A Graph-Based Language for Direct Manipulation of Procedural Models

Wolfgang Thaller*, Ulrich Krispel†, René Zmugg*, Sven Havemann*, and Dieter W. Fellner*†‡

*Institute of Computer Graphics and Knowledge Visualization, Graz University of Technology, Graz, Austria
{w.thaller, r.zmugg, s.havemann}@cgv.tugraz.at

†Interactive Graphics Systems Group (GRIS), TU Darmstadt, Darmstadt, Germany
u.krispel@gris.tu-darmstadt.de

‡Fraunhofer IGD and TU Darmstadt, Darmstadt, Germany
d.fellner@igd.fraunhofer.de

***Abstract*—Creating 3D content requires a lot of expert knowledge and is often a very time consuming task. Procedural modeling can simplify this process for several application domains. However, creating procedural descriptions is still a complicated task. Graph based visual programming languages can ease the creation workflow; however, direct manipulation of procedural 3D content rather than of a visual program is desirable as it resembles established techniques in 3D modeling. In this paper, we present a dataflow language that features novel contributions towards direct interactive manipulation of procedural 3D models: We eliminate the need to manually program loops (via implicit handling of nested repetitions), we introduce partial reevaluation strategies for efficient execution, and we show the integration of stateful external libraries (scene graphs) into the dataflow model of the proposed language.**

Keywords-procedural modeling, dataflow graphs, loops, term graphs

I. Introduction

This is a revised and augmented version of "Implicit Nested Repetition in Dataflow for Procedural Modeling", which appeared in the Proceedings of The Third International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking (COMPUTATION TOOLS 2012) [1].

Conventional 3D models consist of geometric information only, whereas a procedural model is represented by the operations used to create the geometry [2]. Complex man-made shapes exhibit great regularities for a number of reasons, from functionality over manufacturability to aesthetics and style. A procedural representation is therefore commonly perceived as most appropriate, but not so many 3D artists accept a code editor as user interface for 3D modeling, and only few of them are good programmers. Recently, dataflow graph based visual programming languages for 3D modeling have emerged [3], [4]. These languages facilitate a graphical editing paradigm, thus allowing to create programs without writing code. However, such languages are not always easier to read than a textual representation [5]. Therefore, the goal is a modeler that allows direct manipulation of procedural content on the concrete 3D model, without any knowledge of the underlying representation (code), while retaining the expressiveness of dataflow graph based methods.

In this paper, we present a term graph based language for procedural modeling with features that facilitate direct manipulation. First, we give an overview of related work in Section II. Then, we give a summary of the requirements for the language in Section III. Next, in Section IV the language is formally defined, and a compilation technique to embed such models in existing procedural modeling systems is described. Section V describes how the language can be applied to different modeling operations. Going beyond our previous work in [1], Section VI describes a method for incrementally reevaluating a procedural model expressed in our language in response to user interaction. We conclude with a discussion and some points of future research.

II. Related Work

Procedural modeling is an umbrella term for procedural descriptions in computer graphics. As a procedural description is basically just a computer program, there are many possibilities to express procedural content.

One category are general purpose programming languages with geometric libraries, for example C++ with *CGAL* [6] or the Generative Modeling Language (*GML*) [2] which utilizes a language similar to Adobe's PostScript [7]. *Processing* [8] is an open source programming language based on Java with a focus on computer programming within a visual context.

As many professional 3D modeling packages contain embedded scripting languages, these can be used to express procedural content. Some representatives are for example MEL script for Autodesk Maya [9] or RhinoScript for Rhinoceros [10].

Some domain specific languages have successfully been applied to express procedural content. For example, based on the work of Stiny et al. [11] who applied the concept of formal grammars (string replacements) to the domain of 2D shapes, Wonka et al. [12] introduced *split grammars* for automatic generation of architecture. These concepts have further been

extended by Mueller et al. [13] into CGA Shape, which is available as the commercial software package CityEngine [14] that allows procedural generation of buildings up to whole cities.

Visual Programming Languages (VPLs) allow to create and edit programs using a visual editing metaphor. Many VPLs are based on a dataflow paradigm [15]; the program is represented by a graph consisting of *nodes* (which represent operations) and *wires* along which streams of *tokens* are passed. Some examples in the context of procedural modeling are the procedural modeler Houdini [4] and the Grasshopper plugin for Rhinoceros [10], which both feature visual editors for dataflow graphs. Furthermore, the work of Patow et al. [16] has shown that shape grammars can also be represented as dataflow graphs. Such a representation also allows established interaction metaphors to be accessible for procedural modeling packages [17].

Term Graphs [18] arose as a development in the field of term rewriting. While term graphs are intuitively similar to dataflow graphs, there is no concept of a stream of tokens. Term graphs are a generalization of terms and expressions which makes explicit sharing of common subexpressions possible. Formally, we base our work on the definitions given in [19] rather than on any dataflow formalism.

III. Language Requirements

Dataflow languages have a number of properties that make them very desirable for interactive procedural modeling. They allow efficient partial reevaluation in order to interactively respond to "localized" changes, they are expressive enough to cover traditional domains of procedural modeling such as compass-and-ruler constructions and split-grammars, and they can be extended in various ways to support repeated structures/repeated operations.

We are currently researching direct manipulation based user interfaces for dataflow-based procedural modeling. This means that the dataflow graph itself is not visible to the user; instead, the user interacts with a concrete instance of the procedural model, i.e., a 3D model generated from a concrete set of parameter values. The basic usage paradigm is that the user selects objects in this 3D view and applies operations to them; these operations are added to the graph.

The goal of keeping the graph hidden during normal user interaction leads to additional requirements for the language that differ from traditional approaches.

A. Repetition

Loops should not be represented explicitly, i.e., loops should not be represented by an object that needs to be visualized so the user can interact with it directly. Operations should be implicitly repeated when they are applied to collections of objects.

It must be possible to deal with **nested repetitions** as part of this implicit repetition behaviour. Existing dataflow-based procedural modeling systems use a "stream-of-tokens" concept, i.e., a wire in the dataflow graph transports a linear stream of tokens that all get treated the same by subsequent operations. Nested structures are not preserved in this model.

When directly interacting with a 3D model, we expect the user to frequently zoom to details of the model. For example, consider a model of a building façade that consists of several stories, each of which contains several identical windows, which in turn contain several separate window panes. A user will zoom in to see a single window on their screen and then proceed to edit that archetypal window, for example by applying some operation to two neighbouring window panes of that same window. All operations in the modeling user interface should always behave consistently, independent of whether the user is editing a model consisting of just a single window, or one of many windows. In both cases, the system needs to remember that a collection of window panes belongs to a single window. Thus, flat token streams are not suited to direct-manipulation procedural modeling.

B. Failures

There are many modeling operations that do not always succeed, e.g., intersection operations between geometric objects. When applying volumetric split operations, a volume might become empty, rendering (almost) all further operations on that volume meaningless.

Often, these failures have only local effects on the model, so aborting the evaluation of the entire model is excessive; rather, we propagate errors only along the dependencies in the code graph — if its sources could not be calculated, an edge is not executed. In many cases, this is exactly the desired behaviour and allows to easily express simple conditional behaviours such as "if there is an intersection, construct this object at the intersection point" or "if there is enough space available, construct an object".

