Implicit Nested Repetition in Dataflow for Procedural Modeling

Wolfgang Thaller, Ulrich Krispel, Sven Havemann
Institute of Computer Graphics and Knowledge Visualization
Graz University of Technology
Graz, Austria
Email: {w.thaller, u.krispel, s.havemann} @cgv.tugraz.at

Dieter W. Fellner
Fraunhofer IGD and TU Darmstadt
Darmstadt, Germany
Email: 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 a novel approach to handling loops in the context of direct interactive manipulation of procedural 3D models and show compilation techniques to translate it to traditional languages used in procedural modeling.**

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

I. Introduction

Conventional 3D models consist of geometric information only, whereas a procedural model is represented by the operations used to create the geometry [1]. 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 [2], [3]. 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 [4]. 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 2. Then we give a summary of the requirements for the language in Section 3. Furthermore, in Section 4 the language is formally defined, and a compilation technique to embed such models in existing procedural modeling systems and examine error handling in the context of partial model evaluation is described. Section 5 contains examples and some benchmarks showing optimization results. The last section concludes with 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* [5] or the Generative Modeling Language (*GML*) [1] which utilizes a language similar to Adobe's PostScript [6]. *Processing* [7] 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 [8] or RhinoScript for Rhinoceros [9].

Some domain specific languages have successfully been applied to express procedural content. For example, emerging from the work of Stiny et al. [10] who applied the concept of formal grammars (string replacements) to the domain of 2D shapes, Wonka et al. [11] introduced *split grammars* for automatic generation of architecture. These concepts have further been extended by Mueller et al. [12] into CGA Shape, which is available as the commercial software package CityEngine [13] 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 [14]; 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 [3] and the Grasshopper plugin for Rhinoceros [9], which both feature visual editors for dataflow graphs. Furthermore, the work of Patow et al. [15] has shown that shape grammars can also be represented as dataflow graphs.

Term Graphs [16] 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 [17] 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 facade 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.

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 [17]:

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*),

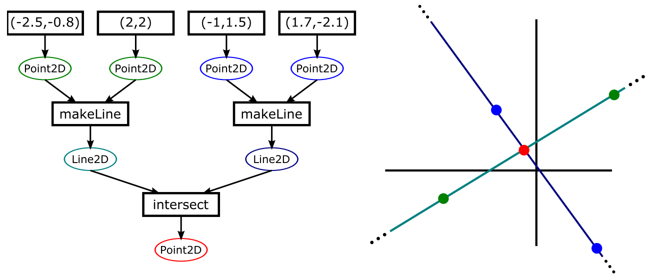

Figure 1. A **code graph** (as presented by [17]) is a hypergraph that consists of nodes that correspond to results and hyperedges that represent operations (left). In this illustration the nodes are represented as ellipses. Hyperedges are visualized as boxes; they can have any number of source and target 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).

- 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 $\mathtt{nType} : \mathcal{N} \to \mathsf{NType}$ 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 been extended to include *loop-boxes*. A loop-box edge label is a tuple $(\mathtt{LOOP}, G', f)$ 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 $\mathtt{List}[\cdots]$ as appropriate:

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

$$\mathsf{edgeInType}\,((G,f))_i := \begin{cases} \mathtt{List}[ti_i] \text{ if } f_i = 1 \\ ti_i \text{ otherwise} \end{cases} \qquad □$$

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 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

