth < 0 ∧ var ≥ stopVal) do
        SceneGraph newSG ← executeChilds(node)
        if newSG ≠ NULL then
            if sgn = NULL then
                sgn ← new SceneGraph
            end if
            sgn.addChild(newSG)
        end if
        var += stepWidth
    end while

    updateHandles(node, sgn, sgn)
    return sgn
end function

```

Figure 27: Pseudocode for ForOperator

4.3 Additional Language Features

The preceding examples used operators that still need to be discussed in detail.

4.3.1 Control Operators

ForOperator is a convenience operator that executes its child nodes in a loop without requiring a cycle in the model graph. The user defines which variable is used for counting, and may access this variable normally in *formulas*. Inside the `for` loop, the user should not use the counting variable for other purposes. The user also defines start, stop, and step width for counting the variable. Pseudocode for ForOperator is listed in Figure 27.

Another important control operator is WhileOperator. The user may state a condition, and child nodes are executed as long as the stated condition holds. The operator works similar to ForOperator, therefore no pseudocode is necessary to understand how it works internally.

4.3.2 Variable Types

`AssignmentOperator` and `StartOperator` are used to initialize *variables*, *records* and *arrays*. The user chooses the number of assignments, and defines variable names, types and values for each assignment. The following basic types may be used:

- `int` stores a 64-bit integer,
- `double` is an IEEE-754 double precision floating-point number,
- `string` stores a linear sequence of characters,
- `int[n]`, `double[n]`, `string[n]` are arrays based on these types. The size of an array can be computed from a formula when declaring the array, and can later be queried using the function `size(arr)` for an array `arr`.

Variable names in `plab` are case sensitive. Remember that a variable is only visible in the model graph where it was defined, so a model graph defines a local *scope* for all its variables. The variables are stored as objects on the *heap*, and are only deleted when execution of the model graph is complete. Further types for variables are discussed in the following subsections. Appendix E summarizes the predefined functions that can be used in formulas in `plab`. The `random` function is of particular use in procedural modeling. It produces a random value v of type `double`, $0 \leq v < 1$.

For an array, the user may state an index in square brackets to access a single array entry. If the index and square brackets are omitted for an array, the entire array is updated. In that case, the variable `index` can be used to query which entry of the array is currently updated. For example, assigning `index` to the entire array will store each entry's index in each entry.

In all operator attributes that accept strings, string variables and other variables may be evaluated. In this case, the names of evaluated variables must be surrounded by `$` to separate them from plain text characters. For example, when saving a file, the string `myFile$round$.xml` is expanded in a loop to file names such as `myFile1.xml`, `myFile2.xml`, `myFile3.xml`, if `round` is the counting variable for the loop.

4.3.3 Records

Records are used to combine several variables of arbitrary types into a *data structure*. Instead of using the variables `red`, `green` and `blue`, it is possible to assign `color.red`, `color.green` and `color.blue` to a record `color` with the members `red`, `green` and `blue`. This stresses that the members belong together, and allows to store a reference to the group of variables. Used wisely, records increase the readability of the code produced. Grouping data into records can also reduce the number of parameters when calling model graphs, which is explained in Section 4.3.5.

Records can be defined explicitly by stating their name with the type `record`. A record stores only its members, but does not store a value itself, so the assignment part of a record declaration is ignored. Records can also be created implicitly: When the

user assigns a value to `<RecordName>.<MemberName>`, `plab` creates a record `<RecordName>`, if the record does not exist, and a member `<MemberName>`. Further members may be added at any time, but reading from members that do not exist results in an error. If a variable of non-record type already exists, it is not possible to add members to it. Assigning a value to `car.tyre.kind` automatically creates records for `car` and `car.tyre`.

Records can be placed in arrays. For example, a variable `cars` with type `record[3]` would be an array with three elements containing records. Contrary to strictly typed languages, `plab` does not enforce that the members of the records in the array `cars` have the same names and types. As a result, `car[0]` may have totally different members than `car[1]`. Even if `car[0]` has members with the same names as `car[1]`, `plab` does not enforce that these members have the same types.

If a model graph needs to work repeatedly with the record `car.tyres`, it is possible to assign `car.tyres` to a record reference named `tyre`. The variable type for `tyre` is `recordRef`. Each assignment to `tyre` also alters `car.tyres` and vice versa, until a new record is assigned to `tyre`.

An example where `recordRef` is of particular use is storing and processing recursive data in records. For example, it would be possible to store a tree using a combination of records and arrays. Each tree node would be reflected with a record, which would contain an array `childNodes`. In this case, it would be possible to store and process the data recursively using a variable `recordRef node` to point to the currently processed node.

4.3.4 Updating variables

While updating a variable, a number of cases must be treated properly. First, the correct variable needs to be found or created. In doing this, the implementation needs to treat arrays and records correctly. Once the correct variable or member was located or created, its value is updated. When evaluating the formula, the result type is given by the type of the variable or record member receiving the value. All values are cast to this type automatically. However, if parts of a formula need to be evaluated to a different type, that part of the formula can be assigned to a variable with the required type.

Some node types require references to geometry created by other nodes, e.g. `AddPolygonComponent`, `InstanceOperator`. In order to support this, every node creates or updates two handle variables after its execution is complete. These handles often store the same value, but not always. The handles are set using the function `updateHandles`, which was already used in the pseudocode of several functions. Figure 28 shows pseudocode for this operation. First, a record variable `<NodeName>` is declared which will receive the handles as members. `<NodeName>.handle` is a handle to the scene graph node created by the node. `<NodeName>.mesh` is a handle to the scene graph node that contains the geometry added by executing the current model graph node. For certain node types, `<NodeName>.mesh` may differ from `<NodeName>.handle`. `<NodeName>.texture` is a handle to the texture generated by the current node, or `NULL` if no texture was generated.

```

function MODELGRAPHNODE.UPDATEHANDLES(String nodeName, Scene-
GraphNode sg, SceneGraphNode mesh, Texture texture)
    Record rec = variable.findRecord(nodeName)
    if rec == NULL then
        rec = new record (nodeName)
        variables.add(rec)
    end if
    rec.setMember("handle", "handle", sg)
    rec.setMember("mesh", "handle", mesh)
    rec.setMember("texture", "handle", texture)
end function

```

Figure 28: Pseudocode for updating handles

4.3.5 Calling Other Model Graphs and Recursion

`CallOperator` is used to run model graphs as a *subroutine* of a model graph. If the model graph calls itself, this results in a *recursion*. `StartOperator` defines the *parameters* for a model graph. If the model graph is run directly, instead of being called from another model graph, these parameters are treated as variables, with a name, type and value. However, if a user creates a `CallOperator` in a model graph, and types in the name of a model graph with a `StartOperator`, `plab` opens that model graph to extract its parameters. The parameters are then offered as attributes of the `CallOperator`, including default parameters defined in the model graph's `StartOperator`. Of course, the values for the call may be edited, but the default values often reduce typing.

`StartOperator` has separate lists for input and output parameters. When a `CallOperator` is executed, *input parameters* are copied into new variables, whereas *output parameters* are stored as references to variables of the calling model graph. Output parameters are passed as references to the variables in the calling model graph, allowing to pass changed data to and from the calling model graph. If an array is passed as a parameter, the type definition in `StartOperator` states its minimum size. If the array equals or exceeds this size, it is passed in full size, otherwise, the default value is used to inflate it to the required size.

`CallOperator` and `string` can be used to implement *dynamic binding* in model graphs. Dynamic binding is a concept from object-oriented programming that allows to decide at runtime which implementation of a function is used, usually by marking functions as `virtual`. In model graphs, this can be implemented by storing the file name of a model graph in a `string`, and evaluating the `string` variable in the `CallOperator` for picking the model graph to execute. This allows to change the model graph used to generate parts of the scene at runtime, by passing different model graph file names using `StartOperator`. For example, this can be used in a model graph that defines the façade of a building in loops, by passing the model graph responsible for generating the elements in the façade as a parameter. Alternatively, if `string[] modelgraphs` is used to store a list of model graphs, one of the model graphs can be selected using `$modelgraphs[i]$`.

```

function CALLOPERATOR.EXECUTE
  SceneFactory otherSF ← new SceneFactory (new TextureManager)
  SceneGraphNode sgn ← NULL
  String filename
  variables.evaluateAttributeToString("FileName", filename)

  if otherSF.loadModelGraph(filename) then
    MultiAttribute inputVars ← node.findAttribute("Input Variables")
    for i=0 → (inputVars.size/3)-1 do
      String varName ← inputVars[3*i]
      String varType ← inputVars[3*i+1]
      String varVal ← inputVars[3*i+2]
      plabVariable var ← otherSF.addVariable(varName, varType)
      variables.evaluate(varVal, var)
    end for

    MultiAttribute outputVars ← node.findAttribute("Output Variables")
    for i=0 → (outputVars.size/3)-1 do
      String varName ← outputVars[3*i]
      plabVariable var ← findVariable(varName)
      otherSF.addCloneVariable(var)
    end for

    if outputVars.size == 0 ∧ childs.size == 0 ∧ MultiThreadingEnabled then
      In separate thread do otherSF.visitAllNodes()
    else
      otherSF.visitAllNodes()
    end if

    sgn ← executeChilds(node, otherSF.scene)
  end if

  delete otherSF
  updateHandles (this, sgn, sgn)
  return sgn
end function

```

Figure 29: Pseudocode for CallOperator

```

function STARTOPERATOR.CREATEVARIABLES(String attrName)
  // Find attribute
  plabMultiAttribute multifield ← FindMultiAttribute (attrName)
  if multifield == NULL then
    return
  end if

  // Store variables
  for i = 0 → multifield.getValueCount()-1 do
    String varName ← multifield.getValueStr (i)
    String varType ← multifield.getValueStr (i+1)
    String varVal ← multifield.getValueStr (i+2)
    i += 3

    // Create variable if it is missing
    plabVariable var ← variables.getVariable(varName)

    // If the variable exists already, it was created and initialized
    // by a calling model graph. Otherwise, create and initialize it
    if var == NULL then
      var ← variables.createVariable (varName, varType, varVal)
    end if
  end for
end function

function STARTOPERATOR.EXECUTE
  // Create variables for input and output parameters if they are missing
  // They may be missing if this model graph is run directly - as
  // opposed to being called from another model graph - or if the other
  // model graph has an incomplete list of parameters for this model graph
  createVariables("Input Variables")
  createVariables("Output Variables")

  // Visit successors
  SceneGraphNode sgn = executeChilds (node)
  factory.setHandle (node, "handle", ro)
  return sgn
end function

```

Figure 30: Pseudocode for CallOperator

In both cases, the parameter lists of the model graphs used for dynamic binding should be similar. If a `CallOperator` lacks a parameter listed by a `StartOperator` in the model graph called by the `CallOperator`, *SceneFactory* will create a member variable with the given name and type using the default value listed in the `StartOperator`. On the other hand, if the `CallOperator` lists a parameter that is not used in the called model graph, or even missing in the `StartOperator`, then that parameter will not have an effect. If the model graph expects a variable with the same name but different type, this may lead to errors. This may happen if the parameter list of the called model graph is changed after the calling model graph was created. Figures 29 and 30 summarize the instructions that compose `CallOperator` and `StartOperator`.

4.3.6 Parallelization

In procedural modeling, the bottleneck for generating geometry is often the CPU. Therefore, *plab* was optimized to take advantage of multicore CPUs. A model graph called from a `CallOperator` can be executed in parallel to its caller provided that there are no output parameters and the `CallOperator` does not have any outgoing edges. Child nodes could add geometry to the parent node's scene graph node, leading to *race conditions*. Output variables may be changed by the current model graph or the called model graph at any time, which would also lead to race conditions. The only synchronization required is at the end of model graph execution, where rendering has to wait until all model graph *threads* have been completed.

Later works use the same approach [32]. Lipp et al. demonstrate an algorithm for rewriting L-strings that takes advantage of massively parallel graphics hardware [87]. In any iteration, parts of the L-strings are assigned to threads. Each thread counts the added total cost of replacements in its L-string segment. These counts are added up in a single thread to find the base addresses for the replaced strings, then the new string can be built by several threads that copy parts of the new string in parallel. Push/pop operations, which use [and] in L-systems, are essentially subroutines that do not return values. These can be executed in parallel to the caller.

Unfortunately, handles and records introduce hurdles to the approach of automatic detection when parallelization is possible: Passing handles or records to other model graphs might allow different threads to simultaneously manipulate the same data structure. This could lead to corrupted data structures, errors and program crashes. Handles are necessary to support operations such as `AddPolygonComponent` and `InstanceOperator`, and allow to store references to geometry. There are several ways of preventing this:

- Use thread-safe containers, e.g. Intel's Threading Building Blocks,
- Forbid to pass handles, records and record references (*recordRef*) as model graph parameters,
- Prevent parallel execution of model graphs in the configuration file.

```

function ROUGHTEXTUREGENERATOR.EXECUTE
    int width ← variables.evaluateAttribute(node, "Width")
    int height ← variables.evaluateAttribute(node, "Height")
    double minRed ← variables.evaluateAttribute(node, "minRed")
    double minGreen ← variables.evaluateAttribute(node, "minGreen")
    double minBlue ← variables.evaluateAttribute(node, "minBlue")
    double maxRed ← variables.evaluateAttribute(node, "maxRed")
    double maxGreen ← variables.evaluateAttribute(node, "maxGreen")
    double maxBlue ← variables.evaluateAttribute(node, "maxBlue")
    Texture texture = new Texture(width, height, GL_RGB)

    for int y = 0 → height-1 do
        for int x = 0 → width-1 do
            double r = minRed + (maxRed-minRed)*random
            double g = minGreen + (maxGreen-minGreen)*random
            double b = minBlue + (maxBlue-minBlue)*random
            texture.setTexel(x, y, r, g, b)
        end for
    end for

    updateHandles(nodeName, NULL, NULL, texture)
    SceneGraphNode sgn ← executeChilds(node)
    updateHandles(nodeName, sgn, sgn, texture)
    return sgn
end function

```

Figure 31: Pseudocode for RoughTextureGenerator

4.3.7 Texture Generators

In plab, *textures* can be loaded from a file or created procedurally. A simple way to create a texture with random noise has been implemented as `RoughTextureGenerator`. The user states the width and height of the texture to create, and minimum and maximum color values c_1 , c_2 . It assigns $X \cdot c_1 + (1 - X) \cdot c_2$ to every texel, with different random values $0 \leq X < 1$ for every texel and colors c_1 , c_2 . Figure 31 shows pseudocode for this operator. `ProceduralTextureGenerator` computes the color for every texel from a formula. See Figure 32 for pseudocode. `TransparentTextureGenerator` works the same way, but adds an *alpha channel* for simulating *transparency*. Both `ProceduralTextureGenerator` and `TransparentTextureGenerator` may read texels from other textures using the functions `getRed`, `getGreen`, `getBlue`, `getAlpha`. `TextureFile` loads the texture from a BMP file. Details on these operators can be found in Appendix C.

```

function PROCEDURALTEXTUREGENERATOR.EXECUTE
    int width ← variables.evaluateAttribute(node, "Width")
    int height ← variables.evaluateAttribute(node, "Height")
    PreparedFunction redTerm ← variables.parseFunction("Red")
    PreparedFunction greenTerm ← variables.parseFunction("Green")
    PreparedFunction blueTerm ← variables.parseFunction("Blue")
    vector<PreparedFunction> assignments
    assignments ← variables.parseMultiattribute("Variables")
    varXName ← getAttributeValue("xindex")
    varYName ← getAttributeValue("yindex")
    Variable varX ← variables.findVariable(varXName)
    Variable varY ← variables.findVariable(varYName)
    Texture texture = new Texture(width, height, GL.RGB)

    for varY = 0 → height-1 do
        for varX = 0 → width-1 do
            assignments.execute
            varRed.evaluate (redTerm);
            varGreen.evaluate (greenTerm);
            varBlue.evaluate (blueTerm);
            texture.setTexel(varX, varY, varRed, varGreen, varBlue)
        end for
    end for

    updateHandles(nodeName, NULL, NULL, texture)
    SceneGraphNode sgn ← executeChilds(node)
    updateHandles(nodeName, sgn, sgn, texture)
    return sgn
end function

```

Figure 32: Pseudocode for `ProceduralTextureGenerator`

Transparent polygons must be sorted by depth, but non-transparent polygons are rendered first. The fewer transparent polygons there are in the scene, the smaller the performance penalty. An efficient technique to render transparent polygons is to sort them into a BSP tree once, and rendering the polygons by traversing the BSP tree in back to front order [25].

4.3.8 Geometry Creation

Operators that *create geometry* are called *components*, and there are several component types in `plab`. The most simple one is `PolygonComponent`, which was described in Section 4.1.2. `QuadStripComponent` is used to create a polygon strip from two functions. `AreaComponent` is used to create geometry from a two-dimensional function. The user may state the number of polygons to create, and vertices are shared by up to four polygons. Pseudocode for `AreaComponent` is shown in Figure 33.

`ConeComponent`, `CubeComponent`, `CylinderComponent` and `SphereComponent` are specialized components to create cones, cuboids, cylinders and spheres. `StemComponent` is useful to produce tubelike surfaces such as branches, stems, pillars or similar organic shapes that occur in nature and architecture.

`AddPolygonComponent` was mentioned in Section 4.2. It allows adding polygons to an existing mesh created by any component or it creates a new mesh. The user may create new vertices and connect any old and new vertices to new polygons to create meshes of arbitrary topology. It works as follows: Its attributes may state several vertices $v_1 = (x_1, y_1, z_1, u_1, v_1), \dots, v_n = (x_n, y_n, z_n, u_n, v_n)$. Vertices are assigned to polygons by stating vertex indices (i_1, i_2, \dots, i_m) in a second multi attribute, and the indices may include vertices that were added to the mesh by other operators, during a previous execution of this model graph node, or during the current execution of the node. Then, the faceset $F = (V, I)$ becomes $F = (V \cup \{v_1, \dots, v_n\}, I \cup \{(i_1, i_2, \dots, i_m)\})$, where n is the number of new vertices and m is the number of vertices in the new polygon. A polygon is only added if $m \geq 3$. Pseudocode for `AddPolygonComponent` is listed in Figure 34.

After execution, every model graph node is represented with a record in the variable scope. This record has the same name as the node and has at least two members, `mesh` and `handle`. For `AddPolygonComponent`, these handles may have different values. This is because `<node name>.mesh` refers to the mesh that contains the newly created polygon, whereas `<node name>.handle` refers to the scene graph created by executing the node and its childs. The handle values are the same if the new polygon is added to a new mesh. If the new polygon is added to an existing mesh, `<node name>.mesh` refers to that mesh.

For `AddPolygonComponent`, the user does not have to state normals, because normals are chosen automatically orthogonal to the created polygon. `AddPolygon-WithNormalsComponent` allows the user to state or compute normals for every vertex in every face from user-defined *formulas*.

```

function AREACOMPONENT.EXECUTE
    sv1 ← variables.evaluateAttribute(Node, "Var. 1 start")
    ev1 ← variables.evaluateAttribute(Node, "Var. 1 end")
    sv2 ← variables.evaluateAttribute(Node, "Var. 2 start")
    ev2 ← variables.evaluateAttribute(Node, "Var. 2 end")
    PreparedFunction xFunc ← variables.parseFunction("x")
    PreparedFunction yFunc ← variables.parseFunction("y")
    PreparedFunction zFunc ← variables.parseFunction("z")
    string textureName ← node.getAttributeValue("Texture Name")
    Texture texture ← variables.findTexture(textureName)
    SceneGraphNode mesh ← new Mesh(texture)
    int pc1 ← ev1-sv1+1
    int pc2 ← ev2-sv2+1
    v1Name ← getAttributeValue("Count var. 1")
    v2Name ← getAttributeValue("Count var. 2")
    x ← variables.find(v1Name)
    y ← variables.find(v2Name)

    // Add vertices to mesh
    for y ← sv2 → ev2 do
        for x ← sv1 → ev1 do
            Point3D p
            p.x = xFunc.evaluate
            p.y = yFunc.evaluate
            p.z = zFunc.evaluate
            double u = (x-sv1) / pc1
            double v = (y-sv2) / pc2
            mesh.addVertex (p, u, v)
        end for
    end for

    // Connect vertices to a surface
    for int i = 0 → pc1-2 do
        for int j = 0 → pc2-2 do
            size_t base ← j*pc1+i
            mesh.addTriangle (base+1, base+pc1, base);
            mesh.addTriangle (base+1, base+pc1+1, base+pc1);
        end for
    end for

    calcNormals(mesh)
    updateHandles(node, mesh, mesh)
    executeChilds(node, mesh)
    return mesh
end function

```

Figure 33: Pseudocode for AreaComponent

```

function ADDPOLYGONCOMPONENT.EXECUTE
  string textureName ← node.getAttributeValue("Texture Name")
  Texture texture ← variables.findTexture(textureName)
  string meshName ← node.getAttributeValue("Container Handle")
  SceneGraphNode mesh = variables.findSceneGraphNode(meshName)
  SceneGraphNode childRoot = NULL
  if mesh == NULL then
    mesh = new SceneGraphNode(texture)
    childRoot = mesh
  end if

  // Add vertices to mesh
  MultiAttribute vertexAttrs = node.findAttribute("New Points")
  int vertexCount = vertexAttrs.size/6
  for i=0 → vertexCount-1 do
    double x ← variables.evaluate (vertexAttrs.value[6*i])
    double y ← variables.evaluate (vertexAttrs.value[6*i+1])
    double z ← variables.evaluate (vertexAttrs.value[6*i+2])
    double u ← variables.evaluate (vertexAttrs.value[6*i+3])
    double v ← variables.evaluate (vertexAttrs.value[6*i+4])
    int index ← sg.addVertex(x, y, z, u, v)
    string indexName ← vertexAttrs.value[6*i+5]
    if indexName.length > 0 then
      variables.update(indexName, index)
    end if
    mesh.addVertex(x, y, z, u, v)
  end for

  // Add a polygon to mesh
  MultiAttribute polygonIndices = node.findAttribute("New Polygon")
  if polygonIndices.size > 0 then
    Polygon poly = sg.addPolygon
    for i=0 → polygonAttrs.size-1 do
      int index ← variables.evaluate (polygonIndices.value[i])
      poly.addVertex(index)
    end for
    poly.computeNormals
  end if