C. Side Effects

Neither dataflow graphs nor term graphs are particularly well-suited for dealing with side-effecting operations; also, to simplify analysing the code for purposes of the GUI, we have a strong motivation to forbid side effects.

However, it is a fundamental user expectation to be able to have operations that *create* objects, and to be able to *replace* or refine objects. Both Grasshopper and Houdini use side-effect free operations and rely on the user to pick one or more dataflow graph nodes whose results are to be used for the final model; this solution is not applicable to a direct manipulation procedural modeler because it would require interacting with the graph rather than with a 3D model.

IV. The Language

Below, we will first define the term graphs that form the basis of our language; we will then proceed to discuss our treatment of side effects, repetition and failing operations.

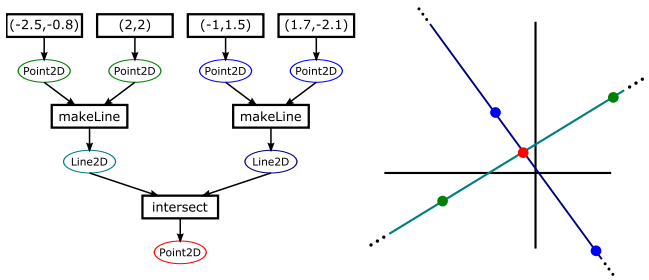


Fig. 1. A **code graph** (as presented by [19]) is a hypergraph that consists of nodes that correspond to results and hyperedges that represent operations (left). In this illustration the nodes are represented as ellipses. Hyperedges are visualized as boxes; they can have any number of source and target nodes. Hyperedges with no source nodes correspond to constants. This example shows a code graph that carries out a simple construction: Two points define a straight line; two lines yield an intersection point (right).

A. Code Graphs

The underlying data structure is a *hypergraph* consisting of nodes, which correspond to (intermediate) values and graphical objects, and *hyperedges*, which represent the operations applied to those values as shown in Figure 1.

Note that we are following term graph terminology here, which differs from the terminology traditionally used for dataflow graphs. In a dataflow graph, *nodes* are labelled with operations, and they are connected with edges or *wires*, which transport values or tokens. In a term graph, *hyperedges* (i.e., edges that may connect more or fewer than two nodes) are labelled with operations or literal constants, and values are stored in nodes, which are labelled with a type.

We reuse the following definition from [19]:

Definition 1: A *code graph* over an edge label set ELab and a set of types NType is defined as a tuple $G = (\mathcal{N}, \mathcal{E}, \mathsf{In}, \mathsf{Out}, \mathsf{src}, \mathsf{trg}, \mathsf{nType}, \mathsf{eLab})$ that consists of:

- a set $\mathcal{N}$ of *nodes* and a set $\mathcal{E}$ of *hyperedges* (or *edges*),
- two node sequences $\mathsf{In}, \mathsf{Out} : \mathcal{N}^*$ containing the *input nodes* and *output nodes* of the code graph,
- two functions $\mathsf{src}, \mathsf{trg} : \mathcal{E} \to \mathcal{N}^*$ assigning each edge the sequence of its *source nodes* and *target nodes* respectively,
- a function $\mathsf{nType} : \mathcal{N} \to \mathsf{NType}$ assigning each node its *type*, and
- a function $\mathsf{eLab} : \mathcal{E} \to \mathsf{ELab}$ assigning each edge its *edge label*. □

Furthermore, we require all code graphs in our system to be acyclic and that every node occurs exactly once in either the input list of the graph, or in exactly one target list of an edge.

Definition 2: Edge labels are associated with an input type sequence and an output type sequence by the functions $\mathsf{edgeInType}$ and $\mathsf{edgeOutType} : \mathsf{ELab} \to \mathsf{NType}^*$. □

Definition 3: An edge e is considered *type-correct* if $\mathsf{edgeInType}(\mathsf{eLab}(e))$ matches the type of the edge's source nodes, and $\mathsf{edgeOutType}(\mathsf{eLab}(e))$ matches the type of its target nodes. A codegraph is type-correct if all edges are type-correct. □

B. Limited Side Effects

In Section III-C, we have noted the need to be able to model *creation* and *replacement* operations. The *scene* is the set of visible objects; we define it as a global mutable set of object references. We only allow two kinds of side-effecting operations: (a) adding a newly-created object to the scene, thus making it visible; and (b) removing a given object reference from the scene.

Replacement and refinement can be modeled by removing an existing object and adding a new one. Object removal is idempotent and only affects object visibility, not the actual object. Object visibility cannot be observed by operations. Therefore, no additional constraints on the order of execution are introduced.

C. Implicit Repetition

When an operation is applied to a list rather than a single value, it is implicitly repeated for all values in the list; if two or more lists are given, the operation is automatically applied to corresponding elements of the lists (cf. Figure 2). It is assumed that the lists have been arranged properly.

We define our method of implicitly handling repetition by defining a translation from codegraphs with implicitly-repeated operations to codegraphs with explicit loops.

1) Explicit Loops:

Definition 4: A *codegraph with explicit loops* is a codegraph where the set of possible edge labels ELab has been extended to include *loop-boxes*. A loop-box edge label is a tuple $(\texttt{LOOP}, G', f)$ where G' is a code graph (the loop body) with n inputs and $f \in \{0,1\}^n$ is a sequence of boolean flags, such that at least one element of f is 1. The intention behind the flags f is to indicate which inputs are lists that are iterated over ($f_i = 1$), and which inputs are non-varying values that are used by the loop ($f_i = 0$). The number of iterations corresponds to the length of the shortest input list. The edge input and output types of a loop are defined by wrapping the input and output types of the loop body (referred to as ti_i and to_i below) with $\texttt{List}[\cdots]$ as appropriate:

$$\mathsf{edgeOutType}((\texttt{LOOP}, G', f))_i := \texttt{List}[to_i]$$

$$\mathsf{edgeInType}\left((\texttt{LOOP}, G', f)\right)_i := \begin{cases} \texttt{List}[ti_i] \text{ if } f_i = 1 \\ ti_i \text{ otherwise} \end{cases}$$ □

2) Codegraphs with Implicit Repetition: To allow implicit repetition, we relax the type-correctness requirement that edge input/output types match the corresponding node types.

A codegraph with implicit repetition is translated to a codegraph with explicit loops by repeatedly applying the following translation; the original codegraph is considered type-correct iff this algorithm yields a codegraph with explicit loops that fulfills the type-correctness requirement.

