# Interactive Visual Editing of Grammars for Procedural Architecture

Markus Lipp* Peter Wonka† Michael Wimmer*

*Vienna University of Technology † Arizona State University

**Figure 1:** *Screenshots from our real-time editor for grammar-based procedural architecture. Left: Visual editing of grammar rules. Middle left: Direct dragging of the red ground-plan vertex and modifying the height with a slider creates the building on the middle right. While dragging, the building is updated instantly. Right: Editing is possible at multiple levels, here the high-level shell of a building is modified.*

## Abstract

We introduce a real-time interactive visual editing paradigm for shape grammars, allowing the creation of rulebases from scratch without text file editing. In previous work, shape-grammar based procedural techniques were successfully applied to the creation of architectural models. However, those methods are text based, and may therefore be difficult to use for artists with little computer science background. Therefore the goal was to enable a visual workflow combining the power of shape grammars with traditional modeling techniques. We extend previous shape grammar approaches by providing direct and persistent local control over the generated instances, avoiding the combinatorial explosion of grammar rules for modifications that should not affect all instances. The resulting visual editor is flexible: All elements of a complex state-of-the-art grammar can be created and modified visually.

**CR Categories:** I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

**Keywords:** Procedural Modeling, Architectural Modeling, Usability, Shape Grammars

## 1 Introduction

Content creation is one of the most important challenges in computer graphics today. The current approach to 3D modeling is to manually create 3D geometry using tools like Autodesk Maya or 3ds Max ™. This process is time consuming, tedious and repetitive, but allows the artist full control over every aspect of the final 3D model. Recently, grammar-based procedural modeling has shown promising results, for example for architectural modeling [Wonka et al. 2003; Müller et al. 2006]. However, these approaches allow only *indirect* control over the final model by changing the underlying grammar, or *global* control by changing some parameters. When more fine-grained control over individual buildings is needed, a tedious change grammar-regenerate- ... cycle is required until the desired output is achieved. Furthermore, current procedural modeling systems are mostly text based and therefore impractical for the intended users, i.e., artists and technical artists.

In this paper, we aim to take the next step in procedural modeling by combining the full generative power of design grammars with the ease of use and flexibility of 3D modeling systems. This is achieved by introducing visual editing, with direct local control of all aspects of the grammar.

**Direct Visual Editing** While text-based production systems are very powerful, end users require a visual frontend to be able to use them productively. It is important to provide both a visual rule editor to create and modify the individual rules of the grammar and immediately see the consequences for the generated models, as well as a visual model editor which allows modifying the instances generated by the rule derivation process.

**Local Modifications** Assume the artist wants to assign a different texture, different window width or different ornamentation rule to a specific window on a facade. In a current text-based procedural modeling system, the artist would have to write several new rules to identify the floor and column of the window and add the modification. In a visual editor, the desired workflow is that the artist simply *selects* the desired window and chooses a new texture, rule or window width. To make this happen, we need to solve the problem how to allow *local* modifications to variables, rule selection and geometry, *without* having to change the underlying grammar.

*{lipp|wimmer}@cg.tuwien.ac.at, 1040 Vienna, Austria

†peter.wonka@asu.edu, Tempe, AZ 85287-0112

**Semantic and Geometric Selection** Local modification should often be applied to several elements, not just one. For example, all windows in a specific floor or column should have a changed appearance. We therefore need to provide mechanisms to select elements based on *semantic* attributes like facade number, floor or column, which are not limited by the derivation hierarchy. These should be paired with standard tools known from 3D modeling like selection rectangles etc.

**Persistence** Local modifications are in a sense more volatile than actual grammar rules. Assume, for example, that a specific window on a facade of a building is modified. If the next modification is to change the height of the building, the whole instance has to be regenerated from the grammar rules. In order not to lose the previous modification, any modification has to be stored *persistently* – this is especially difficult when the structure of the grammar changes.

**Main Contribution** The main contribution of this paper is to enable a visual editing workflow for grammar-based modeling by providing the aforementioned functionalities. In particular, we introduce a set of visual operators for both rule and building editing, and introduce the concept of *exact* and *semantic locators*, which will be used to allow local modifications and semantic selections, as well as to solve the persistence problem. Local modifications are a true extension of the expressive power of design grammars, and bring procedural modeling a step closer to a workflow acceptable to artists.

The remainder of the paper is structured as follows: At first we provide a description of our visual editing concepts in Section 2. Section 3 gives details and discusses how we use exact and semantic locators, while Section 4 discusses the visual building and rule editors. In Section 5 the workflow and performance is evaluated, and implementation details are provided. In Section 6 a discussion is provided comparing our method to previous work.

## 2 Visual Editing Concepts

In this section we provide an overview of design grammars and the problems that occur during visual editing, and our concepts to solve them.