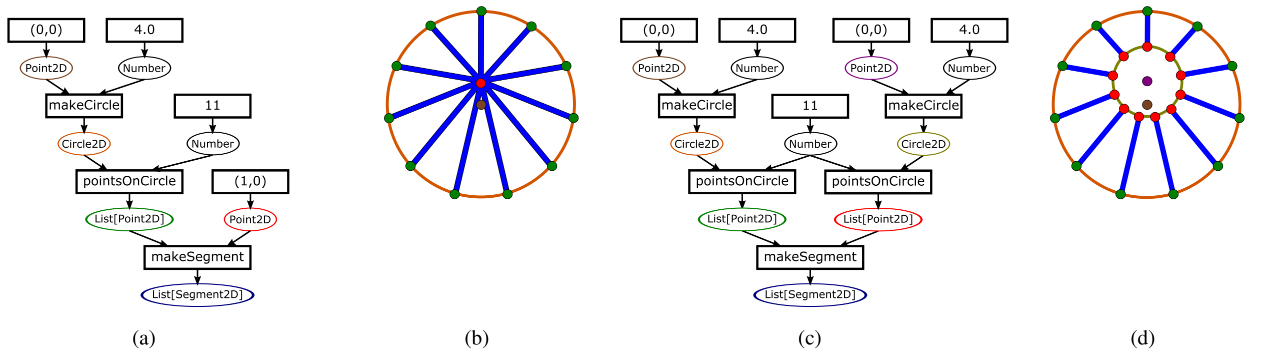

Figure 2. Handling repetitions: The images show examples of simple procedural models ((b) and (d)) that create a list of line segments (blue) and their respective code graphs ((a) and (c)). Points, lines and circles correspond to intermediate results (nodes) of the same color. makeCircle creates a circle 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)).

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 occurence 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.

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

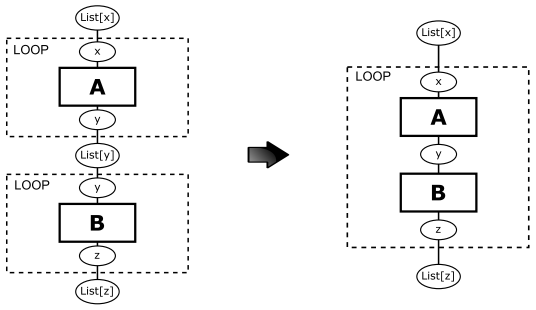

Figure 3. Two consecutive loops containing one operation each that gets applied to every item of the list. Under certain conditions (see text) the loops can be fused in order to simplify the graph.

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

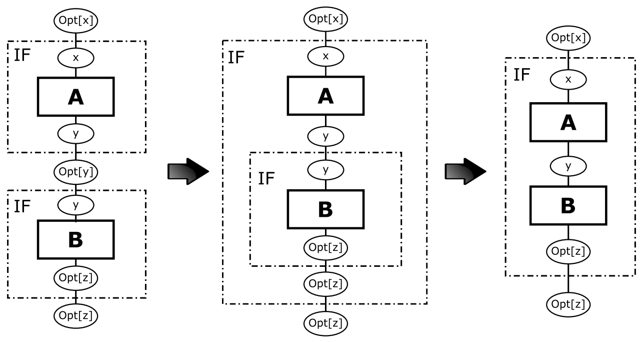

Figure 4. Left: two consecutive if-boxes used for handling potentially-failing operations. The input ($\texttt{Opt}[x]$ at the top) is already the result of a potentially-failing operation. Note that in this example, operation **A** itself cannot fail (result type is plain y), while operation **B** can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

$\texttt{Opt}[\texttt{List}[t]]$ and $\texttt{List}[\texttt{Opt}[t]]$ and $\texttt{Opt}[\texttt{List}[\texttt{Opt}[t]]]$ 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 $\texttt{Opt}[\cdots]$ 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 ERRORs; 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 ($\texttt{Opt}[\cdots]$). 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 $\texttt{Opt}[t]$, the corresponding flag f_i is set to 0. If all flags are set to zero for the inner if-box, the box is elminated by replacing the edge with its body codegraph.

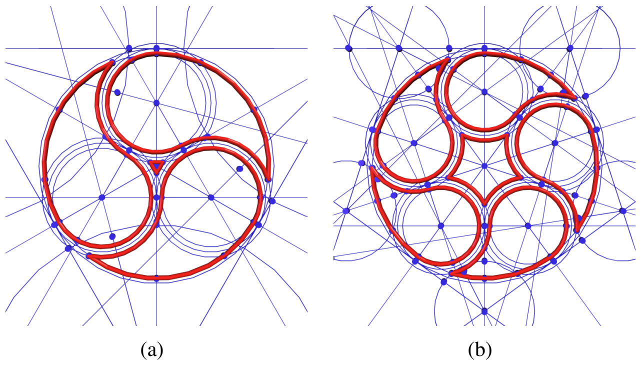

Figure 5. This gothic window construction was created in our test framework using direct manipulation without any code or graph editing. The numnber of repetitions is an input parameter of the model.

