An Integrated Framework for Procedural Modeling

Björn Ganster * Reinhard Klein †

Abstract

This paper proposes a new type of visual language to integrate the features of previous procedural modeling systems into a single modeling environment. As in a visual dataflow pipeline, we let nodes wrap operations, but instead of using pipelines to define dataflow, we use edges to define the order of execution. Models can be created efficiently without needing time-consuming compilation runs or learning an unintuitive syntax, and the new system offers a mechanism that can alter procedural models in the viewport. An example demonstrates how to use the new system to create complex models consisting of buildings, plants and landscapes procedurally without resorting to external tools.

CR Categories: I.3.5 [Computing Methodologies]: Computer Graphics—Computational Geometry and Object Modeling; I.3.7 [Computing Methodologies]: Computer Graphics—Three-Dimensional Graphics and Realism; D.3.3 [Software]: Programming Languages—Language Constructs and Features; I.6.3 [Computing Methodologies]: Simulation and Modeling—Applications

Keywords: Computer Graphics and Modeling, Procedural Modeling, Visual Programming Languages

1 Introduction

Procedural modeling aims at automatic creation of large scenes for films and computer games through algorithms or sets of rules. The parameters and rules producing the scene are often very small compared to the file size of the model. This is known as data amplification [Smith 1984]. The level of detail can be controlled by a parameter that governs the number of polygons created for different parts of the scene. Incorporating random variables allows new variants of the model to be created automatically, but the same random seed can be used to reproduce the model.

We will require some concepts from graph theory: 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. Often, $E \subset V \times V$. If an edge $e \in E$ from $a \in V$ to $b \in V$ is directed, it cannot be followed in the reverse direction, but there may be an opposite edge from b to a. If the edges are undirected, they can be followed in both directions. A path is an ordered set of vertices $v_1, ..., v_n$, where v_i and v_{i+1} are connected by edges. A path that starts and ends with the same vertex $v_1 = v_n$ is called a cycle. If a graph does not contain a cycle it is called acyclic.

L-Systems are a common solution to modeling plants [Prusinkiewicz and Lindenmayer 1990; Lindenmayer 1968], and can also be used for streets and buildings [Parish and Müller 2001]. These systems are based on rules for replacing string parts to derive a high-level description of the model, for example the skeleton of a plant, and generate the geometry in a second step. An alternative grammar for buildings are shape grammars [Stiny 1975; Wonka et al. 2003; Müller et al. 2006]. The drawback of grammars is that they are usually limited to a specific class of models and they require a lot of training and time for experimenting. Furthermore, it is not obvious how to parallelize the rewriting of context sensitive grammars to make use of modern multicore CPUs.

A different procedural approach is used in Xfrog, that also generates very convincing plants. The system allows a user to visually specify so-called p-graphs whose vertices produce primitives or layouts [Lintermann and Deussen 1998]. Xfrog converts the p-graph into the i-tree by replicating multiplier vertices, then it builds the geometry from the i-tree. The conversion is sufficiently fast for interactive display. Although organic architectural designs were among the original design goals, it works best for plants and trees and it is unsuitable for more general modeling tasks. Creating plants is more intuitive in Xfrog than in an L-System.

John Snyder introduced the GenMod system for generative modeling [Snyder 1992] . The system produces 3D shapes from several curves without an intermediate step. It is based on a C interpreter with overloaded operators, therefore it has variables, arrays, loops and formulas in infix notation. More recently, Sven Havemann implemented a similar system that employs a stack-based postfix notation to avoid the need for variables and a parser [Havemann 2005]. His system is called GML, for Generative Modeling Language. GML's applications focus on architecture and its most interesting feature are programmable gizmos. Gizmos are special points that allow the user to edit the parameters of a primitive by moving a handle in the viewport. GML's gizmos allow the user to define how the gizmos are mapped to parameters of an object. This can be duplicated in Maya by taking cubes or spheres as a gizmos, and a script that evaluates their positions to determine new parameters for a model. Programmable gizmos 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 requires human users to perform the conversion from infix to postfix notation although computers can do this faster and more reliable [Aho et al. 1986].