  if childRoot == NULL then
    childRoot = new SceneGraphNode(texture)
  end if
  updateHandles(node, childRoot, mesh)
  executeChilds(node, childRoot)
  return sg
end function

```

Figure 34: Pseudocode for AddPolygonComponent

4.3.9 Animation

`RenderOperator` can be used to create animations in `plab`. Geometry created by its child nodes is deleted immediately after being rendered. In order to reduce CPU load while the model graph runs, it is possible to use `WaitOperator`. Optionally, `RenderOperator` can store a screenshot for every frame. The notation

`< FileName > $frame$.bmp`

is useful to store every frame to a different file.

4.4 User Interface

Every *node* in a model graph has a number of attributes that the user may edit. They are displayed when the user clicks the node in the model graph or the geometry created by them in the viewport. Nodes are connected by dragging the target node onto the source node. Dragging the target node onto the source node again disconnects the nodes. The user may highlight several nodes at once by pressing the shift key. This is useful when several nodes need to be moved or deleted.

The model graph is not restarted after changes in the model graph automatically, as often several changes may be necessary to reach a displayable model. An exception to this rule is described in Section 4.4.2. Also, Menz et al. demonstrate an interactive editor that displays variants for different parameters while the user edits the model [96]. For executing a node, the scene factory needs to parse the attributes, create geometry, and update variables and the scene graph.

Section F gives some general hints for working with model graphs.

4.4.1 Object Picking

In this work, the *mouse ray* is defined as the line segment that starts at the mouse position on the near clipping plane, that ends or passes through the mouse position on the far clipping plane. If a user clicks part of the geometry for a model in the viewport, *plab* intersects the mouse ray with the geometry and displays the node which created the geometry. If the node is in another model graph, the responsible *CallOperator* in the current model graph is displayed instead. This makes it easier for the user to find the node which created the geometry.

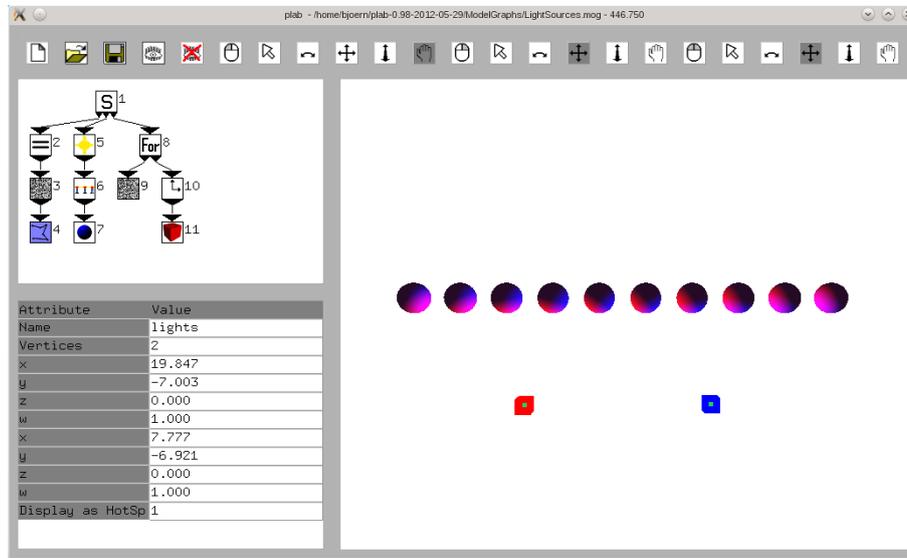


Figure 35: Control points and light sources in plab

4.4.2 Control Points

While modeling solutions like *Maya*, *3ds Max* or *Cinema 4D* allow editing parameters of primitives in the viewport only for certain types of primitives, *plab* and *GML* allow doing this for arbitrary objects, by implementing *control points*. `ControlPointArrayOperator` stores control points in a table. When the user moves control points, the changed values are written to the table, then the model graph is executed to update the geometry. Using the shift key, it is possible to select and move several control points at once.

There are various uses for control points. They can be used to mark salient points on an object, or they can be used to move objects as a whole. Control points do not need to lie on an object's surface. Common uses of control points include defining an object's length or angles between edges and surfaces.

The model graph depicted in Figure 35 uses control points to move light sources. The light sources are rendered as cubes, and the spheres are lit by them. The control points are stored in a `ControlPointArrayOperator` named `lights`. When the user moves the control points, node `lights` is updated and the model graph is restarted interactively. As a result the user may witness the effects of the moving light sources interactively.

4.5 Implementation

Several implementations of the user interface were realized. The variants that use glut and SDL are rather similar. An additional variant uses QT and supports a multi-document interface for model graphs. These variants use OpenGL for rendering. The glut and SDL variants provide their own set of widgets to maximize portability while minimizing external dependencies and support editing and executing model graphs. Alternatively, model graphs can be executed via command line to produce geometry. This way, procedural models can be created without user intervention. plab currently supports Microsoft Windows and Linux.

4.5.1 General Optimizations

Optimizations were implemented in plab to prevent specific bottlenecks rather than optimizing plab in general, because all optimizations need to provide a measurable gain in performance. Optimizing code without measuring performance may lead to code that might be slower for reasons that may not be obvious and should therefore be avoided. Furthermore, fully optimizing plab would not have been possible in a reasonable amount of time.

As a result of differing demands on performance, different paradigms have been used in the implementation of plab. At first, model graphs were a completely interpreted language. This means that all statements were parsed every time a node required it. This allowed for a speedy prototype implementation. When it turned out that this led to poor performance in certain scenarios, new code was added that stored a parsed representation of the nodes and *formulas* that could be evaluated or executed faster using virtual functions. This eliminated many (string) comparisons, reduced parsing and led to worthwhile performance gains. For simplicity, the pseudocodes in this section are based mostly on versions of the functions before optimizations like parse trees were applied. Parse trees are stored in objects, and each node in the parse tree automatically evaluates its child nodes when needed.

Storing every expression in a parse tree still does not guarantee best performance. Parsing an expression takes more time than evaluating that same expression once, because allocating memory for the parse tree takes a certain amount of time. While this added cost pays off quickly when the node is executed several times, it still incurs a penalty for nodes that are evaluated only once. Performance could be improved further either by detecting which nodes are suitable for preparsing, or by allowing the user to choose these nodes. While exporting model graphs as C++ code ensures optimal performance, improving performance of interpreted execution still matters for developing and testing model graphs.

Since *parse trees* store pointers to variables in the local *scope*, they must be recompiled whenever a model graph is executed, but the pointers remain valid for the calling model graph when executing `CallOperator`.

4.5.2 Rendering Optimizations

Similar optimizations turned out to be necessary for rendering. At first, the number of polygons that needed to be rendered was small, and each polygon was allowed to have its own texture. As higher polygon counts were required, rendering requirements shifted. It became necessary to reduce the number of textures by compiling individual textures to *texture atlases*. The texture atlases reduced the number of OpenGL calls required to set the current texture. Because the supported size of textures may be limited in hardware and software, the texture atlas may consist of several bitmaps when many (large) textures are used.

Texture atlases are created dynamically in `plab`. While a model graph is executed, textures created by model graph nodes are added to the texture atlas as virtual textures. The atlas may consist of several container textures that may contain several virtual textures. Any operations involving textures manipulate texels in the texture atlas via the virtual textures. A list of unused texture space is maintained by the texture atlas. This list is used to place new virtual textures into the smallest area that can accommodate the texture. Container textures may be enlarged to add space for new virtual textures when necessary, up to a user-defined size limit. When a new virtual texture cannot be placed in a container texture and it is not possible to enlarge an existing container texture, a new container texture is added to the atlas.

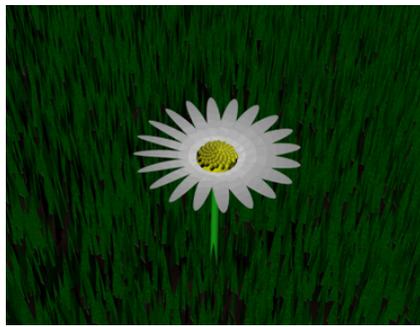
For rendering, all texture coordinates must be divided by the final sizes of the textures. One way to do this is to store absolute, integer coordinates at first, and then switching to floating point texture coordinates by iterating over all vertices and by dividing the texture coordinates by the final texture size once. Alternatively, OpenGL can scale integer texture coordinates using `glMatrixMode` with the `GL_TEXTURE` parameter. The latter method reduces space requirements since these texture coordinates can use integer types like `word`.

Furthermore, the OpenGL programming style shifted from the `glBegin/glVertex/glEnd` paradigm to calls that render several primitives in a single call. However, since `plab` needed to support instancing, and each instance may use different transform and lighting operations, it was not advantageous to bake the entire scene into a single mesh. Still, `plab` offers functionality to add polygons to existing meshes and is optimized to render objects in a single OpenGL call when possible.

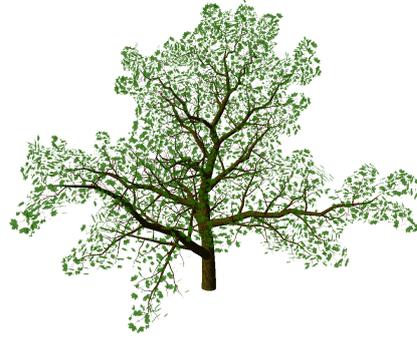
The *components* that produce primitives create a single mesh. `PolygonComponent` creates a separate mesh for each polygon, while `AddPolygonComponent` and `AddPolygonWithNormalsComponent` can add polygons to existing meshes. `plab` needs very direct access to scene graphs and deeply interacts with them. In the opinion of the author, this justifies using a scene graph implementation that was optimized specifically for `plab`.

4.5.3 File Formats

In order to continue working on models later, model graphs can be stored in XML format. For data exchange with other applications, scenes are imported into plab using `MeshFileComponent` and exported as Wavefront OBJ files. Wavefront OBJ files do not support transforms or object hierarchies, therefore the only way to export instanced geometry is to apply the transforms to separate copies of the instances. Textures are loaded using `TextureFile` and stored as BMP files. Model graphs may import and export variables in XML format using `LoadVariablesOperator` and `SaveVariablesOperator`.

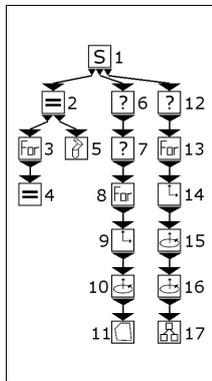


a



b

Figure 36: Example plants created in plab. a: a marguerite, b: a deciduous tree



a

Attribute	Value
Name	AssignmentOperator2
Variables	4
Variable Name	stem[4*i]
Type	double
Value	stem[4*i-4]+gnarliness*2*(random-0.5)
Variable Name	stem[4*i+1]
Type	double
Value	(i-1)/(stemsegs-1)*stemLength
Variable Name	stem[4*i+2]
Type	double
Value	stem[4*i-2]+gnarliness*2*(random-0.5)
Variable Name	stem[4*i+3]
Type	double
Value	stemwidth*(1-i/stemsegs)

b

Figure 37: Creating a deciduous tree. a: model graph, b: assignments in node 4

4.6 Examples

We demonstrate the versatility of the new system with model graphs that generate plants, buildings and landscapes.

4.6.1 Creating Plants

Figure 36 shows a flower and a tree that were created using plab. The *stem* of the *marguerite* was created using *StemComponent*. The *calyx* was created using a deformed sphere. It contains some *pollen* in *phyllotactic layout*. The polygons for the *marguerite's petals* are formed by connecting points along the petal rim with a mirrored rim, and the rims are defined by *Bezier* curves interpolating user-defined control points. By animating the control points, the bloom can be animated to grow, open or close.

The model graph in Figure 37.a creates trees using a technique described by Oppenheimer [111]. Execution starts in the *StartOperator*, node 1. It defines the interface for calling this model graph from another model graph using *CallOperator*. Among these parameters are the length and number of segments in a branch, the thick-

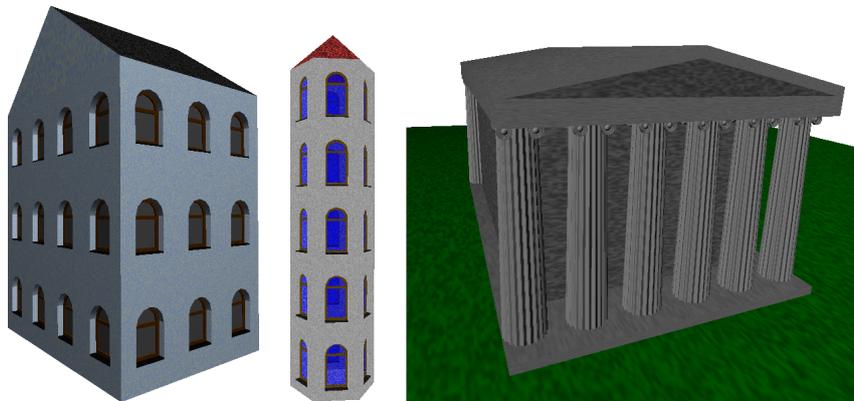


Figure 38: Some buildings that were created using plab

ness of the stem, the number of branch levels to create (`recursions`), and the textures to use.

Node 2 is an `AssignmentOperator`. It declares a new variable `stem` that stores the control points of the current branch. Node 3 is a `ForOperator` executing node 4 in a loop that initializes `stem`. Figure 37.b shows the assignments of node 4. Execution continues with the next child of node 2: Node 5 is a `StemComponent`. It creates the geometry for the trunk of the tree from `stem` and a bark texture.

Node 6 tests if enough recursions have been performed and node 7 tests if a leaf texture was given as a parameter. In that case, nodes 8-11 create the leaves. Node 8 is a `ForOperator` that creates the leaves with the aid of a translation in node 9 and a rotation in node 10. Node 11 creates a leaf polygon every time it is executed, using a partially transparent leaf texture.

Nodes 12-17 create branches by recursion. Node 12 tests `recursions` to decide whether further recursions are necessary. Node 13 is a `ForOperator` that executes nodes 14-17 in a loop to create child branches. Nodes 14-16 transform the new branch. Node 17 is a `CallOperator` that produces the next levels of branches recursively.

4.6.2 Creating Buildings

In this subsection, several model graphs are explained that allow a user to define a building from its corner points. The geometry for the house is computed from these points automatically. In particular, this allows for buildings with a non-rectangular floor plan.

Figure 38 shows *buildings* that were created with plab. The *tower* and the *house* use the same model graph to create *windows*. Window breadth, height, depth and texture can be changed using parameters. The number of polygons for the round window shape is also a parameter. If that parameter is less than 2, a straight window top is created.

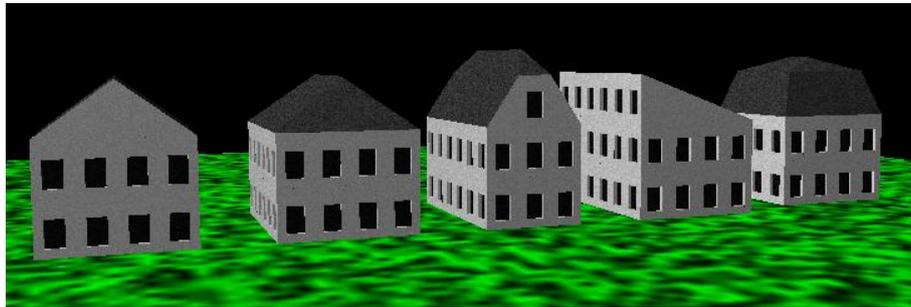


Figure 39: Roof types supported by plab: gable roof, hip roof, half-hip roof, single-sloped roof, mansard roof

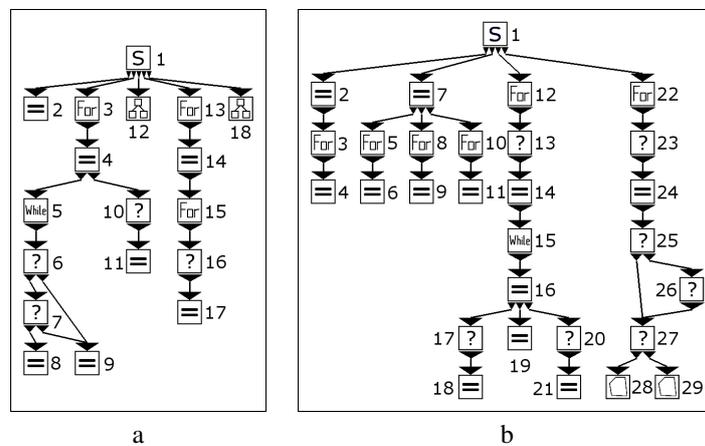


Figure 40: a: model graph that calculates the order of points on a roof (`calcRoofOrder`), b: model graph to create the geometry for a roof (`roof`)

An editing system for buildings was implemented that computes a building from control points that define the corners of a building and its roof. Moving the *control points* allows the user to edit buildings interactively at a lower resolution. Several model graphs are needed to implement this editor. Figure 39 shows example buildings with various roof types created using these model graphs. The first model graph is `calcRoofOrder`, shown in Figure 40.a. It sorts the input vertices by altitude and assigns each vertex a `roofOrder` depending on its altitude. Two vertices have the same `roofOrder` if they are at approximately the same altitude (subject to a parameter `heightTolerance`). This information is used to compute the geometry automatically from points that the user may move interactively. If the user chooses, a façade with windows and an entrance is constructed. By moving the *control points*, buildings can be generated from different floor plans. Using group *selection*, parts of the building, or the entire building, may be moved.

The `StartOperator` (node 1) in `calcRoofOrder` takes the following parameters: `frontPoints` gives the number of vertices defining the façade of the build-

ing. Counting the vertices that share the lowest altitude yields the correct value for `frontPoints` only in some cases. As demonstrated by gable roofs, front vertices are not always at the same altitude, so `frontPoints` must be stated by the user. `vertices` contains the corners of the building and the roof. The vertices defining the façade must be placed in this array first. There are also texture parameters and a flag `highQuality` that decides whether to create a single polygon for each building side or detailed windows and an entrance.

Node 2 declares the array `roofOrder` and several help arrays used later. Nodes 3-11 copy the y vertex positions into an array `heightsFound` that contains each height level only once. A height level is an interval delimited by `h±heightTolerance`. Node 12 calls another model graph that sorts `heightsFound`. Nodes 13-17 assign a `roofOrder` to each vertex by finding the closest height level. Node 18 calls a separate model graph (`generateFaçade`) to produce a façade from the vertices, including windows and entrances. The creation of the façade, windows, and entrances consists mainly of polygon lists and low-level computations, and therefore is not presented in this work.

The model graph for roof creation is shown in Figure 40.b. It has a `StartOperator` taking the same arguments as `calcRoofOrder`, with the addition of `roofOrder`. Nodes 2-11 compute several values from the input parameters. Nodes 12-21 compute the nearest point of the next roof order for each point by Euclidian distance. Node 22 iterates over nodes 23-29 to create geometry for the roof. The overall time complexity of this model graph is $O(n^2)$, where n is the cardinality of `vertices`. A *Voronoi diagram* would reduce asymptotic costs, but would likely be slower for the small numbers of points required to define typical buildings.

The algorithm by Laycock and Day supports several roof types, but the user has to split the building into several parts and assign a roof type to each part [82]. We argue that it is more intuitive and faster to edit buildings using roof corner vertices, as proposed here.

4.6.3 Creating Terrain

This subsection and the following present a tool set of model graphs that can be used to generate eroded terrain. To begin with, we need a terrain that contains a few hills or mountains. Then, we run an erosion simulation to obtain a network of rivers and lakes. Optionally, the user may want to apply a smoothing effect to the resulting terrain. Figure 41 shows an example *terrain* created using the model graphs described here.

The model graph `SlopingHills`, shown in Figure 42, calculates a *digital elevation model (D.E.M.)* for a mountain range, similar to the algorithm by Belhadj and Audibert [9]. `StartOperator1` (node 1) defines its global parameters: the resolution of the grid to create (`xsize`, `ysize`), the number, height, minimum and maximum slope of the mountains. Node 2 declares the arrays `arr` and `delta`. `arr` stores the grid, while `delta` stores the slope of each grid point to its highest neighbor. `delta` is required because we need the slope between two points to be *reproducible*. If the algorithm would choose between the current altitude for a cell and the highest surrounding alti-

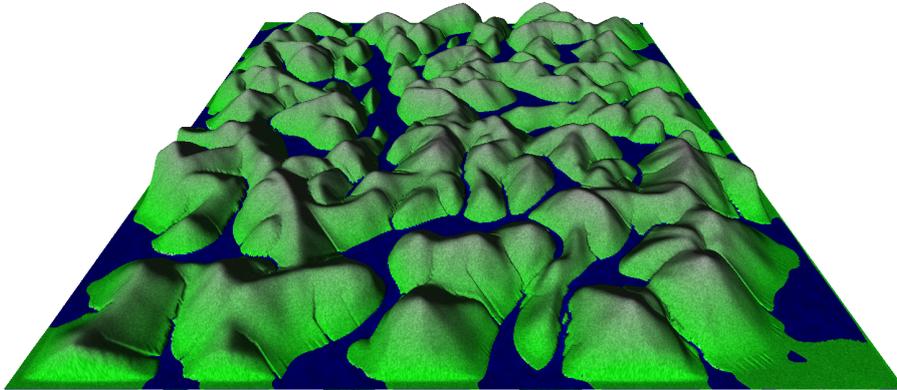


Figure 41: Eroded terrain (grid resolution 512x512)

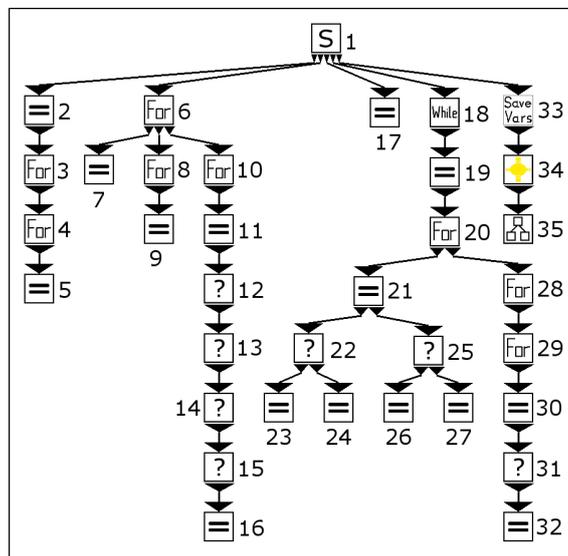


Figure 42: Creating mountain ridges from Bézier curves

tude minus a random value, over the course of several iterations the algorithm would choose ever lower altitudes, reducing the noisy impression that is desired for terrain. By storing the random value that is subtracted from the highest neighbor, we prevent this and keep the fractal look of the terrain. Nodes 3-5 initialize `arr` and `delta`.

Node 6 executes nodes 7-16 in a loop to create the mountain ridges. Nodes 8 and 9 create a *Bezier* curve for the mountain ridge starting from the position selected in node 7. Nodes 10-16 copy that curve into the grid.

The mountain slopes are calculated for each grid point by subtracting `delta` from the highest neighbor, multiplied with $\sqrt{2}$ for diagonal neighbors. Node 18 executes nodes 19-32 until all cells have received their final values. The direction of diffusing altitudes is changed for each iteration of the main cycle, because otherwise, altitudes would be

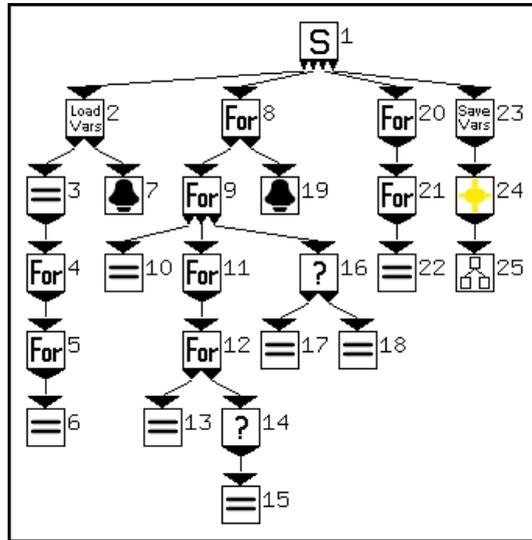


Figure 43: Model graph that smoothes the terrain

diffused too slowly in directions opposing the direction used by the for loops. Thus, the directions of the for loops in nodes 28, 29 are changed every turn. Node 17 defines the directions, nodes 21 to 27 select the appropriate values for nodes 28, 29. Nodes 28-32 diffuse the altitudes through the grid.

After propagation of the altitudes is complete, the results are stored to disk (node 33). Node 34 sets up a light source for display and node 35 calls another model graph that generates polygons for display. While fast algorithms for computing *Voronoi* maps are available, they cannot be applied here, as we are dealing with randomized cell elevations.

4.6.4 Smoothing Terrain

Figure 43 displays a model graph that smoothes terrain. For higher altitudes, a higher fractal dimension is required to create an impression of rough hills, whereas in a valley, a lower fractal dimension dominates. This can be achieved by varying the smoothing radius with the altitude. The model graph's *StartOperator* (node 1) defines the following parameters: *topSmoothRadius* is the smoothing radius that applies at the highest altitudes, while *bottomSmoothRadius* is the smoothing radius that applies to valleys. Node 2 loads the input terrain. Node 3 defines some help variables. Nodes 4-6 find the minimum and maximum altitude in the grid. Node 7 prints some debug information. Nodes 8-9 form the main loop which iterates over nodes 10-19. Node 10 computes the radius for smoothing the vertex, depending on the altitude of the vertex. Nodes 11-15 compute the average altitude of the vertices that are inside the computed radius. Nodes 16-18 assign the average altitude to the vertex if radius is greater than 0. Otherwise, the old value is copied. The altitudes are assigned to a new array to assert that all vertex altitudes are computed from non-smoothed data. Nodes 20-22 copy the computed values back into the original array. Node 23 stores the result to a file, and nodes 24+25 render the results with a light source.

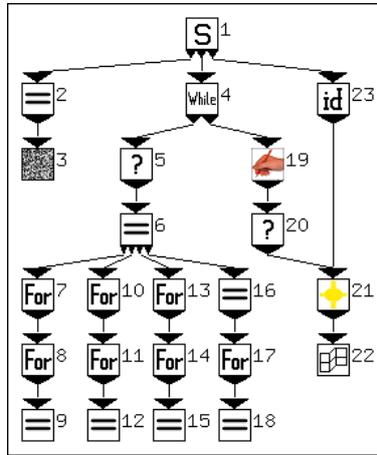


Figure 44: Model graph for midpoint displacement

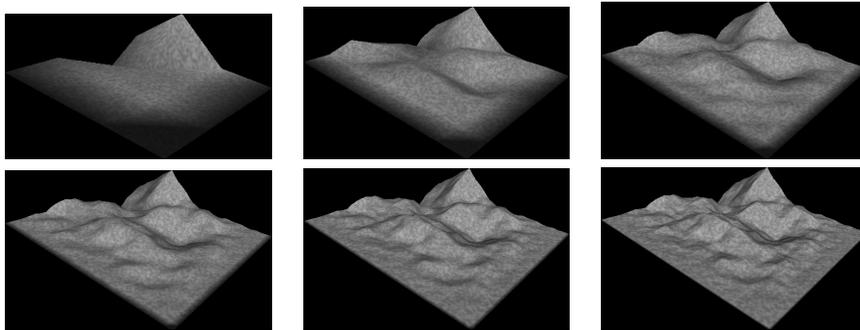


Figure 45: Midpoint displacement: Successive *levels of detail*

4.6.5 Midpoint Displacement

Figure 44 presents an implementation of midpoint displacement using model graphs. The `StartOperator` in node 1 specifies the model graph's parameters: `maxLen` defines the maximum size of the *grid*, while `maxDelta` defines the maximum elevation. Node 2 defines a number of variables that are needed for execution, while node 3 creates a simple, rough texture for the terrain. Node 4 controls the main *while* loop over nodes 5-22. Node 5 makes sure that the terrain is refined only as long as the maximum grid size is not exceeded. Nodes 6-9 resize the array by copying all values. Nodes 10-12 calculate altitudes for vertices between pairs of grid points, similar to splitting horizontal and vertical edges. Nodes 13-15 calculate altitudes for vertices that lie between vertices that were inserted in nodes 10-12, similar to inserting a new vertex into the middle of a quad. Nodes 16-18 copy the computed values back into the original grid. Nodes 19-22 render the grid as an animation showing how the algorithm creates terrain detail by doubling the resolution of the terrain. As `RenderOperator` destroys the geometry of its child nodes immediately after rendering, node 23 is needed to bypass `RenderOperator` to generate geometry for display after the model graphs has completed. Figure 45 shows an example that was produced using these model graphs.

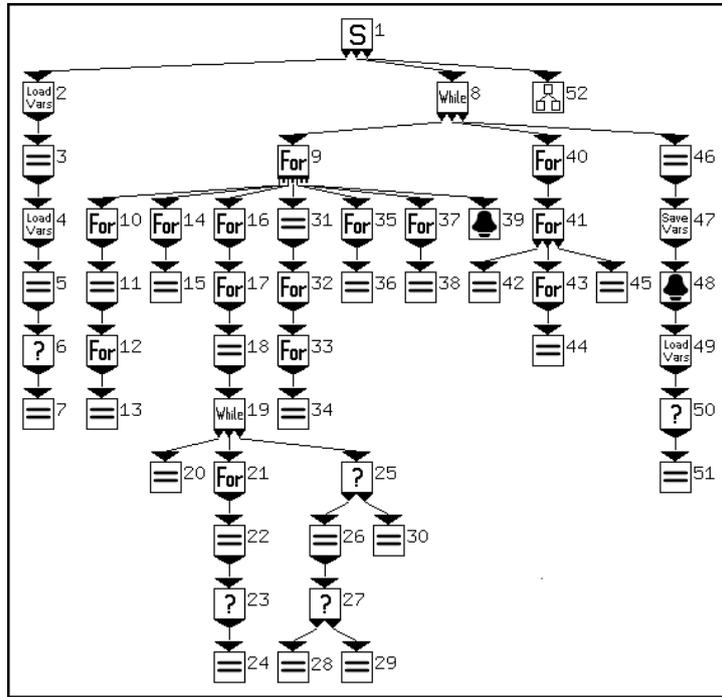


Figure 46: Model graph for erosion simulation

4.6.6 Erosion Simulation

As discussed in Section 3.2.12, *erosion simulations* try to mimic the processes that shape terrain over millennia. The input terrain may be created using the methods discussed in the preceding sections, or the input terrain can be a *digital elevation model* representing real terrain. The simulation adds rain to cells in the grid, calculates water exchanges between the cells, and decreases altitudes depending on the amount of water that was exchanged. Some simulations store the water levels for each cell, whereas other simulations model the water as particles. If particles are used, the speed of the particles can be factored into the altitude decrease. Further options include simulating different layers of material, such as rock, sand, water, and how much dissolved material (sand) is transported by the water. Depending on the set of features implemented, different *formulas* have to be used to model the physical effects.

We chose to implement a simple model of erosion that does not require particles, because erosion simulation with particles is even more computationally expensive than a grid-based simulation. Figure 46 shows the model graph used for this simulation. Node 1 is a *StartOperator* that defines the parameters used in this model graph. Node 2 loads the terrain from an external XML file. Node 3 defines a number of help variables. Node 4 loads additional simulation settings from an external file. These settings are more important than the settings defined in *StartOperator*. Storing these settings in an external file allows the user to edit the settings while the simulation is running, so the model graph has to reload the file at regular intervals. Node 5 defines an array that stores directions to the neighboring cells: $(-1, -1)$, $(-1, 0)$, ... $(1, -1)$, $(0,$

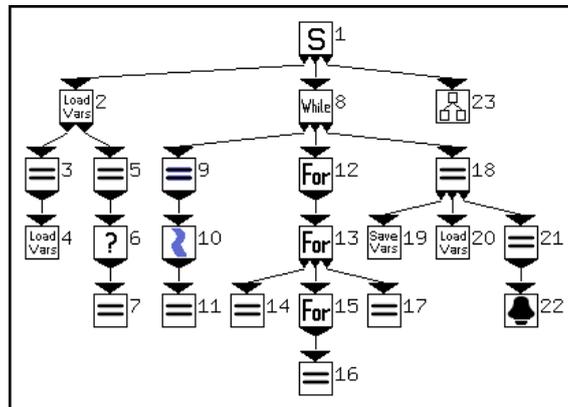


Figure 47: Model graph for erosion simulation that uses an erosion simulation node written in C++ (node 10)

-1). If the terrain loaded in node 2 does not contain water, nodes 6 and 7 initialize a new variable `water`, which stores the water levels for each grid point.

Node 8 is the iterator for the main loop. The main loop aborts if no terrain data was loaded in node 2, or if the user presses any key. The `ForOperator` in node 9 executes nodes 10-39 in a loop that simulates water movement and fluvial erosion. Nodes 10 to 15 add a user-defined amount of rain to every cell in every simulation cycle. Nodes 16-31 implement the actual erosion simulation. For each grid point, nodes 21-24 find the neighboring cell with the lowest sum of terrain altitude and water level. If the water levels between the cell and its nearest neighbor differ by more than a certain constant, node 26 copies some array values into local variables to speed up computations in nodes 27 to 29. Node 27 checks if the full amount of water can be transferred from the current cell to its selected neighbor. If so, the transfer takes place in node 29. Node 28 handles partial transfers.

All water transfers computed so far were not applied immediately. If all transfers were applied immediately, water would be transferred faster in some directions than in others. Instead, we limit the speed of transfers so that water is transferred only from one cell to the next in each iteration, similar to Section 4.6.3. Nodes 31 to 34 apply the transfers to the terrain. Nodes 35-38 set water levels at the grid borders to zero again, so that all water arriving at the grid border simply vanishes. Node 39 reports progress to the console.

After several iterations of fluvial erosion simulation, nodes 40 to 51 are executed once. Nodes 40-45 compute thermal erosion. Through thermal erosion, every cell drops by a certain amount towards its lowest neighbor. Nodes 46-47 store the results of the computation to a file. This allows to view the course of simulation later and to continue the simulation from previous simulation cycles. Node 49 reloads the settings file in case the user made changes to the simulation parameters while the simulation is running. If the user presses any key to abort the simulation, node 52 calls a separate model graph to display the results.

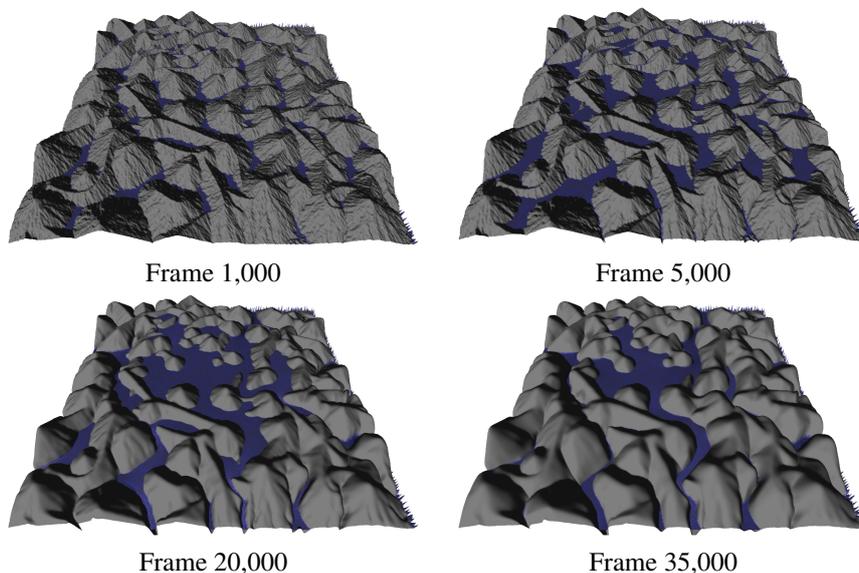


Figure 48: Results for running the integrated erosion simulation in plab on a 512x512 terrain. The parameters were changed during the simulation to increase the smoothing effect

Running the model graph for erosion simulation turned out to be practical only for small terrains of resolutions up to 128x128 due to performance reasons. For larger terrains, the main nodes for simulating erosion were rewritten as C++ code, which can be called using a model graph node. The resulting model graph is depicted in Figure 47. Furthermore, it was possible to implement a parallel variant of the erosion algorithm. While Beneš and Forsbach cut the terrain into several strips, and treat the borders separately [14], we split the array for transferred water amounts into 8 arrays, with one input for every adjacent cell. This allows to transfer water from all cells in parallel. After the exchanges are completed, the input amounts of water are added up for each cell in parallel. We used OpenMP to parallelize the algorithm. The simplified model graph is shown in Figure 47. The unaccelerated model graph computes 0.83 frames per minute on a 2048×2048 grid, while the accelerated model graph computes 158 frames per minute on a grid of the same size. Therefore, the accelerated model graph is 190 times faster. These measurements were taken on an AMD Phenom 1090T with 6 cores.

When an erosion simulation is run on terrain that has not been eroded in any way, the simulation has to run for a long time. First, all basins in the terrain must be filled with rain until they leak at the lowest border. Then and there, a new river bed is formed as water escapes the basin. Lakes are converted to rivers as this process continues, and the simulation has to run until the user is satisfied with the results. This also means that the process has to be supervised at least in part. In our experiments, during one simulation cycle, we test for water transfers between all neighboring cells once, and we required several thousand simulation cycles in total to generate satisfying terrains. The minimal time to create a terrain was half an hour, but it is possible that more experienced users would be able to generate terrains faster. Figure 48 shows selected frames of the erosion simulation.

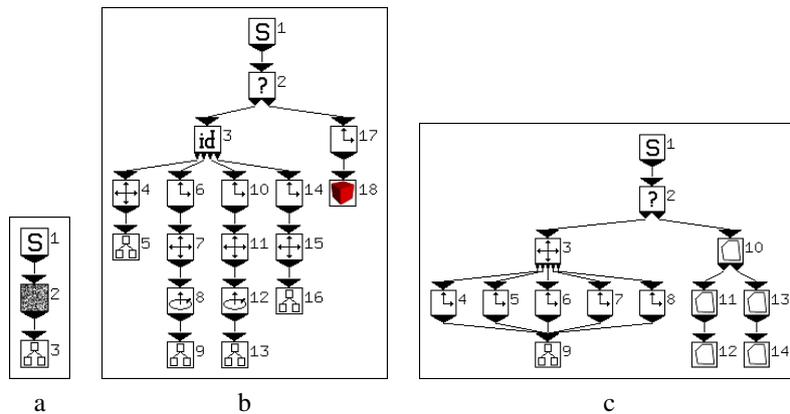


Figure 49: Model graphs that produce fractals. a: non-recursive part for Koch curve, b: recursive part for Koch curve, c: recursive part of Sierpinski pyramid

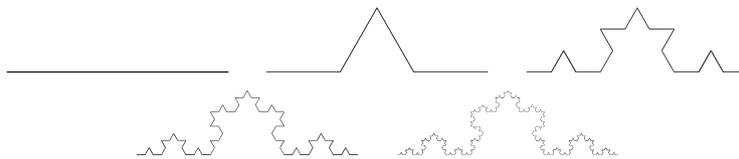


Figure 50: First five iterations of a Koch curve

4.6.7 Fractals

We also tested generating fractals with plab. In order to use the same texture for several levels of recursion, fractals are modeled using two model graphs, where the first model graph sets up the texture to use, and then calls the recursive second model graph. Otherwise, the same texture would be generated by every model graph. While it would be possible to integrate both model graphs into a single model graph, this solution is slightly more efficient because it avoids deciding which part of the single model graph would have to be executed for every recursion. For example, the *Koch curve* starts with a single line segment. Each line segment is split into three pieces recursively, and the middle piece is replaced by two further pieces. In each iteration, the curves grow by a factor of $4/3$. It can be shown that the fractal dimension of the Koch curve is

$$d = \frac{\log 4}{\log 3} \approx 1.2619. \quad (5)$$

Figure 49.a shows the first model graph for a Koch curve, Figure 49.b shows the second model graph. The first part of the model graph is executed once. It creates the texture once and executes the recursive, second model graph. In the second model graph, node 1 is the `StartOperator`. Its first parameter states the thickness for rendering the line, whereas the second parameter states the number of recursions used to approximate the Koch curve. If further recursions are needed, node 2 executes nodes 3 to 16 to perform the recursions. Node 3 is an `idOperator` which is used because the `Comparator` in node 2 calls only one successor, but nodes 4, 6, 10, and 14 need