Consider an edge e where the type-correctness condition is violated. If any of the output nodes is not a list, or if any of the mis-matching input nodes is not a list, abort; in this case, the input codegraph is considered to be invalid. Replace the edge e by a loop edge e'. The repetition flags f_i for the new loop edge are set to 1 for every input with a type mismatch, and

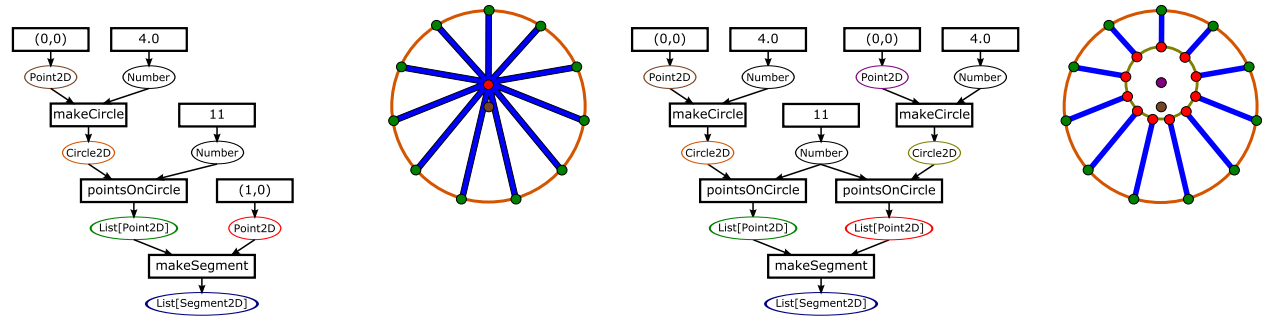


Fig. 2. Handling repetitions: The images show examples of simple procedural models ((b) and (d)) that create a list of line segments (blue) and their respective code graphs ((a) and (c)). Points, lines and circles correspond to intermediate results (nodes) of the same color. makeCircle creates a circle out of a point and a radius, pointsOnCircle creates a list of evenly distributed points on a circle and makeSegment creates a straight line segment between two points. This operation can be implicitly repeated to create segments from a list of points (on a circle) to a single point ((b)), or between two lists of points on circles ((d)) using makeSegment. Multiple graphical elements are represented by single nodes in the corresponding code graphs ((a) and (c)).

to 0 otherwise. The loop body G' is a codegraph containing just the edge e; the types of its input and output nodes are chosen such that the edge e' becomes type-correct within the outer codegraph. The translation is then applied to the loop body G'.

3) Fusing Loops: The result of the above translation is a codegraph that contains separate (and possibly nested) loops for each edge. This is undesirable for two reasons, namely performance and code readability. Performance is relevant whenever the operations used in the codegraph edges are relatively cheap, such as, for example, compass and ruler constructions, as opposed to boolean operations on 3D volumes (constructive solid geometry, CSG). Code readability is important because a procedural model might still need to be modified after it has been exported from our system to a traditional script-based system.

Consecutive loops, i.e., loops where the second loop iterates over an output of the first, can be fused if both loops have the same number of iterations and if the second loop does not, either directly nor indirectly, depend on values from other iterations of the first loop.

To determine which loops have the same number of iterations, we will annotate each occurrence of `List` in each node type with a symbolic item count, represented by a set of variable names. Each variable is an arbitrary name for an integer that is unknown at compile time. A set denotes the minimum of all the contained variables. $\texttt{List}_{\{a\}}[t]$ means a list of a items of type t, and $\texttt{List}_{\{a,b\}}[t]$ means a list of $\min(a, b)$ items.

All `List` types that appear as outputs of non-loop edges are annotated with a single unique variable name each. Every loop edge is annotated with a symbolic iteration count that is the minimum (represented by set union) of the symbolic item counts of all the lists it iterates over. Annotations on nested `List` types are propagated into and out of the loop bodies. The resulting `List` types of a loop box are annotated with a symbolic item count that is equal to the symbolic iteration count of the loop.

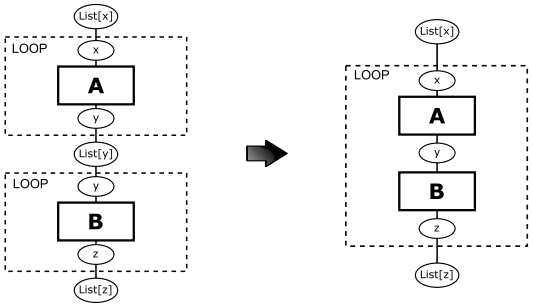


Fig. 3. Two consecutive loops containing one operation each that gets applied to every item of the list. Under certain conditions (see text) the loops can be fused in order to simplify the graph.

Two consecutive loop edges e_1 and e_2 can be *fused* when the symbolic iteration counts of the loops are equal, the repetition flag f_i is set to 1 for all inputs of e_2 that are outputs of e_1, and e_2 is not reachable from any edge that is reachable from e_1, other than e_1 and e_2 themselves.

If all these conditions are fulfilled for a given pair of edges, the edges can then be replaced by a single edge (cf. Figure 3); the fused loop body is the sequential concatenation of the two individual loop bodies. The inputs for the fused edge are the inputs of e_1 and all nodes that are inputs of e_2 but not outputs of e_1. The flags f_i for the fused edge are equal to the corresponding flags for inputs of e_1 and e_2. The outputs for the fused edge are all nodes that are either outputs of e_1 or of e_2. This fusing operation is applied until no more edges can be fused.

D. Handling Errors

The desired error-handling behaviour can be described by regarding `ERROR` as a special value which is propagated through the codegraph. If an operation fails, all its outputs are set to `ERROR`; an operation is also considered to fail whenever any of its inputs are `ERROR`.

In a naive translation, all arguments need to be explicitly checked for every single operation. To arrive at a better

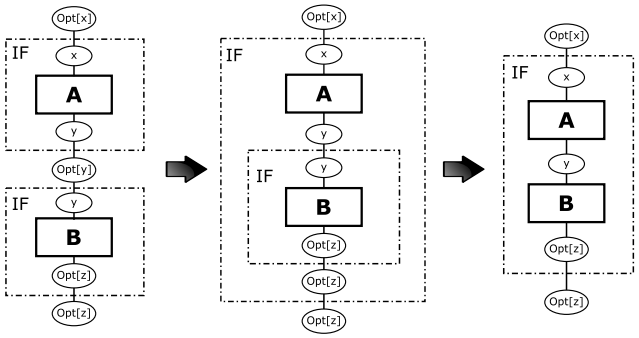
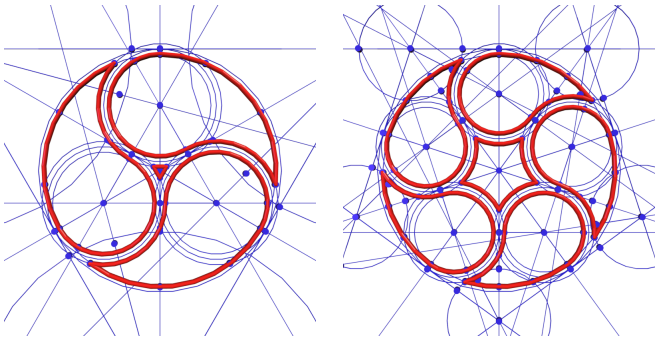


Fig. 4. Left: two consecutive if-boxes used for handling potentially-failing operations. The input (`Opt`$[x]$ at the top) is already the result of a potentially-failing operation. Note that in this example, operation **A** itself cannot fail (result type is plain y), while operation **B** can (result type is `Opt`$[z]$). They can be combined by nesting the second box inside the first (center). This often exposes opportunities for eliminating redundant error checks (right).



(a) (b)

Fig. 5. This gothic window construction was created in our test framework using direct manipulation without any code or graph editing. The number of repetitions is an input parameter of the model.

translation, we use a similar method as for the loops above; we first make the error checking explicit and then introduce a rule for combining consecutive error-checks.