The main concept of a design grammar as used for example for architecture is based on a *shape grammar* utilizing a rulebase (note that in this paper, we do not introduce a formal grammar notation, but loosely follow the notation of [Müller et al. 2006], while the concepts work for other grammars as well): Starting from an initial axiom shape (for example a ground plan), rules are applied, replacing shapes with other shapes. A rule has a *labeled shape* on the left hand side, called *predecessor*, and one or multiple shapes and *commands* on the right hand side, called *successor*. Commands are macros creating new shapes or commands. Three commands were introduced in [Müller et al. 2006]: *Split* of the current shape into multiple shapes, *repeat* of one shape multiple times and *component split* creating new shapes on components (e.g. faces or edges) of the current shape. Every rule application creates a new *configuration* $C_i$ of shapes. During a rule application, a hierarchy of shapes is generated corresponding to a particular *instance* created by the grammar, by inserting the successor shapes as children of the predecessor shape. This *production process* is executed until only terminal shapes are left. An example rulebase is visualized in Figure 2, from this rulebase the instance and associated shape hierarchy in Figure 3 are *automatically* generated.

The power of design grammars lies in their capability to produce *variations*. This means that each instance created by the grammar

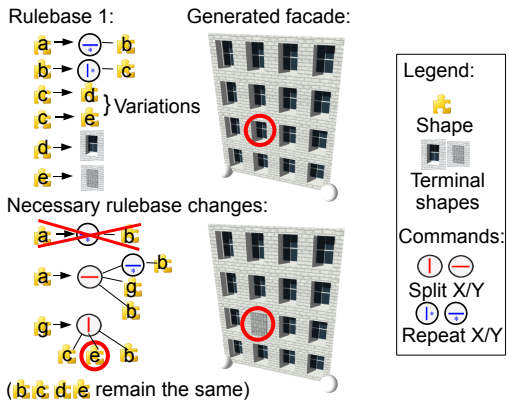


**Figure 2:** *The rulebase on the top has two possible windows, enabling variations during generation. An example output is shown on the top right. If we want to specify the window type to be used for the encircled window, we have to manually rewrite rules in order to set the window. The necessary rulebase changes are shown at the bottom, creating the new rendering. We found this rewriting to be tedious and error prone, even when just one variation is controlled.*

will look different. The following mechanisms are available to introduce variation in design grammars:

1. multiple possible production rules for a shape (chosen stochastically)
2. parameters (e.g., window width) chosen according to variables set by the user (in a text file)
3. random parameter assignments in rules

Note that all these mechanisms are *global* in nature, i.e., they are typically only chosen once for a whole instance. If a shape that uses a particular variable appears in several nodes in the shape hierarchy, there is no way to assign different values to the different nodes. Instead, for each node that should differ from the rest, a set of new rules have to be introduced which expose the desired variability via new variables, as shown in Figure 2. This is tedious and quickly leads to an explosion of the rulebase.

**Direct Control of Variation** In this paper, we propose a new paradigm for rule-based modeling: *direct local* control of the shape hierarchy, shown in Figure 3.

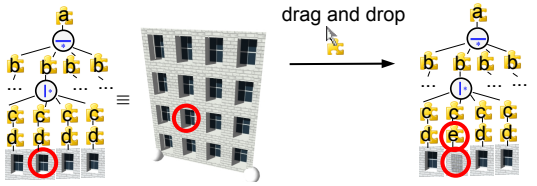


**Figure 3:** *On the left, the automatically generated shape hierarchy corresponding to the facade in Figure 2 is shown. Only the second floor is visualized, in order to increase readability. Utilizing direct control, the user can drag and drop the desired window on the rendered facade, automatically changing the underlying shape hierarchy, as seen on the left. No manual rewriting is necessary for the user.*

At first, we provide tools to directly modify the shape hierarchy of an instance via so-called *locators*, which are stored externally to the

grammar, thus decoupling local modifications from the grammar. Direct changes are important to create an artist-controlled unique look for a specific instance. In particular, we introduce the following operators which can act locally on any level of the shape hierarchy:

1. modify the variables used in a particular node in the shape hierarchy
2. select the production rule applied to a shape (selected from multiple possibilities given by the grammar, or even arbitrary rules)
3. pin random choices
4. directly set the geometry and textures used in a particular terminal shape

Note that the items selected in 2 to 4 can be (automatically) expressed as variables and are therefore special cases of 1. While these operators may seem to be straightforward to implement, actually they are not. Since these direct modifications need to be provided in a *visual editor*, we have identified two major problems:

**Selection Problem** Every variable is used in at least one, but typically in several nodes in the shape hierarchy. For example, the window width is a variable that can be set once for a building and is used in every window tile. Obvious ways to influence the window width are to set one width for all windows (the default), or to set the width of some windows individually. However, the most common required action will be to change the variable for a certain subset of nodes in which the variable is used. One solution is to provide *hierarchical* assignments, i.e., a variable can be assigned a value at any node in the hierarchy. However, that is often not sufficient. We also require *semantic* assignments, for example, selection of all windows in a floor, or all windows in a column. Both subset selection methods are shown in Figure 4.


**Figure 4:** *Using hierarchical selections, all shapes underlying a specific shape in the shape hierarchy are selected. Semantic selections allow selecting multiple shapes that share common semantic properties. Please note that it is impossible to select a whole column just by using one hierarchical selection in this shape hierarchy, as there is no rule that directly represents a whole column.*

**Persistence Problem** Most modifications $m_i, 1 \leq i \leq 4$, require the instance along with its shape hierarchy to be regenerated from the grammar rules. However, as shown in Figure 5, this will obliterate any previous modification to the instance. Therefore, any modification needs to be stored persistently. However, rebuilding the shape hierarchy could lead to a different hierarchy, so an important problem to solve is how to apply a modification to a different shape hierarchy.