Other possibilities to describe a procedural model are to code it in a programming or scripting language, using scene graph class libraries or storing a list of primitives. A scene graph is a directed acyclic graph whose nodes wrap computer graphics operations and whose edges declare functional dependences [Strauss and Carey 1992]. While scripts are interpreted and therefore do not offer full performance, programs need to be compiled. Thus, it depends on the task at hand whether a program or a script is more handy. For a compiled application, changing parameters requires recompilation, unless a user interface is also implemented.

In a visual dataflow pipeline, each operation is wrapped by a node that may have several input and output attributes. One node's output attribute can be connected with another node's input attribute to form a pipeline, hence the name visual dataflow pipeline (VDFP). VDFPs are used for many tasks in computer graphics, e.g. they can be used to write shaders [Goetz et al. 2004], and they control the dataflow and object dependences in Maya [Gould 2002]. Ackerman established several properties of dataflow languages [Ackerman 1982], among them the single assignment

*e-mail:mail@bjoern-ganster.de
†e-mail:rk@cs.uni-bonn.de

System	Gizmos	Variables	Arrays	Formulas	Local Loops	Modularization	Autoparallelization	Model types
GenMod	No	Yes	Yes	Yes	Yes	Yes	No	any
GML	Yes	No (stack)	Yes	(postfix)	Yes	Yes	No	any
L-Systems	No	No	No	Yes	No	Limited [Hanan 1992]	No	plants, streets
Scripts	Yes: MEL	Yes	Yes	Yes	Yes	Yes	No	any
VDFPs	No	No	Yes	(visual)	Yes	Yes	Yes	any
Xfrog	No	No	No	Yes	No	No	No	plants
Model Graphs	Yes	Yes	Yes	Yes	Yes	Yes	Yes	any

Table 1: We propose model graphs to integrate the features of previous approaches to procedural modeling into a single system

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

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

property, that assert that nodes can be executed concurrently when their inputs have arrived. In a VDFP, the single assignment property is embodied in the fact that every input may be connected to only one output, but one output may be connected to several inputs.

Visual languages visualize structural dependences and the parameters for function calls very well 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 require users to connect icons for each operator in a formula, therefore 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 optimum performance. If each node would be run on a separate processor 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 outline solutions to these problems, but they report these issues to be still open [2004].

2 Problem Statement

Tab. (1) compares important aspects of procedural modeling systems. Gizmos allow a user to model parts of a scene directly in the 3D viewport by dragging handles. For parameters related to positions, this is the most intuitive method of editing. Variables and arrays store values that change over time or are computed iteratively. They are useful for procedural modeling because they can store state for more complex operations. Procedural modeling requires models to be described by formulas, therefore formulas are an important aspect of design for any procedural modeling language. Modularization is required to divide complex models into smaller components and it improves reusability of models. Local loops reflect the ability to loop code fragments as opposed to the global loop of replacements over the text of an L-System. Automatic parallelized execution removes the burden of parallel programming from the user. This is necessary for full performance on multicore CPUs.

Standard 3D graphics suites include scripting languages, store their state in scene graphs, and often include visual dataflow pipelines. But the solutions found in Maya, 3DS Max and Cinema 4D differ too little from their generic implementation to warrant a detailed discussion, so Tab. (1) does not include them. Tab. (2) gives an overview of these packages.

Let us summarize our findings on previous procedural modeling systems. 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 the nodes for formulas may take longer than typing the formula on the keyboard. L-Systems yield results comparable to those of Xfrog, but they require even more time to achieve them. Scripting languages do not offer the ease-of-use found in Xfrog or VDFPs. We conclude that there is no single system bringing together all strengths of procedural modeling.

Similar limitations have been addressed by forming tool pipelines [Deussen et al. 1998; Parish and Müller 2001; Müller et al. 2006]. However, interactions between various parts of the scene must be reflected in the pipeline and require the definition of exchange file formats. Therefore, tool pipelines tend to be more inflexible than integrated solutions. In addition, all separate tools need to be maintained, by paying upgrade fees, installing the upgrades, and training, and code produced for one solution may not be usable in the context of another tool. An integrated solution allows all objects to be viewed and edited in a single application. Additional functionality should be added in the form of plugins.