Definition 5: `Opt`$[t] := t \cup \{$`ERROR`$\}$ for all types t, i.e., `Opt`$[t]$ is a type that can take any value that type t can, or a special error token. `Opt`$[t]$ is idempotent: `Opt`$[$`Opt`$[t]] =$ `Opt`$[t]$. Also note that `Opt` can nest with `List` — the types `Opt`$[$`List`$[t]]$ and `List`$[$`Opt`$[t]]$ and `Opt`$[$`List`$[$`Opt`$[t]]]$ are three different types. □

Definition 6: An *if-box* edge label is a tuple $(\texttt{IF}, G', f)$ where G' is a codegraph with n inputs and $f \in \{0,1\}^n$ is a sequence of boolean flags, such that at least one element of f is 1. The edge input and output types of a loop are defined by wrapping the input and output types of the loop body with `Opt`$[\cdots]$ as appropriate, analogously to the treatment of loop boxes (cf. Definition 4). When an if-box is executed, all input values for which $f_i = 1$ are first checked for `ERROR`s; if any of the input values is equal to `ERROR`, execution of the box immediately finishes with a result value of `ERROR` for each output. If none of the inputs are `ERROR`, the body G' is executed; its output values are the output values of the if-box. □

Predefined operations that can fail will return optional values (`Opt`$[\cdots]$). For every edge in the code graph, if-boxes have to be inserted if necessary to make the codegraph type-consistent.

Two consecutive if-box edges e_1 and e_2 can be *fused* when the flag f_i is set to 1 for at least one input e_2 that is an output of e_1, and e_2 is not reachable from any edge that is reachable from e_1, other than e_1 and e_2 themselves.

Fusing of if-boxes happens by moving the edge e_2 into the body of the if-box e_1, yielding two nested if-boxes (cf. Figure 4). The inputs for the fused edge are the inputs of e_1 and additionally all nodes that are inputs of e_2 but not outputs of e_1; the flags f_i for the additional flags are all set to 0, which means that the outer box does not need to check these inputs against `ERROR`, because the inner box will do so if necessary. For the nested if-box inside the fused edge, we next check whether that box is still required; first, for every input whose node type is not of the form `Opt`$[t]$, the corresponding flag f_i is set to 0. If all flags are set to zero for the inner if-box, the box is eliminated by replacing the edge with its body codegraph.

V. Modeling Vocabularies

In this section, we describe some common modeling operations and their realization within our framework. The examples in this section have been created using direct manipulation on a visible model only (without visualization of the underlying code graph), the concrete user interface is, however, outside the scope of this paper. Refer to [20] and [21] for accounts of different applications of our system.

A. Compass & Ruler

Compass and ruler operations have long been used in interactive procedural modeling [22]; these operations are well suited to a side-effect free implementation, and usually return only a single result per operation. Our addition of repetition allows for new constructions (Figure 5).

B. Split Grammars

We can use a methodology similar to Patow et al. [16] to map split grammars to code graphs (see Figure 6). A model is described by a set of replacement rules. Successive application of rules gradually refines the result (coarse to fine description). Just as in CGA Shape [13] and the work of Thaller et al. [23], a shape consists of a bounding volume called *scope*, a individual local coordinate system, and geometry within the scope. These shapes are partitioned into smaller volumes by operations split and repeat (replacement as side-effect). The split operation partitions the scope in a predefined number of parts, whereas with the repeat operation the number of parts is determined by the size of the scope at the time of rule application.

A complex example of a façade that was realized through our system is shown in Figure 7.

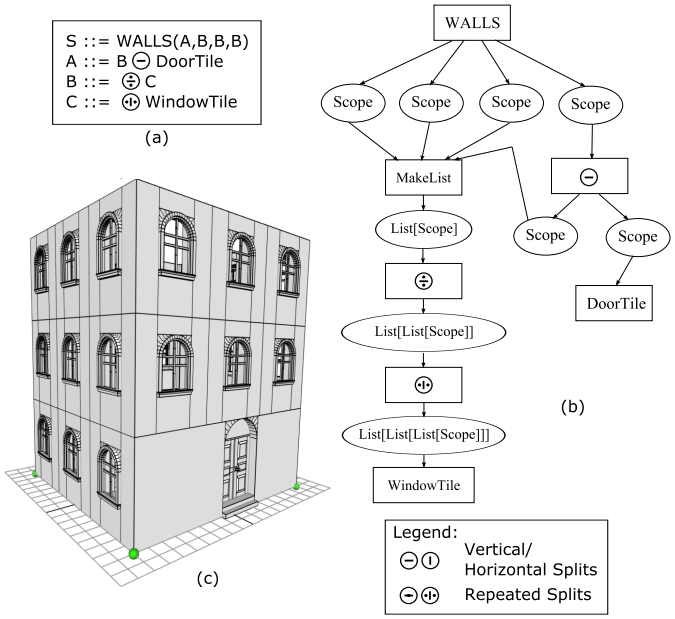
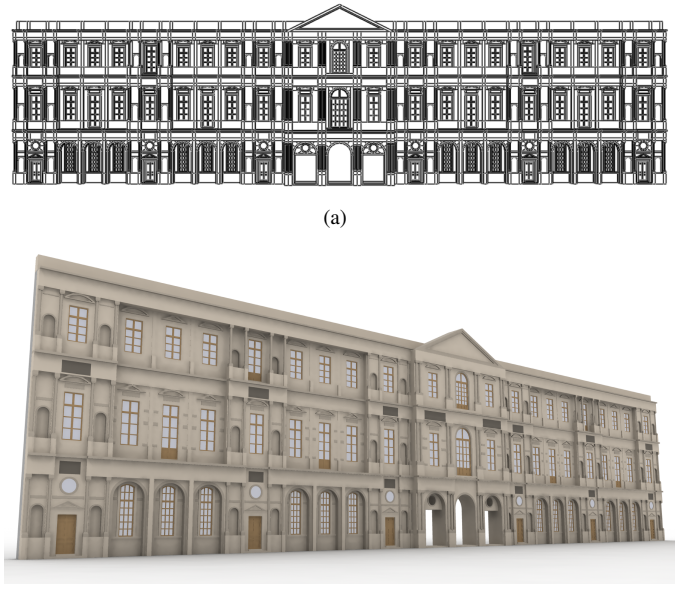


Fig. 6. Split grammar example: A simple shape grammar with split and repeat operations can be expressed using a textual description (a). This structure can be mapped to a codegraph (b) and executed, which yields (c).



(a)

(b)

Fig. 7. A complex façade example realized with our system. These images stem from parts of the Louvre that were reconstructed in the work of Zmugg et al. [20]. Figure (a) shows the hierarchical split layout of the façade, which led to the rendering (b).

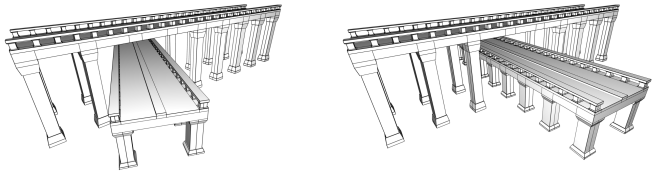


Fig. 8. Illustration of interconnected structures: The pillars of the bridges are constructed using ray casting for obstacle detection; The pillars of the larger bridge are constructed with respect to the position of the lower bridge.