We solve both of these problems using *semantic annotations* which can easily be specified in the visual editor, and provide some heuristics how to set these annotations automatically. We define two types of instance locators: *exact instance locators* and *semantic instance locators*. An exact locator stores the exact location of a shape in

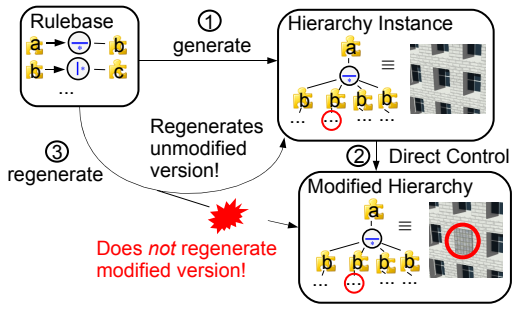


**Figure 5:** *A persistence problem occurs when we at first generate a shape hierarchy from the rulebase, and then modify this hierarchy utilizing direct control. When we need to perform a regeneration (for example because the house height has changed) the unmodified version is generated, thus all direct modifications are lost.*

the shape hierarchy. This is essentially done by storing all information occurring along the path from the shape to the axiom. Using an exact instance locator we can *unambiguously* locate shapes, and thus retain selections after a regeneration. However, this does not allow semantic selections based for example on floor and column numbers. We therefore introduce *semantic tags* that can be attached to structural commands in the shape hierarchy (e.g., split and repeat commands).

**Visual Rule Editing** Finally, we provide an intuitive user interface for editing the grammar in a visual editor. We have identified the following important operations, which will be further explored in Section 4:

- graphical assignment of numeric values (i.e., dragging split planes)
- multiple views on a rule: rendering, visualization of commands and structural overview
- focus and context visualization of rules

## 3 Instance Locators for Local Control

Three problems emerge when we want to enable local control. At first, we have to describe how we *select* shapes in the hierarchy. Secondly, we must define how to *apply modifications* to those selections. Thirdly, those modifications have to be *persistent*. We will describe solutions to those problems in this section, which rely mainly on the concept of *instance locators*.

### 3.1 Selections

There are three types of selections we want to provide: Selection of a *single* shape in the hierarchy, *hierarchical selections* and *semantic selections*. Essentially a selection works as follows: (1) The user clicks on the 3D rendering. Using intersection tests, we calculate which shape in the shape hierarchy the user has clicked on. (2) For this shape an *exact instance locator* or a *semantic instance locator* is calculated and stored in *loc*. (3) To actually highlight all selections in the rendering, we use the locator *loc* by comparing it to the (temporarily created) locator of each shape in the hierarchy, and highlight all matches. Let us now define instance locators in more detail:

**Exact Instance Locator** The selection of a *single* shape in the shape hierarchy is specified the following way: For every shape or command in the hierarchy graph, we sequentially number every outgoing edge from left to right starting from 1. An example numbering is shown in Figure 6. In order to specify one shape $s$, the unique path $p$ from the axiom to $s$ is determined. We walk along this path, and write every shape, command and edge number we encounter into an ordered tuple. We call this tuple *exact instance locator*, as it allows us to uniquely specify a single shape instance. The resulting exact instance locators for Figure 6 are shown in Table 1. Note that it would theoretically suffice to use edge numbers only, but we also include shapes in the locator to be able to check whether the locator is valid after a hierarchy change.

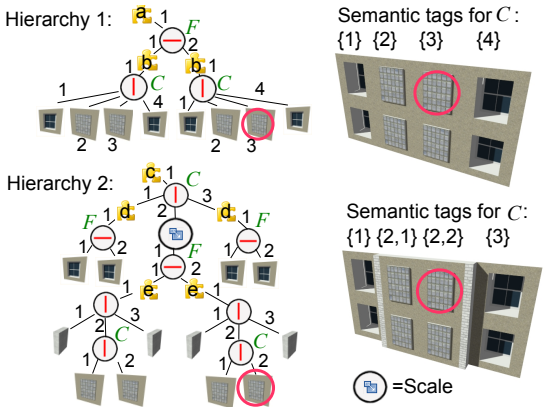


**Figure 6:** *Hierarchy 1 corresponds to the rulebase in Figure 7, hierarchy 2 is a more complex example. Red circles around shapes represent corresponding selections. Edges are sequentially numbered. Over the bottom rendering, the semantic tags and corresponding absolute values of columns are shown. Please note that while the renderings are quite similar, the underlying graphs are significantly different.*


**Figure 7:** *We introduce semantic tags attachable to commands, represented here as F for floor and C for column.*

A hierarchical selection is simply a selection of a single shape that is an internal node of the shape hierarchy, and therefore can also be expressed with an exact instance locator.

**Semantic Instance Locator** Using just the shape hierarchy and exact locators, there are two problems:

1. We cannot select shapes based on semantic attributes, for example *2nd column on the 3rd row*.
2. We cannot perform semantic queries in the form *select all shapes in the 2nd column*.