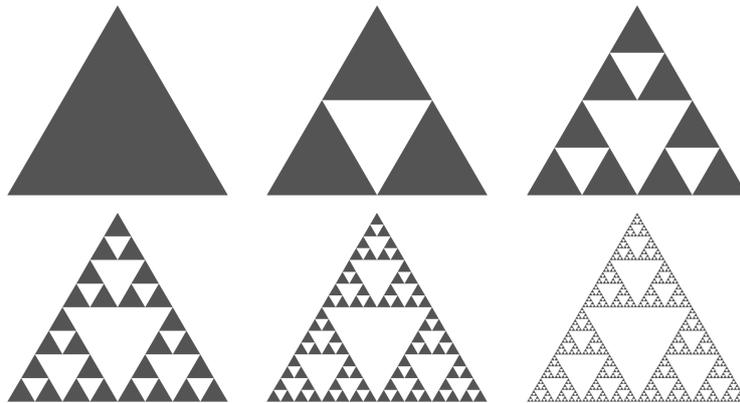


Figure 51: First six iterations of a Sierpinski triangle

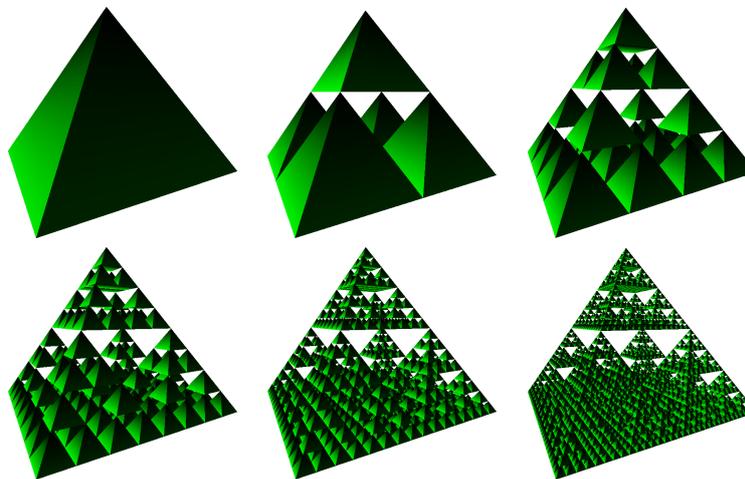


Figure 52: First six iterations of a Sierpinski pyramid

to be executed. Nodes 6, 10, and 14 translate child objects, while nodes 4, 7, 11, and 15 scale them. Nodes 8 and 12 perform the rotation required for the middle pieces, then nodes 5, 9, 13, and 16 call the model graph recursively. If no further recursions are needed, node 18 creates a cube representing the line. Figure 50 shows example outputs.

Constructing the *Sierpinski triangle* starts with a triangle. In each iteration, all sides are split in half to create four new triangles out of each existing triangle, then the middle triangle is removed. The outer border of the surface can be described as a curve with *Hausdorff dimension*

$$d = \frac{\log 3}{\log 2} \approx 1.585. \quad (6)$$

Figure 51 shows an example for this fractal produced using `plab`. Figure 49.c shows the recursive part of a model graph that produces a three-dimensional variant of the

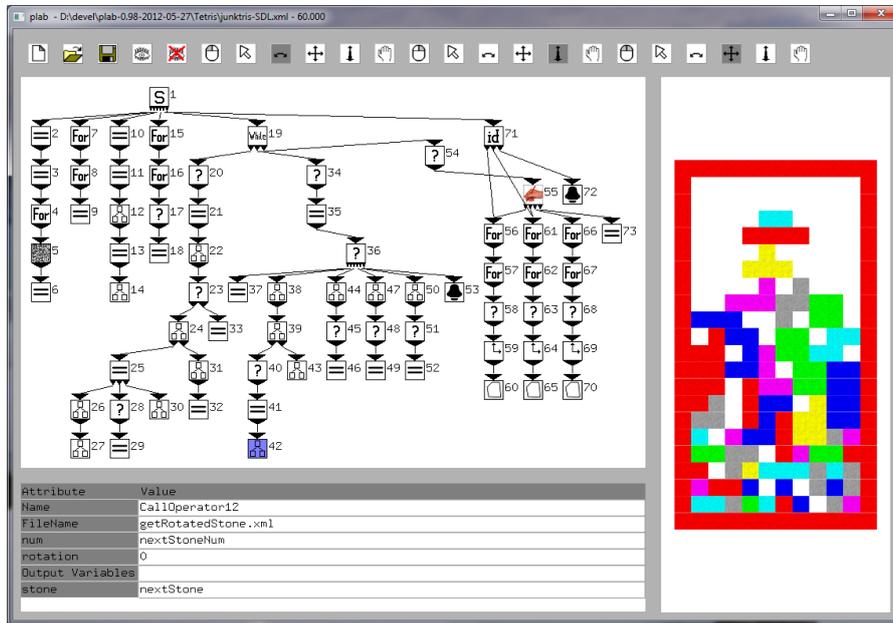


Figure 53: plab executes a model graph that allows to play a Tetris clone

Sierpinski triangle, the Sierpinski pyramid. It is structurally similar to the model graph that produces the Koch curve, which is displayed in Figure 49.b. One difference is that node 9 is used for all recursions, instead of using several CallOperators. This works because the parameters are identical. Nodes 10-14 produce triangles for the pyramids. Figure 52 shows the resulting image.

4.6.8 Animation

As a simple example for an interactive application of plab, a *Tetris* clone was programmed using plab. Figure 53 shows a screenshot of the game. Node 1 defines general parameters, such as the size of the playfield. Also, this node defines the number of random junk lines that are added to the bottom of the playfield to make it more challenging. Nodes 2 and 3 define basic game variables, like colors for the game pieces and statistics. Nodes 4-6 define textures for the game pieces. Nodes 7-9 initialize *playfield*. Nodes 10-12 initialize further variables. Nodes 12 and 13 use an external model graph to obtain the first and second game pieces. Nodes 15-18 initialize the lower playfield lines with random junk. Node 19 implements the main loop for the game. Nodes 20 and 21 check if enough time has passed to drop the game piece by one line. Node 22 checks if the current game piece can drop by one line. If the game piece cannot move down, nodes 24-32 choose the next game piece. Node 27 checks if sufficient space is present for the new game piece. If not, nodes 28 and 29 end the game. Nodes 34-52 contain the keyboard handlers. Nodes 54-70 render the playfield, the current and next game pieces. If the games is finished, node 73 prints final score, and node 72 is an IDOperator that reruns the operators that create geometry to display after model graph execution ends.

4.6.9 Model Graphs that Emulate Turing Machines

As stated by the Church-Turing thesis, any function that can be calculated can be calculated by a Turing machine. Applied to procedural models [154], this means that any shape that can be computed or approximated using finite time and storage can be computed by a Turing machine. As model graphs can emulate Turing machines, they can compute or approximate the same class of shapes as Turing machines. We prove this informally by describing how to construct a model graph that could emulate a given Turing machine.

Turing machines are a machine model designed to investigate theoretical possibilities of algorithms. They were proposed by Alan Turing in 1937. A Turing machine can be thought of as operating on an infinite tape. In each cycle, it reads one symbol from the tape, changes its internal state according to that symbol, writes a symbol to the tape at the same position, and moves the read/write head either right or left. Formally, a *Turing machine* can be defined as a tuple [70]

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F), \quad (7)$$

where

- Q : A finite set of states,
- Σ : A finite set of input symbols, $\Sigma \subseteq \Gamma$,
- Γ : The set of symbols that may be used on the Turing machine's tape,
- δ : The transition function, $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$,
- q_0 : The initial state, $q_0 \in Q$,
- B : The blank symbol, which initially occupies all positions on the tape,
- F : The set of accepting states for the Turing machine, $F \subseteq Q$.

The tape used by the state machine can be represented by an integer array `tape`. When allocating the array for the tape, model graphs require limits for the amount of storage usable for the array. During allocation of the array representing the tape, the array is initialized with the blank symbol. The current state $q \in Q$ can be stored in a variable `state`. The position of the read/write head of the Turing machine is stored in a variable `offset`, which is initialized to point to the start of the input data. The main loop of the model graph is controlled by a `WhileOperator`, that executes its child nodes until a terminal state is reached. The `WhileOperator` has a `Comparator` or `BranchOperator` child node. It checks the `state` variable and branches to another `Comparator` or `BranchOperator`. This operator checks the symbol at the current position of the read/write head by evaluating `tape[offset]`. Depending on that symbol, the model graph branches to an `AssignmentOperator`, which writes a new symbol to the tape, increases or decreases `offset`, and optionally changes the state. This completes the construction and the proof.

Just as π cannot be stored as a number using finite storage, all macroscopic shapes have a microscopic complexity that cannot be captured using a reasonable amount of storage and rendering time. However, it is a principal goal of raster graphics to match the resolution of geometry to the resolution of the screen. As such, approximations are a basic fact of life in computer graphics.



Figure 54: Model graphs can be used to model trees, buildings and landscapes

4.6.10 Putting It All Together

Figure 54 shows a scene that uses the model graphs discussed in this section. Geometry creation takes about 27s using a single thread, and 17s using multiple threads on a dual core AMD Opteron 180 with 2400MHz and 2.5 GB RAM. The mountain is read from a file which was created using the model graph shown in Figure 42. The scene consists of more than 800.000 polygons.

System	Model Types	Viewport Editing	Formula Notation	Variables / Arrays	Split programs into modules	Automatic Parallelization
GenMod	any	No	infix	Yes	Yes	No
GML	any	Yes	postfix	Yes/Yes	Yes	No
L-Sys.	plants, streets	No	infix	Yes [128]	Limited [62]	No
Scripts	any	Can be impl.	infix	Yes	Yes	No
VDFPs	any	No	visual	No/Yes	Yes	Yes
Xfrog	plants	No	infix	No	No	No
Model Graphs	Any	Yes	infix	Yes	Yes	Yes

Table 3: Model graphs integrate the features of previous approaches to procedural modeling into a single system

4.7 Discussion

In plab, the number of variables, polygons and textures are limited only by the hardware. Despite the optimizations, *model graphs* do not reach the performance of compiled applications. This is a limit in case of computationally expensive operations, but it could be overcome by exporting model graphs as compilable code. The *Deutsch limit* states that a computer monitor is usually limited to about 50 icons that can be visible at a time, and that this is insufficient to implement complex applications. This limit does not apply to model graphs since model graphs can be split into several modules and large model graphs can be viewed using scroll bars if a model graph does not fit onto the screen.

Table 3 shows that the new system includes the strengths of its predecessors and avoids their shortcomings. The importance of these features is underlined by the fact that the example in Figure 54 uses all of the features in the table.

While *GML* stores all data on the stack and uses postfix notation, model graphs have variables and complex data types that allow for *formulas* in infix notation. *Infix notation* is more familiar to humans than *postfix notation*, and computers can translate it into machine code faster and more reliably than humans can. Model graphs visualize functional dependencies better than the postfix notation used in *GML*. The common strength of both systems are the *control points* that allow for interactive editing of arbitrary objects in the viewport. plab integrates the features of several specialized tools into a universal procedural modeling tool. While some design decisions in *GML* were justified only with being able to create a simple parser, plab aims at simplifying the modeling process for the user. plab's infix notation, variables and graphical user interface are arguably easier to use than *GML*'s stack-based reversed Polish notation. plab tries to minimize parsing by storing parsed expressions as C++ classes, allowing for faster execution of model graphs.

Both *VDFPs* and model graphs wrap operations with nodes, but the meaning of the edges is fundamentally different. *VDFPs* use edges to transport data and nodes may be executed when all input data has arrived. While *VDFP* edges transport anonymous data, variables in model graphs are named, and the names can be used to explain the purpose of the variables. Model graphs use edges to define a strict order of execution. *VDFPs* require a single node for each operator in a formula, while model graphs allow functions to be typed faster in infix notation, and procedural models heavily use mathematical formulas. Model graphs achieve high performance since they require little synchronization for parallelism and arrays can be changed efficiently, whereas a *VDFP* needs to create copies for all changes to an array, or stores the changes in a list and efficient execution requires careful scheduling. Model graphs have very intuitive notations for loops.

As stated earlier, visual languages like *VDFPs*, *Xfrog* and model graphs have advantages over textual languages since they visualize functional dependencies and function parameters better, and for all node parameters, a descriptive text is shown instead of requiring the user to memorize keywords or a parameter list. Using grammar-based systems, such as *L-systems*, requires a lot of experience and time for experimentation in order to create plants or buildings, but the model graph nodes closely reflect basic concepts in computer graphics and allow for interactive modeling in the viewport by moving *control points*. While *Xfrog*'s nodes are well-suited for plants, model graph nodes are more versatile, as proven by the examples in Section 4.6.

Both *Xfrog* and *plab* are visual procedural modeling systems. While *Xfrog* excels in creating and animating plants, *plab* is far more universal and offers variables and modularization. *plab*'s variables allow changing important parameters in a central location. *plab*'s nodes are finer grained, allowing for greater flexibility. While *plab* does not specialize in modeling plants, procedural algorithms to create plants can be implemented. In contrast to *Xfrog*, *plab* offers automatic parallelization, arrays, *viewport editing*, and supports models of all types, instead of editing only treelike structures.

Standard 3D modeling suites have powerful methods of freehand editing. *plab* offers freehand editing of arbitrary objects using model graphs with control points, similar to *GML*. Grammar-based systems, including *L-systems*, are usually implemented for a single class of objects, such as plants. An *L-system*'s replacement operations can be translated to adding geometry or changing record values in *plab*, but *plab* is not limited to a single class of objects. Compared to *VDFPs*, *plab* has the advantage of having variables, rather than passing all values using pipelines. As there tend to be numerous variables in a procedural model, a *VDFP* for a procedural model would likely contain many pipelines and nodes, whereas *plab* requires only a single edge between a pair of model graph nodes to define the order of execution. `AssignmentOperator` allows to specify several formulas in a single node, while a *VDFP* requires a single node for each operator in a formula.

Using model graphs, it is easy to change all instances of geometry created by an operator, but editing single instances may require editing the model graph. In the next chapter, we analyze how an editor can enable selective changes to a single instance, several instances, or all instances.

The goal to create closed meshes in plab was in conflict with fast rendering. In OpenGL, fast rendering requires *indexed facesets*. Polygons are stored as vectors of indices into the vertex buffer. If a vertex appears twice, only with different (u, v, w) coordinates or another normal, the vertex must be stored twice. It would be possible to detect vertices with identical x, y, z coordinates automatically, but this would require additional data structures on top of the indexed facesets.

4.8 Conclusion

This chapter discussed a new visual paradigm that integrates the features of previous procedural modeling systems into a single modeling environment without compromising performance. It is the first procedural modeling language designed for versatility and usability. An artist's mind is tuned towards visual perception, and model graphs appeal to this thinking. Furthermore, models can be altered efficiently in the viewport. The examples demonstrate how to use this for editing buildings. The new system is the first to create complex models consisting of buildings, plants and landscapes procedurally without resorting to external tools.

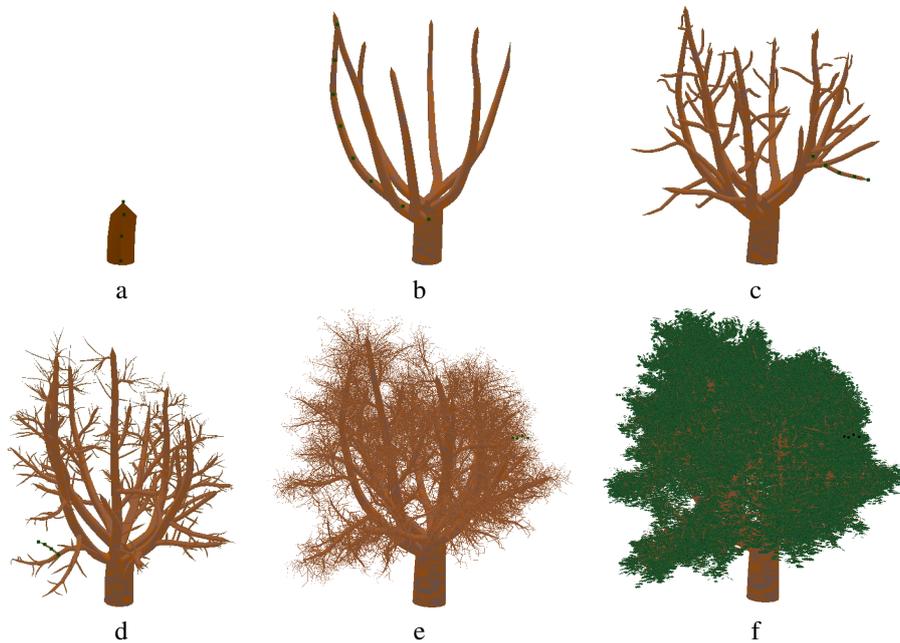


Figure 55: Modeling a plane tree in 1-2-tree. a: Editing a tree starts at the trunk. b: The trunk and the main branches are modeled individually. c+d+e: Further branches are added and edited using mass updates. f: Result after adding leaves.

5 Increasing Usability of Procedural Modeling with an Editing Approach

This chapter proposes a workflow for procedural modeling that differs from the one described in the last chapter. For this reason, we temporarily leave model graphs behind, but a possible synthesis of the approaches is presented in Section 6. Most previous algorithms for procedural modeling work in three phases, including the approach discussed in the last chapter:

- First, the user edits the parameters for a model.
- Then, the procedural algorithm is executed to create geometry for the model.
- Finally, the new model is displayed to the user, who may start the process again by further adjusting the parameters.

Editing models in this manner is time-consuming and frustrating for several reasons:

- **Lack of performance:** When large scenes are recreated, it is often not possible to achieve interactive editing, because recreating the entire scene simply takes too long.
- It may be difficult to judge how much a scene will change when its parameters are changed - even small changes to the parameter set may change the resulting scene drastically.

- **Flickering:** When the scene is recreated with a different random seed, all random parts of the scene change, even when the user makes only slight changes. If the same random seed is used, flickering may still occur when the changes cause the same pseudorandom values to be used in a different local order.

In order to allow editing specific entities, lists of exceptions can be stored. When changes are made, the procedural algorithm needs to check the exception list for every item that needs to be created, which complicates the algorithm. After adding or deleting objects, matching problems may occur. These problems can be addressed in the following manner:

- Instead of storing a list of exceptions, parameters can be stored directly in the scene graph. This solves the problem of finding the parameters for a branch.
- **Selective changes:** Storing the parameters directly in the scene graph also allows for selective changes. It is possible to select items based on their position in the scene graph, or selecting items based on their properties.
- Instead of recreating the entire scene whenever parts are changed, the procedural model can selectively update parts of the scene graph that are affected by any changes. This also helps reduce flickering and improves performance for editing.
- **Direct Feedback:** Viewport changes should be executed and visualized interactively.

We demonstrate these solutions with an editor for trees. Our system treats *tree* organs such as the *trunk*, *branches*, *twigs*, *roots*, *leaves*, *blossoms* and *fruits* as *semantic entities*. The entities have simple *parameters* such as size and color. Further parameters control placement and orientation of the tree organs. In this chapter, we focus on the branches and leaves as the most important entities that define the tree structure and visual appearance, but future implementations could include other entities. As in other publications, the term “branches” includes the trunk, any twigs, and even roots.

The entities are stored in a *scene graph* [146], where all nodes represent branches. Algorithms that alter parameters or update geometry have *recursive scene graph traversal* as a built-in feature. We expose the recursive data structure updates to the user as *selection options*. Thus, the user may choose to apply parameter changes to a single branch or several branches at once. A notation for *positional information* ensures variety.

Every entity has two representations: one is its *parameterization*, the other is its *geometry*. Both representations are stored in the scene graph node. This has a number of benefits: When the user selects an entity in the viewport, our system displays its parameters. When the user edits the geometry in the viewport, our system adjusts the entities’ parameters. Any changes to the parameters or the geometry are interactively applied to the other representation and optionally to other instances. Only the geometry for the affected nodes needs to be rebuilt.

The new system is designed to allow for fast and easy creation of trees, thus its name 1-2-tree (“easy as counting one-two-t(h)ree”).

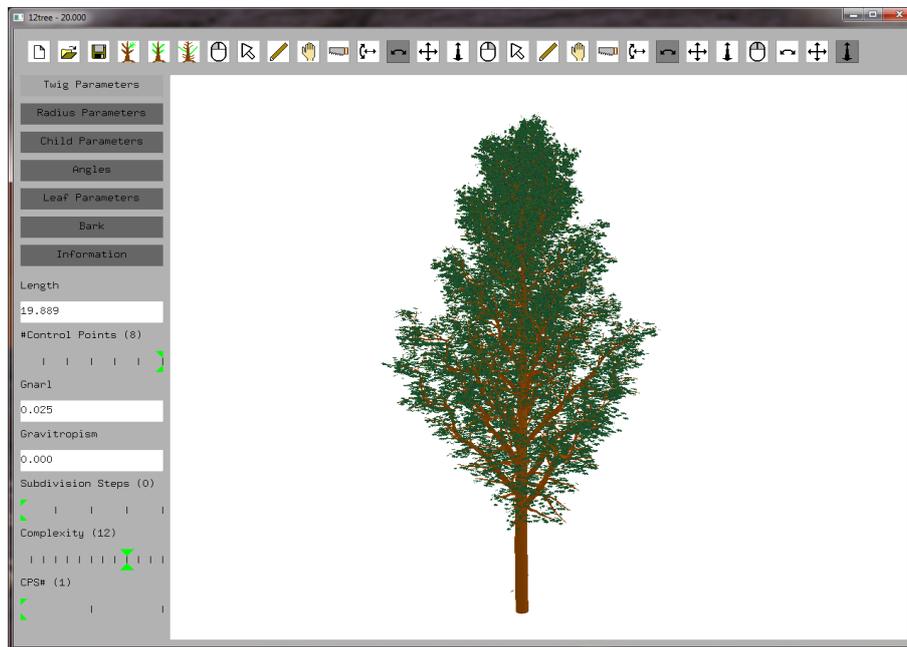


Figure 56: Screenshot of 1-2-tree. The viewport is used for selecting and editing branches directly, and the parameters are sorted into categories in a pane on the left.

5.1 Using 1-2-tree

We propose the following *user interface*: When the user starts the system, the trunk for a new tree is displayed. Branches are added by setting the number of child branches to nonzero or by *sketching* them in the viewport. All branches can be edited directly in the viewport or by adjusting their parameters. For capturing the uniqueness of specific trees, it often makes sense to model the trunk and some of the main branches individually, but as modeling a tree progresses, minor branches and twigs are usually best edited using mass updates. Leaves are usually added as a last step, as they hide the tree's inner structure. Figure 55 demonstrates the workflow, and Figure 56 presents the user interface.

5.1.1 Selection

The user may select branches in the viewport. In order to perform *mass updates*, further branches can be added to the *selection* using the following selection modes:

- *Same Level*: Selects all branches on the same *branch level*,
- *Recursive*: Adds the selected branches' children to the *selection*.

If both options are active, changes affect all branches on the same level and below the currently selected branch. If the trunk is selected and recursive selection is active, editing operations affect the entire tree.

There may be several kinds of child branches on a branch. For the trunk, there may be a few main branches, many small branches near its top, and the roots can be modeled as branches, too. The user may define groups of branches by assigning *tags*, and may limit updates to branches that carry the selected branch's tag. With properly assigned tags, it is possible to edit only the main branches, for example.

Since all editing operations are applied to all selected branches, the selection modes permit the user to perform powerful editing operations in an intuitive manner.

5.1.2 Viewport Tools

Each branch is defined by *control points*, *radii* and parameters, which are stored in the branch's scene graph node. The following tools may be used to manipulate the control points in the viewport:

The “*Move*” *tool* allows moving branch control points. All following control points on the same branch, its child branches and all branches in the *selection* are translated by the same vector. Another viewport tool interactively rotates the currently selected branches against their parents in the viewport. The parameters “Length” and “Angle against Parent” are updated for the entire selection.

The “*Sketch*” *tool* allows for *sketching* branches directly in the viewport. The user moves the mouse over the parent branch to select a starting point and then sketches the new branch. After sketching, the new branch lies in a plane perpendicular to the viewing direction, but the user can move the control points from a different perspective. Alternatively, depth information could be deduced from a shadow sketch [30].

The currently selected branch can be deleted by pressing the DEL key. The user may delete the trunk in this manner and sketch a new one. The “*Saw*” *tool* can be used to cut off branch parts at the mouse position. It was implemented as a replacement for pruning.

5.1.3 Parameters

While sketching and moving control points work well for editing individual branches, editing many branches at a time can often be accomplished more easily by manipulating parameters. In order to increase variety during *mass updates*, we created a simple notation that allows the user to combine parameter increases or decreases along the parent branch with a noise function. This is a simplified form of positional information [129]. For describing the notation, we again use *Backus-Naur form* [6]. The notation $\langle a \rangle - \langle b \rangle$ increases or decreases the value along the parent branch linearly from a to b , where the user inserts numerical values for a and b . For example, $2 - 1$ means that the parameter varies from 2 down to 1 along a branch. In order to introduce random variations, a parameter can be stated as $\langle c \rangle_u \langle d \rangle$, meaning a uniform distribution with an expected value of c and a maximum deviation of d . Both notations can be mixed: $\langle a \rangle_u \langle b \rangle - \langle c \rangle_u \langle d \rangle$. This notation allows to interpolate between randomly distributed values, therefore we call it an *interpolated distribution* for short.

Our parameters are similar to those of Weber and Penn [159], except that we store individual parameters for each branch, and parameters are sorted into categories. Category “*Branch Parameters*” defines basic branch properties, such as the branch’s length and its number of control points. “Gnarl” defines the maximum random deviation from a straight branch. “Gravitropism” is a gravitational effect bending a branch down or up. “Subdivision Steps” gives the number of subdivision steps used to produce smooth branches. “Complexity” is the number of polygons to create for the generalized cylinders composing the branches. “CPS” allows assigning the currently selected branches to a different child parameter set, using numeric tags. This is useful to protect branches from changes.

The category “*Radius Parameters*” contains parameters for editing the branch radius and buttons for calculating radii automatically. There are parameters for the radius at the start (“Radius”) and at its end (“Min. Radius”). The button “Calculate Radius” calculates radii from “Min. Radius” using da Vinci’s law for branch radii [68].

The category “*Child branches*” governs child branches. Its parameters are the “Number of Branches”, and “Anchor values”. It is possible to configure these parameters for several separate child parameter sets (CPS).

The category “*Angles*” pools parameters that model a branch’s angle against sibling and parent branches. These parameters can interpret interpolated distributions. The first value describes the angle for the first branch child, the second angle is assigned to the last branch. The angles for all other branches are linearly interpolated between these. The category “*Leaf parameters*” contains parameters similar to the parameters explained before, but for leaves. We calculate the leaf’s breadth from the leaf length and the texture’s aspect ratio, so a “Leaf Breadth” parameter is not needed.

Category “*Bark*” allows creating a simple noise texture to use as bark. Small green branches can be used to produce the needles for a conifer. In that case, “Control Point Count”, “Complexity” and “Subdivision Steps” should be set to minimum values to reduce the number of polygons created.

For many trees, smaller branches and leaves are concentrated near the convex hull of the tree. We approximate this effect by introducing the “anchor values” parameter: Every branch’s anchor value selects the control points to use for placing the branch. The starting point \mathbf{q} for a child branch with anchor value a and n parent branch control points p_0, \dots, p_{n-1} is computed by interpolation

$$\mathbf{q} = \begin{cases} \mathbf{p}_k + (j - k)(\mathbf{p}_{k+1} - \mathbf{p}_k) & \text{if } 0 < a < 1 \\ \mathbf{p}_0 & \text{if } a \leq 0 \\ \mathbf{p}_{n-1} & \text{if } a \geq 1 \end{cases} \quad (8)$$

where $j = a(n-1)$ and $k = \lfloor j \rfloor$. The user may restrict branches and leaves to the outer areas of a tree by specifying an interval for the anchor values. For example, assigning “0.5-1” as an anchor value will place branches only on the outer half of a branch.

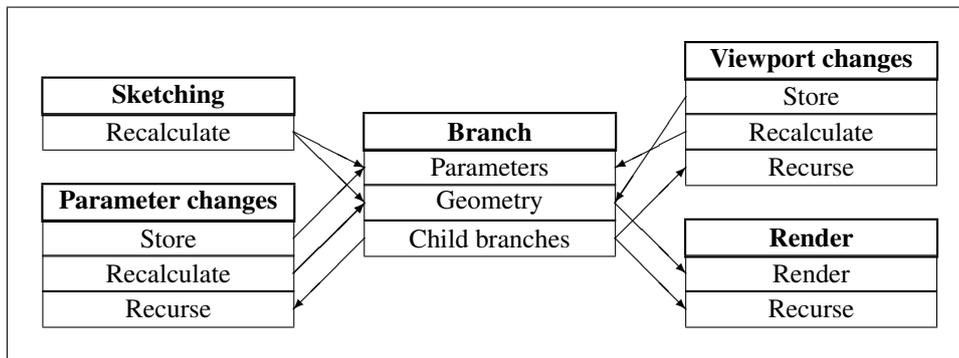


Figure 57: Processes affecting the tree's central data structure: scene graph node type Branch

5.2 Implementation

Figure 57 summarizes how moving control points, sketching, changing parameters, and rendering affect scene graph nodes for branches. For viewport changes, the geometry of the current branch is updated, new parameters are deduced, and the deduced parameters are applied to all currently selected branches. While the user is sketching, 1-2-tree recalculates the parameters and vertices interactively. When the user changes a parameter, the geometry and parameters are updated recursively. Rendering requires a recursive traversal of all scene graph nodes. Textures are compiled to *texture atlases*.

5.2.1 Selection in the Viewport

Clicking small branches that occupy only few pixels in the viewport may be difficult for the user, because the mouse driver does not notify the system of every pixel the cursor runs through. Even if the user clicks the right pixel, it is possible that the ray through the pixel center does not intersect a polygon that was rendered into the pixel. Moreover, the user may not want to zoom into a single branch, because he wants to view changes to a number of branches. In these cases, it is easier for the user to move the cursor to cross the branch instead of clicking it directly. The algorithm detects the selected branch by intersecting a *mouse polygon* formed from the *mouse rays* at the current and last known mouse positions with the geometry. A mouse ray was defined in Section 4.4.1 as the line segment that starts at the mouse position on the near clipping plane, that ends or passes through the mouse position on the far clipping plane.

5.2.2 Parameter Updates

There is an update function for every parameter that optionally allows for recursive application. The new value, the start level for applying the change and the stop level have to be passed as parameters for the update functions. Changes are applied to branch levels that equal or exceed the start level, and recursion continues until we reach the stop level. In order to change only a single branch, the parameter setter is called for that branch, with start and stop levels set to the branch's level. For applying the changes to all children of a branch recursively, the function that changes the parameter is called

for that branch with the stop level set to a large number. In order to apply changes to all branches on a single level, the function changing the parameter is called for the trunk and the start and stop levels equal the level of the currently selected branch. Hierarchy traversal is unaffected by tag checking, but parameters are updated only for branches with the specified tag. After a branch was moved or changed as a result of moved control points or changed parameters, all of its child branches need to be adjusted. The current branches and their child branches have to be translated to new positions. Parameter “length” must be recalculated.

Changing the parameters “Angle against Parent/Sibling” requires recursive application of a rotation matrix. The rotation axis \mathbf{r}_s for the parameter “Angle against Sibling” is given by the parent branch’s control points \mathbf{p}_i before and after the anchor value,

$$\mathbf{r}_s = \frac{\mathbf{p}_i - \mathbf{p}_{i-1}}{|\mathbf{p}_i - \mathbf{p}_{i-1}|}, \quad (9)$$

and the rotation axis \mathbf{r}_p for “Angle against Parent” is perpendicular to the first two control points $\mathbf{b}_0, \mathbf{b}_1$ of the branch and the difference vector of the parent branch control points before and after the branch,

$$\mathbf{r}_p = \mathbf{r}_s \times \frac{\mathbf{b}_1 - \mathbf{b}_0}{|\mathbf{b}_1 - \mathbf{b}_0|}. \quad (10)$$

Attempts to set “Angle against parent” to 0 are ignored.

5.2.3 Viewport Tools

Whenever any branch’s control points are moved, its child branches and the following control points on the same branch must be moved as well. Otherwise, the child branches would be torn off their parents and hover unsupported. If the branch’s first control point is moved, its anchor value is recalculated. The child branches’ anchor values a_j allow to decide quickly which branches need to be moved when control point p_i is moved. These are the branches b_j with anchor value a_j where $a_j n > i - 1$ for a branch with n child branches. All selected branches’ control points are moved by an equal distance, and their “Length” parameter is recalculated.

For sketching, the start point for the new branch is selected with the same algorithm as for *viewport selection*, as described in Section 5.2.1. After that, all mouse positions are stored in an array and branch control points are selected equally spaced from the array in every frame. This allows to render the new branch interactively during sketching.

5.2.4 Geometry Creation and Rendering

When creating new branches procedurally, the direction of each new branch segment is calculated from the formulas

$$\mathbf{d}' = \mathbf{d} + \frac{c_{gn}}{c_{cpc}} \cdot (R_1 \cdot \mathbf{n}_1 + R_2 \cdot \mathbf{n}_2), \quad (11)$$

$$\mathbf{p}' = \mathbf{p} + l \cdot \frac{\mathbf{d}'}{|\mathbf{d}'|}, \quad (12)$$

where

\mathbf{d}' :	new direction,
\mathbf{d} :	previous direction,
c_{gn} :	gnarliness parameter,
$R_1, R_2 \in \mathbb{R}$:	uniformly distributed random variables, $-1 < R_1 < 1$, $-1 < R_2 < 1$,