C. Interconnected Structures

A drawback of (context-free) shape grammar systems is that they lack mechanisms for connecting structures across different parts of the top-down modeling hierarchy. The solution proposed in [24] is to extend a text-based shape grammar system by a feature called "containers". The idea is to pass *mutable* containers, currently implemented via nested arrays, as parameters to shape grammar rules. During the evaluation of the rules, objects which are potential *attaching points* can be appended to these arrays as a side effect of the grammar evaluation. These arrays can later be used to define structures that connect elements from independent parts of the model hierarchy. These connecting elements can follow different connection patterns based on geometric queries, such as ray casting (see Figure 8).

We can directly translate the idea of containers in [24] to our system, with the only difference being that attaching points have to be explicitly grouped in arrays. This does not cause additional complexity; receiving a container as an input and explicitly adding objects should take about the same "effort" as explicitly grouping objects and returning a nested list as an output. In the context of direct manipulation of procedural models, however, our approach has two advantages, both of which stem from the absence of side effects in our system:

- A list in our system has a concrete visual representation — the user can click on it; by contrast, a *mutable* container has different states throughout its lifetime, and it is created as an empty container before objects are added. As such it is a "virtual" object for which no concrete 3D representation seems possible.
- Passing a mutable container enforces a linear execution order; different operations that access the same container must always be executed in the same order, or the meaning will change, preventing efficient partial reevaluation of the scene. This is not a problem in the context of [24], which focuses on offline generation of geometry.

D. Scene Graphs

Many three-dimensional scenes have a hierarchical structure where individual objects are placed relative to a parent object, e.g., pieces of furniture are placed relative to the room that they are located in, but the objects on a table are placed relative to the table. The structure describing such relations is referred to as a *scene graph*. When objects that occur more

Fig. 9. Scene graphs allow the representation of hierarchical dependencies; As the TV is placed in dependence of the table, changing the table's position will also move the TV accordingly. Furthermore, scene graphs are memory efficient through instancing: the two chairs in this scene refer to the same geometry with different transformations.

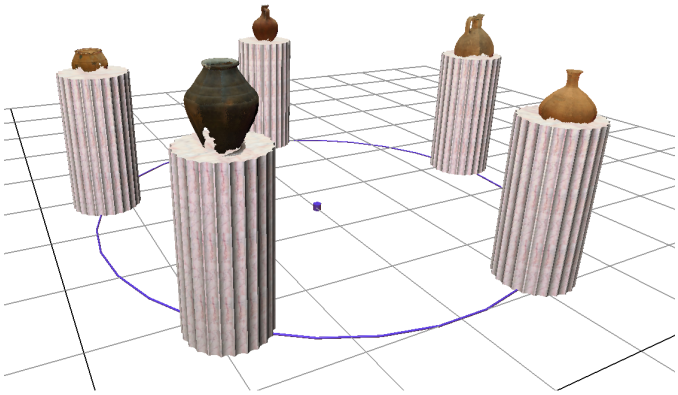


Fig. 10. A procedural scene graph: Scene graph nodes with pedestals are placed at points distributed on a circle. On top of each pedestal, a museum exhibit is placed. The input for the code graph that represents this procedural model is a list of file paths to load the exhibits from.

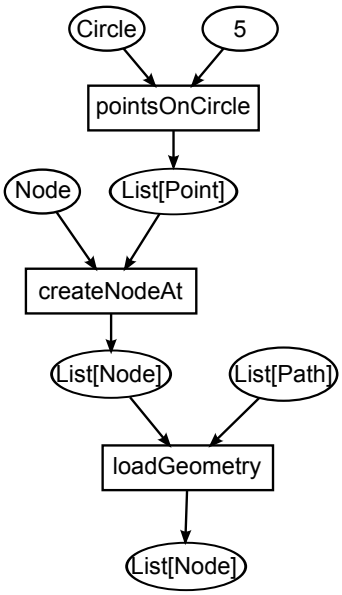


Fig. 11. The code graph representing Figure 10. For simplicity, the operations representing the pedestals have been left out.

than once in a scene are taken into account, the scene graph is an acyclic graph instead of a tree. Each node in a scene graph contains a transformation and, optionally, geometry. The transformation that is applied to each piece of geometry — defining its placement in the scene — is the product of all transformations on the path from the root to the node (see Figure 9). By gradually changing the transformations of nodes, animated objects can be achieved easily.

To embed a scene graph in our system, we first need a type `Node` to represent scene graph nodes; we will assume that one instance of that type, the *root* of the scene graph, will get passed to the code graph as an input. Child nodes are created using operations that take the parent as an input; in particular, there is an operation createNode that creates a transformation node (without any visible geometry) and an operation loadGeometry that creates a node with pre-generated geometry loaded from a file. Finally, the toGlobal converts a point from the local coordinate system of a scene graph node to global coordinates; this operation allows creating structures that connect objects that reside in different parts of the scene graph.

Building on top of the createNode and toGlobal operations, we can also provide a createNodeAt operation that places a child node at a point given in global coordinates (instead of the usual parent-relative transformation).

The first question that has to be asked is whether this vocabulary fulfills the requirements of our code graph formalism, in particular the limitations on side effects. There are no object removal or replacement operations; both createNode and loadGeometry are intended as object creation functions, but they actually modify the parent node's set of children rather than a global set of visible objects. This is, however, equivalent to having one global set of objects, where each object can contain a reference to its parent object. Storing individual sets of child nodes at each node rather than a single global set can thus be seen as a performance optimisation that is transparent from the semantics point of view.

For real-world applications, the range of node-creating operations needs to be extended, but the basic structure will remain the same. This is a straightforward mapping of a scene graph to the code graph, which is important because it allows the user interface to present standard scene graph semantics to the user. The work of Zmugg et al. [21] describes this from an application's point of view.

Figure 10 shows a variant of a use case described in [21]; the corresponding code graph can be seen in Figure 11. The inputs for this code graph are the number of museum exhibits to be shown and a list of paths to files containing the 3D models of the exhibits. The requested number of models is loaded from the list and placed in scene graph nodes arranged in a circle.

As the transformations could be represented explicitly as values in the code graph, a code graph based system is necessarily at least as powerful as a scene graph system.

However, there are two reasons why it is desirable to use existing scene graph systems (such as OpenSG [25]) as a modeling vocabulary for a code graph based system:

- Scene graphs, with their hierarchical way of managing object placement, provide a useful abstraction; dealing with transformations as values in a code graph can be hard to understand for the end user. A scene graph node, on the other hand, ties the transformation to a concrete object and can thus be represented in a more intuitive way in a graphical user interface.
- Scene graph systems are available as ready-to-use libraries and already solve many problems related to efficient rendering and animation. It is therefore desirable to be able to use them from the procedural modeling system, rather than having to re-implement their functionality.

VI. Incremental Updates

During interactive manipulation of a procedural object, it is usually a single parameter that is being modified, for example using a mouse dragging operation. This parameter often only affects a small part of the model, so, for reasons of performance, it is not desirable to reevaluate the entire model. Instead, we want to perform the minimum amount of work required, i.e., to only reevaluate those individual operations that really depend on the changed input.

The straightforward method is thus to reevaluate all hyperedges that are below the changed value (i.e., that consume it directly or indirectly). Reevaluating an operation entails first undoing all side effects caused by that operation, before re-executing the operations with updated parameters.

In the course of this section, we will first show why this approach is not sufficient and then proceed to describe a method that takes the discussed problems into account.