Those problems are caused by the lack of semantic information in the shape hierarchy. We therefore introduce *semantic tags* that can be attached to split or repeat commands. For example, in Figure 7 the semantic tags $floor$ and $column$ are attached to rules. The tag $facade$ can be used to differentiate facades. Actually defining semantic tags is very easy for the user: In our rule editor, a simple drag and drop operation of a tag on a command applies the tag. This has to be performed only once when defining rules, everything else is done automatically. Please note that in Figure 6 (bottom) not every vertical (X)-split has the tag $column$ applied, in this way the user can specify what is considered a unique column and what not.

*Automated semantic tagging* is possible by using the following heuristic: The user specifies threshold $f_m$ for the height of multiple floors and $f_s$ for the height of one floor. Y-Splits/Repeats where the scope height is above $f_m$ are automatically tagged as $floor$, directly succeeding shapes or commands having a scope height below $f_s$ are excluded from this tagging. Analogously X-Splits/Repeats are tagged as $column$ using the width as threshold. All component splits having a height above a threshold are tagged as $facade$.

A *semantic instance locator* for shape $s$ can now be constructed from a hierarchy graph the following way: At first, we construct the unique path $p$ from the axiom to $s$. We walk along this path, and when we encounter a tagged command, the assignment $tag = edgenumber$ is added to the locator. $edgenumber$ is the sequential numbering of the edge to the next shape already introduced for exact instance locators. It is important to note that tags can occur multiple times in the locator. By using the sequential edge number, we exploit the spatial ordering split and repeat rules provide. Example semantic instance locators for the encircled shapes in Figure 6 are shown in Table 1. We can improve the granularity of semantic locators by additionally saving the symbol of shape $s$.

| Marked S. | Exact Instance Locator |
|---|---|
| Hierarchy 1 | $\{a, 1, S_y, 2, b, 1, S_x, 3\}$ |
| Hierarchy 2 | $\{c, 1, S_x, 2, Scale, 1, S_y, 2, e, 1, S_x, 2, S_x, 2\}$ |

| Marked S. | Semantic Instance Locator |
|---|---|
| Hierarchy 1 | $(F = 2, C = 3)$ |
| Hierarchy 2 | $(C = 2, F = 2, C = 2)$ |

**Table 1:** *Exact and semantic instance locators for the encircled shapes in Figure 6 are shown here. $S_x, S_y$ represent a Split X/Y command, $Scale$ a scale command.*

Using semantic instance locators it is easy to solve the previously mentioned problems:

1. We can perform selections based on semantic attributes simply by specifying a semantic locator, and searching for shapes with matching locators.
2. Selections of whole columns are done by ignoring the $floor$ tags during selection searching. For entire floor selection this works analogously.

Semantic assignments are assignments applied to shapes having a specific semantic tag. Please note that semantic assignments can be flexibly combined with hierarchical assignments, by attaching semantic assignments to internal nodes of the hierarchy graph.

Exact instance locators are used by the rule editor to specify selections in the GUI. Both exact and semantic locators can be used in the building editor.

**Anchor Points** We can construct locators relative to an *anchor point*. Possible Anchor points are either left, right, or center in horizontal direction and bottom, top, or center in vertical direction. Per default the closest anchor point is chosen (with priority on the borders for equal distance to the center), because that proved to be most intuitive in our experience. Additionally, the user has the option to modify an anchor point. The actual generation of anchored locators is straightforward: Instead of $tag = edgenumber$ we use $tag = numEdges - edgenumber$ for locators anchored to right

or top, and $tag = numEdges/2 - edgenumber$ for locators anchored to the center. For example, using anchored locators it is very easy to specify a selection that should always contain the center column.

### 3.2 Direct Modifications and Persistence

Using a selection specified in an instance locator, we can specify *where* a modification should be performed. The next step is to define *what* should actually be modified. As we already mentioned in Section 2, all local modifications (node variables, rule to use, random choices, geometry and textures) can be expressed as variable assignments, thus a variable assignment describes *what* to modify.

Using both a locator $loc_m$ and a variable assignment $v_m$, a modification is exactly specified. In order to allow hierarchical assignments, all variable assignments have the following properties: The assignment extends its scope to the underlying *subtree*, thereby modifying the value of variables used in any rule application of this subtree. Subsequent assignment on lower levels override assignments on higher levels, thus providing local control for every shape. An example assignment is shown in Figure 8.

The key to *persistence* is that instead of applying $v_m$ to the current hierarchy, we *save* $loc_m$ *and* $v_m$ *externally*. Of course we can save multiple modifications externally, thus preserving previous modifications.

The actual application of $v_m$ is carried out in the following way: First we delete the current shape hierarchy, and start the production process from the axiom (configuration $C_0$). Now, every time we insert a shape $s$ into a configuration $C_n$, we create an instance locator $loc_s$ for $s$. When $loc_m = loc_s$, we attach the variable assignment $v_m$ to $s$. Note that in a longer editing session, some variables can be assigned different values using the same locator. In this case later assignments override previous ones.

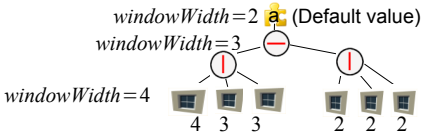


**Figure 8:** *On every shape we can assign values to variables. Assignments extend their scope to all underlying shapes. Assignments on lower levels override assignments on higher levels. Numbers below windows show the values of* $windowWidth$.