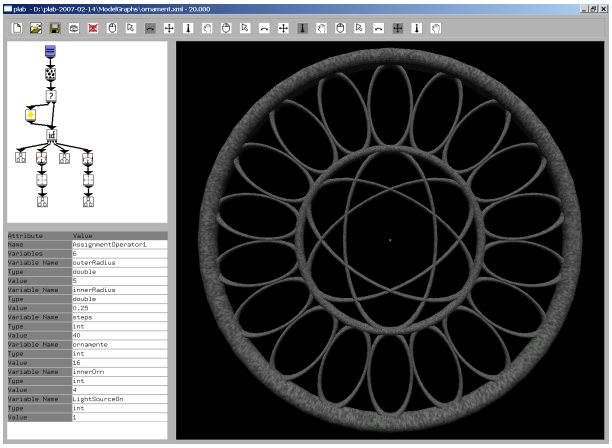

Figure 1: Screenshot of the new system. The model graph is edited in the upper left, the attributes of the selected vertex are displayed below that, and the right half displays the results of running this model graph.

3 The New Visual Language

We propose a new visual system that allows entering formulas in infix notation. Infix notation requires variables, but normal variables defy the single assignment rule of dataflow and thus VDFPs [Ackerman 1982]. The basic idea of the new language is this: A visual language that stores named variables on a heap does not require pipelines to transport data, as a VDFP does. Instead, we can use directed edges to define the order of execution, and have special nodes that perform variable assignments. 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 to justify an own name. We will call them model graphs.

Fig. (1) shows a screenshot of the system. 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. All attributes with numeric results can be computed from formulas which may include comments. The formulas may use the operators +, -, *, /, % (modulo division) and the functions in Tab. (3). Nodes are connected by dragging the target node onto the source node. Dragging the target node onto the source node again disconnects the nodes.

A model graph node may have any number of incoming and outgoing edges, and may be connected to any other node, but not to itself. The nodes in a model graph are visited by depth-first traversal. We display starting edges below a node, and incoming edges above it. Outgoing edges are executed in the same order as their edges start under the node, from left to right. The nodes, their attributes and edges completely define the model graph, which is equivalent to a program.

We introduce the operators that create textures first. All texture operators also define diffuse and specular reflection and specular shininess for use in the Phong lighting model:

Function	Calculates
random	Random number between 0 and 1
min (a, ...,z)	Minimum of comma-separated list
max (a, ...,z)	Maximum of comma-separated list
abs (x)	Absolute $\|x\|$ of x
sign (x)	Sign of x
sin (α)	Sine of α
cos (α)	Cosine of α
tan (α)	Tangent α
asin (α)	Inverse sine
acos (α)	Inverse cosine
atan (α)	Inverse tangent
trunc (x)	Truncate to integer
sqrt (x)	Square root
length ($\vec{x}$)	Length of a vector $\vec{x}$
size (A)	Number of elements in an array
getRed (T,u,v)	red component for texel at (u,v) in texture T in the range [1;0]
getGreen (T,u,v)	Like getRed, for green component
getBlue (T,u,v)	Like getRed, for blue component
getAlpha (T,u,v)	Like getRed, for alpha component
bezierX (array, value)	Evaluate array including x, y, z values as a bezier curve, returns x component
bezierY (array, value)	Like bezierX, for y component
bezierZ (array, value)	Like bezierX, for z component

Table 3: Functions supported in model graphs. All angles are in degrees.

TextureFile:
Loads a texture from a file. It has a parameter stating the file name for the texture.

RoughTextureGenerator:
Chooses a random value $X \in [0;1[$ for every texel and stores $X \cdot \vec{C}_1 + (1-X) \cdot \vec{C}_2$ in a texture. Depending on how different the user-defined colors $\vec{C}_1$, $\vec{C}_2$ are, this produces a rougher or smoother texture.

ProceduralTextureGenerator:
Evaluates formulas for the red, green and blue channels and optional interim values for every texel of a texture. It is more versatile than RoughTextureGenerator, but also slower.

TransparentTextureGenerator:
Similiar to ProceduralTextureGenerator, but it adds an alpha channel.

All operators that create geometry are referred to as components. They require a texture as a parameter:

PolygonComponent:
Creates a single polygon from an arbitrary number of vertices that are given by x, y, z space coordinates and u, v texture coordinates.

AddPolygonComponent:
Creates a new mesh or adds a polygon to an existing mesh. For every new point, a record variable is created that contains the index and coordinates of the new point. The new polygon is created by stating point indices. This operator is useful to create closed meshes.

ConeComponent:
Creates the geometry for a cone. The user may state the radius, length, and number of triangles.