$\mathbf{n}_1, \mathbf{n}_2$:	normals to \mathbf{d} ,
\mathbf{p}' :	new point coordinates,
\mathbf{p} :	old point coordinates,
l :	the length for the new segment,
c_{cpc} :	number of control points already in the branch.

In Equation 11, dividing by c_{cpc} reduces the gnarliness along the branch.

The modified butterfly scheme is used [44] to produce smooth branch curves from the control points. It guarantees that the branch curve contains the control points. Then we compute *generalized cylinders* for the branches from the vertices and radii. The polygons are stored in the scene graph node.

Leaf polygons are managed and stored in their parent branch. Leaf texels are interpreted as fully transparent or completely opaque. This allows us to use alpha-testing rather than the more expensive depth sorting for rendering the leaves.

5.2.5 Persistence

In procedural modeling, the persistence problem refers to the difficulty of retaining user edits after parts of the model were changed [19, 86]. A naive approach to change the number of branches on a parent branch could use one of the old branches as a *template* for creating the new branches. However, if one of the deleted branches was edited with individual parameters, these parameters would be lost. Instead, if the number of branches is reduced, some branches will have to be deleted, but the branches closest to the new positions should be moved to new positions to keep user-edited parameters. If the number of branches is increased, existing branches are moved to new positions, and their parameters are used to produce further branches. The positional information stored in the branches can be used to recreate geometry for the branches at their new position without losing user edits. Assigning tags further reduces the impact of changing the number of branches, because it allows to change branches with a specified tag.

5.2.6 File Formats

Each branch has two descriptions. The parameters, radii and branch control points reflect a high level representation of the tree, from which geometry can easily be recreated. The lower-level representation of the tree consists of the tree's geometry. The branch node's duality of parameters and geometry is reflected by two separate storage formats. When the user saves a tree to a file, he can store its parameters, control points and radii in an XML file. This format retains the scene graph hierarchy and allows changes later. After loading the file, geometry is recomputed from the control points and radii.

For data export to other 3D applications, we chose the OBJ file format for its simplicity and widespread support. While the format cannot reflect the hierarchy of semantic entities, the output is optimized for rendering: All textures are compiled into an atlas, and all vertices use a single coordinate system.

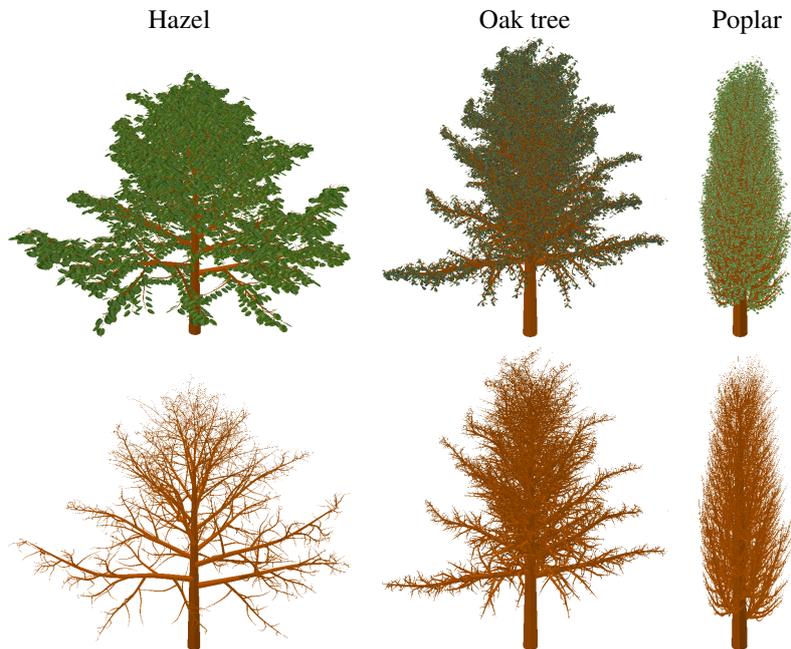


Figure 58: Example trees I

5.3 Results

Table 4 (page 101) lists memory usage and time needed to model the trees shown in this section. The amount of memory needed to store the geometry usually exceeds the memory needed for the parameters by a factor of at least 2. Given today's memory sizes, the added cost of individual parameters for each branch is affordable for editing single trees. Figures 58, 59, and 60 show some results obtained using the proposed user interface.

At minimum *level of detail*, a branch is isomorphous to a tetrahedron. While a tetrahedron has four points, OpenGL's `glDraw` command forces us to store five points, because the texture coordinates for the first and last points are not the same and rendering using `glBegin/glVertex/glEnd` incurs a performance penalty we wanted to avoid. In double precision, we require $5 \cdot 3 \cdot 8 = 120$ bytes for storing the vertex positions, or 60 bytes in single precision. Normals require the same amount of memory. Texture coordinates require $5 \cdot 2 \cdot 8 = 80$ bytes in double precision, or 40 bytes in single precision. The indices into these data buffers require $3 \cdot 3 \cdot 4 = 36$ bytes. Summing up, we need $2 \cdot 120 + 80 + 36 = 356$ bytes in double precision, and $2 \cdot 60 + 40 + 36 = 196$ bytes in single precision. The parameters require 368 bytes, not counting the 11 external buffers for parameters in string representation, which have variable length. In double precision, the parameters require about as much memory as the geometry at its lowest resolution, and we usually aim for a higher level of detail. So the parameters double memory consumption only in the worst case, and we think that the benefits of storing individual parameters outweigh the cost.

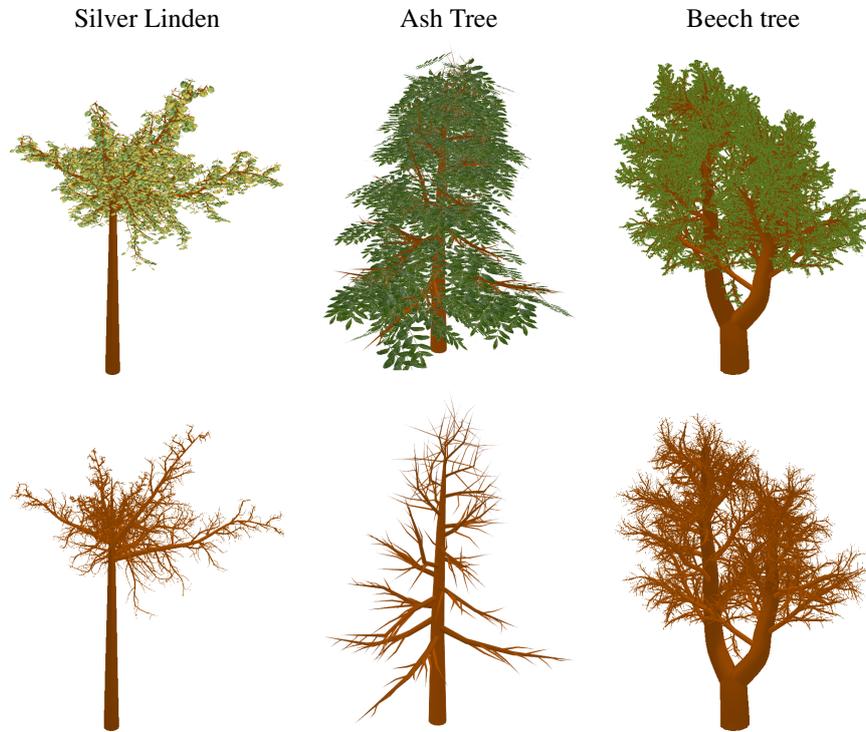
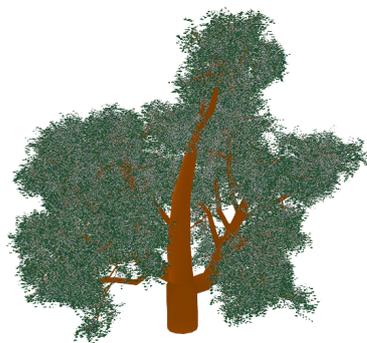


Figure 59: Example trees II. For the ash tree, the final level of branches was modeled as a texture.

Species	Modeling Time / min	Branches	Leaves	Branch Levels	Geometry / MB	Parameters / MB
Hazel	4:58	2,221	17,939	4	21.9	1.1
Oak	5:03	9,931	57,228	4	22.3	4.7
Silver Linden	7:33	2,211	7,048	4	21.7	1.1
Poplar	3:29	6,631	27,483	4	32.4	3.1
Ash Tree	4:37	221	680	4	0.7	0.1
Beech Tree	5:57	7,517	187,000	5	23.8	3.5
Maple	ca. 25	34,001	393,822	5	75.5	15.7
Birch Tree	8:37	23,011	99,190	5	114.0	10.9
Plane Tree	ca. 9	14,373	204,138	5	18.8	7.0

Table 4: Statistics for creating the trees in Figures 55, 58, 59, and 60

Maple



Birch Tree

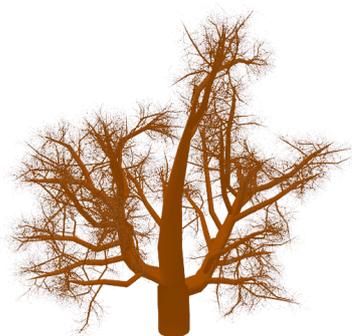
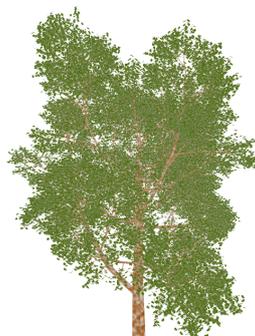


Figure 60: Example Trees III

5.4 Discussion

In this section, we compare 1-2-tree to other tree modeling systems. L-systems are a long success story in modeling plants, but we preferred to operate directly on a scene graph for a number of reasons. First, the works of Weber and Penn [159], Lintermann and Deussen [84] demonstrate that viable alternatives to L-strings exist. Secondly, 1-2-tree's editing functions build on hierarchy traversal, which can be implemented more efficiently on hierarchical data structures than on an L-string. Thirdly, while any alteration requires completely parsing and copying the L-string, updates are very efficient with a hierarchical data structure because only the geometry of the affected scene graph nodes needs to be updated. However, the editing approaches presented in this chapter could be applied to trees created by L-systems.

L-systems have to build and parse the L-string every time geometry is created, whereas we parse user input once on assignment and then use it stored form. That forces us to store the string representation and numeric value, but this cost can be afforded given today's large amounts of memory. Furthermore, L-systems require twice the memory of the L-string during parallel rewriting, as the old L-string has to be stored separately from the new version.

While the algorithm by Weber and Penn stores parameters for every branch level [159], 1-2-tree stores similar parameters for every single branch, and 1-2-tree can be configured to apply changes to all branches on a single level. This is only one of several modes supported by 1-2-tree, but arguably a very useful one. However, 1-2-tree also allows to model branches individually. Whereas Weber and Penn's system allows varying parameters only randomly, we use interpolated distributions for positional information, which allows mixing random influences with parameter increases or decreases along the branch.

Xfrog provides node types with numerous parameters. The variety of plants that can be modeled with *Xfrog* may be comparable to the power of L-systems. *Xfrog* stores the rule system describing a plant in the so-called p-graph. *Xfrog*'s algorithm for geometry creation takes the p-graph and a separate exception list as input to create the model's geometry. These three data structures are integrated into a single data structure in 1-2-tree, which reduces computation time and algorithmic complexity. 1-2-tree specializes in modeling trees rapidly and individually. In contrast to *Xfrog*, users of 1-2-tree do not require knowledge about loops and mathematical formulas. 1-2-tree's *selection modes* and *viewport editing* allow for more intuitive modeling, because 1-2-tree relieves the user of finding the p-graph node that produces a certain effect.

The system by Boudon et al. stores individual parameters for each branch in the *decomposition graph* and the user may inherit branch properties from parent to child branches [19]. In order to edit branches, the user has to locate the branch in the decomposition graph and edit its parameters. By contrast, 1-2-tree allows to model branches directly in the viewport. When the user clicks a branch in the viewport, its parameters are displayed immediately. Thus, 1-2-tree uses the viewport as a replacement for the decomposition graph. The user may copy changes to other branches by intuitive selection settings rather than using inheritance settings.

Like 1-2-tree, the system by Boudon et al. stores individual parameters for each branch [19]. In 1-2-tree, copying changes to other nodes is governed by intuitive selection settings rather than inheritance settings. 1-2-tree integrates the functionality of the *decomposition graph*, and the branching structure browser into the 3D viewport to further ease modeling: Instead of browsing the decomposition graph, the user simply clicks branches in the viewport to select them and to display their parameters.

The system by Okabe et al. [109] is orthogonal to 1-2-tree. It focuses on sketch- and example-based modeling, whereas 1-2-tree focuses on procedural modeling and direct editing methods. Both approaches have their advantages, and future users might want to have a system that integrates the strengths of both systems.

Boudon et al. report an average modeling time of 3 hours for each model [19], and the models by Okabe et al. took less than 10 minutes on average [109]. In 1-2-tree, most models take between 4 and 10 minutes to create.

1-2-tree differs widely from plab. While plab is a generic visual procedural programming language, 1-2-tree is an editor for trees. While model graphs are interpreted scripts that can be changed flexibly, the algorithms in 1-2-tree are hard coded and run as natively compiled code on the CPU. This increases performance at a certain cost of flexibility, but it also eases editing individual branches. In plab, when instances of objects that are created in a loop in the model graph need to be edited individually, this requires editing the model graph directly. 1-2-tree conveniently supports editing individual branches, as all scene graph nodes store a full parameter set by default. Both systems use scene graphs as a core data structure. In plab, the scene graph is recreated fully when the model graph is executed. In 1-2-tree, the scene graph is updated only when and where necessary, which also improves performance and reduces flickering.

5.5 Conclusion

We tried to balance the various approaches and use sketching to model individual branches. We feel that deriving parameters from viewport input and applying the derived parameters to other instances nicely combines procedural modeling and sketch-based interfaces. We focused on creating a system that gives the user maximum control over the process of modeling trees without requiring him to learn concepts like loops, grammars and formulas. This reduces the flexibility of 1-2-tree, but greatly enhances the *ease of use*. Simple tree models can usually be created within a few minutes, and arbitrary precision can be obtained by modeling individual branches.

This chapter has demonstrated the benefits of storing branches as semantic entities in a hierarchical data structure for modeling trees. Among these benefits are the powerful *selection options* for changing single entities or masses of branches. Moreover, we have shown that this data structure can be used to propagate viewport changes back to the parameters and vice versa.

6 Editing 3D models using templates

Hardware capabilities keep growing at an exponential rate, and graphics hardware can handle increasingly complex scenes. While 3D modeling suites offer convenient tools to manipulate shapes, managing the primitives is left mostly to the user, which can be cumbersome or impractical for large scenes. Furthermore, content producers have to keep costs under control. As a result, there is an increasing gap between hardware capabilities and the amount of detail that is provided in computer games and films. Procedural modeling aims for editing methods that work on a higher level of abstraction, rules that guide how elements are placed and changed by user edits. Unfortunately, the complexity of procedural modeling is an obstacle that prevents wider adaption. This chapter presents methods that are intended to ease wider adaption of procedural methods.

A procedural model can be treated as a black box. While many people do not understand how some technical devices work internally, they are still able to use them if they are presented with a suitable interface. In the past, procedural models have often been made accessible by wrapping them with a suitable interface. Procedural algorithms are often able to create different instances by varying the input parameters. This chapter will explore how such procedural models can be used as *templates*, from which instances are created, and how this can be made easy for the user.

Procedural models are often implemented as external applications, possibly with their own, unusual interface conventions, which certainly contributes to a notion of complication surrounding procedural models. Content that is produced by external applications must be imported into 3D modeling suites, further complicating the use of procedural models. Seamlessly integrating procedural models into 3D modeling applications might allow users to work at a higher level of abstraction with the tools that they use every day.

6.1 Design Considerations

While current 3D modeling suites offer generic tools for editing objects, many objects in scenes belong to categories that can be edited more efficiently using specialized tools. We propose to address the challenge of modeling similar objects using *templates*. An object template consists of several scripts. The *base script* sets up the parameters, geometry and textures for a 3D object. A template optionally contains further scripts that are executed as tools to change certain aspects of the model. These are called *tool scripts*. The user forms concrete objects using the base scripts and customizes them using tool scripts. Object types such as trees, roads, landscapes and street networks require different templates. The controls are integrated seamlessly into the user interface, and hide the details of the implementation, but the user may view the scripts when needed. Our goals include:

- *Stability*: Modeling application crashes should be prevented when possible, as crashes may lead to data loss for the user. Therefore, errors in a script should not crash the modeling application.

- *Performance*: Changes should be possible very quickly, optimally changes would occur immediately.
- *Targetted Updates*: The system should *reproduce* elements only when necessary, to increase CPU efficiency and performance, and also to reduce flickering.
- *Persistence*: Updating only the parts of a model that are affected by a change is also a step towards persistence, which asserts that user edits are maintained unchanged as long as possible. When a parameter is changed, all other parameters should stay unchanged.
- *Customizability*: Advanced users should be able to customize every aspect of creating and editing the objects. This can be achieved by using and editing scripts, rather than hardcoding the procedural system.
- *Ease of Use*: Instead of programming changes to the scene in a procedural model, the user should be able to use the same modeling metaphors for creating and transforming objects that are generated procedurally as he would for objects that are normal part of his modeling solution.
- *Viewport Editing*: Changes would best be made in the viewport immediately.
- *Sketch-based modeling*: Touchscreens have proven to be a versatile and intuitive modeling tool. A scripting language capable of constructing models and altering them using sketched inputs could turn out to be a valuable addition to the tool set.
- *Accessibility*: The scripts should give the user the artistic freedom he needs.
- *Intuitive Tools*: At the same time, the user should know at all times how to achieve any desired effect, and the system's limitations.
- *Semantic Selection*: The user should be able to select objects based on their properties. Updating only selected scene graph nodes also ensures persistence of unaffected items.
- *Context-sensitive tools*: The user should only be presented with tools suitable for the selected scene graph nodes.

Semantic information includes information on how to recreate geometry for single objects in the scene graph, which tools could be applied to the object, its parameters and child objects. Semantic information may be assigned directly by the user or automatically by model graphs.

6.2 Implementation

We chose to demonstrate this approach by integrating *plab* into *Cinema 4D* as a plugin called *plabC4D*, but different combinations of scripting languages and 3D modeling suites could be used instead. *plab* was designed to offer a good compromise between performance, *ease of use* and learning, interactivity, and power of expression. Table 2 (page 38) demonstrates that visual scripting languages are in widespread use in computer graphics. In *plabC4D*, base scripts and tool scripts are model graphs. This text uses *base model graph* and *tool model graph* as synonyms for *base script* and *tool script*.

6.2.1 User Interface

We approach the design of templates from the user's perspective. Using a 3D modeling suite, an artist's approach to modeling a house might be to assemble buildings from predefined textured parts. Optionally, in order to further individualize the building, some parts could be edited manually. How can a computer better assist these steps? First off, the user would state that he intends to place a house. Knowing that the artist is modeling a house allows an editor to make several assumptions. For example, that the sides of the building stand vertically to the ground, and the polygons for the roof can be computed automatically as demonstrated in Section 4.6.2. Going further, the system could assume a rectangular building layout and roof type, and simply ask for the side lengths and height of the building. The type and parameters of the house, material properties, types of windows, balconies would be chosen next, and the procedural system would be charged with generating the details and geometry, possibly depending on the viewing distance. The user should be able to execute mass updates to install similar balconies or roofs on a selected number of buildings.

Figure 61 shows an image of *Cinema 4D* with the *plabC4D* plugin. Procedural content is added to a scene in a 3D modeling suite by instantiating a *template* node in the suite's scene graph (1) and by associating a base model graph with the node (2). The *base script* is responsible for setting up the object's variables, creating initial geometry, and it lists the *tool scripts* that can be used to edit the generated object. The user executes the base model graph to obtain geometry for the node (3). Before the base model graph or the tool model graph is run, the user may select the parameters for the model graph (4). After the model graph is executed, *Cinema 4D* displays the geometry that was created (5). Tool model graphs may be executed in order to edit the node and its geometry. Objects are selected by clicking them in the viewport. Future implementations could support *control points*, and altering the control points or editing the attributes would invoke a script to recreate geometry for the selected object and possibly its child objects. After moving the control points, parameters of the object would have to be updated, and after editing parameters, the control points would have to be updated, as demonstrated in Section 5.

Every instance of a template is stored as a separate node in *Cinema 4D*'s scene graph. Each instance has its own geometry. This has the advantage that for any editing operations, only the geometry of the affected instance needs to be updated. *plab* allows strict control over normals and texture coordinates.

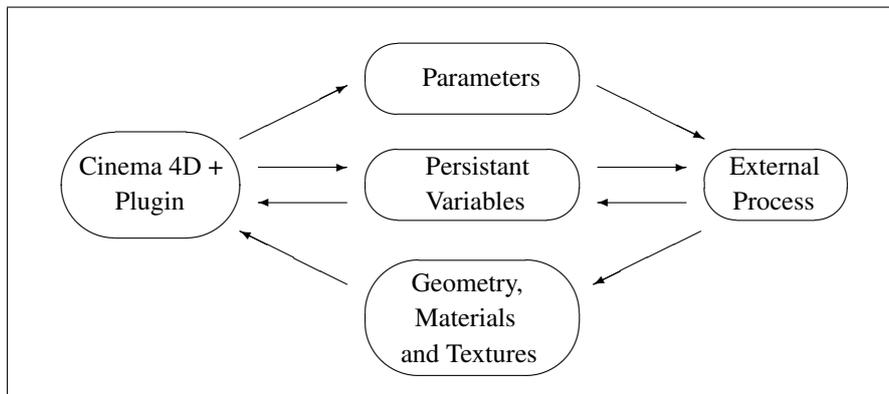


Figure 62: Inter-process communication between the plugin and the external process

- **Geometry and textures** are generated and exported by the external process. The plugin loads the geometry and texture for display.

Figure 62 demonstrates how the plugin and the external process communicate using the file system. These files are stored in a folder for temporary files and each plabC4D node uses its own set of temporary files, to prevent different instances of a single object *template* from interfering with each other. For running a model graph, its parameters are written to the parameters file. The external process loads the parameters from that file, runs the model graph, and writes new geometry to another temporary file. The name of the variables file is passed as an optional argument - a model graph may read and update the file as needed. After execution of the model graph completes, the variables are read into memory. This has to be done in order to pass the variables to other tool model graphs and in order to store them to a file when the scene is written to file. When the model graph completes execution, plab stores the geometry, material and texture to an external file, and these files are read by plabC4D.

When the scene graph in Cinema 4D is stored in a file, any plabC4D nodes must be stored as well. As in Section 5.2.6, there are two main representations for a plabC4D node – its metadata and geometry. For plabC4D nodes, both representations are written to disk. Stored metadata include the base model graph associated with the plabC4D node and its variables. Cinema 4D stores additional data for each object node, such as the object’s transforms. A model graph may use random values to produce geometry, so running the same model graph may lead to different results. This may be undesirable when reloading a scene, because the user expects to see the exact same geometry as before the file was stored. Storing the geometry in the file and loading it later, instead of recreating the geometry, prevents this. Loading and storing the geometry often takes less time than recreating it in an external process.

6.3 Results

The plugin allows to execute all model graphs in Cinema 4D, but plab's control points are currently not fully implemented in Cinema 4D. Control points may be edited in plab, and plab uses the positions of the control points stored in the model graph when executed via plabC4D, but plabC4D currently does not allow to interactively move the control points directly in Cinema 4D. Templates have been created that aid in modeling architecture and terrain, and model graphs for plants and trees can also be used as templates. However, a model graph used as a template lacks tool model graphs.

6.3.1 Architecture

A *template* for generating simple building models has been created. The base model graph is `house/base.mog`. Its parameters are the number of window elements on the front and sides of the building. Furthermore, a base model graph defines the tool model graphs that can be used with the template. `EditFront.mog` can be used to alter several window elements at once, for example, to replace them with balconies. `Edit single front element.mog` allows to replace single window elements, for example, to add an entrance to the building. This would also allow to use different window styles or to replace a window with a wall. Certainly, more complex editing operations would be required in order to use this template in a productive environment. For non-rectangular buildings, several solutions are possible. The first solution is to adapt the model graphs from Section 4.6.2. Another solution is to provide special base model graphs for L-, T-, H-, and +-shaped buildings. These could be altered using control points. A third solution is to leave modeling the base shape entirely to the user. The user would provide a low-polygon model, then the user would choose polygons to replace with window and door tiles.

6.3.2 Terrain

A template has been created that includes the model graphs presented in sections 4.6.3, 4.6.4 and 4.6.6. `terrain/base.mog` creates a plane terrain that is used as the basis for further editing. `SlopingHills.mog` creates a number of hills. `erosion-native.mog` simulates erosion on this terrain. `smooth-landscape.mog` can be used to smoothen the surface after these computations. As with buildings, this template currently does not support interactive editing in the viewport, but integration with Cinema 4D eases running the model graphs, compared to a workflow based solely on plab. Figure 63 shows an example of terrain created in this manner.

6.3.3 Trees and plants

For trees and plants, the model graphs discussed in Section 4.6.1 can be used, but they have not yet been updated to work as templates. Additional programming would be needed to properly support this. A single model graph would have to support creating several Cinema 4D nodes, and these new nodes would have to support loading their variables either from a common file, or plabC4D would have to support creating these files. Therefore, 1-2-tree is currently the better solution for editing trees.

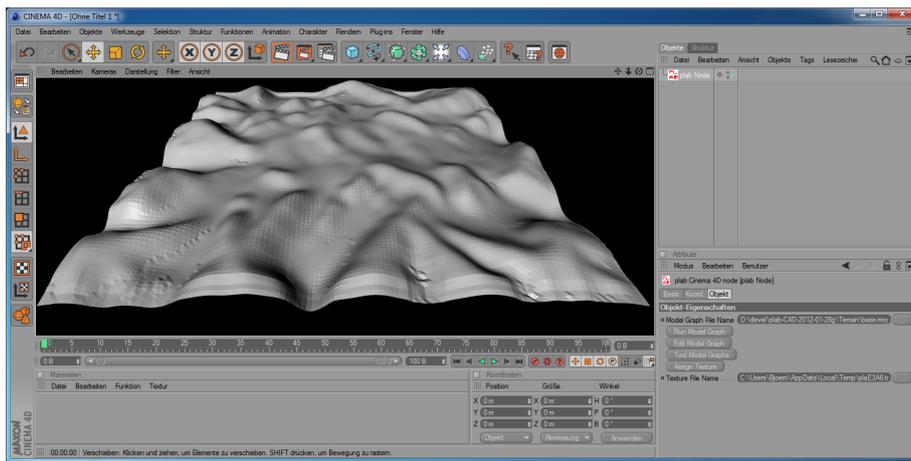
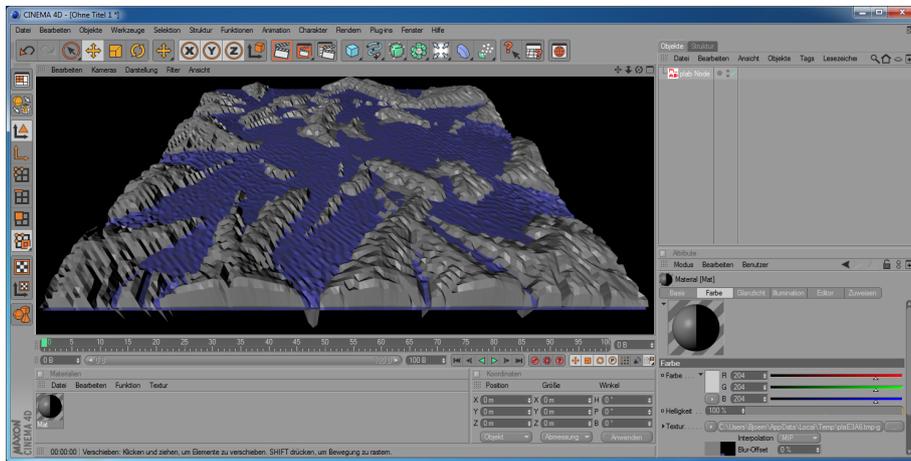
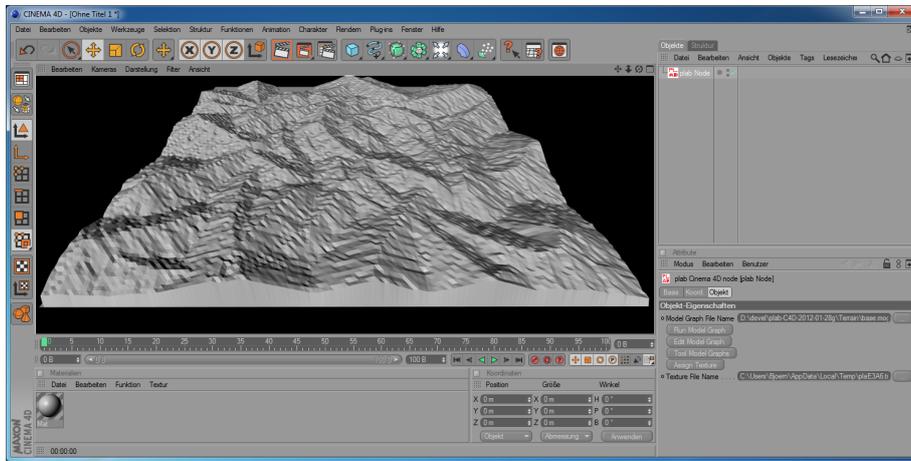


Figure 63: plabC4D Template for creating terrain. Top: SlopingHills.mog was used to create hills. Middle: erosion-native.mog was used to simulate erosion on this terrain. Bottom: smooth-landscape.mog was used to smoothen the eroded terrain.

6.4 Discussion

In this subsection, the goals defined in this section's introduction are compared with the results. *Stability* was ensured by running *SceneFactory* in an external process. This allows the Cinema 4D plugin to recover from crashes in the external process, reducing the risks of data loss due to application crashes. While the scripting language and the inter-process communication incur some overhead compared to a plugin that is executed natively in the context of the Cinema 4D process, mostly the *templates* process small amounts of data where this overhead is acceptable because the updates target specific small instances. Each template knows which tools can be used with it, therefore the tools are automatically *context-sensitive*. *Ease of use* is ensured by encapsulating the code for the templates with a simple GUI. When the user still needs to edit the scripts, the visual scripting language was specifically designed for ease of use.

Of course, there is still room for improvement. *Sketch-based modeling* could be integrated. For this, additional tools would have to be implemented that allow to alter shapes based on this type of input. Also, *semantic selection* has not been implemented. It could be implemented by allowing to query for instances of templates that have certain values for variables. *Targetted updates* ensure that selected objects are affected by editing operations, while other objects are not touched (*persistence*). This was achieved in *plabC4D* in a similar fashion as in *1-2-tree*: By updating only those nodes in the scene graph that are actually affected by the changes. After the user selects a node by clicking it, only tool model graphs relevant for the node type are displayed. This ensures that templates are *context-sensitive*.

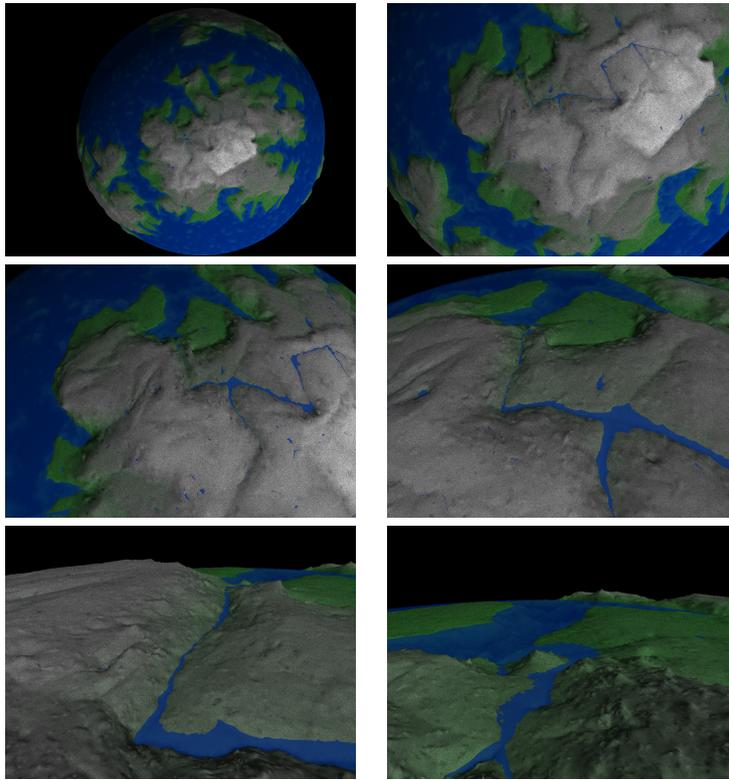


Figure 64: Zooming into a planet created using the adaptive procedural technique

7 Procedural Terrain with River Networks

Movies, simulations and computer games allow to explore a wide variety of realistic, fictional terrains. In some cases, using real terrain would break the illusion of exploring unknown planets. For this reason, the computer games *Spore* (2008) and *Civilization* (1991) employ procedural models to generate realistic planets from a set of rules automatically, as discussed in Section 3.5.3. While the procedural models may use numerous parameters, default values and help texts allow the user to tweak the planets as desired with minimal effort. In recent years, these algorithms were improved to interactively adapt the geometry to a moving camera, in order to support view-dependent *level of detail*.

While terrain can be generated quickly using previous procedural models, these terrains lack realistic rivers. Rivers are vital for life, and can be important for navigation, as rivers often lead to communities, or to the sea. Erosion simulations model the natural processes that form rivers, as discussed in Section 3.2.12. Unfortunately, these algorithms are computationally expensive and can therefore not be used to generate a locally adaptive, high resolution landscape during a fly-through. In this work, we propose a novel algorithm that allows to generate realistic rivers at adaptive *level of detail* with minimal preprocessing. Instead of attempting to recreate the physical processes of erosion, we aim for *river networks* that obey the following observations:

- Precipitation is transported by the river network along the steepest decline until it reaches the sea or an endorheic basin.
- River networks are surrounded by valleys between mountains and hills.
- Rivers do not cross and they are mostly above ground.

We use these criteria to define consistent river networks. Our new procedural algorithm creates complete planets and landscapes with plausible river networks without performing an erosion simulation, rendering it suitable for on-the-fly generation of terrain. The algorithm starts by creating a coarse representation of the terrain. Additional geometry has to be produced in order to locally adapt the geometry to the camera position and perspective as the user traverses the terrain. This must be done in a manner that is consistent with the plausibility constraints named above. In summary, the main contribution of this section is a novel algorithm that combines the following properties:

- **Adaptive level of detail:** We store terrain metadata in the edges and vertices of a mesh and describe rules to refine terrain based on that information. As a result, the algorithm generates river networks at adaptive levels of detail.
- **Plausible river networks without an erosion simulation:** Water levels are computed directly and rivers carve into the landscape without an iterative simulation of water movement. Plausibility of the river networks is maintained while adapting the level of detail.
- **Fast terrain synthesis:** The algorithm allows us to generate a base mesh in less than a second and to process refinements in real time.

This enables us to make the following **main contribution:** We generate planets with eroded terrain and plausible river networks at adaptive level of detail with minimal setup time. Figure 64 shows an example. The following section describes the author's contributions to the paper by Derzapf et al [39]. Section 7.6 describes additional material that went into that paper.

7.1 Analysis

A system designed to generate terrain that can be explored immediately would have to meet the following criteria:

- **Fast precomputation:** Exploring the planet should be possible without delay.