**Transformations between Hierarchies** Semantic locators are normally quite robust with respect to modifications in the visual editor, like changes of the floor plan, modifications of variables etc. However, sometimes the hierarchy changes in a way that a semantic locator does not fit the hierarchy anymore. This can happen when the rulebase is edited in the rule editor, or when substantial changes are made to the choices of productions in higher levels in the shape hierarchy. Still, we want to give the user the possibility to recover modifications and apply them to the new hierarchy by *transforming* the locator. Please note that such an algorithm can also be used to transfer modifications from one building to another.

We classify semantic locators into three categories: (a) valid, (b) structurally invalid and (c) semantically invalid. Valid locators correspond to a path in the current shape hierarchy. A locator $loc_m$ is structurally invalid when a *prefix* of $loc_m$ matches the locator $loc_s$ of some shape $s$ in the shape hierarchy, but the $edgenumber$ following the prefix is greater than the number of outgoing edges of $s$. This usually happens when resizing scopes, causing subtrees attached to repeat commands to disappear. Finally, semantically invalid locators are those that are not valid or structurally invalid, i.e., they do not fit the current shape hierarchy at all.

In general, artists prefer direct control over heuristics that run automatically. Therefore, in order to deal with invalid locators, we offer the user a command to transform semantically invalid locators to the current hierarchy. In our implementation a list of locators is offered, color coded according to their category. By selecting a semantically invalid locator from this list and confirming with a button, this locator can be automatically transformed to the new hierarchy using the algorithm described in the following. Note that it does not make sense to transform structurally invalid locators since they often become valid again due to further operations (resizing, ...).

In the following we describe how to transform a semantically invalid locator using so-called *sequential orderings* for semantic tags. Optionally, all calculations can be restricted to the subtree of the shape hierarchy that does not include a possibly matching prefix in order to preserve the maximum semantic context. The idea is to assign one *ordering value* per tag to every shape, regardless of how often the tag appears in the semantic locator. The ordering value depends on the spatial directions of the tags encountered along the way, since we typically have tags "split" by other tags. When externally saving semantic locators, we also save the corresponding ordering values.

Example sequential ordering directions are shown in Figure 9 (top). Using this sequential ordering, a transformation of a semantic locator $loc$ is done the following way: We compare the *ordering values* (instead of the edge numbers) of this locator with the ordering values of the other hierarchy. When a match is found at shape $s$, the transformed semantic locator is $loc_s$.

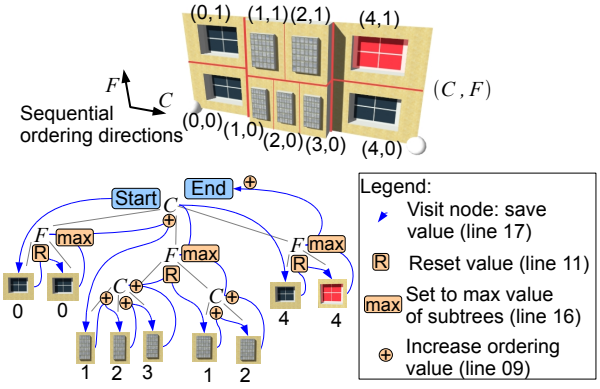


**Figure 9:** *Example for ordering value calculation. The rendering corresponds to the hierarchy graph. To increase readability, we only show elements having a tag attached in the graph, as other elements do not influence the algorithm. Using a blue line we illustrate the traversal of the algorithm during calculation of tag* $C$*, while orange circles highlight important events occurring during the traversal. The resulting sequential numbering is overlaid in the rendering. Line numbers correspond to Figure 10.*

The algorithm to calculate ordering values for a tag `currentTag` and each node in the shape hierarchy is illustrated as pseudo code in Figure 10 and works as follows: A modified post-order traversal of all nodes is performed, and after each visited subtree it is decided if the ordering value `seqValue` should be increased. Basically a node `node` associated with `currentTag` increases `seqValue` by one if `currentTag` does not occur in the last visited subtree. There are two possibilities if a different tag `otherTag` is asso-

ciated with `node`: (1) That tag is ignored for the calculation of `seqValue`. (2) The tag is defined to be a "semantic splitter", i.e., `seqValue` is calculated in each of the subtrees of the tag *independently*. The overall increase of `seqValue` is determined by the *maximum* increase in any of the subtrees. For example, floors and columns are mutual semantic splitters, thus preventing a column tag to be counted on multiple floors. Semantic splitters for arbitrary tags can be defined in a table, but usually correspond to tags associated to splits/repeats in alternating coordinate axes.

```
currentTag = ...; ordValue=0; //Init values
01:CalcAbs(node) {
02: isSplitter = Is tag attached to node semantic
                 splitter of currentTag?
02: hasCurrentTag = Is tag attached to node equal currentTag?
03: previousOrd= ordValue;  //Save current ordering value
04: int maximum= 0;
05: for all children of node {
06:  Postorder traversal: recurse into CalcAbs(child)
07:  //After subtree was visited:
08:  if (hasCurrentTag&(currentTag not found in subtree ))
09:   ordValue++;  //Increase ordering count!
10:  if (isSplitter) { //Special handling of splitters
11:   ordValue= previousOrd; //Restore previous ordValue
12:   maximum= max(ordValue, maximum);
13:  }
14: }
15: if (isSplitter)
16:  ordValue= maximum; //Set to maximum subtree count
17: node.tag.ordValue= ordValue; //VISIT node: save value
18:}
```