A. The Problem of Aggregate Values

Excessive recalculation can happen whenever an input of a code graph edge is of an aggregate type, i.e., any type that consists of several parts such that some of these parts might stay unchanged. Lists are an obvious example of an aggregate data type in our language; if a change in an input parameter causes a change in one element of an intermediary result of list type, we do not want to undo and recalculate all operations that use the unchanged elements.

Individual modeling vocabularies can add further aggregate data types which can also cause excessive recalculation; changing the color of an object might, for example, only affect the colors of objects that depend on it, but not their shape. Recalculating the geometry of those objects might be a lot more expensive than just updating their color.

In particular, this problem affects our use of scene graphs as described in Section V-D. One of the main strengths of scene graphs is that a scene graph can be efficiently animated by changing the transformation on a node; this affects all children of the node without requiring that subgraph to be changed. The code graph representation of a scene graph, however, encodes dependencies that do not actually exist. Each scene graph node depends on its transformation, and all children of a scene graph node depend on the parent node.

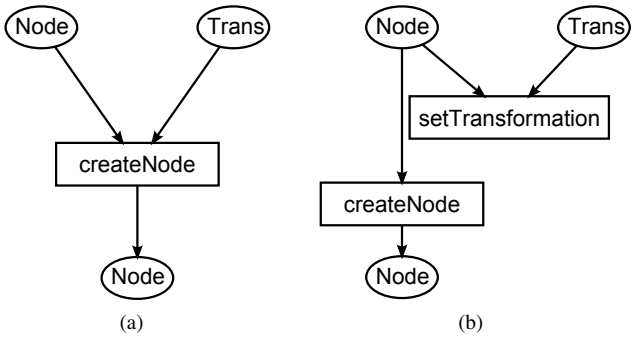


Fig. 12. With a simple representation of scene graphs, child nodes will depend on the transformation (a); this can be avoided by having separate createNode and setTransformation operations (b). This will avoid a reevaluation of createNode after changing the transformation.

Thus, when a naive method is used to evaluate the code-graph, all children of a node are rebuilt from scratch when the transformation of the parent is changed.

The "museum" example from Figures 10 and 11 constitutes a further example. When the number of museum exhibits to be displayed is changed, already-loaded objects should never be re-loaded. Ideally, only new objects should be loaded, while all objects that were previously visible should be re-used.

B. A Possible Alternate Interface to Scene Graphs

A possible way to solve this problem is to make the code graph encode the dependencies more accurately. In particular, this means using several separate operations to achieve the work of createNode and related operations. In particular, node creation needs to be separated from setting a node's transformation (Figure 12). A node's children will thus depend on the parent node's existence, but not on its transformation.

This hypothetical setTransformation instruction introduces some problems. It is obviously a side-effecting operation, and it does not seem to fit any of the allowed side effects described in Section IV-B. It can, however, be interpreted as an object creation function that creates an invisible "transformation object" that contains the transformation and a reference to the scene graph node to be transformed. After code graph evaluation is complete, the scene can be scanned for these transformation objects and the transformations applied to the scene graph nodes.

The toGlobal operation can not be supported directly by this approach, as it would need to observe the set of transformation objects that are part of the scene, which is not allowed according to Section IV-B. Instead, the actual transformation will need to be passed to it using an explicit connection in the code graph.

This approach thus fulfills all the formal criteria, but it has serious disadvantages for the user interface. Direct manipulation of the setTransformation operation by itself is next to impossible, as it does not have any result that can be

visualized in a 3D GUI. The GUI layer will probably have to present a simplified view of the situation to the user, where the createNode and setTransformation operations are folded back together.

Likewise, note that the loadGeometry operation used in the example in Figure 11 is responsible for both loading a model from a file and placing it in the scene. Separating a loading operation from a placement operation would introduce additional complexity at the user interface level, as the user interface would need to have a visual representation of "loaded but not displayed" models. It is thus highly desirable to have a single operation for loading and placement.

It is therefore preferable by far to define a method to efficiently handle incremental updates on codegraphs that use the more straightforward representation of scene graphs described earlier.

C. Incremental Update for Individual Operations

To define the semantics of incremental updates, we will follow a bottom-up approach. We will start by defining updates for individual hyperedges in a code graph, i.e., for individual operations, before building up to entire code graphs.

Modeling operations are defined outside the code graph. When basic term graph evaluation is used to evaluate code graphs, each modeling operation can be implemented by a simple function in the underlying language of the system.

To partially reevaluate a code graph, the side effects of the affected operations have to be taken into account; in particular, old side effects have to be undone before an operation can again be executed with new parameters. Therefore, for each modeling operation op we require a modeling operation library to provide at least the two implementation functions $\mathsf{evaluate}_{op}$ and undo_{op}. The former performs the operation (potentially causing side effects) and the latter reverses those side effects.

$$\begin{aligned} \mathsf{evaluate}_{op} &: (in_1 \cdots in_n) \\ &\rightarrow (\mathtt{S}_{op}, out_1 \cdots out_m) \\ \mathsf{undo}_{op} &: \mathtt{S}_{op} \rightarrow () \end{aligned}$$

In addition to the modeling operation's output values, the $\mathsf{evaluate}_{op}$ function also returns a value that gets passed to the next invocation of undo_{op}. The type $\mathtt{S}_{op}$ of this state value depends on the operation. For operations without side effects, the undo_{op} function does nothing and $\mathtt{S}_{op}$ contains no information.

To allow reevaluations to be optimized by taking advantage of previous evaluations and to allow partial updates of aggregate values such as lists and scene graph nodes, modeling operation libraries may also provide a update_{op} function.

The input and output values of the update_{op} function will each be annotated with a tag. A tag is intended to specify if a value has changed, and if so, in what way it has changed. For many datatypes, it will be enough to track whether a value has changed or not. For scene graph nodes, we need to distinguish between changes that only affect the transformation and changes that should trigger a complete reevaluation of subsequent operations.

We therefore define a parametric datatype Tag, i.e., for each data type t, we define a datatype $\mathtt{Tag}[t]$. The individual Tag datatypes should be defined individually according to the requirements for the datatype t. We require that every Tag datatype has at least the two special values NEW and UNCHANGED.

The update_{op} function is also supplied with the outputs of the previous evaluation of the operation; its signature is thus as follows:

$$\begin{aligned} \mathsf{update}_{op} &: (\mathtt{S}_{op}, in_1 \cdots in_n, \\ &\qquad \mathtt{Tag}[in_1] \cdots \mathtt{Tag}[in_n], \\ &\qquad oldout_1, \cdots oldout_m) \\ &\rightarrow (\mathtt{S}_{op}, out_1 \cdots out_m, \\ &\qquad \mathtt{Tag}[out_1] \cdots \mathtt{Tag}[out_m]) \end{aligned}$$

If the update function is not defined, a default implementation is provided based on the evaluate and undo functions; it can be seen in Listing 1.