CubeComponent:
Creates the geometry for a cube. The parameters describe the side lengths.

CylinderComponent:
Creates the geometry for a cylinder. There are parameters for the length, radius, and polygon count of the cylinder.

SphereComponent:
Constructs the geometry of a sphere from parameters for the x, y, z radius, and the number of stacks and slices.

Objects are usually created at the origin, and we use matrices to move them. The operators defining the matrices apply to child nodes:

TranslateOperator:
Defines a translation matrix. It has parameters for x, y, z translation.

RotateOperator:
Defines a rotation matrix. The parameters give the axis and the angle of rotation.

ScaleOperators:
This operator scales all subsequent geometry by a scale factor for the different coordinate axes.

TransformOperator:
This operator allows to define arbitrary 4×4 matrices.

The following operator assigns variables:

AssignmentOperator:
Declares, initializes and updates variables, records and arrays. Each of these has a name, type, and value. See Fig. (2) for an example.

Variables are strictly typed as `int`, `double`, or `record`. It is possible to create arrays of these data types. Additional data types are `recordRef`, which stores a reference to a record, and `handle`, which is used to store pointers to geometry.

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)

Figure 2: An example of an AssignmentOperator

The introduction of an if-node allows to us to branch during execution. It defies depth-first traversal in that only one successor is visited:

Comparator:
Compares pairs of values until one comparison evaluates to true. The number of this comparison yields the number of the edge that is followed to the node which is executed next. If there is an additional edge, it will be used in case all comparisons failed, as an ”else” branch.

Model graphs may contain cycles that equal loops and could be discontinued using Comparators. However, using the following control operators is more convenient and less error-prone:

ForOperator:
Counts a variable from a start value to an end value and calls its child nodes for each value.

WhileOperator:
Executes its children while the specified condition holds.

Model graphs may become quite complex, therefore we implemented a means to break them into several modules:

CallOperator:
Executes a model graph as a subroutine, passing user-defined parameters.

Execution starts in any node without incoming edges. But we need a means to define its parameters:

StartOperator:
Defines the input and output parameters for calls from other model graphs. Each parameter has a name, a type and a default value.

If the model graph is run directly, not called from another model graph, the default values in the StartOperator are used to initialize the parameters. After the user has entered the file name of a model graph into a CallOperator, the system extracts the parameters from the StartOperator in that graph and fills in the default parameters. The user may edit these. When the model graph is executed, input parameters are copied into variables, and output parameters are passed as references to variables. In order to perform a recursion, the model graph simply calls itself using CallOperator.

StartOperator and CallOperator provide the means to export repetitive tasks as separate model graphs, allowing additional primitives, mathematical operations and algorithms to be implemented as model graphs. Several vector operations have been implemented in this fashion. Every model graph has its own variable scope. Therefore, variables defined in a model graph are only visible in that model graph for that call, unless passed as an output parameter. There are no global variables.

Specialized components for creating geometry include:

QuadStripComponent:
Adds a strip of quads constructed from two functions for the x, y, z, u, v coordinates with user-defined start, stop and step width parameters.

AreaComponent:
Constructs a surface from a two-dimensional function. The parameters give the start and end values for the variables, and the function is entered as x, y, z, w components.

StemComponent:
Connects a series of generalized cylinders from an array of points and radii to a tube-like surface. It is helpful for trunks, branches and twigs of plants or arches on buildings.

Normally, the result of a model graph is displayed after it has been executed. The following operator allows for interactive model graphs:

RenderOperator:
This operator creates a scene from its subsequent vertices for immediate display. This makes animations and interactive applications such as games possible.

WaitOperator:
This operator waits for a specified time. This is useful to reduce CPU load in animations.

Some other important node types are:

GizmoOperator:
Stores an array of points that the user may edit by keyboard or move in the viewport. In the latter case, the model graph is executed automatically to produce a new model from the parameters derived from the gizmos.

AlertOperator:
Displays the values of functions and variables when processed. It is useful for debugging and displaying progress information during lengthy operations.

SaveVariablesOperator:
The user may state a number of variables to store in an XML file. This is useful for storing the results of a simulation or other computation for later use.

LoadVariablesOperator:
This operator will load all variables from an XML file that was created using SaveVariablesOperator or any other file that uses a compatible format. This allows a user to separate algorithms and data, or to include data from external sources.