**Figure 10:** *Pseudocode for ordering value calculation. Essentially this is a modified postorder traversal with special measures to increase the count of the ordering value after each subtree was visited. Lines that handle arbitrary nesting of tags are marked orange, lines that actually increase the ordering value are yellow.*

Please note that this algorithm exploits structural information created by split/repeat commands, *not* actual world-space positions of shapes. Therefore, when scope translation commands occur in lower-level shapes, it may produce incorrect sequential orderings—in the worst case this can lead to modifications being transferred to shifted positions. To prevent this, world-space positions would have to be considered for shapes with translation commands, for example by casting a ray along the ordering direction to determine the world-space ordering.

## 4 Interactive Visual Editor for Grammars

We will now explore how the introduced concepts fit together creating a new visual editing paradigm for grammars. Inspired by the observation of direct versus indirect control, we have separate windows for a building editor and a rule editor in our GUI, seen in Figure 11. The rule editor provides indirect control, allowing the creation of rulebases from scratch, and the building editor provides direct variation control to the artist. All windows can run concurrently, and are linked in several ways: It is possible to edit a rule, and show the effects of this edit on a building. Further drag and drop functionality is provided between the windows, allowing for example direct application of textures.

### 4.1 Building Editor

In our building editor shown in Figure 12, we want to allow direct variation control to configurations, without changing the rulebase. In order to specify the position of a variation, we need to enable selections of shapes at first. This is done using *picking* in the 3D rendering: A user can click on a shape, we internally create an *instance locator* for this shape. In order to specify *hierarchical selections* we provide a button to move the selection up one level in the hierarchy. *Semantic selections* can be specified by using checkboxes

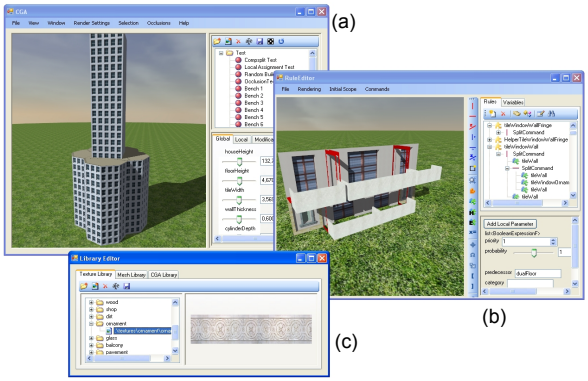


**Figure 11:** *Three windows make up our GUI: (a) A building editor enables direct variation control on buildings (b) Rulebases can be visually created from scratch in the rule editor, providing indirect control (c) Textures and meshes are stored in the library editor.*

for "On Whole Row", "On Whole Column" or "On All Facades" in the menu.

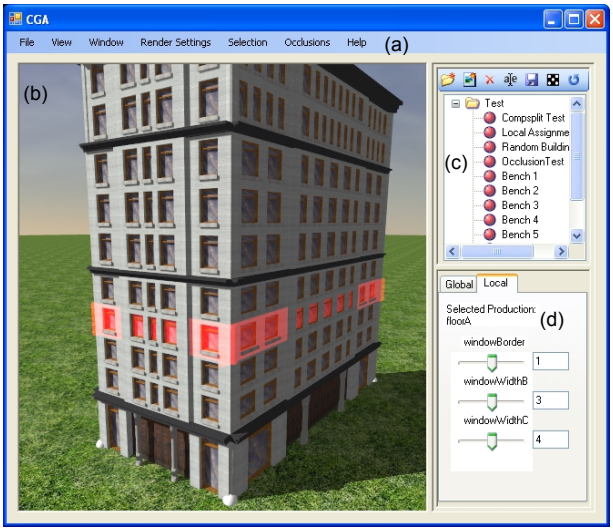


**Figure 12:** *Interactive building editor providing direct variation control. (a) Menu with various rendering and derivation controls (b) Real-Time rendering of result. Currently a floor is selected. (c) List of all building instances (d) All parameters occurring in the currently selected shape.*

To actually perform modifications on selections, we expose controls for variable aspects: Sliders are automatically created for every parameter occurring in the current selection. The user can now locally adjust those parameters, implicitly creating a *variable assignment* attached to the current selection. In order to specify a production rule to be used, the user can *drag and drop* a specific production from a rule library on the selection. Modifications of textures and meshes are performed by *drag and drop* from a texture or mesh library. Persistence is maintained as previously described: We store the instance locator together with the modification to be performed externally.

An example showing hierarchical selections and modifications is shown in Figure 13. Hierarchical selections allow artists to specify the granularity of their direct modifications.