V. Examples and Results

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 still in a preliminary stage.

A. Compass & Ruler

Compass and ruler operations have long been used in interactive procedural modeling [18]; 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. [15] to map split grammars to code graphs (see Figure 6). Just as in CGA Shape [12], volumes called *Scopes* are partitioned into smaller volumes by operations split and repeat (replacement as side-effect). split partitions the scope in a predefined number of parts, whereas with repeat the number of parts is determined by the size of the scope at the time of rule application.

C. Optimization Benchmark

We benchmarked the loop fusion and error handling optimizations on three different models. The code graphs are compiled to GML, 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).

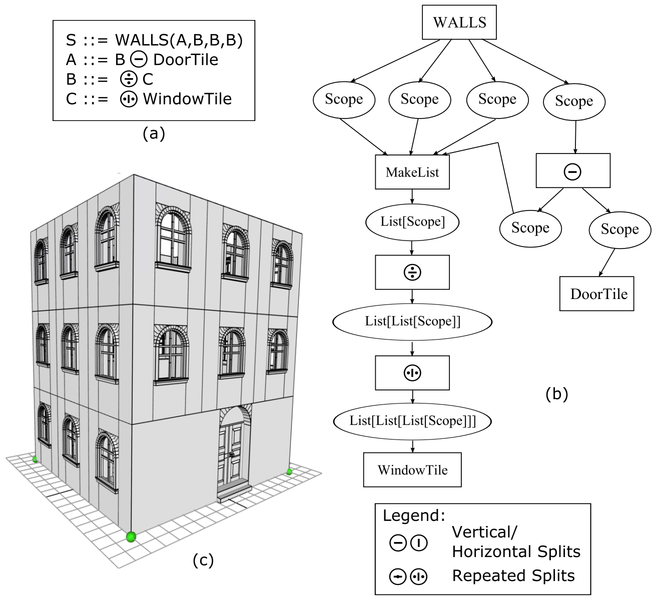

Figure 6. Split grammar example: A simple shape grammar with split and repeat operations can be expressed using a textual description (a). This structure can be mapped to a codegraph (b) and executed (c).

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

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

VI. CONCLUSION AND FUTURE WORK

We have presented a formal framework for the representation of procedural models, with a focus on implicit loop representations and improved partial error handling which is particularly suited for direct manipulation of procedural 3D content. We have further described algorithms that allow translation of these models to traditional programming-language based procedural modeling systems.

Using the framework presented in this paper, we believe it will soon be possible to create procedural constructions of medium complexity without writing code or using a visual programming language.

There are many research opportunities for adapting existing techniques 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'.

REFERENCES

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

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

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

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

[5] CGAL, "Computational Geometry Algorithms Library," [retrieved: 2012, 05]. [Online]. Available: http://www.cgal.org

[6] Adobe Inc., *PostScript Language Reference Manual*, 3rd ed. Addison-Wesley, 1999.

[7] Processing, "Processing," [retrieved: 2012, 05]. [Online]. Available: http://www.processing.org

[8] 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.

[9] Robert McNeel & Associates, "Rhinoceros 3D," [retrieved: 2012, 05]. [Online]. Available: http://www.rhino3d.com

[10] 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.

[11] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky, "Instant architecture," *Proc. SIGGRAPH 2003*, pp. 669 – 677, 2003.

[12] 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.

[13] Esri, "CityEngine," [retrieved: 2012, 05]. [Online]. Available: http://www.esri.com/software/cityengine/

[14] 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.

[15] G. Patow, "User-friendly graph editing for procedural buildings," *Computer Graphics and Applications, IEEE*, vol. PP, no. 99, p. 1, 2010.

[16] 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., 1999, pp. 3–61.

[17] 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, vol. 3929, pp. 147–160.

[18] Y. Baulac, "Un micromonde de géométrie, cabri-géométre," Ph.D. dissertation, Joseph Fourier University of Grenoble, 1990.