LightSourceOperator:
Defines a light source. Its position is given by x, y, z, w coordinates, its brightness is given as red, green and blue channels, and there are parameters for attenuation. Reflected light is simulated by a parameter named ambient factor.

idOperator:
This operator does nothing other than visit its child nodes.

MeshFileComponent:
In order to exchange geometry with other programs, models may be exported in .obj format and can be included through this component.

We store frequently used functions as parse trees in C++ classes to reduce parsing overhead, therefore repeated arithmetic operations and access to variables or constants incur only a virtual method call compared to native code. Model graphs are cached to further reduce loading and parsing time. Full performance on multicore hardware requires parallel execution of code, but parallel execution is beyond the knowledge of most computer users. A model graph called from a CallOperator can be executed parallel to its caller provided that there are no output parameters and the CallOperator does not have any outgoing edges. This kind of parallelization requires no action on the part of the artist. 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. These optimizations assure the performance we strive for. The new system was implemented in C++ and uses the OpenGL, PThreads and FreeGlut libraries. The model graphs are stored in XML file format.

4 Examples

We will prove the versatility of the new system by demonstrating how model graphs can be used to model trees, buildings and landscapes.

The first model graph in Fig. (3) creates trees using a technique described by Oppenheimer [1986]. Numbers are displayed beside

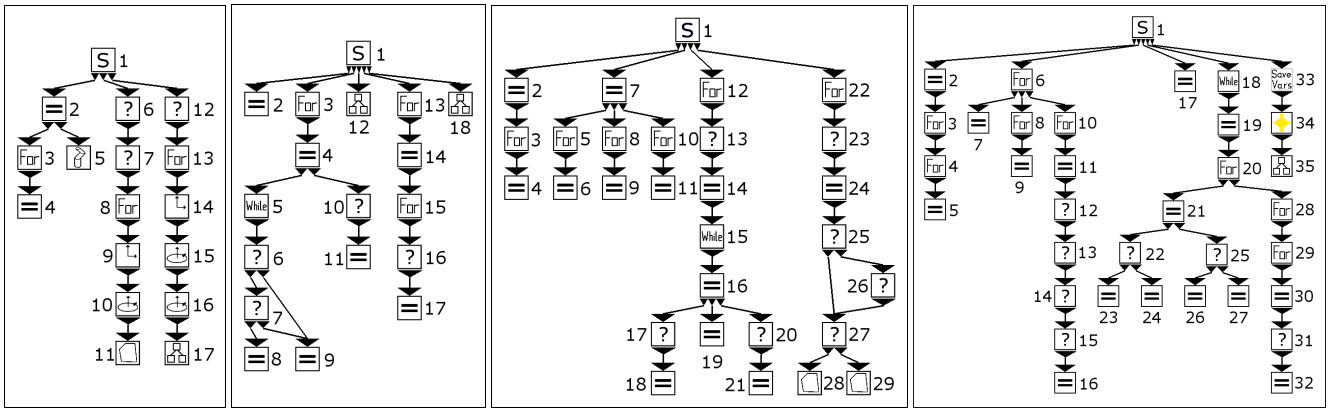

Figure 3: From left to right: the first model graph creates a tree, the second calculates the order of points on a roof (`calcRoofOrder`), the third creates the geometry for a roof (`roof`) , and the fourth creates a hill landscape

the icons to clarify the order of execution, but they are not visible in the editor. Execution starts in StartOperator1 (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 twig, the thickness of the stem, the number of recursions to perform, and the textures to use.

Node 2 is an AssignmentOperator. It declares a new variable that will store the starting positions of the new branches (`stem`). Node 3 is a ForOperator executing node 4 in a loop that initializes `stem`. Fig. (2) shows the assignments of node 4. Execution continues with the next child of node 2, node 5. It creates the geometry for the trunk of the tree from `stem` and a bark texture.

Nodes 6 and 7 test if enough recursions have been performed and 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 in a loop with the aid of a translation in node 9 and a rotation in node 10. Node 11 creates a leaf polygon.

Nodes 12-17 create twigs by recursion. Node 12 tests if further recursions are skipped by testing the parameter `recursions`. Node 13 is another ForOperator that runs nodes 14-17 in a loop to create twigs. Nodes 14-16 transform the new twig. Node 17 is a CallOperator that performs the recursion necessary to produce the next level of twigs.