**Figure 13:** *Sequence showing hierarchical modifications. (1) A shape representing a floor is selected. (2) The parameter $windowHeight$ is modified - all shapes on lower levels are automatically modified (3) Selecting a specific shape allows overriding the parameter on a lower level.*

### 4.2 Rule Editor

We implemented a rule editor based on the language elements of CGA shape [Müller et al. 2006], all of which are visually editable, and we can create rulebases from scratch. Three views on the currently edited rule make this possible:

First, a 3D rendering of the derivation overlaid with a visualization of shapes and commands provides direct visual feedback, seen in Figure 14(b). Second, a treeview displaying all rules occurring in the current derivation allows easy navigation and provides an overview, as seen in Figure 14(c). Finally, all parameters of the currently selected shape or command are automatically mapped to standard GUI elements like sliders and checkboxes, seen in Figure 14(d). This way, all elements that are not visualized can be edited. An example workflow using this editor is shown in Section 5. Let us now describe those concepts in greater detail:

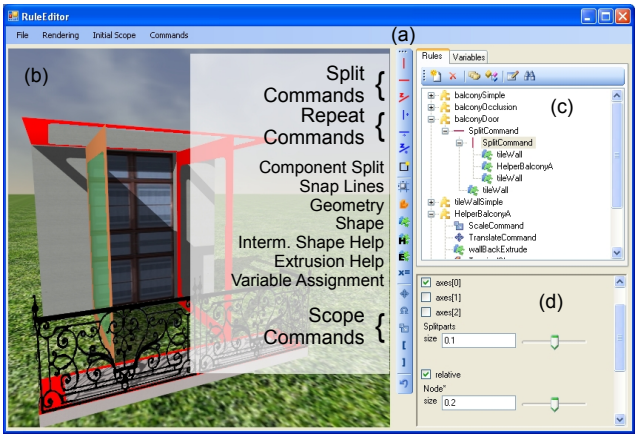


**Figure 14:** *Interactive Rule Editor. (a) Tool palette allows creation of new commands and rules (button descriptions were added to the screen shot) (b) Real-time rendering of result and visualization (c) Linked tree-view. Yellow puzzle icons represent predecessor shapes, blue/green puzzle icons represent shapes occurring in a successor. (d) Parameters are automatically mapped to GUI elements.*

**3D Visualization** In order to edit rules and commands in a rendering, they have to be visualized. Split and repeat commands are visualized by rendering their dividing planes. We use a plane with a rounded rectangular hole in the middle (achieved with alpha tests) to reduce occlusion issues. Depth cues by reducing brightness for distant objects are employed to make distinctions of different visualized commands easier. Currently selected shapes or commands are highlighted with a surrounding transparent box. An example visualization can be seen in Figure 14 (b). We also experimented with visualizations of other aspects, like relationships and nesting of rules. However, we felt that a linked tree-view, as explained later, is more suitable for those aspects.

**Visual 3D Editing** Utilizing the visualization we allow direct editing: The user can pick a visualized element (for example a dividing plane), and drag the element around. When multiple choices are possible, a context menu allows specific selections. Internally, this works as follows: When a dividing plane is picked, we construct a perpendicular helper polygon going through the intersection location of the picking ray, as seen in Figure 15. We need this polygon to restrict the possible mouse positions and to fix the depth $z$ of the intersection – without this polygon we experienced oscillations of the dividing plane during editing. The helper polygon is stored until a new plane is picked.

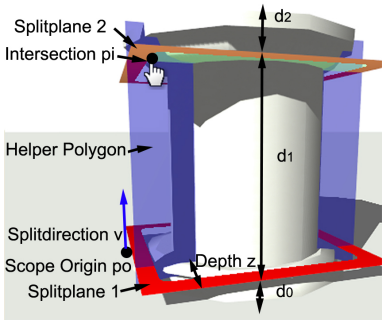


**Figure 15:** *Geometry during picking and direct dividing plane movements.*

When the user moves the mouse cursor, we intersect the picking ray with the helper polygon, yielding an intersection point $p_i$. Now we have to recalculate the parameters of the command containing the picked plane, meeting the following condition: After regeneration of the edited rule, the dividing plane has to intersect the helper polygon on the intersection point $p_i$.

For repeat commands, this calculation is trivial: A dot product of $(p_i - p_o)$ with the repeat axis vector $v$, divided by the number of the modified plane, yields the new value for the repeat with. Split commands are more difficult, because split sizes can be defined relative or absolute [Müller et al. 2006]. We recalculate the parameters as follows: At first, the desired sizes $d_i$ are calculated, this is trivially done using dot products and subtractions. Then we need to calculate split size parameters $s_i$ that generate sizes $d_i$. Absolute sizes $sabs_i$ are simply set to $d_i$. Relative sizes $srel_i$ are set to $d_i \cdot (\sum srelold)/(scopesize - \sum sabs)$ with $srelold$ being the relative sizes before the mouse movement.

When variables are involved in the parameters, we simply add an offset to this variable (e.g. $var$ gets $var + 0.3$), so no variables are lost during visual editing.