- **Minimal user interaction:** At most, the user can be expected to give a general description of the planet.
- **Plausible River Networks:** River networks and mountains can be vital clues for navigation. A simulated populace could rely on the river networks for drinkable water and transport of goods.
- **Adaptive Level of Detail:** As the user explores the planet, the level of detail of the terrain has to be locally adjusted to the camera position. Therefore, the procedural model must be capable of adding geometry while satisfying plausibility constraints for river and mountains.

Algorithm/ Authors	Input	Output	Precomputation	Water Effects	Level of Detail
Midpoint Displacement	Parameters	Mesh	Very fast (Create tetrahedron)	Global sea level	Adapted inter- actively
Kelley et al. [76]	Parameters	Mesh	Very fast	Plausible river networks	Fixed
Erosion Simulation	Grid	Layered Grid [13]	Minutes or hours, see Section 4.6.6	Various	Fixed
Constrained Modeling	Constraints	Grid	2.97s for 1024^2 samples [8]	Global sea level	Limited
Transplant Terrain Detail	Two grids	Grid	2.5 s for $2794 \times$ 394 samples [20]	Global sea level	Fixed
Hnaidi et al. [66]	Control curves	Grid	0.3 s for 1024^2 samples	Rivers, consis- tency depends on user input	Limited
Proposed algorithm	Parameters	Mesh	Very Fast	Plausible river networks	Adapted inter- actively

Table 5: Comparison of features in previous algorithms.

For dry planets, midpoint displacement is suitable to generate geometry. For planets with large amounts of water, an erosion simulation could be added to control the creation of rivers, lakes and seas. However, the experiments in Section 4.6.6 revealed that very many simulation cycles are required to compute *river networks* for uneroded terrain. Furthermore, the user has to watch over the process and adapt the parameters in order to obtain a plausible result. For computer games, creating visually plausible terrain in a time-efficient manner is more important than accurately simulating erosion.

Table 5 summarizes the features of several algorithms presented in Section 3. For a number of algorithms, the *level of detail* is fixed. In other cases, the level of detail is limited by the functions that are used to define the terrain. At high polygon counts, additional polygons make features rounder but do not add further detail. In these cases, Table 5 reports the level of detail as limited. Midpoint displacement can be used to add further detail to a terrain [8]. Unfortunately, the algorithm lacks the necessary rules to prevent introducing mountain peaks into rivers. Some algorithms do not support river networks. However, water effects are vitally important as river networks are a defining element of natural landscapes. As a result, in related work, either the level of detail is limited, river networks are missing, or producing the terrain takes too long.

Both midpoint displacement and the method of Kelley et al. [76] come very close to satisfying the stated requirements. The one lacks consistent river networks, while the other lacks adaptive *level of detail*. Both methods also operate on a mesh data structure. This chapter demonstrates how these algorithms can be combined into a new algorithm that offers level of detail and river networks that expand as the user zooms into the landscape. A parallel implementation is required to reach sufficient performance for interactive applications.

7.2 Overview

As none of the related algorithms meet the specified criteria, we chose to pursue a fractal approach that consists of two phases. The first phase is *planet creation*, where the algorithm creates a rough representation of the planet to be explored in less than a second. The second phase is called *interactive exploration*, where the algorithm interactively and adaptively refines the terrain while the user moves about freely. This is made possible by combining *midpoint displacement* with semantic information and the algorithm by Kelley et al. for creating *river networks*. The algorithm is designed to create plausible river networks as defined above. It is based on a mesh consisting of triangles and vertices in 3D space.

As our procedural model describes entire worlds, it could be used to fill a fictional universe with planets. In particular, computer games could benefit from the described technique, because every time such a game is played, different planets could be generated. Trade simulations, vehicle simulations, tactical and strategical games could use this technique. Another area of application could be driving training and tests. In many cases, the decision of whether a license for steering a vehicle should be granted is made depending on a test taking place in a local environment, but the license is valid for places that offer dramatically different challenges. In order to increase fairness in driving and flying tests, the tests could take place in a randomly created virtual environment that contains a fair mix of challenges.

Our algorithm stores mountain ridges, coast lines and rivers as edges. While it would be possible to store the planet in a displacement map wrapped around a sphere, only eight directions are possible for transporting water between neighboring cells, and a solution would be needed that can exceed these eight directions when zooming in. Otherwise, parallel rivers would emerge. Instead, we use a mesh to store the terrain, which allows us to vary the vertex positions to prevent such artifacts. A marker in each polygon stores whether the polygon belongs to the sea or to a continent, and edges are typed as sea, coast, river or mountain. If a vertex is incident to at least one edge of a given type, it will be referred to as a vertex of that type. For example, a vertex with a river edge is a river vertex. A single vertex may have incident edges of all the mentioned types, so a single vertex can be a sea, coast, river and mountain vertex at the same time, but a vertex that has only edges of a single type is called a *pure vertex*. A vertex that is incident to a continental polygon is called a continental vertex, or sea vertex if incident to a sea polygon. Thus, coast vertices are both sea and continental. Appendix G lists the parameters used by our procedural system.

The algorithm is designed for planets with sea, continents and *river networks*. These require a balance of a number of physical factors. The planet must have an atmosphere, otherwise the water would either freeze or escape into space. Atmospheric pressure and temperature also have to support water in both gaseous and liquid form. The gaseous form is required for precipitation, which feeds the river system. There must be enough water to form rivers and lakes. For simplicity, this chapter focuses on water, but all liquids follow the same laws of physics. Earth is not the only satellite known to have lakes. Saturn's moon Titan may have precipitation, rivers and lakes composed of methane and ethane.

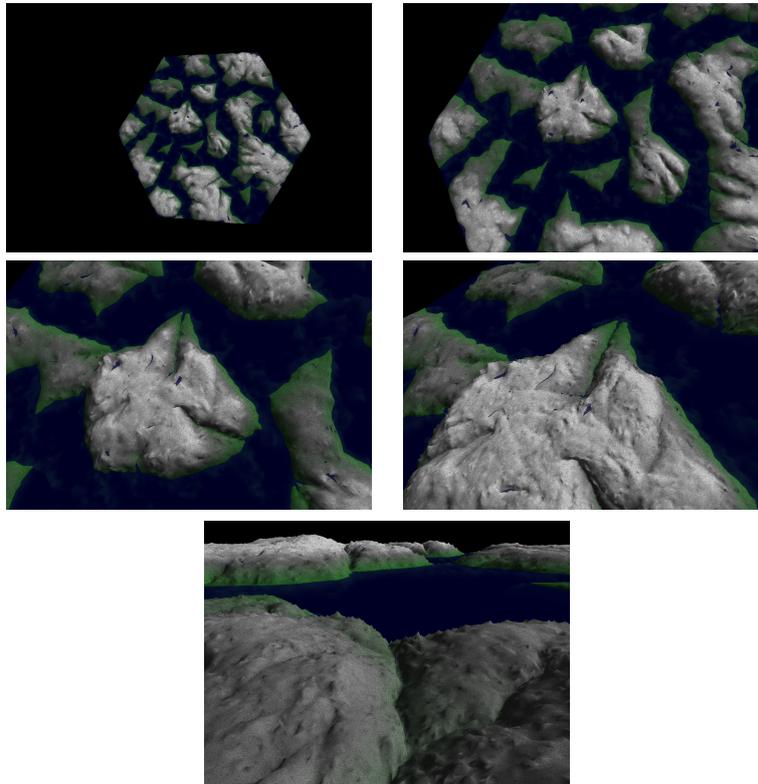


Figure 65: When only part of a planet can be seen, the base shape can consist of planar polygons

7.3 Planet Creation

As we strive to create terrain that can be explored without delay, the *planet creation* phase was designed to complete in a fraction of a second:

1. Create base shape,
2. Define continents: assigns each polygon to either the sea or to a continent,
3. Produce initial river networks: converts edges to rivers so that they form river networks,
4. Assign vertex altitudes,
5. Compute water levels: sums up precipitation for the river network,
6. Assign vertex colors,
7. Create geometry for lakes and rivers.

These preprocessing steps will be discussed in this subsection.

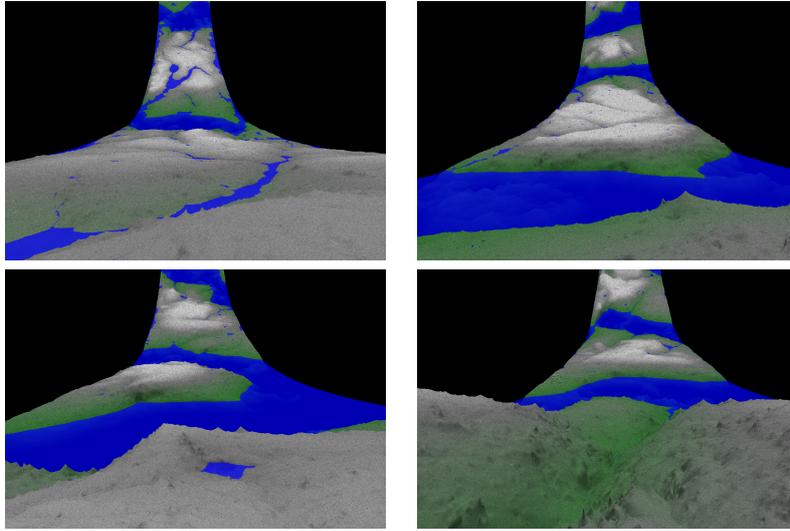


Figure 66: RingWorld [108] - fictional habitats on the inside of a huge ring, where gravity is simulated by the centrifugal force, as a result of rotating the ring. However, unlike gravity on a planet, the centrifugal force pushes outward, therefore such a ring would have to be inhabited on the inside

7.3.1 Create Base Shape

When exploring a planet, the base shape can usually be approximated by a sphere or ellipsoid, but the technique works with other base shapes as well (Figure 65, Figure 66). For a sphere, each new vertex with coordinates \mathbf{v} needs to be lifted to the base shape's surface:

$$\mathbf{v} \leftarrow (r + a) \cdot \frac{\mathbf{v} - \mathbf{c}}{|\mathbf{v} - \mathbf{c}|} + \mathbf{c}, \quad (13)$$

where

- r : the sphere's radius,
- a : the vertex's altitude,
- \mathbf{c} : the sphere's center.

If only a small part of a planet is going to be explored, the base shape could be one or more planar polygons. Positions for new vertices \mathbf{v} are chosen from a randomized weighted sum of the surrounding vertices:

$$\mathbf{v} \leftarrow \frac{r_1 * \mathbf{p}_1 + (1 - r_1) * \mathbf{p}_2 + r_2 * \mathbf{p}_3 + (1 - r_2) * \mathbf{p}_4}{2}, \quad (14)$$

where

- $\mathbf{p}_1, \dots, \mathbf{p}_4$: the vertices surrounding \mathbf{v} ,
- r_1, r_2 : random weights, $r_{min} < r_i < 1 - r_{min}$,
- $r_{min} > 0$: asserts that a new vertex is not placed too near an existing vertex.

The paper presents an alternative equation [39]. Current implementations use $r_{min} = 0.25$ to avoid artifacts such as acute triangles. Since the common longest edge is split, the point set union of both triangles is convex, so defining the position of the new vertex by a randomized weighted sum of the triangle vertices with weights $0 \leq r_i < 1$ produces a point inside these triangles.

7.3.2 Define Continents

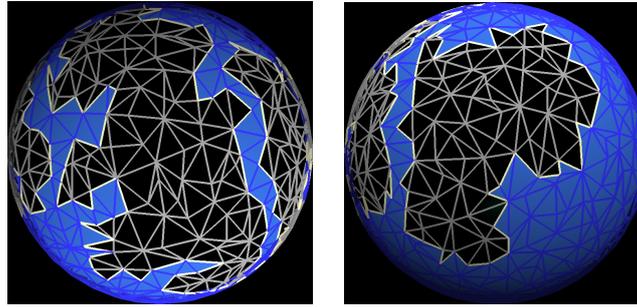
In the context of this work, a *continent* is a connected land mass above sea level. Initially, all polygons are labeled as sea. For every continent, a starting face is selected and labeled as a continent. Polygons that have at most one pure sea vertex can be added to the continent, the other vertices in a new polygon must already belong to the continent. This ensures that two continents are always separated by an edge. If all three vertices in the new polygon are continental, there may be only one sea edge, because adding a polygon with three continental vertices and two sea edges could result in the creation of inland seas or merging continents. The polygon is labeled to belong to the continent, and its edges are labeled as mountain edges. This is repeated until the percentage of the total land mass would exceed a user-defined threshold. Any edges between a continental and sea polygon are marked as coast edges.

While most rain that falls onto land gathers in rivers that run to the sea, on Earth about 18% of the water ends up in *endorheic basins* [61], where it can only escape by *seepage* or *evaporation*. This effect could be modeled by marking some continental vertices as sinks for endorheic basins. River edges may be connected to the river network or the endorheic basins' sinks. At the sink of the endorheic basin, there should be a lake, whose area can be assumed to be proportional to the area that transports precipitation to it. Instead of having one ocean separating the continents, planets drier than Earth could be covered by a single continent with a number of endorheic basins, instead of a single, connected sea.

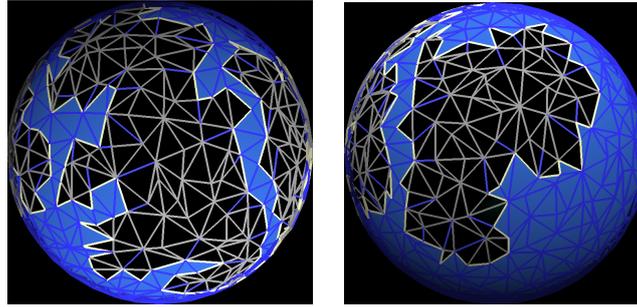
7.3.3 Produce Initial River Networks

At this point, continents consist only of continent and coast edges and vertices. We still need to generate river networks and compute continental vertex altitudes. Creating the river networks starts at the river mouths. All vertices are checked in pseudorandom order, looking for vertices that are pure mountain and adjacent to a coast vertex. In a river mouth, typically only one river mouths into the sea, therefore the coast vertex should not have a river edge yet. The edge between the chosen vertices is flagged as a river edge.

In order to complete the river networks, edges that connect a river vertex with a pure mountain vertex are considered in pseudorandom order and are converted to river edges. Two rivers may merge in a river vertex, but if possible, alternative river edges should be used to prevent merging more than two rivers in a single vertex. When all continental vertices have been connected to the river network, the river networks are complete.



a: Situation before river network creation.



b: Adding river edges to vertices that are adjacent to coast vertices.

Figure 67: Two examples for creating the initial river network, part 1. Yellow edges belong to the coast, brown edges are mountain, and blue edges denote sea and rivers.

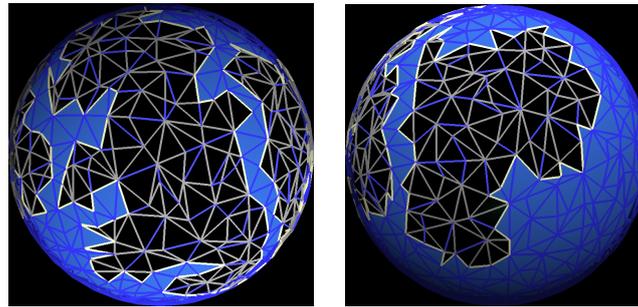
While the river network is created, altitudes are assigned to the river vertices, starting from the coast vertices at sea level:

$$a_v \leftarrow a_u + e \cdot l \cdot X, \quad (15)$$

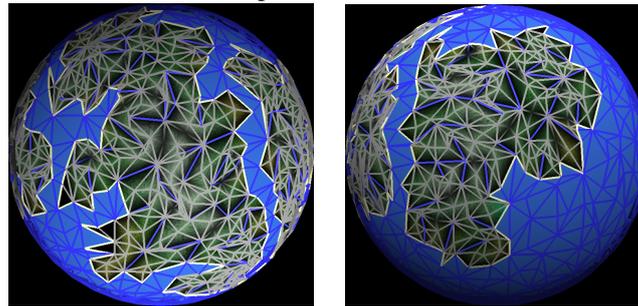
where

- v : the current vertex,
- u : reached by v 's outgoing river edge,
- a_u, a_v : the altitudes of u, v ,
- e : average elevation,
- $l = |\mathbf{u} - \mathbf{v}|$: the length of the edge between \mathbf{u}, \mathbf{v} ,
- X : a random variable satisfying $0 \leq X < 1$.

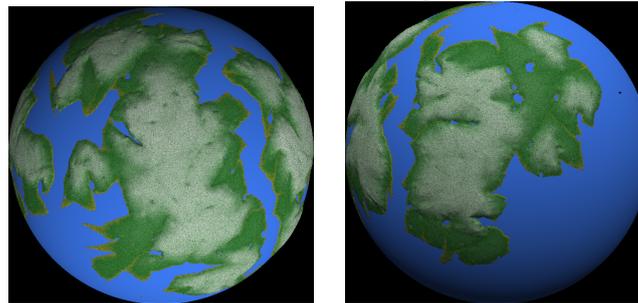
At this point, mountain edges may connect river vertices that belong to different rivers, but we need to separate the rivers by introducing mountain ridges between the rivers. Otherwise, water could flow along mountain edges between different rivers. Therefore, we insert a pure mountain vertex into every mountain edge between two river vertices or between a river vertex and a coast vertex. Coast edges with two river mouth vertices and mountain edges with one coast and one river vertex must be split in the same manner. Note that after these insertions, all new polygons are connected to the river network, since the new vertices only split edges with two vertices that are connected to the drainage system. Figure 67 and Figure 68 illustrate the process of creating river networks with examples.



c: Completed river network.



d: Mountain vertices inserted to separate river beds.



e: Result, after further refinements.

Figure 68: Two examples for creating the initial river network, part 2. Yellow edges belong to the coast, brown edges are mountain, and blue edges denote sea and rivers.

7.3.4 Assign Altitudes

Pure mountain vertices \mathbf{v} that were inserted to separate the rivers must be placed at higher altitude than their surrounding river vertices, so their altitude a is computed using the altitude a_r of the highest adjacent river vertex \mathbf{v}_r :

$$a = a_r + e_m \cdot l_e \cdot \xi, \quad (16)$$

where

- e_m : the elevation of mountain edges,
- l_e : the horizontal length of the edge between \mathbf{v} and \mathbf{v}_r ,
- $0 \leq \xi < 1$: a random number.

If altitudes need to be generated for undersea areas, pure midpoint displacement can be used. However, if altitude rises above water level again, an island is created. Coast

edges must be calculated for the island, mountain edges between two coast vertices must be split and a river network must be created. Alternatively, it is possible to use a distribution that prevents the creation of islands.

7.3.5 Compute Water Levels

In order to render the rivers, *water levels* need to be computed once for all rivers and lakes, using a simple fluid simulation. Each polygon has a river vertex, and river vertices are placed below pure mountain vertices, so each polygon's lowest vertex should have a river edge that transports water away from the polygon. Precipitation proportional to each polygon's size is added to that polygon's lowest river vertex. Following that, water exchanges are computed between the cells. The number of simulation cycles required until water levels stabilize equals the number of edges in the longest path from a river spring to that river's mouth. The water levels can be computed in a fraction of a second, whereas a full erosion simulation would take far more time. However, a full erosion simulation is not needed, since the terrain already contains river beds. All river and coast vertex altitudes are reduced by the water levels, therefore the water levels at river mouths are at sea level. The paper assumes a linear mapping between altitude and water levels of river vertices [39].

7.3.6 Assign Vertex Colors

An area's average color depends on the dominant material, whether it is submerged, its geographical position, as well as its altitude, climate and season. Biomass may be responsible for the coloration of large parts of a planet, and plants on other planets could have surprising colors [78]. Combining color interpolation with a simple noise texture increases the perceived *level of detail* at low memory cost.

In order to texture the terrain, texture coordinates must be assigned to the vertices. Rendering with vertex buffer objects and `glDrawElements` requires that either each vertex uses the same texture coordinates for all triangles it appears in, or the vertex must be copied for each occurrence that uses different texture coordinates. The latter possibility also incurs a performance penalty, so we chose to assign a single texture coordinate to each vertex that is used for all triangles that contain the vertex. Normally, it follows from the *Four Color Theorem* that four pairs of texture coordinates would be sufficient to texture the mesh: (0,0), (1,0), (1,1), (0,1). However, when inserting a vertex into an edge between two triangles, the surrounding vertices may already use the four pairs of texture coordinates. It might be possible to fix this locally, but it is faster to allow the assignment of a fifth texture coordinate. In addition, texture coordinates are not supposed to change, because this would lead to flickering. The fifth coordinate must not lie on a diagonal of the existing texture coordinate pairs. If we used (0.5, 0.5) as a fifth texture coordinate, some polygons would use (0, 0) and (1, 1) or (0, 1) and (1, 0) as additional texture coordinates. In these cases, all texture coordinates would lie on a line segment, resulting in undesirable artifacts. Instead, we chose (0.5, 0.25) as a fifth coordinate, which does not lie on a line segment formed from the other selected texture coordinates. Unfortunately, (0.5, 0.25) may still lead to local distortion of the texture, as the edges have different length.

The texture is the same for all polygons in the mesh and all *levels of detail*. It is merely intended to increase the perceived amount of geometry. Furthermore, every vertex is assigned a color depending on its altitude. It would be possible to implement more complex texture schemes that generate a unique texture for each polygon, taking into account elevation, altitude, orientation and geographical location.

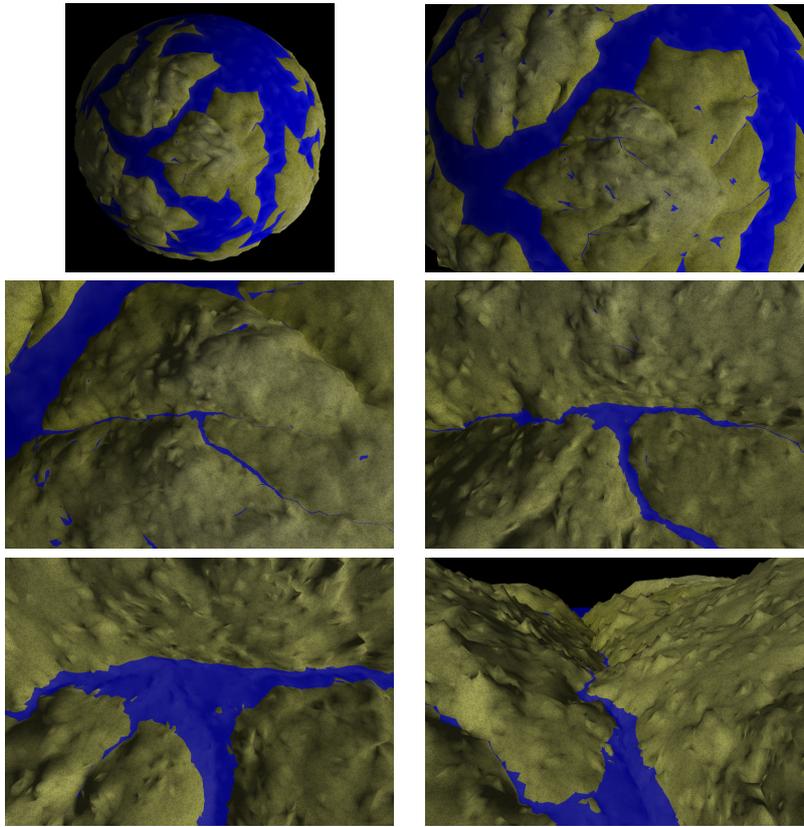


Figure 69: Screenshots showing different steps of adaptive refinement. The last image traces the river to the sea.

7.4 Adaptive Level of Detail

During planet creation, a rough representation of a planet was created quickly. While the user interactively explores the planet, the *level of detail* for this representation is adapted as required by camera movement. We use edge splits to increase the level of detail and vertex collapses to decrease the level of detail.

7.4.1 Refinement

It is possible to use a fractal approach for *refinement* because the same laws of physics apply to different scales. Polygons are refined using edge splits, and an edge is split if

$$\frac{l}{d} > c, \quad (17)$$

where

- l : length of the edge,
- d : the distance to the polygon,
- c : a constant.

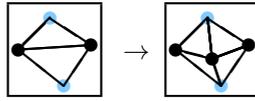


Figure 70: Splitting a mountain vertex, case 1

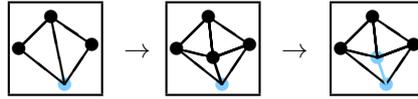


Figure 71: Splitting a mountain vertex, case 2

Furthermore, an edge is split only if it is the longest edge in both triangles that share the edge, to prevent acute triangles. Figure 69 shows a planet that is explored with adaptive refinements.

Polygons formed from continental polygons remain continental, or sea otherwise. If a river edge is split, two river edges and two mountain edges will be created. If a coast edge is split, there will be two coast edges, one sea edge, and one mountain or river edge. If a sea edge is split, there will be four new sea edges. If a mountain edge was split, the new vertex v_n will have four new mountain edges, as depicted in Figure 70.

In order to maintain the guarantee that each polygon is connected to the *river network*, we may need to convert the new vertex to a river spring. The first case where this applies is if there is only one river vertex for the four new triangles. Figure 71 demonstrates the necessary actions. The middle diagram shows that the two top triangles would be cut off from water. This is prevented by converting one edge to the river type. The other case where the new vertex must be converted to a river spring is if there are two river vertices in the new triangles that are connected by a river edge. In that case, there is one triangle that does not have a river vertex yet. In order to separate the river beds, the mountain edge between the two river vertices must be split by inserting a pure mountain vertex, as depicted in Figure 72.

The polygons that were replaced in the refinement process were connected to the river network, so there must be at least one river vertex v_r in the four new triangles. By changing edge $\overline{v_n v_r}$ to river type, all polygons are again connected to the river network. As during planet creation, we prefer to select a river or coast vertex for v_r with minimal number of incident river edges.

v_n 's altitude depends on the types of its incident edges. If v_n was inserted into a river edge, its altitude is between the old edge's vertices, and the water level is inherited from the vertex with higher altitude. New river springs are placed below the surrounding mountain vertices and above the river vertex at the end of v_n 's river edge, using Equation 15 if possible.

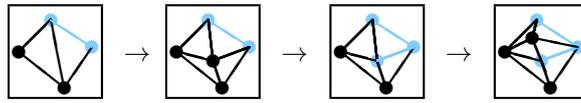


Figure 72: Splitting an edge, case 3

Altitudes for pure mountain vertices must be chosen with care. If a pure mountain vertex is placed too low, nearby rivers may take an alternate path. This leads to a number of complications, because if an entire polygon is flooded, the river may look much broader there. Also, the rivers may look diffuse if they split and merge several times while running downhill. On the other hand, always placing the mountain vertices above their adjacent river vertices leads to hills that become increasingly steep as the user zooms into them. Instead, when a mountain vertex is created to separate riverbeds, this is stored in a flag of the river vertex. If a mountain vertex is inserted into an edge with two mountain vertices with that flag, the new vertex inherits that flag. At high *levels of detail*, forcing a vertex above the water may require unrealistic elevations. Also, with an active water simulation, vertices that were forced above water level may still be submerged. In these cases, the flag to force river vertices above water should not be inherited to new vertices. Therefore, each new vertex is tested for submergence, and if one of the mountain vertices is submerged, the flag is not inherited to new mountain vertices that split its incident edges. As rivers are surrounded by mountain vertices at higher altitude, rivers always follow the steepest decline.

By lowering a river vertex, and increasing that vertex's water level by the same amount, *lakes* can be added to the terrain. The lake vertex should have at least one incoming river edge and must not have coast or sea edges. While planet creation required only adding up water levels, water simulation may have to level off water between several adjacent vertices during interactive exploration, because of the higher resolution of the terrain. Refinements cause slight changes in the distribution of precipitation between all vertices. Optionally, a water simulation can be run to recompute water levels for all vertices after refinements.

7.4.2 Reversing and Reapplying Refinements

In order to prevent that the amount of geometry created during refinements increases beyond practical limits, at some point it becomes necessary to decrease the *level of detail* for parts of the planet's mesh by coarsening. Instead of deleting a pair of polygons after refining them, the polygons are stored in a hierarchical data structure, where the polygons at the higher level of detail are stored as child polygons of the two older, coarser polygons. This allows us to undo the refinement operation by replacing the group of new triangles with the predecessors later. There is a frame-to-frame coherence: Polygons that have been rendered in the last frame will likely be rendered again in the next frame. Rather than traversing the entire polygon hierarchy every frame, we store the set of polygons currently selected for rendering in a mesh and adapt it to the new viewer's position.

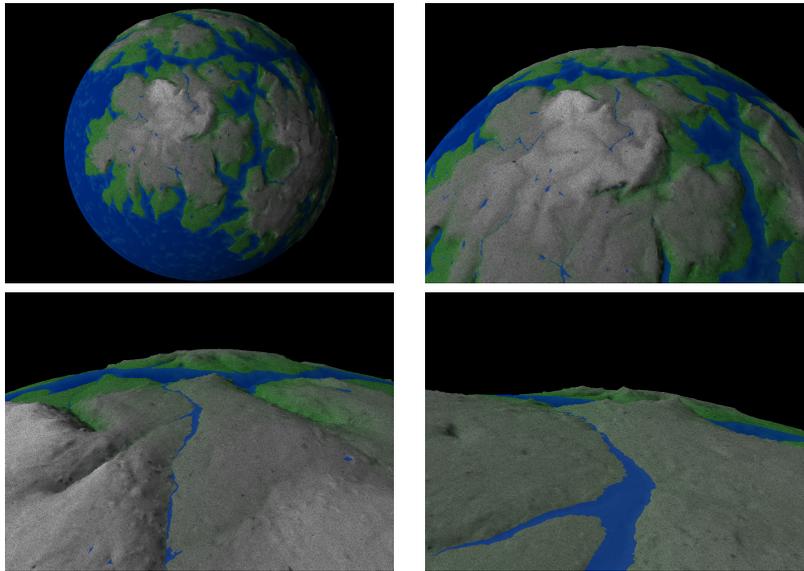


Figure 73: Zooming into an Earthlike planet

As evidenced by figures 70, 71, and 72, each refinement operation replaces two triangles by a group of four or six new triangles. All triangles that were created in a single refinement operation must be in the mesh for reversing the refinement operation. If this is not the case, other refinement operations must be undone first. The edges inside the old and new groups are consistent, and the edges outside the groups do not change, therefore no T-vertices are introduced.

As the user moves further away from parts of the terrain, we undo refinements for polygons whose parents satisfy $l/d \leq c$, but the polygons at the higher *level of detail* are retained in case the user comes back later. In that case, the polygons at the higher level of detail are reinserted into the mesh, if both parent polygons are in the mesh.

7.4.3 Reproducibility

In a computer game involving several planets, a player might wish to come back to a planet later. This mandates that the game must be able to recreate planets. A simple approach would store the results from previous visits and refine them further during exploration. However, as the user keeps exploring the planet, the system produces geometry. This requires memory, which may be used up at some point. Furthermore, in a networked scenario, several players could be exploring the planet independently, and meet at various times. In this case, it must be ensured that all players see the same geometry, otherwise one player might follow a route that on the other player's computer would lead through a mountain. Sending all new geometry over network may not be feasible due to bandwidth restrictions in the network and synchronization cost. Therefore, other ways have to be explored to ensure a consistent experience for all players.

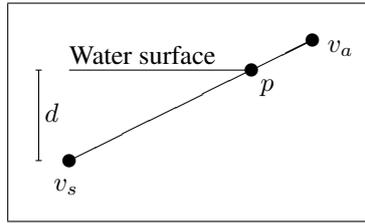


Figure 74: Computing the position \mathbf{p} where edge $\overline{v_s v_a}$ exits the water

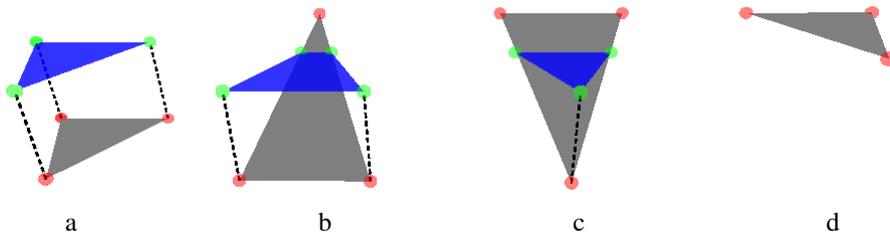


Figure 75: Generating water polygons for different cases of submerged ground polygons: a. fully submerged polygon, b. two submerged vertices, c. one submerged vertex, d. polygon fully above water.

Normally, the results of a procedural algorithm can be *reproduced* by using the same random seed. This is not the case here, because a different path of exploration would apply the same random values in a different local order, and global refinements are usually not an option. A possible solution is to store the planet after preprocessing when it requires little memory and to ensure the results of refinement are the same regardless of the path of exploration. This can be implemented by assuring that refinements are guaranteed to take place in a certain local order, and that the same random values are produced locally, independent of refinement order. The paper presents a solution to this problem [39].

7.4.4 Create Geometry for Lakes and Rivers

The following algorithm can be used to *create geometry* for the lakes and *river networks*. For every partially submerged edge, we need to compute a vertex \mathbf{p} at the water's surface, as demonstrated in Figure 74:

$$\mathbf{p} \leftarrow \mathbf{v}_s + \frac{d}{\text{alt}(\mathbf{v}_a) - \text{alt}(\mathbf{v}_s)} \cdot (\mathbf{v}_a - \mathbf{v}_s), \quad (18)$$

where
 \mathbf{v}_a : the edge's vertex above water level,
 \mathbf{v}_s : the edge's submerged vertex,
 d : water depth,
 $\text{alt}(q)$: q 's altitude.

In order to render water, we generate one polygon for each terrain polygon representing the water above the terrain polygon. Figure 75 summarizes the different cases. If all three vertices are submerged, the water polygon is a triangle consisting of the three vertices, lifted to the surface of the water (Figure 75.a). If two vertices are submerged, the water polygon is a quad consisting of the submerged vertices, again lifted to the water surface, and two vertices along the edges from the submerged vertices to the vertices above water (Figure 75.b). If one vertex is submerged, the water polygon is a triangle consisting of the submerged vertex, lifted to the water surface, and two vertices along the edges to the other vertices (Figure 75.c). If no vertex in the polygon is submerged, no water polygon is needed (Figure 75.d). However, each continental polygon is connected to the drainage network. Therefore every vertex must have a water polygon, and a polygon with no submerged vertex would be considered as an error.

The water polygon must be updated if its relevant water levels change, therefore the water polygon vertices are not inserted into the terrain mesh. The terrain polygons reference the water polygons instead.

Again, the paper presents a different solution to this problem [39]. In the paper, water vertices are allowed to lie below ground, water is always represented by triangles, and parts of the water triangles that lie below the ground are culled in the depth test.

7.5 CPU Implementation

In the CPU implementation, planet creation takes about 200ms to produce planets of about 6000 polygons. Reproducibility was implemented by storing all polygons even if they are not currently visible. In early implementations of the algorithm, the *level of detail* was adjusted in every frame. When updating more than 33,000 triangles every frame, the framerate drops below 15 frames per second. Therefore, rendering and mesh adaptation have been implemented to use separate *threads*. While one thread processes edge splits and vertex collapses, the other thread is used for rendering. The refinement thread produces indexed facesets for the terrain and water meshes. These are uploaded to vertex buffer objects and stored on the GPU. This reduces bus traffic and ensures that the geometry buffers are not altered while rendering. This allowed to render 120,000 triangles at 60 frames per second, while the refinement thread ran at ca. 4 frames per second. At that level of detail, the refinements take around 90 ms, undoing refinements takes about 70 ms, creating a copy of the terrain for rendering in the separate thread takes 110 ms, and uploading the geometry to the GPU takes 12 ms. The experiments were performed on a PC with a six-core AMD Phenom II X6 1090T (3.2 GHz). Rendering could be improved using continuous level of detail [134], but the success of the GPU implementation made it unnecessary to test this.



Figure 76: Sample image of the GPU implementation

7.6 GPU Implementation