A sketch-system for buildings has been implemented using model graphs. Pascal Müller's CityEngine System [Parish and Müller 2001; Müller et al. 2006] allows placing and editing buildings in the viewport, but we aim for a solution that makes interactive editing of roofs possible. Our sketch system for buildings computes a building from gizmos that define the corners of a building and its roof. Moving these points allows the user to edit buildings interactively at a lower resolution. This requires several model graphs. `calcRoofOrder` sorts the input points by height and assigns each point a `roofOrder` depending on its height, see Fig. (3). Two points have the same `roofOrder` if they have approximately the same height (subject to a parameter `heightTolerance`). This information is used to compute the geometry automatically.

calcRoofOrder's StartOperator (node 1) takes the following parameters: `frontPoints` gives the number of points defining the facade of the building. `frontPoints` is a parameter since it cannot be computed. If all points defining the front have the same height, counting them would give `frontPoints`, but as demonstrated by saddle roofs, not all front points need to have the same height. `points` contains the corners of the building and the roof. The points defining the facade 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 coordinates of the points array into an array `heightsFound` that contains each height level only once (height levels: intervals of `heights`$\pm$`heightTolerance`). Node 12 calls another model graph that sorts `heightsFound`. Nodes 13-17 assign a `roofOrder` to each point by finding the closest height level. Node 18 calls a separate model graph (`generateFacade`) to produce a facade from the points, including windows and entrances. Since the creation of the facade, windows, and entrances consists mainly of polygon lists and low-level computations, these will not be presented in detail.

The model graph for roof creation (`roof`) has a StartOperator taking the same arguments as that in `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 the roof polygons. The overall time complexity of this model graph is $O(n^2)$. A voronoi diagram would reduce asymptotic costs, but would likely be slower for the small numbers of points required to define a building.

The model graph SlopingHills, shown right in Fig. (3), calculates a heightfield for a mountain range. StartOperator1 (node1) defines its global parameters: the resolution of the heightfield to create (`xsize`, `ysize`), the number, height and slope of the mountains. Node 2 declares the arrays `arr` and `delta`. `arr` will store the heightfield, while `delta` stores the slope of a point in the heightfield to its highest neighbour. `delta` is required because we need the slope between two points to be reproducible. Nodes 3-5 initialize `arr` and `delta`.

Node 6 executes nodes 7-16 in a for loop to create the mountain ridges. Nodes 8 and 9 create a bezier curve for the mountain peak