Listing 1 Default update function for operations

function $\mathsf{update}_{op}(s,\ arg_{1\cdots n},\ atag_{1\cdots n},\ oldout_{1\cdots m})$
  if all $atag_i$ are UNCHANGED **then**
    return $(s,\ oldout_{1\cdots m},$
      UNCHANGED $\cdots$ UNCHANGED)
  else
    $\mathsf{undo}_{op}(s)$
    $(s',\ out_1 \cdots out_m) \leftarrow \mathsf{evaluate}_{op}(arg_0 \cdots arg_n)$
    return $(s',\ out_{1\cdots m},$ NEW $\cdots$ NEW)
  end if
end function

To solve the problem of transformations causing subsequent scene graph nodes to be recreated, we need to define tags for scene graph nodes that can describe the situation that only the transformation has changed:

$$\mathtt{Tag}[\mathtt{Node}] := \{\mathtt{NEW}, \mathtt{UNCHANGED}, \mathtt{TRANSFORMED}\}$$

For scene graph nodes that are TRANSFORMED, most update operations will simply reuse their old outputs, but tag any scene node outputs as TRANSFORMED as well; the toGlobal operation, however, which actually depends on the transformation of the input, will calculate a new result and tag it as NEW.

D. Incremental Update for Entire Code Graphs

We are now ready to define $\mathsf{evaluate}_G$, undo_G and update_G on entire codegraphs.

For the purpose of this section, we assume G to be a code graph without implicit repetition or error handling.

Definition 7: For each code graph G, we define $\mathsf{evaluate}_G$ to be the evaluation function on the entire code graph. Evaluating a code graph entails calling the individual $\mathsf{evaluate}_{op}$

functions for each edge of G in an arbitrary topologically sorted order. The state value returned is a map associating each edge of G with the state value returned by the $\mathsf{evaluate}_{op}$ invocation for that edge. □

Definition 8: The undo function undo_G on a code graph G calls the individual undo_{op} for all the edges of G in an unspecified order. Each individual undo_{op} is passed the appropriate state value. □

Definition 9: The function update_G calls each update_{op} function for each edge of G in an arbitrary topologically sorted order. Each individual update_{op} is passed the appropriate state value from the input state, and the state value returned is again a map associating each edge of G with the state value returned by the update_{op} invocation for that edge. □

Updating of a code graph can be optimized under the assumption that the individual update functions will return UNCHANGED results when all their inputs are UNCHANGED. That way, a complete traversal of the code graph can often be avoided.

E. Incremental Update of Error Checks and Loops

We can handle implicit loops and error handling by first using the translation given in Sections IV-C and IV-D to translate these features to explicit loop-boxes and if-boxes. We then treat the $(\texttt{LOOP}, G', f)$ and $(\texttt{IF}, G', f)$ families of edge labels as regular operations and define appropriate implementation functions for them.

The $\texttt{Opt}[t]$ types used for error handling are not aggregate datatypes. We do not need any special tag values beyond those defined for the underlying type t, so we define $\texttt{Tag}[\texttt{Opt}[t]] := \texttt{Tag}[t]$ for all types t.

To handle update for an if-box, the state value will be either the state for the contained graph, or the value ERROR, if the graph was not evaluated because the error check failed. This is used to decide whether the contents of the if box should be evaluated, updated or undone, and whether any ERROR outputs should be marked as NEW or as UNCHANGED.

The implementation of $\mathsf{update}_{(\texttt{IF},G',f)}$ can be seen in Listing 2; the implementations of $\mathsf{evaluate}_{(\texttt{IF},G',f)}$ and $\mathsf{undo}_{(\texttt{IF},G',f)}$ trivially forward to the corresponding functions on the contained graph G' and are therefore left out for brevity.

Dealing with repetition is more complicated, as $\texttt{List}[t]$ is an aggregate type. When individual items in a list are changed, we want to reevaluate only the corresponding iterations of loops that iterate over that list. The tag types $\texttt{Tag}[\texttt{List}[t]]$ therefore need to store individual tags for the list elements.

Definition 10: We define the tag for a list type to be either NEW, UNCHANGED or a list of tags for the individual list elements, or, more formally:

$$\texttt{Tag}[\texttt{List}[t]] := \{\texttt{NEW}, \texttt{UNCHANGED}\} \cup \texttt{Tag}[t]^*$$

where $*$ denotes the Kleene closure. □

Note that this definition does not allow tracking movement of elements within an array; the added complexity of such a system does not seem worthwhile at this time. Permuting or swapping list elements in response to a parameter change will therefore require all involved list elements to be marked as changed.

Listing 2 The update function for if boxes

```
function update_(IF,G',f)(s, arg_1...n, oldout_1...m)
    if all atag_i are UNCHANGED then
        return (s, oldout_1...m,
                   UNCHANGED ··· UNCHANGED)
    end if

    if any arg_i with f_i = 1 is ERROR then
        if s is ERROR then
            for all outputs do
                (out_i, otag_i) ← (ERROR, UNCHANGED)
            end for
        else
            undo_G'(s)
            for all outputs do
                (out_i, otag_i) ← (ERROR, NEW)
            end for
        end if
        s' ← ERROR
    else
        if s is ERROR then
            (s', out_1...m) ← evaluate_G'(arg_1...n)
            otag_1...m ← NEW
        else
            (s', out_1...m, otag_1...m)
                ← update_G'(s, arg_1...n, tag_1...n)
        end if
    end if
    return (s', out_1, otag_1 ··· out_m, otag_m)
end function
```

The persistent state s for a loop box is a list of the persistent states for the individual iterations. Thus, the update function will calculate the number of iterations required and compare it with the number of iterations done the previous time. States that are still needed are updated using $\mathsf{update}_{G'}$. If fewer iterations are needed, extra states are destroyed using $\mathsf{undo}_{G'}$. If the number of iterations has increased, new states are created using $\mathsf{evaluate}_{G'}$.

The update function for loop edges, $\mathsf{update}_{(\texttt{LOOP},G',f)}$, can be seen in Listing 3. Extracting the proper arguments for specific loop iterations happens as described in Section IV-C. The definitions of $\mathsf{evaluate}_{(\texttt{LOOP},G',f)}$ and $\mathsf{undo}_{(\texttt{LOOP},G',f)}$ are straightforward and are left out for brevity.

This concludes the extensions to the language (cf. Section IV). They cover the incremental updates of individual operations up to incremental updates of whole code graphs, as well as the handling of implicit loops and error checks in this context.

VII. Discussion and Conclusion

We have presented a formal framework for the representation of procedural models that is particularly suited for direct manipulation of procedural 3D content.

Listing 3 The update function for loop boxes

```
function update_(LOOP,G',f)(s, arg_1...n, oldout_1...m)
    if all atag_i are UNCHANGED then
        return (s, oldout_1...m,
                    UNCHANGED ··· UNCHANGED)
    end if

    n_new ← min(lengths of relevant input arrays)
    n_old ← length of s
    for i ← 1, max(n_new, n_old) do
        iarg_1...n ← extract arguments for iteration i
        itag_1...n ← extract tags for iteration i
        if i > n_old then
            (s'[i], out_1...m[i]) ← evaluate_G'(iarg_1...n)
            otag_1...m[i] ← NEW
        else if i > n_new then
            undo_G'(s[i])
        else
            (s'[i], out_1...m[i], otag_1...m[i])
                ← update_G'(s[i], iarg_1...n, itag_1...n)
        end if
    end for
    return (s', out_1...m, otag_1...m)
end function
```

The design space for the dataflow language is constrained by three main considerations: simplicity, efficiency and interactivity.

Simplicity in this case means minimizing the number of entities that do not have a natural visual representation in the GUI. A three-dimensional shape can be represented directly in the GUI; a repeated three-dimensional shape can also be represented. A repetition operator, on the other hand, is an abstract concept, not a three-dimensional object. By introducing *implicit* looping and error handling constructs (Section IV), we have avoided this problem.