The number of polygons required for realistic scenes and advances in hardware made it necessary to redesign the algorithm to support more than two threads. Evgenij Derzapf et al. presented a massively parallel implementation that runs on graphics hardware [39]. Planet creation works similar to the algorithm discussed earlier in this chapter, but a new pipeline was designed for massively parallel edge splits and vertex collapses. New memory for splits is allocated in advance, and safety limits on the proximity of polygons affected by the operations assert that the operations can be carried out in parallel. The algorithm was implemented to be reproducible from a random seed to reduce memory use, as currently invisible polygons do not need to be stored using this approach. The GPU implementation also includes algorithms that ensure that rivers meander and that the terrain around the rivers is smoother than the hilltops. It adds backface culling and view frustum culling for controlling the *level of detail*. Water is rendered as a shader effect. The system uses other equations in some cases, describes how to generate meandering rivers, and the system does not require an erosion simulation.

While the other images in this section were produced using the CPU implementation, Figure 76 shows an image produced using the GPU implementation.

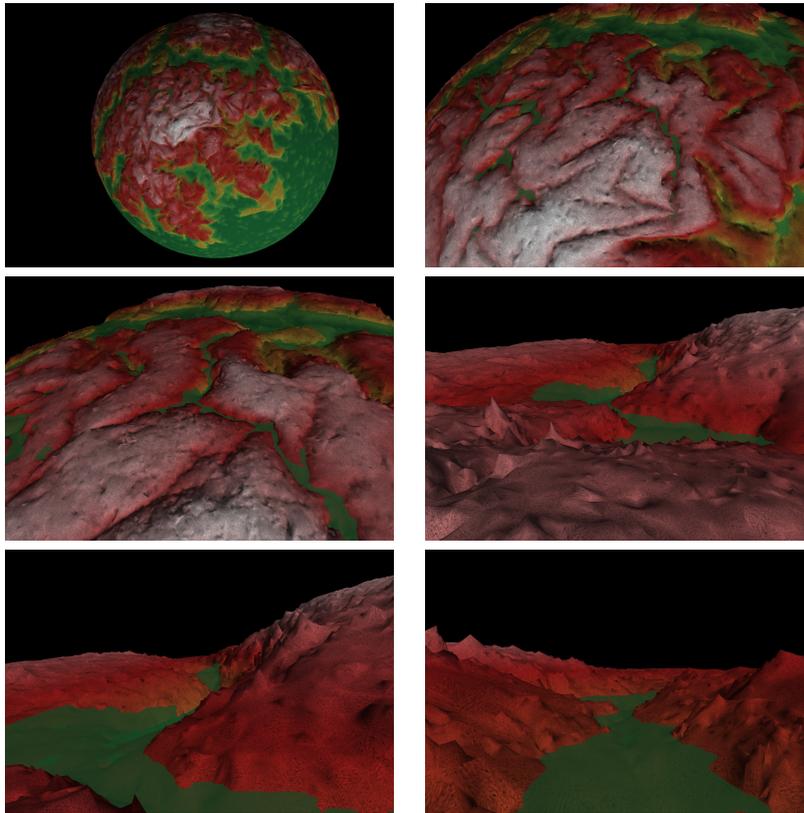


Figure 77: Zooming into a toxic planet

7.7 Discussion

We have proposed a new procedural algorithm that spontaneously creates planets or terrain parts with continents and realistic river networks. It was specifically developed for view-dependent adaption, and a massively parallel GPU implementation demonstrates that the algorithm can generate high *level of detail* interactively. While previous algorithms are not able to generate planets at adaptive level of detail within seconds while ensuring consistency of the river networks, we are the first to present a parallelized pipeline that combines these features. The algorithm correctly models how valleys and mountain tops differ in that rivers flow through the valleys. While many applications use a fixed level of detail, interactively adapting the level of detail to the camera perspective is a necessity when dealing with large terrains, such as planets. The GPU implementation does not require storage except for the mesh representing the terrain, and the geometry does not have to be streamed over network, even when several users wish to explore the same terrain. Only a small parameter set and an initial random seed are needed to completely recreate a planet because of reproducibility.

The system by Kelley et al. produces only a single river network, and it does not support refinement [76]. Our technique is suitable for interactively adapting large terrains to a moving camera. The only other technique that can do this is the midpoint dis-

placement algorithm by Fournier et al. [47], which does not produce river networks. When using midpoint displacement on realistic terrain, rivers may be interrupted by mountains, which would likely block the rivers, resulting in large endorheic basins or inconsistencies. In contrast to that, the rules implemented by the proposed algorithm ensure consistency of the river networks.

Erosion simulation can also produce eroded terrain with continents and river networks, but that family of algorithms requires much more computation time and has to run supervised. Otherwise, either too many large lakes remain, or the rivers carve too deeply into the terrain. If the simulation ran too long with wrong parameters, this can only be fixed by restarting the simulation. The algorithm presented in this chapter does not require an erosion simulation, and thus produces viable solution much faster. While it may be possible to combine erosion simulation with adaptive level of detail, ultimately similar problems would have to be solved. In addition, users might note changes in the terrain caused by the simulation if the simulation runs simultaneously while the user explores the terrain. In contrast to that, the terrain generated by our algorithm is immediately realistic and stable. Erosion simulation is not superceded by our algorithm, as landscapes produced by our algorithm can be further improved by simulating erosion.

The algorithm can be used for simulations that require spontaneously created terrain. This includes computer games and learning to steer vehicles, e.g. flight school.

8 Systems Comparison

In this section, we compare previous systems with the systems introduced in this work. The tools and algorithms presented in this work can be understood as samples of various tradeoffs. One way of comparing the systems is in terms of *specificness* and *generality*: Is a system or algorithm geared towards a specific purpose or can it be applied to general problems? Another perspective on using the tools described in this work is in terms of modeling effort. The tools described in this work can also be compared by *power of expression*. Turing machines are hypothesized to be able to compute any function that can be computed algorithmically [151]. This includes the ability to compute shapes. Therefore, any Turing complete language can be used to compute any shape that can be computed using finite resources. There appears to be a tradeoff between power of expression and *ease of use* in programming languages. The more powerful a language is, the more difficult it becomes to learn and use, as it incorporates more concepts that need to be mastered. Research continues for finding notations for programming languages that are easier to learn and use. Any model created using procedural methods could still be edited further manually at the polygon level, with as much effort as necessary to achieve a specific result, but the goal of procedural modeling is to minimize such efforts.

We start by comparing previous systems. Early L-systems focused on modeling trees. Since then, an enormous amount of research was put into increasing their flexibility. Early L-systems were simple, difficult to use, limited in capability, but every generation of implementations added new features. As this discussion progresses, important improvements to the basic implementation of L-systems are noted. As there are no absolute measurements of the tradeoffs mentioned above, the tradeoffs can be quantified only by comparing various systems, so this discussion starts by comparing L-systems with GML. GML uses postfix notation to form a 3D modeling language. L-systems and GML fare badly in terms of usability, because both use cryptic notations. The effect of text replacements in L-systems can be difficult to predict, and it requires a lot of thinking to figure out the meaning of the rules. The same goes for GML. As a PostScript variant, GML is Turing complete. GML has loop control, and a dictionary could be used to emulate the Turing machine's tape.

Early L-systems were not Turing complete: The number of replacement operations is a global parameter that prevents unbounded loops. More recent implementations keep replacing string parts until only terminal symbols remain in the L-string. While this may lead to unterminated loops, this is a general problem shared by most complex computing languages, and formally known as the halting problem (Halteproblem). With the addition of full programming languages in systems such as L+C [128], Turing completeness may have been achieved in L-systems. No languages that are more powerful than Turing-complete languages are known, so GML marks a maximum in expressional power. For L-systems and modeling grammars, a wide range of implementations are available, covering a wide range of usability and power of expression. Other systems increased the breadth of shapes that can be described [79] or improved editing methods [86]. While early implementations of L-Systems would score low in terms of usability and power of expression, CGA shape is very usable because it offers very many features [157].

In Xfrog, the cryptic syntaxes used by L-systems and GML are replaced by far more intuitive, visual symbols. Xfrog uses recursive loops to create tree organs, but it lacks certain constructs that would make the language Turing complete, such as variables. While both early L-systems and Xfrog strive to model trees, GML and recent implementations of L-systems can generate a wide variety of models. Xfrog, L-systems and GML form a class of *user-programmable* procedural algorithms. While L-systems are arguably programmable, they are not Turing complete, so the algorithms in the class of user-programmable procedural models are not always Turing complete. As a non-Turing complete language, Xfrog's power of expression is smaller than that of a Turing complete language.

Various examples of non-user-programmable procedural algorithms exist. Examples of algorithms include midpoint displacement, as well as numerous early tree generators such as the algorithm proposed by Weber and Penn [159]. These algorithms have fixed parameter sets that require simple, numeric parameters. The systems may incorporate parameters to control loops. For midpoint displacement, the number of global refinement operations is a parameter. In Weber and Penn's algorithm, branches may be split by cloning, and the cloning process is also controlled by parameters. These parameters control loops, yet the functionality is essentially hard coded into the algorithm. On the other hand, of course, model graphs or GML programs could be written to implement non-user-programmable procedural algorithms.

We can now evaluate the algorithms presented in this work by the terms discussed above. A Turing-complete language can approximate the geometry of any model with polygons [154]. This means that plab provides the full set of concepts necessary for describing shapes that can be computed using finite resources. plab's versatility and power of expression come at the cost of mastering a programming language. While these concepts may be difficult to learn for people who do not have a computer scientific background, the concepts have been implemented as a visual language. There are no keywords that the user has to remember, and hints are displayed for every parameter that the user needs to provide. However, editing individual instances of rules that apply to masses of objects requires manually editing the model graph, which is cumbersome. Modeling objects using plab can take quite some time, but it is useful when the time needed to create the procedural model is less than the time needed to create all instances manually.

1-2-tree was designed to make modeling trees as simple and efficient as possible for average users. It allows to create trees in a matter of minutes. Users do not have to learn concepts like loops or variables from computer science, instead the modeling workflow is based on manipulating the model directly via parameters or the viewport. While model graphs require any instances that require exceptional parameters to be hard coded into the model graph, 1-2-tree stores all branches in a scene graph and allows the user to edit either single branches, or edit several branches at once based on selection settings. 1-2-tree can model only trees. While 1-2-tree is non-user-programmable, and every entity in 1-2-tree has a fixed parameter set, branches may have individual parameters. This allows the user to model trees more precisely than by using a single set of parameters that applies to all branches. 1-2-tree allows the user to model trees, in much the same way as a text editor allows to edit text: While reason-

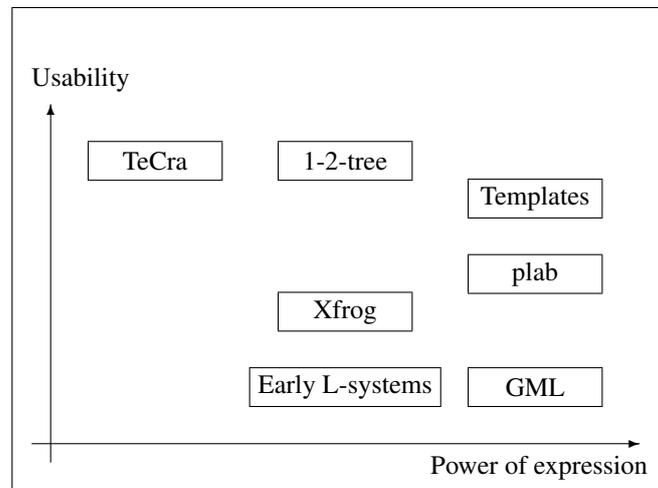


Figure 78: Rating systems and algorithms analyzed in this work in terms of power of expression and usability

ably powerful for modeling trees, it is not designed for more general tasks. However, the idea of applying operations to semantic entities, which are selected by properties, certainly applies to more general modeling tasks. 1-2-tree has elements that allow to model on a semantic level, editing similar branches easily. As it focuses on a particular type of model, that type of model can be edited very efficiently, at the cost of being limited to trees. The power of expression could be improved by integrating a scripting language to solve unforeseen problems.

In the following, the name TeCra is used for the algorithm that created terrain with river networks. It is an example of a non-user-programmable procedural modeling system. TeCra only requires choosing few general parameters, such as the ratio of land and sea and colors. Following that, terrain is created automatically. Here, it is the algorithm's task to ensure that consistent terrain is created, but the user does not have much control over it. Therefore, TeCra requires very little modeling effort, instead the user may explore the terrain immediately. While it would be possible to implement editors, or add a scripting language, often it suffices to export the resulting terrain and edit it using conventional modeling tools.

Templates are specific to a certain type of model by nature, but templates could be created for all types of objects. The exception here are unique objects, which may comply to a set of rules, but which can be modeled more efficiently using conventional modeling tools. Templates require as much effort to create as any plab model graph, but once created, objects can be generated and edited easily.

Figure 78 evaluates the various systems and algorithms proposed in this work regarding power of expression and usability. GML and L-systems are likely the most difficult systems to use, as all other systems offer better support to guide the user to the 3D models he wants to create. GML, plab and templates score equally high in terms of user control and power of expression since these are all Turing complete procedural model-

ing systems. There are too many individual implementations of L-systems to mark in the diagram, so the diagram only covers the early implementations of L-systems. Xfrog is certainly more convenient to use than GML, but in turn the tool does not allow for computations that are as powerful as GML's or plab's. plab and 1-2-tree are more convenient than GML because they allow for a visual notation. plab is rated slightly higher than Xfrog because plab allows for *viewport editing*, at least when the model graph was designed to take advantage of it. *Templates* are an extension of model graphs, so they are as powerful as model graphs, but more convenient to use. 1-2-tree does not have a Turing complete set of instructions, but it has simple support for functions that modify parameters along a branch and randomization. 1-2-tree is much more user convenient than Xfrog, because the user is not dealing with abstract visual nodes, but instead directly editing objects in the viewport and sketching. It can be argued that both Xfrog and 1-2-tree offer the tools for fully editing trees, so they are equally powerful. TeCra takes only few input parameters – modeling could not be easier, but it also leaves very little control to the user. TeCra also represents the class of non-user-programmable modeling languages.

9 Conclusion

This work has presented user interfaces and algorithms for procedural modeling. In standard 3D modeling approaches, the user moves and edits single primitives, whereas procedural modeling allows to manipulate objects based on rules. While earlier procedural modeling approaches focused on modeling a specific class of objects, such as plants or buildings, current systems are capable of modeling different classes of objects. In particular, this work demonstrates that *plab* can be used to model any type of model that can be computed using finite storage and computation time, by the *Church-Turing thesis*. The Church-Turing thesis states that for every abstract algorithm, a Turing machine can be designed. This work proves that model graphs can emulate Turing machines, therefore, model graphs can also compute any algorithm. This means that model graphs can also approximate geometry for any object, as long as the computation uses finite time and storage.

This work also provides practical examples that demonstrate that *plab* can be used to model various types of models. Examples were given for plants, trees, terrain, buildings, and games. Buildings can be modeled easily and interactively in *plab* by moving corner points. Parameters can be adjusted to control geometry creation for façade elements. For modeling terrain, a variety of modeling algorithms have been integrated into *plab*. Terrain generation starts with randomly placed B-splines that model mountain ridges. By diffusing elevations through the grid, simple terrains can be generated efficiently. Such terrains lack plausible *river networks*, but these can be created using erosion simulation. Opportunities for parallelism are detected and used automatically (Section 4.3.6). *plab* allows to generate eroded models with reasonable effort.

plab allows to create simple tree and plant models using parameters that apply to the entire plant. However, in many cases, more direct editing methods are desired. *plab* currently does not offer the necessary performance for editing trees interactively, and more powerful *viewport editing* methods would have been required to allow more flexible ways of editing trees. Instead, a new user interface was presented that is specifically tailored to modeling trees. Each branch has its own set of parameters, and parameter values can be entered as numbers or they can be edited in the viewport. Users are able to adjust parameters of single branches or several branches at once using selection options. 1-2-tree focuses on making the common editing functions easy to use, and much of the difficulty in modeling trees is hidden from the user, but it remains a very powerful system because of the *targetted updates*.

Procedural models have often been criticized as being cumbersome and difficult to use. Often, the algorithms are incorporated into their own user interfaces. While this does allow for controls that are well-suited for the task of creating the supported types of objects, often objects have to be exported to a standard 3D modeling suite for viewing them in their intended surroundings. By offering Xfrog as plugins into various 3D modeling suites, trees can be modeled in the suites without exporting models. In order to deliver the same convenience with *plab*, *plab* was integrated into Cinema 4D as a plugin as well. Instead of using a single model graph to edit a model, the user is offered several tool model graphs that are designed to offer artistic freedom.

While plab supports editing terrain, some applications require terrain that can be created immediately without user intervention. Examples include computer games that allow players to visit a large number of planets. In that case, it is not possible to model a vast number of planets in advance at high detail and stream it to users as needed. Instead, procedural models can be used to generate planets as needed. This work presents a system that combines plausible river networks with adaptive *level of detail*. It creates planets in a fully automated manner, using procedural algorithms and semantic information stored in the primitives. In contrast to previous procedural algorithms that produce terrain, these constraints are satisfied with minimal preprocessing.

Procedural models often require users to specify numerous parameters, which the user has to understand and get right. Choosing different settings or another random seed may lead to vastly different results, and recreating the entire scene may take a long time. The approaches presented in this work address some of these issues. While parameters in procedural modeling are complex, the impact of altering parameters can be reduced by recreating only the parts of a scene that are affected by the change. This has been implemented in 1-2-tree using selections and targeted updates. Control points also work well to enhance an interactive modeling experience.

10 Future Work

10.1 Terrain Modeling

Normally, midpoint displacement calculates new vertex positions from the average of the surrounding vertices, and displaces these positions randomly. This could be improved by passing a recursive factor that changes the amount by which the new vertex can be offset. This factor should be increased if there is a river spring in a polygon and for high altitudes.

10.2 Crust Movements

Crust movements cause the creation of new terrain in some places, while other parts of the crust may be lifted. Crust movements counter the rounding effects of thermal and eolian erosion. Existing erosion simulations could achieve more realistic results by integrating these effects.

Mesh data structures are well-suited to simulate how crust movements counter erosion, because every vertex can be assigned a floating point movement vector. Plate movements cause some edges to grow larger, while others are shortened. Long edges can be split, while edges that are too short can be collapsed. Simulating plate movement would create steep terrain where plates collide, while in other places, effects of erosion prevail.

10.3 User-Controlled Modeling of Planets

The procedural algorithm for planets with river networks could be adapted to allow a user to model a planet. The idea is to associate editing tools with the different stages of planet creation and refinement. The tools need to assert consistency of the resulting planet, including altitudes, rivers running downhill, and that every polygon is connected to the *river network*.

After selecting the base shape, the user may sketch continents, and the system finds the edges nearest to where the mouse ray intersects the base shape. Vertices that lie near a sketched line may be moved onto that line. If the sketch deviates too far from an existing edge, the mesh can be refined locally. All polygons inside the encircled region are marked as continental. If the sketch touches existing continents, the selected polygons and existing continents are joined into a single continent. Otherwise, a new continent is created. While a user-supplied sketch is usually smooth, coast lines should look scraggy. Therefore, a user-drawn coast line should be used only as a sketch that requires additional detail.

Another tool would be used to sketch river networks. Again, the system finds the nearest edges, and adds them to the river network. The sketched line defines a river that mouths where the sketch first crosses an existing river, lake or the sea, and the sketched river will flow towards this point. This ensures that a newly sketched river mouths into only one other river, lake or the sea. Altitudes have to be assigned so that they decrease

slightly along the path of the river. Detail and additional river networks can be added by the algorithm, and the user may edit them by moving control points. Each river network may share at most one vertex with the coast, and forms an endorheic basin if such an edge is not present. Another tool could be used to edit lakes.

Altitudes are assigned and updated while the user edits the terrain, but the user may adjust the coordinates by dragging control points. The user would define colors using a 2D diagram, where one dimension is used to store latitude, while the other stores altitude, not longitude. Furthermore, the user would be allowed to define different local color patterns using an airbrush tool. This would allow creating color schemes for planets easily, including icecaps at the poles and desert at the equator. The user should be able to place special objects, and the entire scene should be reproducible.

Vegetation stores water, slowing down evaporating and movement towards river systems. Simulating the interaction between weather and vegetation would lead to more natural distribution of areas with plants and deserts. As a side result, the weather simulation would allow to render planets that are covered by realistic clouds. While weather prediction still requires lots of computational power, it should be possible to obtain a realistic distribution of fertile areas, deserts and clouds with less overhead.

While terrain at higher levels of detail would be created procedurally, the artist might want to edit regions at higher *levels of detail* as well. Restrictions might apply for placing certain objects. For example, an artist might wish to enforce that unique objects are placed only once, or that they are placed relative to or near other objects. Model graphs could be used to reflect such conditions.

The advantage over doing this in a normal modeling suite is that many levels of detail can be edited in this manner, which would be necessary for viewing the landscape at various levels of detail.

10.4 Transportation Networks

If the explored planet is populated by a civilization, there may be cities and a transport network to connect the cities. These have to be considered during planet creation. As before, when starting to explore a planet from orbit, and zooming into details of the terrain, it is not possible to store and display the entire planet at a single high *level of detail*. So again, additional detail needs to be produced as the user explores the scene.

We use the term transport network to abstract from the fact that the network may consist of roads, rails, waterways or other transportation infrastructure, even pipelines or a mix of these. As a transport network is designed to connect cities, we start by placing the cities. Often, the size of a city is limited by resources available to the population, including drinkable water and land that can be farmed. Therefore, river mouths and coast vertices are suitable places for the cities. As streets cannot be generated for previously explored areas, the system would need to know the positions of all cities and major roads before they are displayed. Further cities can only be created close to existing roads in unexplored regions. During planet creation, there may be so few vertices

that every river or coast vertex should be a seed for a city, but during refinement, we need to test for nearby cities before creating new cities. At high level of detail, a city can cover many polygons in the terrain mesh, but in case of large terrain polygons, there may be several cities on a single polygon. So, every polygon may reference several cities and vice versa. The same holds for the transport network: In case of roads, several roads may pass through a large terrain polygon, but smaller terrain polygons may not be connected to the road network at all.

For creating the transport network, we consider random pairs of cities. If the cities are not connected to the transport network, a direct route between the cities can be created. If a route exists, a more direct route should be constructed between the cities if

$$\frac{l_{\text{air}}}{l_{\text{roads}}} > f(p_1 + p_2), \quad (19)$$

where

- p_1, p_2 : the populations of the cities,
- l_{air} : distance between the cities,
- l_{roads} : distance over the transport network,
- $f()$: a function depending on the population of both cities.

If parts of the new route run parallel and near other parts of the transport network, a single road could be created. In that case, the breadth and number of driving lanes may have to be adjusted to the increased amount of traffic. If the route passes near another city, the route could be adjusted to pass close to the city. Transport network crossings should occur with preference near cities, but long-distance routes do not normally pass through the city center. The generative system may evaluate several different routes, and choose the one with best relative cost, length, and number of cities connecting to the new part of the transport network [49].

The iterations to check for routes between cities and adapting the transport network must be stopped before they take too much time. However, the transport network for a single continent must still be a strongly connected component, meaning that all pairs of cities must be reachable using the transport network. The transport network can be stored as a graph, referenced from the terrain mesh. *Level of detail* selection must decide whether to reflect the transport network only in textures, or when to create geometry for it. Exceptions where the road network would be split into two or more components are the borders between enemy states.

After the continental transport network has been created, we can create cities using the algorithms proposed by Parish and Müller [114]. Different parts of a city may contain different types of buildings. Vegetation can be loaded from a database and integrated into a scene immediately when needed with far less planning. When placing objects, visible repetitions should be avoided, but the algorithm must be efficient enough to guarantee an interactive framerate. Perhaps recursive blue noise or Wang tiles can help solve this. If possible, it should not be necessary to store the locations of individual plants, to reduce memory requirements, so reproducibility is a goal here, too.

Different kinds of roads include trails, pavement, sideways, car's roads, country roads and highways. These differ in how geometry must be created for them, they have different parameters, and they have different types of crossings, bridges and tunnels. Between the roads are blocks that can be filled with wild forest, parks, farmed areas, and rural areas.

The level of detail approach requires that a coarse representation of the transportation network and cities is chosen early. If the user has explored an area, it is not possible to add cities or roads to that area, because the user might note the difference. It would break the illusion of reality of the virtual environment.

10.5 Placing Lakes and Endorheic Basins

In the terrain exploration algorithm, during refinement, creating new river springs may lead to vertices that have more than two incoming river edges. It is possible to create a lake around the vertex with more than two incoming rivers. Another option is to mark the new vertex as an endorheic basin instead, allowing subsequent refinement steps to add river edges to the basin sink. The basin would have to be surrounded by mountains that ensure that no water escapes to other river networks.

10.6 Compiled model graphs

It would be possible to generate C++ code that represents one or more model graphs. The parsed model graph nodes would have to generate equivalent C++ code. By linking the code with a library, it would be possible to execute model graphs natively on the CPU. This would certainly yield large performance bonuses. Compiled model graphs would be loaded as DLL files into plab or Cinema 4D. For use with Cinema 4D, appropriate string, dialog and resource files would also be required, but plab would not be required to run as an external process, further improving performance and integration with Cinema 4D.

Records currently pose a problem for compiling model graphs. In plab, record members may be used without a record definition (Section 4.3.3). As a result, it is difficult to predict which members a record that is used in plab will have. Two methods could be used to correct this:

- It would be possible to introduce a new node type that allows the user to create record definitions, `RecordDefinition`. This could be used for tighter storage of records – currently any record in plab needs to store the names, types and values of all members. Using this node type, a record would store a reference to its definition and the values, but not duplicate member and type definitions.