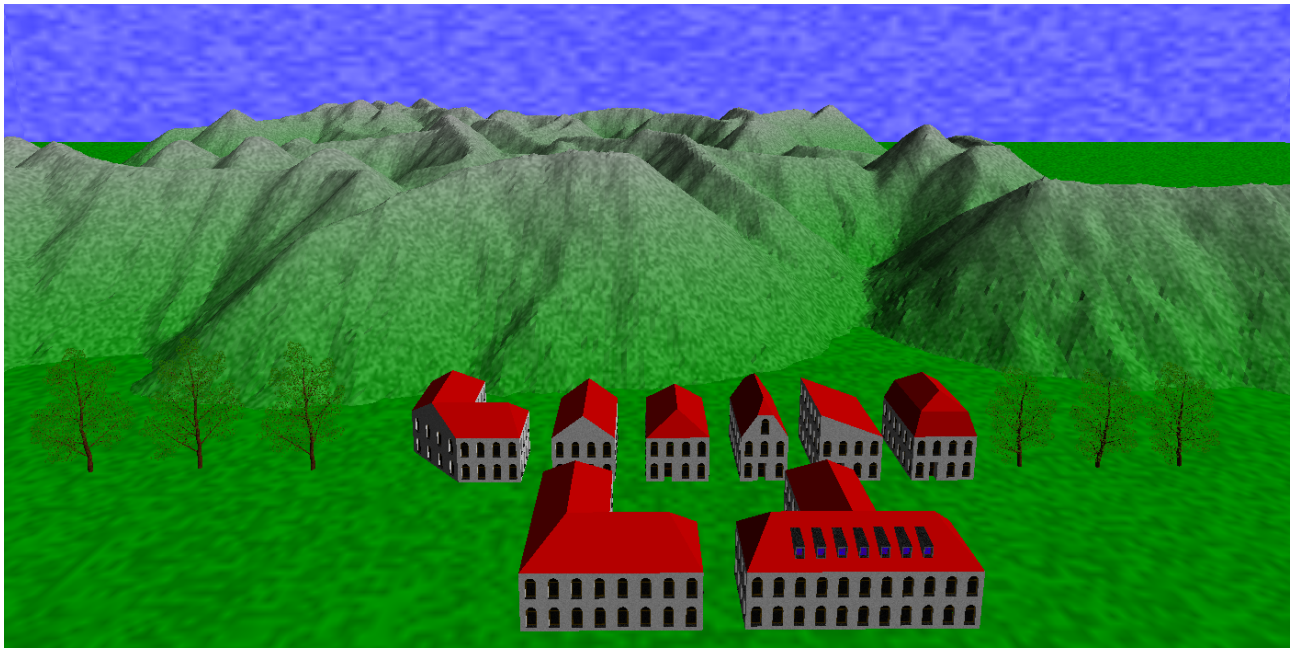

Figure 4: Model graphs can be used to model trees, buildings and landscapes

starting from the position selected in node 7. Nodes 10-16 copy that curve into the heightfield. The mountain slopes are calculated for each cell in the heightfield by subtracting `delta` from the highest neighbour, multiplied by that neighbour's distance ($\sqrt{2}$ for diagonal neighbours, 1 otherwise). Node 18 executes nodes 19-32 until all cells have received their final values.

To speed up computation of neighbours, the directions of the for loops in nodes 28, 29 have to be changed every turn, otherwise height values would be propagated by only one cell against the direction of the for loops every turn. Node 17 defines the directions, nodes 21 to 27 select the appropriate values for nodes 28, 29. Finally, nodes 28-32 propagate the height values through the heightfield.

After propagation of the height values 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 converts the heightfield into polygons. A separate model graph smoothes the hills created above depending on the height of a cell. This is useful to recreate the effect of rolling hills.

Fig. (4) shows an image that uses the model graphs discussed in this section. Geometry creation takes about 27s using a single thread, and 17s using multiple threads on an AMD Opteron 180 with 2400MHz and 2.5 GB RAM. The mountain is read from a file. The scene consists of more than 800.000 polygons.

5 Discussion

The number of variables, polygons and textures are limited only by the hardware. Despite the optimizations, model graphs are currently an interpreted language and thus do not reach the performance of compiled applications. This is a limit in case of computationally expensive operations, but it could be overcome through just-in-time-compilation or by exporting model graphs as C++ 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 distributed into several modules and large model graphs benefit from a scroll bar.

Tab. (1) 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 uses all of the features in the table.

While GML stores all of its data on the stack, 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 reliable than humans can. Model graphs visualize functional dependences better than the postfix notation used in GML. The common strength of both systems are the gizmos that add interactive editing in the viewport to procedural modeling.

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 data has arrived, model graphs store data on a heap and use edges to define a strict order of execution. As a consequence, VDFPs require nodes for all operators in a formula, while model graphs allow functions to be typed faster in infix notation. 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 delicate scheduling. Model graphs have a very intuitive notation for loops.

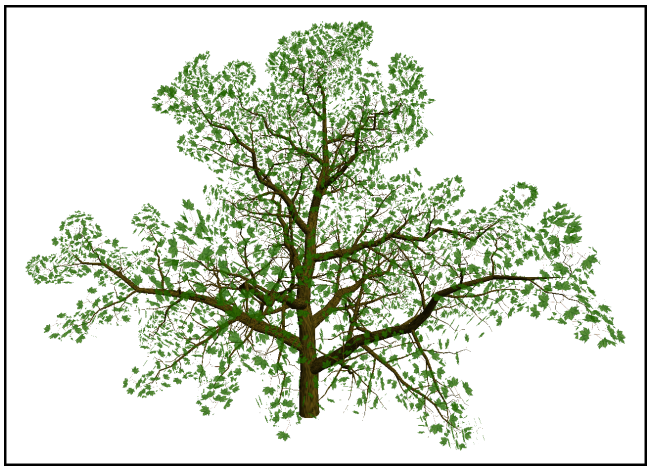

Figure 5: A tree produced using model graphs

As stated earlier, visual languages like VDFPs, Xfrog and model graphs have advantages to textual languages since they visualize functional dependences and function parameters better. Using grammar-based systems requires a lot of experience and time for experimenting 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 gizmos. While Xfrog's nodes are well-suited for plants, model graph nodes are more versatile, as proven by our example.