Next is *efficiency*. Even simple procedural models can, by virtue of their procedural nature, generate relatively large amounts of geometry data; efficiency is thus always a concern. We have benchmarked the loop fusion and error handling optimizations on three different models. The code graphs are compiled to GML [2], a language syntactically similar to PostScript. The measurement is based on the number of executable statements, or tokens; this is independent of model parameters (repetition counts) and of the implementation quality of basic operations. See Table I for the results of optimizing loops (Opt A) and loops and error handling (Opt B).

TABLE I
OPTIMIZATION BENCHMARK: EFFECTS OF FUSING LOOPS (OPT A) AND LOOPS & ERROR HANDLING (OPT B) ON MODEL SIZE.

Model	Tokens	Opt A	Opt B
gothic ornament	1322	992	789
simple house	408	258	225
complex façade	69769	30846	24865

The third and final consideration was *interactivity*. Procedural models are not always evaluated from scratch; this is especially the case in an interactive procedural editor, where the user can adjust individual parameters of a model using the mouse. For a procedural modeling system to perform well in that situation, it needs to avoid doing unnecessary recalculations. We have found that it is not enough to do this at the level of individual objects, as the granularity of these objects is dictated by the requirement of simplicity. An entity perceived as a single object by the user might in fact consist of several parts that can be updated individually. The method we have described in Section VI addresses this by allowing the implementations of the modeling operations to cooperate in providing incremental update functionality.

Taken together, these individual aspects form a system that constitutes a solid base for a direct manipulation based graphical procedural modeler that can be used with different modeling vocabularies depending on the concrete application. Since the publication of our conference paper [1], we have successfully used systems based on this framework for different applications of procedural modeling [20], [21].

A. Future Work

Interactive performance could, in theory, be improved further by taking advantage of the fact that during interactive manipulation of a procedural model, the same parameters are often changed repeatedly. Applying a form of constant folding to the tag values described in Section VI might serve to eliminate a lot of redundant checking. Parallel execution of modeling operations would be very beneficial for large and complex models, as well.

There are also many research opportunities for adapting existing programming language concepts to our framework and to the context of direct manipulation procedural modeling. Defining modules or functions is a well-known technique, but it is unknown how well they can be adapted to the special requirements imposed by direct manipulation. Complex procedural 3D models will necessarily suffer from the same problems as complex software does in general; so at some point it will be necessary to investigate methods of 'shape refactoring'.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the support from the Austrian Research Promotion Agency (FFG) for the project "V2me - Virtual Coach Reaches Out To Me" (825781), which is part of the AAL Joint Programme of the European Union.

REFERENCES

[1] W. Thaller, U. Krispel, S. Havemann, and D. Fellner, "Implicit nested repetition in dataflow for procedural modeling," in *COMPUTATION TOOLS 2012*, T. Ullrich and P. Lorenz, Eds. IARIA, 2012, pp. 45–50.

[2] S. Havemann, "Generative mesh modeling," Ph.D. dissertation, Technical University Braunschweig, 2005.

[3] Robert McNeel & Associates, "Grasshopper for Rhino3D," [retrieved: 2012, 05]. [Online]. Available: http://www.grasshopper3d.com/

[4] Side Effects Software, "Houdini," [retrieved: 2012, 05]. [Online]. Available: http://www.sidefx.com

[5] T. Green and M. Petre, "When visual programs are harder to read than textual programs," in *Proceedings of ECCE-6*, 1992, pp. 167–180.

[6] CGAL, "Computational Geometry Algorithms Library," [retrieved: 2012, 05]. [Online]. Available: http://www.cgal.org
[7] Adobe Inc., *PostScript Language Reference Manual*, 3rd ed. Addison-Wesley, 1999.
[8] Processing, "Processing," [retrieved: 2012, 05]. [Online]. Available: http://www.processing.org
[9] D. Gould, *Complete Maya programming: an extensive guide to MEL and the C++ API*, ser. Morgan Kaufmann series in computer graphics and geometric modeling. Morgan Kaufmann Publishers, 2003.
[10] Robert McNeel & Associates, "Rhinoceros 3D," [retrieved: 2012, 05]. [Online]. Available: http://www.rhino3d.com
[11] G. Stiny and J. Gips, "Shape grammars and the generative specification of painting and sculpture," in *The Best Computer Papers of 1971*. Auerbach, 1972, pp. 125–135.
[12] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky, "Instant architecture," *Proc. SIGGRAPH 2003*, pp. 669 – 677, 2003.
[13] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. V. Gool, "Procedural modeling of buildings," in *ACM SIGGRAPH*, vol. 25, 2006, pp. 614 – 623.
[14] Esri, "CityEngine," [retrieved: 2012, 05]. [Online]. Available: http://www.esri.com/software/cityengine/
[15] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Comput. Surv.*, vol. 36, no. 1, pp. 1–34, Mar. 2004.
[16] G. Patow, "User-friendly graph editing for procedural buildings," *Computer Graphics and Applications, IEEE*, vol. PP, no. 99, p. 1, 2010.
[17] S. Barroso, G. Besuievsky, and G. Patow, "Visual copy & paste for procedurally modeled buildings by ruleset rewriting," *Computers & Graphics*, vol. 37, no. 4, pp. 238–246, 2013.
[18] D. Plump, "Term graph rewriting," in *Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools*, H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, Eds. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1999, pp. 3–61.
[19] W. Kahl, C. Anand, and J. Carette, "Control-flow semantics for assembly-level data-flow graphs," in *Relational Methods in Computer Science*, ser. Lecture Notes in Computer Science, W. MacCaull, M. Winter, and I. Düntsch, Eds. Springer Berlin / Heidelberg, 2006, vol. 3929, pp. 147–160.
[20] R. Zmugg, U. Krispel, W. Thaller, S. Havemann, M. Pszeida, and D. W. Fellner, "A new approach for interactive procedural modelling in cultural heritage," in *Proc. Computer Applications & Quantitative Methods in Archaeology (CAA 2012)*, 2012.
[21] R. Zmugg, W. Thaller, M. Hecher, T. Schiffer, S. Havemann, and D. W. Fellner, "Authoring animated interactive 3D museum exhibits using a digital repository." in *VAST*, D. B. Arnold, J. Kaminski, F. Niccolucci, and A. Stork, Eds. Eurographics Association, 2012, pp. 73–80. [Online]. Available: http://dblp.uni-trier.de/db/conf/vast/vast2012.html#ZmuggTHSHF12
[22] Y. Baulac, "Un micromonde de géométrie, cabri-géométre," Ph.D. dissertation, Joseph Fourier University of Grenoble, 1990.
[23] W. Thaller, U. Krispel, R. Zmugg, S. Havemann, and D. W. Fellner, "Shape grammars on convex polyhedra," *Computers & Graphics*, vol. 37, no. 6, pp. 707 – 717, 2013, Shape Modeling International (SMI) Conference 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0097849313000861
[24] L. Krecklau and L. Kobbelt, "Procedural modeling of interconnected structures," *Computer Graphics Forum*, vol. 30, no. 2, pp. 335–344, 2011. [Online]. Available: http://dx.doi.org/10.1111/j.1467-8659.2011.01864.x
[25] D. Reiners, G. Voss, and C. Neumann, "OpenSG," 2013. [Online]. Available: http://www.opensg.org/