- For records that do not use a type defined by `RecordDefinition`, plab could compile a list of all possible record members and create a single record definition that includes all possible members. The resulting record definitions are bloated. While any member is used by the compiled model graph, many instances require only a subset of these record definitions. Therefore, for compiled model graphs, it is advised to use records with `RecordDefinition`.

Compiled model graphs could be used to integrate templates as plugins. In contrast to plabC4D, these plugins could be executed directly in Cinema 4D's environment, instead of running in an external process. As a result, these plugins would be more efficient to execute than plabC4D templates.

10.7 Parallel Execution of Model Graphs in Case of Several Root Nodes

If a model graph has more than one root node, the order of execution of the root nodes is not defined. Future implementations could exploit this for parallel execution: Each root node would be executed in its own thread. To prevent race conditions, in this case, each root node should have its own *scope*. Even graphs with several root nodes could have shared nodes. The nodes could be executed simultaneously without problems, because each node would be executed using the variables and context defined by the thread that executes it, which may result in different output. While any integrated development environment (IDE) should ease programming as much as possible, ultimately, it is the programmer's task to ensure the correctness of the model graphs.

10.8 Templates and Scene Graph Hierarchies

If a model graph creates a hierarchy of objects, this hierarchy should be reflected in Cinema 4D using additional scene graph nodes. Construction of these Cinema 4D scene graph nodes must be controlled by plab, but most nodes and model graphs that were executed while creating geometry are not supposed to show up in Cinema 4D later. Instead, model graphs need a new node type to define that geometry from its child nodes belongs into child nodes of Cinema 4D's scene graph. The new node type may specify a new base model graph for reflecting the new node in Cinema 4D's scene graph. For this reason, an XML-based file format might be better suited to store geometry that may require new child nodes. When the file is read, new Cinema 4D nodes need to be created automatically. Example applications include houses and plants. This mechanism could be used to reflect branches with their own nodes in Cinema 4D. For houses, the need for using different base model graphs becomes evident: At the highest level of abstraction, a house is represented as a list of about a dozen polygons. The next lower level of abstraction splits a house into the roof and façade. The façade further consists of window and door elements. Each of these levels of abstraction requires its own specialized base model graphs for the plabC4D nodes. Of course, a viewpoint of *level of detail* may lead to a different hierarchy than these levels of abstraction.

10.9 Further Improvements to Templates

Currently, the workflow of using templates is to create a plabC4D node, and then select a base model graph. Handling templates could be improved by displaying additional menu items in Cinema 4D that create plabC4D nodes with predefined model graphs. This would require a command plugin to Cinema 4D that adds new templates to the plugin menu. This command plugin would analyze the base model graph and generate customized resource files. A plabC4D node created this way would not require the user to select a model graph, as the base model graph would be statically compiled into

the resource files. Furthermore, currently tool model graphs are started using the tool model graphs button, and then selected from a context menu. By contrast, the resource files could integrate buttons for each tool model graph.

Currently, there is no good solution for handling textures in plabC4D. The path of the currently used texture is displayed to the user, and the user has to manually copy this path to a material node. The author so far has not found a more convenient way to solve this. If this cannot be automated, model graphs could be adapted to use static textures. That way, the textures would need to be assigned only once, whereas currently, the textures are regenerated when plab is run from plabC4D and need to be reassigned.

A Control Operators in plab

All operators are derived from `OperatorBase`, which in turn is derived from `NodeBase`. The following operators may deviate from the usual order of execution for model graphs, and are called *control operators*:



BranchOperator:

This operator allows to calculate the number of the edge that is followed to the next node to execute. The edges are numbered from 0 to (count of edges)-1. The operator can be used to jump to a random node if the parameter uses the function `random`. If no edge with the given number exists, no child node is executed. It has the following attributes:

- `int selected Branch`: Number of the edge to the next node to execute



CallOperator:

Allows executing the same or another model graph as a subroutine or recursion. The model graph to call is given as a file name that may be computed from *formulas* containing strings and other variables. Relative paths for the model graph are relative to the filename of the current model graph. When the file name of an existing model graph is typed into the operator, plab opens the model graph to extract the model graph's parameters from its `StartOperator`, if present. These parameters are split into input and output parameters. Input parameters are copied into new variables, so changes to these values do not affect the calling model graph. Output parameters must be variables and may also be used to pass values to the called model graph, but changes in that model graph affect the same variable in the calling model graph. The operator has the following attributes:

- `string FileName`: File name of the model graph to execute
- Further parameters are extracted from the `StartOperator` in the called model graph



Comparator:

This operator compares pairs of operands, testing for `<`, `<=`, `=`, `>`, `>=`, `!=`, `<>`. The first comparison that evaluates to true yields the number of the edge which is followed to the next node to execute. Further tests are skipped. If there is one more edge than there are tests, this final edge is followed to the next node in case all comparisons failed, similar to an else branch in a text-oriented programming language. The operator has the following multi attribute:

- `double Operand 1`
- `string Test`: one of the operators `<`, `<=`, `=`, `>`, `>=`, `!=`, `<>`
- `double Operand 2`



ForOperator:

Executes its child nodes for all values of a counting variable that is iterated from a start value to an end value with given step width, until the counting variable becomes greater than the end value for positive values of Step, or smaller for negative values. The operator has the following attributes:

- string Counting Variable: Name of an integer variable used for counting. If the variable does not yet exist, it is created as a double variable
- int Start: Start value
- int Stop: End value
- int Step: Step width



WhileOperator:

This operator executes its child operators while the stated condition holds. The operator has the following attributes:

- double Operand 1
- string Test: One of the operators <, <=, =, >, >=, !=, <>
- double Operand 2

B Operators in plab

The following operators are executed in depth-first search order, as discussed in Figure 20:



`AlertOperator`:

This operator can be used to print variables or functions to the console for debugging or as a progress indicator. If a variable is given, the type is determined from the variable. All other cases are evaluated as double values. It has the following multi attribute:

- `Formula`



`AssignmentOperator`:

This operator type allows to declare, initialize and update variables, records and arrays that can be read or written by all following nodes. For initializing arrays, formulas are evaluated for every array member, and these formulas may evaluate the `index` variable to query their own index. `AssignmentOperator` has the following multi attribute:

- `string Variable Name`: Name of the variable
- `string Type`: Type of the variable, in case of declarations. The following types are supported: `int`, `double`, `handle`, `record`, `recordRef`, `string`. In order to create arrays of these types, either the type declaration or the variable name may specify the size of the array in square brackets.
- `string Value`: Default value, calculated per the type set above.



`ControlPointArrayOperator`:

This operator allows to define *control points*, which may be moved freely in the viewport. The point set may be accessed as an array called `<component name>.points`. When the points are moved in the viewport, the changed coordinates are written to this operator, and the model graph is restarted to update the geometry. It has one multi attribute:

- `double x, y, z, w`: Coordinates of the points



ErosionSimulator:

This node type allows to run a simple erosion simulation on an array. The simulation is based on exchanging amounts of water between neighboring cells, rather than a particle simulation. Thermal erosion, material dissolved in water and sand layers are not implemented. Parameters:

- `string Terrain Array`: States the variable containing the terrain grid altitudes.
- `int Terrain size, x direction`
- `int Terrain size, y direction`
- `string Water Array`: States the variable containing the water levels for each grid point.
- `double Rain`: The amount of rain added to each cell in each round.
- `double Fluvial Erosion Factor`: This factor controls how much altitude is decreased as a portion of the water exchanged.
- `int Cycles`: How many exchanges are calculated for each cell in the grid.



idOperator:

This operator does not do anything, it only visits its child nodes. It allows to define alternative call paths for nodes, or executing several nodes from operators that call only a single node. Furthermore, it is useful because it generates a common handle for the geometry of all child objects, and it can be used for cosmetic changes to model graphs.



initDoubleArrayOperator:

Initializes a double array. Its multi attribute defines the values:

- `double Value`



InstanceOperator:

This operator inserts a link to existing geometry at further places of the scene graph. This is called *instancing*. The operator has only a single parameter:

- `Mesh Handle`: A handle to the geometry that should be instanced.



LightSourceOperator:

This operator defines light sources. While material parameters are defined by the textures, the following parameters have to be specified for the light sources:

- `int count`: The number of light sources to create by evaluating the formulas below
- `double x, y, z, w`: Position
- `double Red, Green, Blue`: Relative brightness
- `double Constant Attenuation`
- `double Linear Attenuation`
- `double Quadratic Attenuation`
- `double Ambient Factor`



LoadVariablesOperator:

This operator allows to load variables from an XML file. The file must have been created by SaveVariablesOperator or use a compatible format. Child nodes are executed only if at least one variable was loaded from the XML file. It has the following attributes:

- `string File Name`: May be absolute or relative to the current model graph's path



MultiplyAlongVectorOperator:

This operator multiplies geometry generated by child nodes along a vector. It has the following attributes:

- `int Count`: The number of instances to create
- `bool MultiplyComponents`: Whether to multiply geometry for each instance or to use the same geometry in the scene graph every time (0: reference instance, 1: multiply geometry)
- `double x, y, z`: Functions defining the vector between the instances



MultiplyOnCircleOperator:

This operator multiplies child objects on a circle. Its parameters include:

- double Radius
- int Count: The number of instances to create
- bool MultiplyComponents: Whether to multiply geometry for each instance or to reuse the same geometry in the scene graph every time (0: reference instance, 1: multiply geometry)
- double xnormal, ynormal, znormal: Normal of the circle around which the objects are rotated
- int FaceOutward: Defines whether all objects face in the same direction (0) or away from the center (1)



RenderOperator:

Child nodes of this node are rendered immediately, and the resulting scene graph for these nodes is deleted immediately after that, but variables created by child nodes remain. If the model has been rendered previously, the camera settings are reused for interactive rendering. Otherwise, a default perspective is used. This operator has the following attributes:

- uint Frames: Number of frames to display
- string Screenshot Filename: Optionally allows storing a screen shot to a file



RotateOperator:

This operator rotates child objects around a definable axis. Its parameters include:

- double Rotation axis, x
- double Rotation axis, y
- double Rotation axis, z
- double Rotation angle: Measured in degrees.



SaveTextureOperator:

Saves part of a texture as a file. Its attributes are:

- string Texture Name: plab texture handle variable
- string File Name: File name to use for saving the texture
- uint Left, Top, Width, Height: Define the part of the texture to store



SaveVariablesOperator:

This operator allows to store a list of variables in an XML file. These files can be reloaded by LoadVariablesOperator or other programs. Its attributes are:

- `string File Name`: May be relative to the current model graph's path or absolute
- `string Variable Name`: A multi attribute containing the names of the variables to store in the file



ScaleOperator:

This operator scales child objects. It has the following attributes:

- `double xscale`
- `double yscale`
- `double zscale`



StartOperator:

This operator works much like an AssignmentOperator, except that it defines the parameters for the model graph as a whole when called using CallOperator. This operator normally does not have a parent node, and all other nodes should be child nodes of this node. If the model graph is executed directly, variables are created for all parameters using the specified default parameters.

- Multi attribute for input parameters:
 - `string Variable Name`: Name of the parameter/variable
 - `string Type`: Type of the variable. May include the size of an array in square brackets.
 - `string Value`: Default Value, used when this model graph is executed directly and not called from a model
- Multi attribute for output parameters:
 - `string Variable Name`: Name of the parameter/variable
 - `string Type`: Type of the variable. May include the size of an array in square brackets.
 - `string Value`: Default Value, used when this model graph is executed directly and not called from a model graph.



SplitEdgeOperator:

This operator allows to split an edge in a mesh by inserting a vertex. It has the following attributes:

- Mesh Handle
- Vertex 1 index
- Vertex 2 index
- x, y, z, u, v : Coordinates of the new vertex



TransformOperator:

This operator allows to define arbitrary transform matrices, e.g. for shear matrices. Its attributes define the 16 matrix values:

- `double value[16]`



TranslationOperator:

This operator allows to translate child objects. Its attributes include:

- `double x, y, z`: Translation vector



WaitOperator:

This operator stops executing the model graph for a number of microseconds. It is useful to reduce CPU load while rendering using `RenderOperator`, or for games that are implemented as model graphs. It has one parameter:

- `int n`: Number of microseconds to wait

C Texture Generators

Nodes that create textures are called texture generators, which are derived from `TextureBase`. These node types also set material values for *diffuse* and *specular reflection*. When executed, a texture generator creates the following variables:

- `<texture name>.texture`: A handle which is specified in order to apply the texture with a component,
- `<texture name>.width`: Width of the texture,
- `<texture name>.height`: Height of the texture.

Textures are added to *texture atlases* to increase rendering performance, so widths and heights do not need to be powers of 2, see Section 4.5.2.



`TextureFile`:

This node type allows to read a texture from a file. It has the following attribute:

- `string File Name`



`RoughTextureGenerator`:

This node type generates a texture from random noise that interpolates between two colors to create the impression of a rough surface. It has the following attributes:

- `int width, height`
- `double minRed, minGreen, minBlue, maxRed, maxBlue, maxGreen`



`ProceduralTextureGenerator`:

Evaluates a user-defined function for every texel. While calculating the texels, it may be useful to query the variables `xindex`, `yindex` for the current texel position. Its attributes include:

- `int width, height`
- **Variables:** Additional variables and assignments that are executed for every pixel
 - Variable Name
 - Type
 - Value
- `double red, green, blue`: These functions are evaluated for every texel



`TransparentTextureGenerator`:

Similar to `ProceduralTextureGenerator`, but adds an alpha channel. Its attributes include:

- `width, height`
- `double red, green, blue, alpha`: These functions are evaluated for every texel

D Components

Components create scene graph nodes and add geometry. These nodes are derived from `ComponentBase`. In `plab`, geometry is always textured, so a texture must be given as a parameter. This parameter is not listed below, as it is shared by all components. Please see Section 4.3.3 for additional information on how handles are created and used by the components.



`AddPolygonComponent`:

Creates a new mesh consisting of a single polygon or adds a polygon to an existing mesh. For named vertices, a record variable is created that contains the index and x, y, z, u, v coordinates of the new point. The new polygon is created by stating vertex indices. Polygons may use vertices from other polygons in the same mesh, so this operator can be used to create closed meshes. Normals are assumed to be orthogonal to the polygon for all vertices in the polygon. `AddPolygonComponent` has the following attributes:

- `Container Handle`: Every component creates a handle `<component name>.mesh`. Such handles may be stated here to add geometry to a mesh created by another component.
- `Index Base`: The base index for accessing the vertex coordinates
- `Multi attribute` – defines vertices:
 - `double x, y, z`: Vertex coordinates
 - `double u, v`: Texture coordinates
 - `string Name`: Name of a variable that stores the index of the new vertex – ignored if empty
- `Multi attribute` – connects vertices to polygons
 - `int Index`: Vertex index



AddPolygonWithNormalsComponent:

Similar to AddPolygonComponent, but allows the user to state normals for the vertices. It has the following attributes:

- Container Handle: Every component creates a handle `<component name>.mesh`. Such handles may be stated here to add geometry to a mesh created by another component.
- Index Base: The base index for accessing the vertex coordinates
- Multi attribute – defines vertices
 - double `x, y, z`: Vertex coordinates
 - double `nx, ny, nz`: Coordinates of the normal
 - double `u, v`: Texture coordinates
 - string `Name`: Name of a variable that stores the index of the new vertex – ignored if empty
- Multi attribute – connects vertices to polygons
 - int `Index`: Vertex index



AreaComponent:

Creates an area from a two-dimensional function at the given resolution. The surrounding polygons of each vertex are used to calculate a normal for that vertex (smooth shading). It has the following attributes:

- string `Name of count variable 1`
- double `Variable 1 start value`
- double `Variable 1 end value`
- string `Name of count variable 2`
- double `Variable 2 start value`
- double `Variable 2 end value`
- double `x, y, z, w`: Functions defining the vertex positions. These will be evaluated for variables 1 and 2 as defined above. The variables are incremented by 1 for each iteration.



ConeComponent:

This node type creates a cone. It has the following attributes:

- double `Thickness`
- double `Length`
- int `Triangle Count`



CubeComponent:

This node type creates a cube. It has the following attributes:

- double x Length
- double y Length
- double z Length



CylinderComponent:

This node type creates a cylinder. It has the following attributes:

- double Thickness
- double Length
- int Triangle Count



MeshFileComponent:

This node type allows loading geometry from external files. It has the following attribute:

- string File Name



PolygonComponent:

This node type is used to define polygons. The number of vertices in the polygon may be chosen freely, with texture coordinates. plab calculates a normal perpendicular to the polygon, which is used for the entire polygon. PolygonComponent has the following multi attribute:

- double x, y, z, u, v



QuadStripComponent:

Creates an area from two functions defining its top and bottom. The surrounding polygons of each vertex are used to calculate a normal for that vertex (smooth shading). QuadStripComponent has the following attributes:

- int Polygons: The number of polygons to create
- string Count Variable: The name of the counting variable
- double Start: Starting value of the counting variable
- double Stop: Stop value of the counting variable
- double x1, y1, z1, u1, v1: Function 1
- double x2, y2, z2, u2, v2: Function 2



SphereComponent:

Creates an ellipsoid. It has the following attributes:

- double x Radius
- double y Radius
- double z Radius
- int Stacks
- int Slices



StemComponent:

This component uses an array to generate a tubelike form from a number of generalized cylinders. The resulting mesh is closed and well-suited for objects like branches, pillars or tori. StemComponent has the following attributes:

- double[] Skeleton Points: the first three components define the x, y, z coordinates of each vertex, the fourth gives the radius of the mesh for that vertex
- int Start Index
- int Segments
- int Complexity

E Functions

The following functions may be used in plab:

Function	Returns
random	A random number between 0 and 1
min (a, \dots, z)	Minimum of a comma-separated list of numbers
max (a, \dots, z)	Maximum of a comma-separated list of numbers
abs (x)	Absolute value $ x $ of x
sign (x)	Sign of x : -1, 0 or 1
sin (α)	sine of α
cos (α)	cosine of α
tan (α)	tangent of α
asin (α)	arcsine, inverse of sine, in degrees
acos (α)	arccosine, inverse of cosine, in degrees
atan (α)	arctangent, inverse of tangent, in degrees
trunc (x)	truncates x
sqrt (x)	\sqrt{x} , 0 if negative
length (\vec{x})	Euclidian length of \vec{x}
size (A)	The number of elements in array A
getRed (T, u, v)	Red color component of a texel $(u, v) \in [0; 1]^2$ in texture T , as a value between 0 and 1
getGreen (T, u, v)	similar to getRed, for green component
getBlue (T, u, v)	similar to getRed, for blue component
getAlpha (T, u, v)	similar to getRed, for alpha component
bezierX, bezierY, bezierZ	Evaluate an array containing x, y, z values as a <i>Bezier</i> spline
time	Returns the number of seconds since midnight, January 1st, 1970, UTC, as a floating-point number
keypressed	Returns 1 if a key was pressed that can be read using readkey
readkey	Returns the key code of the last key that was pressed
round(x)	Rounds a floating-point value x to the nearest integer
round(x, y)	Rounds x to the nearest integer multiple of y
LSdistance	Computes the distance between two line segments. Each line segment is defined by two points, which are stored in a <code>double array[16]</code>
triangleCount(handle)	Returns the number of triangles in a mesh
triangleVertex(handle, index)	Returns the index of a vertex in a triangle
quadCount(handle)	Returns the number of quads in a mesh
quadVertex(handle, index)	Returns the index of a vertex in a quad
meshVertexX(handle, index, var)	Returns the x coordinate of a vertex
meshVertexY(handle, index, var)	Returns the y coordinate of a vertex
meshVertexZ(handle, index, var)	Returns the z coordinate of a vertex
meshVertexU(handle, index, var)	Returns the u coordinate of a vertex
meshVertexV(handle, index, var)	Returns the v coordinate of a vertex

F Hints for Programming in plab

A number of classic hints for programming apply also for plab. As it may not be obvious how they apply in the context of a visual programming language, here are some hints:

- Use comments in formulas:
To place a comment at the end of a line, prefix the comment with //.
- Use meaningful names:
plab names operators automatically, but often other names give the user a better idea of the purpose of an operator, variable, parameter or model graph.
- Replace constants with variables:
Instead of using literal values several times in a model graph, it is preferable to use variables that are assigned only once. This eases changing constants that turn out to depend on other parameters.
- Break model graphs into smaller parts where possible:
This will increase readability and reusability of your code.
- Export important settings as parameters:
In order to increase reusability of code, carefully choose which parameters are exported.
- Pass textures as parameters:
Passing textures as parameters means that called model graphs do not need to reproduce them. This may cut from texture memory requirements, but users should be careful to avoid visible repetitions.

G Parameters for Terrain Modeling

The following parameters are used to define a planet in our procedural model:

- Basic shape type (sphere, ring, flat polygons),
- Size or radius,
- #Continents,
- Percentage of land area to total planet's surface,
- Minimum and maximum edge length at top LOD, or
- #Triangles for base shape,
- Colors for sea, land, mountain tops,
- Min. and max. mountain elevation,
- Min. and max. river slope,
- Precipitation: rain per polygon or unit area (on average, this equals evaporation),
- Probability that a river vertex is turned into a lake,
- Lake depth to lake size ratio.

H Modeling Software

Modeling suites:

- Autodesk 3ds Max 2013, <http://www.autodesk.com/3dsmax>
- Autodesk Maya 2013, <http://www.autodesk.com/maya>
- Maxon Cinema 4D 13, <http://www.maxon.de>
- Luxology Modo 601, <http://www.luxology.com/modo/>
- Autodesk Softimage 2013, <http://www.softimage.com>
- Houdini 12.0, <http://www.sidefx.com/>
- Lightwave 3D 11, <http://www.newtek.com/lightwave/>

Plant Editors:

- Xfrog, <http://www.greenworks.de>
- Garden Suite 2.133, <http://www.onyxtree.com>
- natFX, <http://www.bionatics.com>
- SpeedTree 6, <http://www.speedtree.com/>
- Plant-Life und Treemagik G3, <http://www.aliencodec.com>
- Arbaro 1.9.8, <http://arbaro.sourceforge.net>

CAD:

- Autodesk Inventor 2013, <http://www.autodesk.com/inventor>

Architecture:

- AutoCAD Architecture 2013, <http://www.autodesk.de/autocadarchitecture>
- Autodesk Revit 2012, <http://www.autodesk.de/revitarchitecture-trial>
- Bentley Architecture V8i, <http://www.bentley.com>
- Computerworks Vectorworks 2012 Architekt, <http://www.computerworks.de/vectorworks.html>
- Nemetschek Allplan 2012, <http://www.nemetschek.de>
- ArchiCAD 15, <http://www.graphisoft.com/products/archicad>

Fractal Worlds:

Software Title	Website
Bryce 7	http://www.daz3d.com
Daylon Leveller 4.0	http://www.daylongraphics.com
Mojoworld 3.1	http://www.pandromeda.com
Terragen 2.4	http://www.planetside.co.uk
VUE Infinite 10	http://www.e-onsoftware.com
WorldBuilder Pro 4	http://www.digi-element.com
World Machine 2.2	http://www.world-machine.com

Freeware:

- Blender 2.62, <http://www.blender.org>
- Wings3D 1.4.1, <http://www.wings3d.com>
- K-3D 0.8, <http://www.k-3d.org>

References

- [1] William B. Ackerman. Data flow languages. *IEEE Computer*, 15(2):15–25, 1982.
- [2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing, Boston, 1986.
- [3] Fabricio Anastacio, Mario Costa Sousa, Faramarz Samavati, and Joaquim A. Jorge. Modeling plant structures using concept sketches. In *NPAR '06: Proceedings of the 4th international symposium on Non-photorealistic animation and rendering*, pages 105–113, 2006.
- [4] Kenneth Appel and Wolfgang Haken. *Every Planar Map is Four-Colorable*. American Mathematical Society, 1989.
- [5] Eric Armstrong. Improving real-time motion. In *SIGGRAPH '07: ACM SIGGRAPH 2007 sketches*, New York, 2007. ACM.
- [6] J. W. Backus. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. *Proceedings of the International Conference on Information Processing, IFIP Congress, UNESCO*, pages 125–132, July 1959.
- [7] James M. Bardeen. Fractal rivers in fractal landscapes. *Physica Scripta*, T 38(5), July 1991.
- [8] Farès Belhadj. Terrain modeling: a constrained fractal model. In *Afrigraph*, pages 197–204, 2007.
- [9] Farès Belhadj and Pierre Audibert. Modeling landscapes with ridges and rivers. In *VRST '05: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 151–154, New York, 2005. ACM.
- [10] Bedrich Benes, Ondrej Stava, Radomir Mech, and Gavin Miller. Guided procedural modeling. In *Computer Graphics Forum (Eurographics)*, 2011.
- [11] Bedřich Beneš. Hydraulic erosion by shallow water simulation. In *The 4th Workshop on Virtual Reality Interaction and Physical Simulation - Vriphys'07*, 2007.
- [12] Bedřich Beneš, Nathan Andryscio, and Ondrej Št'ava. Interactive modeling of virtual ecosystems. In *Eurographics Workshop on Natural Phenomena 2009*, 2009.
- [13] Bedřich Beneš and Rafael Forsbach. Layered data representation for visual simulation of terrain erosion. In *SCCG '01: Proceedings of the 17th Spring conference on Computer graphics*, page 80, Washington, DC, USA, 2001. IEEE Computer Society.
- [14] Bedřich Beneš and Rafael Forsbach. Parallel implementation of terrain erosion applied to the surface of mars. In *AFRIGRAPH '01: Proceedings of the 1st international conference on Computer graphics, virtual reality and visualisation*, pages 53–57, New York, 2001. ACM.

- [15] Bedřich Beneš and Rafael Forsbach. Visual simulation of hydraulic erosion. In *WSCG Proceedings of the 10-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, pages 120–132, 2002.
- [16] Jules Bloomenthal. A representation for botanical trees using density distributions. In *1st Int'l Conf. on Engineering and Computer Graphics*, 1984.
- [17] Martin Bokeloh and Michael Wand. Hardware accelerated multi-resolution geometry synthesis. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games, I3D '06*, pages 191–198, New York, 2006. ACM.
- [18] George Boole. *An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities*. Macmillan, 1854.
- [19] Frederic Boudon, Przemyslaw Prusinkiewicz, Pavol Federl, Christophe Godin, and Radoslaw Karwowski. Interactive design of bonsai tree models. In *Proceedings of Eurographics 2003: Computer Graphics Forum 22 (3)*, pages 591–599, 2003.
- [20] John Brosz, Faramarz F. Samavati, and Mario Costa Sousa. Terrain synthesis by-example. In *Proceedings of the first International Conference on Computer Graphics Theory and Applications*, 2006.
- [21] E. Bruneton and F. Neyret. Real-time rendering and editing of vector-based terrains. *Computer Graphics Forum*, 27(2):311–320, 2008.
- [22] Brent Burley and Dylan Lacewell. Ptex: Per-face texture mapping for production rendering. In *Eurographics Symposium on Rendering 2008*, pages 1155–1164, 2008.