A demo version of the software is available from www.procedural-modeling.com/plab.

6 Future Work

The new system could be used to integrate the tool pipelines outlined in [Deussen et al. 1998] and [Parish and Müller 2001], [Müller et al. 2006] into a single modeling environment to reduce user efforts of data exchange between the tools. This will leave artists more time to create virtual landscapes.

Currently, primitives have to be added by editing the model graph directly. The interface could therefore be improved by introducing additional ways to edit the scene in the viewport and let the program reflect the necessary changes in the model graph automatically, similar to standard 3D modeling packages. If editing and replacing steps are recorded in a model graph, this may pave the way for intuitive, visual L-Systems. Beside procedural modeling, the system could be used as a visual environment for producing portable games, since the platform supports Windows and Linux.

Operators could be created for various procedural modeling techniques, including Euler operators, constructive solid geometry (CSG), NURBS, subdivision surfaces, hardware shading and shadows.

References

ACKERMAN, W. B. 1982. Data flow languages. *IEEE Computer 15*, 2, 15–25.

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing, Boston.

DEUSSEN, O., HANRAHAN, P., LINTERMANN, B., MĚCH, R., PHARR, M., AND PRUSINKIEWICZ, P. 1998. Realistic modeling and rendering of plant ecosystems. In *Proceedings of ACM SIGGRAPH 98*, ACM Press, New York, 275–286.

GOETZ, F., BORAU, R., AND DOMIK, G. 2004. An XML-based visual shading language for vertex and fragment shaders. In *Web3D '04: Proceedings of the ninth international conference on 3D Web technology*, ACM Press, New York, 87–97.

GOULD, D. A. D. 2002. *Complete Maya programming - An extensive guide to MEL and the C++ API*. Elsevier, San Francisco.

HANAN, J. S. 1992. *Parametric L-systems and their application to the modelling and visualization of plants*. PhD thesis.

HAVEMANN, S. 2005. *Generative mesh modeling*. PhD thesis, TU Braunschweig.

JOHNSTON, W. M., HANNA, J. R. P., AND MILLAR, R. J. 2004. Advances in dataflow programming languages. *ACM Comput. Surv. 36*, 1, 1–34.

LINDENMAYER, A. 1968. Mathematical models for cellular interactions in development, parts I and II. *Journal of Theoretical Biology 18*, 280–315.

LINTERMANN, B., AND DEUSSEN, O. 1998. A modelling method and user interface for creating plants. *Computer Graphics Forum 17*, 1, 73–82.

MÜLLER, P., WONKA, P., HAEGLER, S., ULMER, A., AND GOOL, L. V. 2006. Procedural modeling of buildings. *ACM Transactions on Graphics 25*, 3, 614–623.

OPPENHEIMER, P. E. 1986. Real time design and animation of fractal plants and trees. In *Computer Graphics (Proceedings of SIGGRAPH 86)*, ACM Press, New York, 55–64.

PARISH, Y. I. H., AND MÜLLER, P. 2001. Procedural modeling of cities. In *Proceedings of ACM SIGGRAPH 2001*, ACM Press, E. Fiume, Ed., New York, 301–308.

PRUSINKIEWICZ, P., AND LINDENMAYER, A. 1990. *The algorithmic beauty of plants*. Springer-Verlag, New York.

SMITH, A. R. 1984. Plants, fractals, and formal languages. In *Computer Graphics (Proceedings of ACM SIGGRAPH 84)*, ACM Press, New York, 1–10.

SNYDER, J. M. 1992. *Generative modeling for computer graphics and CAD: symbolic shape design using interval analysis*. Academic Press Professional, San Diego.

STINY, G. 1975. *Pictorial and Formal Aspects of Shapes and Shape Grammars*. Birkhauser, Basel, Switzerland.

STRAUSS, P. S., AND CAREY, R. 1992. An object-oriented 3D graphics toolkit. In *Computer Graphics (Proceedings of ACM SIGGRAPH 92)*, ACM Press, New York, 341–349.

WONKA, P., WIMMER, M., SILLION, F., AND RIBARSKY, W. 2003. Instant architecture. *ACM Transactions on Graphics 22*, 3, 669–677.