- [23] S. Burris. The laws of Boole's thought. Manuscript, 2000.
- [24] Nathan A. Carr and John C. Hart. Meshed atlases for real-time procedural solid texturing. *ACM Transactions on Graphics*, 21(2):106–131, 2002.
- [25] S. Chen and D. Gordon. Front-to-back display of BSP trees. *IEEE Computer Graphics & Algorithms*, pages 79–85, September 1991.
- [26] Xuejin Chen, Boris Neubert, Ying-Quing Xu, Oliver Deussen, and Sing Bing Kang. Sketch-based tree modeling using Markov random field. In *Proceedings of the Siggraph 2008 Asia*, 2008.
- [27] Norishige Chiba, Kazunobu Muraoka, and Kunihiko Fujita. An erosion model based on velocity fields for the visual simulation of mountain scenery. *The Journal of Visualization and Computer Animation*, 9(4):185 – 194, 1997.
- [28] Lydia Choy, Ryan Ingram, Ocean Quigley, Brian Sharp, and Andrew Willmott. Rigblocks: Player-deformable objects. In *SIGGRAPH '07: ACM SIGGRAPH 2007 sketches*, New York, 2007. ACM.
- [29] John Clark and Derek Allan Holton. *A first look at graph theory*. World Scientific Publishing, Co. Pte. Ltd., 1991.

- [30] Jonathan M. Cohen, Lee Markosian, Robert C. Zeleznik, John F. Hughes, and Ronen Barzel. An interface for sketching 3d curves. In *I3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 17–21, New York, 1999. ACM.
- [31] Kate Compton, James Grieve, Ed Goldman, Ocean Quigley, Christian Stratton, Eric Todd, and Andrew Willmott. Creating spherical worlds. In *SIGGRAPH '07: ACM SIGGRAPH 2007 sketches*, New York, 2007. ACM.
- [32] Carlúcio Cordeiro, Wagner Corrêa, and Luiz Chaimowicz. Parallel lazy amplification: Real-time procedural modeling and rendering of multi-terabyte scenes on a single pc. In *Proceedings of the SBGames 2008*, 2008.
- [33] Barbara Cutler, Julie Dorsey, Leonard McMillan, Matthias Müller, and Robert Jagnow. A procedural approach to authoring solid models. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 302–311, New York, 2002. ACM.
- [34] Carsten Dachsbacher and Marc Stamminger. Rendering procedural terrain by geometry image warping. In *EGSR04: 15th Eurographics Symposium on Rendering*, pages 103–110, 2004.
- [35] Phillippe de Reffye, Claude Edelin, Jean Françon, Marc Jaeger, and Claude Puech. Plant models faithful to botanical structure and development. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 151–158, 1988.
- [36] Paul E. Debevec, Camillo J. Taylor, and Jitendra Malik. Modeling and rendering architecture from photographs: a hybrid geometry- and image-based approach. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 11–20, New York, 1996. ACM.
- [37] David DeBry, Henry Goffin, Chris Hecker, Ocean Quigley, Shalin Shodhan, and Andrew Willmott. Player-driven procedural texturing. In *SIGGRAPH '07: ACM SIGGRAPH 2007 sketches*, New York, 2007. ACM.
- [38] Patrick Degener, Jan Meseth, and Reinhard Klein. An adaptable surface parametrization method. In *The 12th International Meshing Roundtable 2003*, September 2003.
- [39] Evgenij Derzapf, Björn Ganster, Michael Guthe, and Reinhard Klein. River networks for instant procedural planets. *Special Edition of Computer Graphics Forum (Pacific Graphics 2011)*, 30(7):2031–2040, 2011.
- [40] Oliver Deussen, Pat Hanrahan, Bernd Lintermann, Radomir Měch, Matt Pharr, and Przemyslaw Prusinkiewicz. Realistic modeling and rendering of plant ecosystems. In *Proceedings of ACM SIGGRAPH 98*, pages 275–286, New York, 1998. ACM Press.
- [41] Oliver Deussen and Bernd Lintermann. *Digital Design of Nature – Computer Generated Plants and Organics*. Springer, 2005.
- [42] Willibald Dörfler and Werner Peschek. *Einführung in die Mathematik für Informatiker*. Hanser, 1988.

- [43] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. Roaming terrain: real-time optimally adapting meshes. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 81–88, Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.
- [44] N. Dyn, J. Gregory, and D. Levin. A 4-point interpolatory scheme for curve design. In *CAGD 4 (1987)*, pages 257–268, 1987.
- [45] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.
- [46] Alexei A. Efros and William T. Freeman. Image quilting for texture synthesis and transfer. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 341–346, New York, 2001. ACM.
- [47] Alain Fournier, Don Fussell, and Loren Carpenter. Computer rendering of stochastic models. *Commun. ACM*, 25(6):371–384, 1982.
- [48] James Gain, Patrick Marais, and Wolfgang Straßer. Terrain sketching. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games*, New York, 2009.
- [49] Eric Galin, Adrien Peytavie, Nicolas Maréchal, and Eric Guérin. Procedural generation of roads. In *Computer Graphics Forum: Proceedings of Eurographics*, volume 29, Norrköping, Sweden, May 2010. Eurographics Association.
- [50] Björn Ganster and Reinhard Klein. An integrated framework for procedural modeling. In *Spring Conference on Computer Graphics 2007 (SCCG 2007)*, pages 150–157. Comenius University, Bratislava, April 2007.
- [51] Björn Ganster and Reinhard Klein. 1-2-tree: Semantic modeling and editing of trees. In O. Deussen, D. Keim, and D. Saupe, editors, *Vision, Modeling, and Visualization 2008 (VMV 2008)*, pages 51–60. Akademische Verlagsgesellschaft Aka GmbH, Heidelberg, October 2008.
- [52] Martin Gardner. Mathematical games - the fantastic combinations of John Conway's new solitaire game "life". October 1970.
- [53] Björn Gerth, René Berndt, Sven Havemann, and Dieter W. Fellner. 3D modeling for non-expert users with the Castle Construction Kit v0.5. In *VAST 2005: 6th International Symposium on Virtual Reality, Archaeology and Intelligent Cultural Heritage*, pages 49–58, November 2005.
- [54] Frank Goetz, Ralf Borau, and Gitta Domik. An XML-based visual shading language for vertex and fragment shaders. In *Web3D '04: Proceedings of the ninth international conference on 3D Web technology*, pages 87–97, New York, 2004. ACM Press.
- [55] David A. D. Gould. *Complete Maya programming - An extensive guide to MEL and the C++ API*. Elsevier, San Francisco, 2002.
- [56] H. Gravelius. *Flusskunde*. Goschen, Berlin, 1914.

- [57] P.J. Green. Reversible jump Markov chain Monte Carlo computation and Bayesian model determination. In *Biometrika* 82, pages 711–732, 1995.
- [58] N. Greene. Voxel space automata: Modeling with stochastic growth processes in voxel space. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 175–184, 1989.
- [59] Xianfeng Gu, Steven J. Gortler, and Hugues Hoppe. Geometry images. *ACM Transactions on Graphics*, 21(3):355–361, 2002.
- [60] Paul E. Haeberli. Conman: a visual programming language for interactive graphics. *SIGGRAPH Computer Graphics*, 22(4):103–111, 1988.
- [61] U. Theodore Hammer. *Saline Lake Ecosystems of the World*. 1986.
- [62] James Scott Hanan. *Parametric L-systems and their application to the modelling and visualization of plants*. PhD thesis, 1992.
- [63] John C. Hart and Thomas A. DeFanti. Efficient antialiased rendering of 3-d linear fractals. *SIGGRAPH Computer Graphics*, 25(4):91–100, 1991.
- [64] Sven Havemann. *Generative mesh modeling*. PhD thesis, TU Braunschweig, 2005.
- [65] Olof Henricsson, Andre Streilein, and Armin Gruen. Automated 3-d reconstruction of buildings and visualization of city models. In *Proceedings of the Workshop on 3D City Models*, Bonn, Germany.
- [66] Houssam Hnaidi, Eric Guérin, Samir Akkouche, Adrien Peytavie, and Eric Galin. Feature based terrain generation using diffusion equation. *Computer Graphics Forum (Proceedings of Pacific Graphics)*, 29(7):2179–2186, 2010.
- [67] M. Hoffmann. Verse: Paralleles arbeiten am gleichen Modell. *Digital Production*, March/April 2007.
- [68] M. Holton. Strands, gravity and botanical tree imagery. *Computer Graphics Forum 13 (1)*, pages 57–67, 1994.
- [69] Hisao Honda. Description of the form of trees by the parameters of the tree-like body: Effects of the branching angle and the branch length on the shape of the tree-like body. *Journal of Theoretical Biology*, pages 331–338, 1971.
- [70] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley Longman, 2001.
- [71] Carl Hultquist, James Gain, and David Cairns. Affective scene generation. In *AFRIGRAPH '06: Proceedings of the 4th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, pages 59–63, New York, 2006. ACM.
- [72] Takashi Ijiri, Shigeru Owada, and Takeo Igarashi. The sketch L-system: Global control of tree modeling using free-form strokes. In *6th International Symposium on Smart Graphics 2006, LNCS 4073*, pages 138–146. Springer Verlag, 2006.

- [73] Takashi Ijiri, Shigeru Owada, Makoto Okabe, and Takeo Igarashi. Floral diagrams and inflorescences: Interactive flower modeling using botanical structural constraints. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 720–726, 2005.
- [74] Martin Ilcik, Stefan Fiedler, Werner Purgathofer, and Michael Wimmer. Procedural skeletons: Kinematic extensions to CGA-shape grammars. In *Proceedings of the Spring Conference on Computer Graphics 2010*, pages 177–184. Comenius University, Bratislava, 5 2010.
- [75] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys (CSUR)*, 36(1):1–34, 2004.
- [76] Alex D. Kelley, Michael C. Malin, and Gregory M. Nielson. Terrain simulation using a model of stream erosion. *SIGGRAPH Computer Graphics*, 22:263–268, June 1988.
- [77] Tom Kelly and Peter Wonka. Interactive architectural modeling with procedural extrusions. *ACM Transactions on Graphics*, 30(2):14:1–14:15, April 2011.
- [78] Nancy Y. Kiang. The color of plants on other worlds. *Scientific American*, pages 28–35, April 2008.
- [79] L. Krecklau, D. Pavic, and L. Kobbelt. Generalized use of non-terminal symbols for procedural modeling. *Computer Graphics Forum*, 2010.
- [80] Lars Krecklau and Leif Kobbelt. Procedural modeling of interconnected structures. *Computer Graphics Forum*, 30(2):335–344, 2011.
- [81] Peter Kristof, Bedřich Beneš, Jaroslav Krivanek, and Ondrej Št'ava. Hydraulic erosion using smoothed particle hydrodynamics. In *Proceedings of Eurographics 2009: Computer Graphics Forum 28 (2)*, 2009.
- [82] R. G. Laycock and A. M. Day. Automatically generating roof models from building footprints. In *WSCG*, 2003.
- [83] Aristid Lindenmayer. Mathematical models for cellular interactions in development, parts I and II. *Journal of Theoretical Biology*, 18:280–315, 1968.
- [84] Bernd Lintermann and Oliver Deussen. A modelling method and user interface for creating plants. *Computer Graphics Forum*, 17(1):73–82, 1998.
- [85] Markus Lipp, Daniel Scherzer, Peter Wonka, and Michael Wimmer. Interactive modeling of city layouts using layers of procedural content. *Computer Graphics Forum (EG 2011)*, 2011.
- [86] Markus Lipp, Peter Wonka, and Michael Wimmer. Interactive visual editing of grammars for procedural architecture. In *Proceedings of ACM SIGGRAPH 2008*, 2008.
- [87] Markus Lipp, Peter Wonka, and Michael Wimmer. Parallel generation of L-systems. In Holger Theisel Marcus Magnor, Bodo Rosenhahn, editor, *Vision, Modeling, and Visualization Workshop 2009 (VMV)*, November 2009.

- [88] Yotam Livny, Soeren Pirk, Zhanglin Cheng, Feilong Yan, Oliver Deussen, Daniel Cohen-Or, and Baoquan Chen. Texture-lobes for tree modelling. In *ACM SIGGRAPH 2011 papers*, SIGGRAPH '11, pages 53:1–53:10, New York, USA, 2011. ACM.
- [89] Yotam Livny, Feilong Yan, Matt Olson, Baoquan Chen, Hao Zhang, and Jihad El-Sana. Automatic reconstruction of tree skeletal structures from point clouds. *ACM Transactions on Graphics, (Proceedings SIGGRAPH Asia 2010)*, 29(5):151:1–151:8, 2010.
- [90] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *SIGGRAPH Computer Graphics*, 21(4):163–169, 1987.
- [91] Frank Losasso and Hugues Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 769–776, New York, 2004. ACM.
- [92] Chongyang Ma, Li-Yi Wei, and Xin Tong. Discrete element textures. In *ACM SIGGRAPH 2011 papers*, SIGGRAPH '11, pages 62:1–62:10, New York, 2011. ACM.
- [93] Benoit B. Mandelbrot. *The fractal geometry of nature*. Freeman and Company, New York, 1983.
- [94] Jean-Eudes Marvie, Julien Perret, and Kadi Bouatouch. The FL-system: A functional L-system for procedural geometric modeling. *The Visual Computer*, 21:329–339, 2005. 10.1007/s00371-005-0289-z.
- [95] Xing Mei, Philippe Decaudin, and Bao-Gang Hu. Fast hydraulic erosion simulation and visualization on GPU. In *15th Pacific Conference on Computer Graphics and Applications, Pacific Graphics 2007, November, 2007*, pages 47–56, Maui, Hawaii, USA, November 2007. IEEE.
- [96] Stefan Menz, Holger Dammertz, Johannes Hanika, Michael Weber, and Hendrik P. A. Lensch. Graphical interface models for procedural mesh growing. In *Vision, Modeling, and Visualization (VMV 2010)*, 2010.
- [97] Paul Merrell. Example-based model synthesis. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games, I3D '07*, pages 105–112, New York, 2007. ACM.
- [98] Paul Merrell and Dinesh Manocha. Continuous model synthesis. *ACM Transactions on Graphics*, 27:158:1–158:7, December 2008.
- [99] Paul Merrell and Dinesh Manocha. Constraint-based model synthesis. In *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling, SPM '09*, pages 101–111, New York, 2009. ACM.
- [100] Paul Merrell and Dinesh Manocha. Technical section: Example-based curve synthesis. *Computer Graphics*, 34:304–311, August 2010.
- [101] Paul Merrell, Eric Schkufza, and Vladlen Koltun. Computer-generated residential building layouts. *ACM Transactions on Graphics*, 29:181:1–181:12, December 2010.

- [102] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. *ACM Transactions on Graphics*, 25(3):614–623, 2006.
- [103] F. Kenton Musgrave, Craig E. Kolb, and Robert S. Mace. The synthesis and rendering of eroded fractal terrains. *SIGGRAPH Computer Graphics*, 23(3):41–50, 1989.
- [104] Radomír Měch and Przemyslaw Prusinkiewicz. Visual models of plants interacting with their environment. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 397–410, New York, 1996. ACM.
- [105] Benjamin Neidhold, Oliver Deussen, and Markus Wacker. Interactive physically based fluid and erosion simulation. In *Eurographics Workshop on Natural Phenomena*, 2005.
- [106] Boris Neubert, Thomas Franken, and Oliver Deussen. Approximate image-based tree-modeling using particle flows. *ACM Transactions on Graphics*, 2007.
- [107] J. Von Neumann and A. W. Burks. Theory of self-reproducing automata. *ACM Transactions on Graphics*, 1966.
- [108] Larry Niven. *Ringworld*. Ballantine Books, 1970.
- [109] Makoto Okabe and Takeo Igarashi. 3d modeling of trees from freehand sketches. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Sketches & Applications*, pages 1–1, 2003.
- [110] Makoto Okabe, Shigeru Owada, and Takeo Igarashi. Interactive design of botanical trees using freehand sketches and example-based editing. In *Proceedings of Eurographics 2005: Computer Graphics Forum 24 (3)*, pages 487–496, 2005.
- [111] Peter E. Oppenheimer. Real time design and animation of fractal plants and trees. In *Computer Graphics (Proceedings of SIGGRAPH 86)*, pages 55–64, 1986.
- [112] Alexandrina Orzan, Adrien Bousseau, Holger Winnemöller, Pascal Barla, Joëlle Thollot, and David Salesin. Diffusion curves: a vector representation for smooth-shaded images. *ACM Transactions on Graphics*, 27:92:1–92:8, August 2008.
- [113] Wojciech Palubicki, Kipp Horel, Steven Longay, Adam Runions, Brendan Lane, Radomír Měch, and Przemyslaw Prusinkiewicz. Self-organizing tree models for image synthesis. *ACM Transactions on Graphics*, 28:58:1–58:10, July 2009.
- [114] Yoav I. H. Parish and Pascal Müller. Procedural modeling of cities. In *Proceedings of ACM SIGGRAPH 2001*, pages 301–308, New York, 2001. ACM Press, E. Fiume, Ed.
- [115] Heinz-Otto Peitgen, Hartmut Jürgens, and Dietmar Saupe, editors. *Chaos and Fractals – New Frontiers of Science – Second Edition*. Springer, 2004.
- [116] Ken Perlin. An image synthesizer. *SIGGRAPH Computer Graphics*, 19:287–296, July 1985.

- [117] Ken Perlin. Improving noise. *ACM Transactions on Graphics*, 21:681–682, July 2002.
- [118] Adrien Peytavie, Eric Galin, Jérôme Grosjean, and Stéphane Mérillou. Procedural generation of rock piles using aperiodic tiling. *Computer Graphics Forum*, 28(7):1801–1809, 2009.
- [119] Adrien Peytavie, Eric Galin, Stéphane Merillou, and Jerome Grosjean. Arches: a Framework for Modeling Complex Terrains. *Computer Graphics Forum*, 28:457–467, 2009.
- [120] Kevin P. Picott, Brent McPherson, Angus W. Davis, and Ichanahalli V. Nagnendra. System and method for using dependency graphs for the control of a graphics creation process, 1998. United States Patent US 5,929,864.
- [121] Kevin P. Picott, Brent McPherson, Angus W. Davis, and Ichanahalli V. Nagnendra. System and method for using dependency graphs for the control of a graphics creation process, 1999. United States Patent US 5,929,864.
- [122] Joachim Pouderoux, Jean-Christophe Gonzato, Ireneusz Tobor, and Pascal Guiton. Adaptive hierarchical RBF interpolation for creating smooth digital elevation models. In *GIS '04: Proceedings of the 12th annual ACM international workshop on Geographic information systems*, pages 232–240, New York, 2004. ACM.
- [123] Joanna L. Power, A. J. Bernheim Brush, Przemyslaw Prusinkiewicz, and David H. Salesin. Interactive arrangement of botanical L-system models. In *I3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 175–182, 1999.
- [124] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing (3rd Edition)*. Cambridge University Press, 2007.
- [125] P. Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer-Verlag, New York, 1990.
- [126] Przemyslaw Prusinkiewicz and Mark Hammel. A fractal model of mountains with rivers. In *Graphics Interface '93*, pages 174–180, 1993.
- [127] Przemyslaw Prusinkiewicz, Mark James, and Radomír Měch. Synthetic topiary. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 351–358, New York, 1994. ACM.
- [128] Przemyslaw Prusinkiewicz, Radoslaw Karwowski, and Brendan Lane. The L+C plant modeling language. *Functional-Structural Plant Modelling in Crop Production*.
- [129] Przemyslaw Prusinkiewicz, Lars Mündermann, Radoslaw Karwowski, and Brendan Lane. The use of positional information in the modeling of plants. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 289–300, New York, 2001. ACM.

- [130] Long Quan, Ping Tan, Gang Zeng, Lu Yuan, Jingdong Wang, and Sing Bing Kang. Image-based plant modeling. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 599–604, 2006.
- [131] Alex Reche-Martinez, Ignacio Martin, and George Drettakis. Volumetric reconstruction and interactive rendering of trees from photographs. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 720–727, 2004.
- [132] William T. Reeves. Particle systems — a technique for modeling a class of fuzzy objects. *ACM Transactions on Graphics*, 2(2):91–108, 1983.
- [133] William T. Reeves and Ricki Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 313–322, 1985.
- [134] S. Roettger, W. Heidrich, P. Slusallek, and H.-P. Seidel. Real-time generation of continuous levels of detail for height fields. In *Proceedings of the WSCG '98*, pages 315–322, 1998.
- [135] Lorenz Rogge, Christian Lipski, and Marcus Magnor. Visualization of the continental drift in real-time. volume 18, pages 9–16, Plzen, Czech Republic, 3 2010. UNION Agency – Science Press.
- [136] Tatsumi Sakaguchi and Jun Ohya. Modeling and animation of botanical trees for interactive virtual environments. In *VRST '99: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 139–146, 1999.
- [137] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 4.0)*.
- [138] Ilya Shlyakhter, Max Rozenoer, Julie Dorsey, and Seth Teller. Reconstructing 3d tree models from instrumented photographs. *IEEE Computer Graphics and Applications*, 21(3):53–61, 2001.
- [139] Ruben Smelik, Tim Tutenel, Klaas Jan de Kraker, and Rafael Bidarra. Integrating procedural generation and manual editing of virtual worlds. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games, PCGames '10*, pages 2:1–2:8, New York, 2010. ACM.
- [140] Alvy Ray Smith. Plants, fractals, and formal languages. In *Computer Graphics (Proceedings of ACM SIGGRAPH 84)*, pages 1–10, New York, 1984. ACM Press.
- [141] John M. Snyder. *Generative modeling for computer graphics and CAD: Symbolic shape design using interval analysis*. Academic Press Professional, San Diego, 1992.
- [142] S. Stachniak and W. Stuerzlinger. An algorithm for automated fractal terrain deformation. In *In Proceedings of Computer Graphics and Artificial Intelligence*, pages 64–76, 2005.
- [143] Ondrej Stava, Bedrich Benes, Matthew Brisbin, and Jaroslav Krivanek. Interactive terrain modeling using hydraulic erosion. pages 201–210, Dublin, Ireland, 2008. Eurographics Association.

- [144] Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.
- [145] George Stiny. *Pictorial and Formal Aspects of Shapes and Shape Grammars*. Birkhauser, Basel, Switzerland, 1975.
- [146] Paul S. Strauss and Rikk Carey. An object-oriented 3D graphics toolkit. In *Computer Graphics (Proceedings of ACM SIGGRAPH 92)*, pages 341–349, New York, 1992. ACM Press.
- [147] Jing Sun, Xiaobo Yu, George Baciú, and Mark Green. Template-based generation of road networks for virtual city modeling. In *VRST '02: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 33–40, New York, 2002. ACM.
- [148] R. Szeliski and D. Terzopoulos. From splines to fractals. *Computer Graphics (Proceedings of ACM SIGGRAPH 89)*, 23:51–60, July 1989.
- [149] Jerry Talton, Yu Lou, Steve Lesser, Jared Duke, Radomír Měch, and Vladlen Koltun. Metropolis procedural modeling. *ACM Transactions on Graphics*, 30(2), April 2011.
- [150] Ping Tan, Tian Fang, Jianxiong Xiao, Peng Zhao, and Long Quan. Single image tree modeling. In *SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 papers*, pages 1–7, New York, 2008. ACM.
- [151] Alan M. Turing. On computable numbers with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.
- [152] Stanislaw Ulam. On some mathematical problems connected with patterns of growth of figures. *Mathematical Problems in the Biological Sciences - Proceedings of the Fourteenth Symposium of the American Mathematical Society*, pages 215–224, 1961.
- [153] Torsten Ullrich, Christoph Schinko, and Wolf-Dietrich Fellner. Procedural modeling in theory and practice. pages 5 – 8, Plzeň, Czech Republic, 2010.
- [154] Carlos A. Vanegas, Daniel G. Aliaga, Bedřich Beneš, and Paul A. Waddell. Interactive design of urban spaces using geometrical and behavioral modeling. *ACM Transactions on Graphics*, 28(5):1–10, 2009.
- [155] Juraj Vanek, Bedřich Beneš, Adam Herout, and Ondrej Stava. Large-scale physics-based terrain editing using adaptive tiles on the GPU. *IEEE Computer Graphics and Applications: Special Issue - Digital-Content Authoring*, 2011.
- [156] D. J. Walton and D. S. Meek. Technical section: A controlled clothoid spline. *Computers & Graphics*, 29(3):353–363, 2005.
- [157] B. Watson, P. Müller, P. Wonka, C. Sexton, O. Veryovka, and A. Fuller. Procedural urban modeling in practice. *Computer Graphics and Applications, IEEE*, 28:18 – 26, 2008.

- [158] Basil Weber, Pascal Müller, Peter Wonka, and Markus Gross. Interactive geometric simulation of 4d cities. *Computer Graphics Forum*, 28(2):481–492, 2009.
- [159] Jason Weber and Joseph Penn. Creation and rendering of realistic trees. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 119–128, 1995.
- [160] E. R. Weibull. *Morphometry of the human lung*. Springer Verlag, 1963.
- [161] Andrew Willmott. Fast object distribution. In *SIGGRAPH '07: ACM SIGGRAPH 2007 sketches*, New York, 2007. ACM.
- [162] Jamie Wither, Frédéric Boudon, Marie-Paule Cani, and Christophe Godin. Structure from silhouettes: a new paradigm for fast sketch-based design of trees. *Computer Graphics Forum*, 28(2):541–550, 2009.
- [163] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. Instant architecture. *ACM Transactions on Graphics*, 22(3):669–677, 2003.
- [164] Howard Zhou, Jie Sun, Greg Turk, and James M. Rehg. Terrain synthesis from digital elevation models. *IEEE Transactions on Visualization and Computer Graphics*, 13(4):834–848, July/August 2007.

Index

- 1-2-tree, 91–93, 96, 103–105, 111, 113, 136–140
- 3ds Max, 33, 37, 69
- level of detail, 145
- abstract node type, 43
- Accessibility, 107
- acyclic, 4
- AddPolygonComponent, 53, 57, 61, 64, 66, 71, 157
- AddPolygonWithNormalsComponent, 50, 64, 71, 158
- adjacency list, 5, 42
- adjacent, 4
- AlertOperator, 149
- alpha channel, 62
- anchor value, 95
- animation, 20
- AreaComponent, 64, 65, 158
- arrays, 37, 56
- AssignmentOperator, 44, 51, 56, 74, 86, 89, 149, 153
- attribute, 9, 20, 38, 42
- attribute editor, 51
- Autoparallelization, 37
- Backus-Naur form, 41, 94
- base model graph, 108
- base script, 106, 108
- Bezier, 26, 73, 77, 161
- billboards, 15
- blossoms, 92
- branch level, 93
- branch/rib link, 19
- branches, 43, 92
- BranchOperator, 86, 147
- bud, 7
- buildings, 74
- CallOperator, 58–61, 70, 73, 74, 85, 147, 153
- calyx, 73
- CGA shape, 22
- child link, 19
- Child Parameter Set, 95
- Church-Turing thesis, 139
- Cinema 4D, 33, 34, 37, 69, 108
- Comparator, 43, 45, 52, 83, 86, 147
- component, 43, 64, 71
- ComponentBase, 43, 157
- ConeComponent, 64, 158
- connected, 4
- context-free, 16
- context-sensitive, 16, 107, 113
- continent, 120
- control operators, 43, 147
- control points, 14, 37, 69, 75, 88, 89, 94, 108, 149
- ControlPointArrayOperator, 69, 149
- conversion rules, 21
- CPS, 95
- create geometry, 64, 98, 129
- CubeComponent, 64, 159
- Customizability, 107
- cycle, 4, 39
- CylinderComponent, 64, 159
- D.E.M., 76
- data structure, 56
- database amplification, 11, 12
- decomposition graph, 18, 103, 104
- decussation, 7
- deterministic L-system, 16
- Deutsch limit, 88
- dichasium, 7
- dictionary stacks, 14
- diffuse reflection, 155
- digital elevation model, 76, 80
- directed, 4
- dispersion, 7
- distichy, 7
- double, 56
- dynamic binding, 58
- dynamic procedural content, 12, 34
- ease of use, 105, 107, 108, 113, 135
- edges, 4
- endorheic basins, 120
- erosion simulation, 27, 80
- ErosionSimulator, 150
- evaporation, 120
- extrusion, 12
- fluvial erosion, 27

formulas, 37, 41, 45, 46, 50, 55, 64, 70, 80, 88, 147
 ForOperator, 43, 53, 55, 74, 81, 148
 forward kinematics, 33
 Four Color Theorem, 4, 123
 fractals, 14
 fruits, 92

 generality, 37, 135
 generalized cylinders, 98
 generative modeling, 13
 Genesis effect, 15
 geometry, 92
 geometry clipmap, 29
 gizmos, 14
 glBegin, 6, 71, 100
 glDrawElements, 6
 glEnd, 6, 71, 100
 glVertex, 6, 71, 100
 GML, 13, 69, 88
 Golden angle, 7
 Google Earth, 29
 graph, 4
 Gravelius numbering scheme, 4
 gravitropism, 7, 18, 95
 Greedy evaluation, 11, 12
 grid, 79
 Growth animations, 20

 Hausdorff Besicovitch dimension, *see* Hausdorff dimension
 Hausdorff dimension, 14, 84
 heap, 56
 heliotropism, 7, 17
 house, 74

 IDOperator, 85
 idOperator, 83, 150
 Imposters, 15
 incidence list, 5
 incident, 4
 incoming edge, 4
 indexed faceset, 5, 41, 90
 Infix notation, 88
 initDoubleArrayOperator, 150
 input parameters, 58
 input ports, 9
 InstanceOperator, 47, 57, 61, 150
 instances, 18
 instancing, 7, 47, 150

 int, 56
 interactive exploration, 117
 internode, 7
 interpolated distribution, 94
 intersection tests, 22
 Intuitive Tools, 107
 inverse kinematics, 33

 keyframe, 20
 keyframe animation, 20, 33
 Koch curve, 83

 L-string, 15
 L-system, 15, 21, 37, 89
 lakes, 127
 Lazy evaluation, 11, 12
 leaf link, 19
 leaves, 92
 level of detail, 2, 12, 22, 24, 26, 29, 30, 35, 37, 100, 114–116, 123–125, 127, 128, 131–133, 140, 142, 143
 level set, 13
 levels of detail, 79
 LightSourceOperator, 50, 151
 LoadVariablesOperator, 72, 151, 153
 Loft extrusions, 12
 Loops, 10, 37, 43

 Marching Cubes Algorithm, 13
 marguerite, 73
 mass updates, 93, 94
 Maya, 33, 34, 37, 69
 mesh, 41
 meshes, 5
 MeshFileComponent, 72, 159
 midpoint displacement, 24, 117
 mipmapping, 28, 29
 model graph, 38, 42, 88
 Modularization, 37
 MojoWorld, 25
 monochasium, 7
 Monopodial branching, 7
 mouse polygon, 96
 mouse ray, 68, 96
 Move tool, 94
 multi attribute, 42
 multi edges, 4
 MultiplyAlongVectorOperator, 50, 151
 MultiplyOnCircleOperator, 50, 152

 node, 4, 9, 68

- node type, 42
- NodeBase, 43, 147
- octree, 22
- OperatorBase, 43
- Operators, 43
- outgoing, 4
- output parameters, 58
- output ports, 9
- parameterization, 6, 92
- Parameterized L-systems, 16
- parameters, 58, 92
- parse tree, 70
- particle effects, 33
- particle system, 15, 17, 25
- path, 4
- Performance, 107
- persistence, 107, 113
- petals, 73
- Phototropism, 7
- phyllotactic layout, 73
- pipelines, 9
- plab, 39, 69, 108, 139
- planet creation, 117, 118
- plant skeleton, 15, 17
- playfield, 85
- pollen, 73
- PolygonComponent, 46, 52, 64, 71, 159
- Polygons, 5
- positional information, 16, 92
- postfix notation, 13, 88
- power of expression, 135
- Procedural modeling, 11
- ProceduralTextureGenerator, 62, 63, 155, 156
- Pseudorandom number generators, 11
- pure vertex, 117
- QuadStripComponent, 64, 159
- race conditions, 61
- radii, 94
- random seed, 11
- raytracing, 6, 33
- reconstructing buildings, 31
- record, 56
- RecordDefinition, 144
- recordRef, 57, 61
- recursion, 58
- recursion factors, 17
- Recursive, 93
- recursive scene graph traversal, 92
- refinement, 125
- registers, 14
- RenderOperator, 67, 79, 152, 154
- reproduce, 107
- reproduced, 129
- reproducible, 11, 76
- reverse Polish notation, 13
- river network, 15, 25, 27, 114, 116, 117, 126, 129, 139, 141
- root (comp. sc.), 4
- root node, 42
- roots (botany), 92
- RotateOperator, 47, 152
- rotations, 47
- RoughTextureGenerator, 51, 62, 155
- Same Level, 93
- SaveTextureOperator, 152
- SaveVariablesOperator, 72, 151, 153
- Saw tool, 94
- ScaleOperator, 47, 153
- scene graph, 7, 41, 92
- SceneFactory, 61, 113
- scope, 40, 44, 56, 70, 145
- Scripting language, 37
- seepage, 120
- selection, 75, 93, 94
- selection modes, 103
- selection options, 92, 105
- semantic entities, 92
- semantic selection, 107, 113
- Sierpinski triangle, 84
- single assignment convention, 9
- Sketch tool, 94
- Sketch-based modeling, 30, 107, 113
- sketching, 93, 94
- solid textures, 28
- specificness, 135
- specular reflection, 155
- SphereComponent, 64, 160
- Split grammars, 21
- Split rules, 21
- SplitEdgeOperator, 154
- Stability, 106, 113
- StartOperator, 51, 56, 58, 61, 73, 75, 76, 78–80, 83, 147, 153
- State sorting, 6

static procedural content, 12, 34
 stem, 73
 StemComponent, 64, 73, 74, 160
 stochastic L-system, 16
 strand model, 17
 street network, 20
 street patterns, 21
 subdivision surfaces, 14
 subroutine, 58
 Surfaces of revolution, 12
 sympodial branching, 7

 tags, 94
 targetted updates, 107, 113, 139
 template, 98, 106, 108, 110, 111, 113, 137, 138
 terrain, 76
 Tetris, 85
 TextureBase, 155
 texture, 20, 62
 texture atlas, 6, 21, 28, 71, 96, 155
 texture baking, 28
 texture generator, 43
 TextureBase, 43
 TextureFile, 62, 72, 155
 Texturing, 37
 thermal erosion, 27
 threads, 61, 131
 tool model graph, 108
 tool pipeline, 18
 tool script, 106, 108
 tower, 74
 TransformOperator, 154
 TranslationOperator, 47, 154
 translations, 47
 transparency, 62
 TransparentTextureGenerator, 62, 156
 tree (botany), 92
 tree (computer sc.), 4
 tree growth, 17
 trunk, 92
 Turing machine, 86
 Turing-complete, 38
 twigs, 92

 undirected, 4
 user interface, 93
 user-programmable, 136
 UV unwrapping, 6

 variables, 37, 56

 VDFP, 9, 37, 39, 89
 vertices, 4
 viewport editing, 37, 89, 103, 107, 138, 139
 viewport selection, 97
 visual dataflow pipeline, 9
 Voronoï, 30, 78
 Voronoï diagram, 21, 28, 76
 voxels, 17

 WaitOperator, 67, 154
 water levels, 123
 Weibull numbering scheme, 4
 WhileOperator, 43, 55, 86, 148
 window, 74

 Xfrog, 19, 37, 39, 89, 103