**Linked Views** Additionally to the visualization we created a treeview displaying all rules, shapes and commands occurring in the current derivation, as seen in Figure 14 (c). This treeview is linked with the visual representations: Selections in the treeview automatically select an element in the derivation and the other way round. It is important to note that we have a $(1 : n)$ mapping here: One element in the treeview may correspond to $n$ elements in the derivation, because multiple instances are possible. Therefore, when selecting an element in the treeview, just the first occurring element

in the derivation is selected. Additional features of the tree-view include searching for shapes, and support for drag and drop to copy or move shapes. During evaluation, we found the treeview to be better suitable in representing relationships of rules, while the rendered result is more aimed at adjusting specific commands.

**Focus and Context** When editing high-level rules, for example to model building shells, visualization and rendering of lower level rules may be distracting. Therefore we implemented focusing based on the amount of levels between the edited shape $u_e$ and other shapes $u$: We can set the amount of displayed levels $i$ with a slider. Only shapes where the path between $u_e$ and $u$ contains at most $i$ shapes are displayed. For other shapes, either a proxy geometry (to represent the context) or nothing at all is rendered. An example for focusing is seen in Figure 1 on the right. Focusing can be separately controlled for rule rendering and visualization.

**Parameter View** Not all parameters of commands or shapes are mapped to visualizations. In order to enable visual editing of those parameters, they are automatically mapped to standard GUI elements, as seen in Figure 14(d). When variables occur in numerical parameters, slider adjustments simply add an offset to the variable.

**UV Mapping Control** An element missing from previous design grammars is direct control for UV texture coordinate mapping – this is very important for artists. Therefore, we introduce a new *UV mapping* command: It can be inserted into any rule, and defines a parameterized UV mapping, for example a box UV mapping with parameters tiling and offset. All shapes underlying this command in the shape hierarchy can automatically use this mapping.

**Completeness of Editing** Utilizing the described methods, all CGA shape concepts are visually editable: Snap lines can be inserted into a rule utilizing the tool bar buttons seen in Figure 14(a). They are visualized in the 3D rendering. The properties of the snap line are editable with standard GUI widgets in the parameter view. Scope modifications (translation, rotation, scale, push, pop) are also inserted using tool bar buttons, and can then either be modified using standard 3D manipulators (e.g. Arc rotate) in the 3D view or by modifying sliders. When a rule is selected, a text field in the parameter view allows entering arbitrary conditions using variables and Boolean operators. As occlusions are also defined as conditions (using automatically initialized variables like `isShapeOccluded`), they can be defined analogously.

## 5 Implementation and Results

**Modeling Workflow** As our main contributions enable complete visual editing of building grammars, we will now show a case study creating a building grammar from scratch. In Figure 16 the necessary steps to create a simple building in 3.5 minutes are laid out (please refer to the video for a real-time capture of the entire session). (1) A few empty rules are created and named. (2) Selecting a rule and clicking on the split icons allows easy splitting of shapes. (3) Drag and drop allows easy definitions of shapes to be used as in the split command. (4) Adding terminal shapes allows geometry addition. Textures are dragged from a texture library. Cut and paste of shapes allows fast setting of the wall tiles. (5) Direct dragging of split planes is possible in the 3D view. When `CTRL` is pressed during dragging, the planes are distributed symmetrically. Extrusion of the window can be done with one mouse click. (6) Selecting the high-level rule `house`, adding a component split with a simple click and adding a repeat command creates the building shell. Please note that both top-down and bottom-up modeling approaches

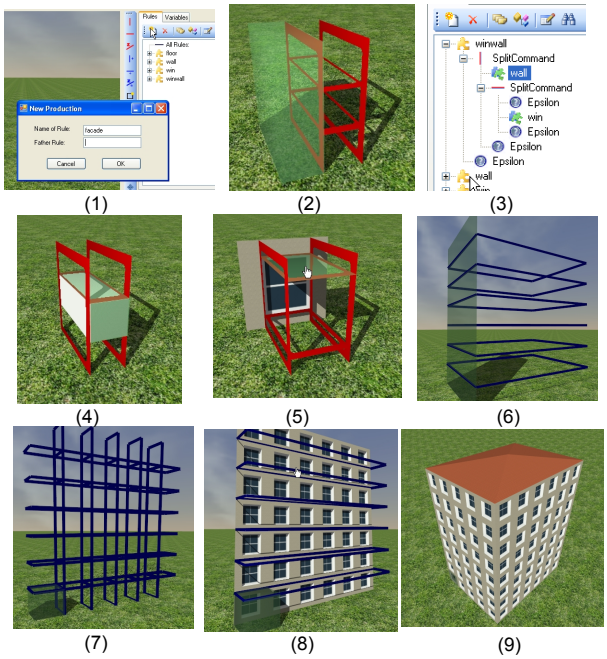


**Figure 16:** *Example workflow using our rule editor. Creating this simple building required 3.5 minutes. These are screenshots from the video accompanying the paper. The screenshots were cropped to magnify important details. A description of the individual screenshots is provided in Section 5. In approximately additional 10 minutes we were able to create the building seen in Figure 12 which has many ornamentation details.*

can be easily combined with our method. (7) Repeat commands are added to distribute windows. (8) The `winwall` rule is dragged into the repeat command. (9) Simple (but automatically adapting) roofs can be added as a terminal shape.

Starting from this simple building, we were able to create the more complex building seen in Figure 12, which has many ornamentation details, in an additional 10 minutes.

**Usability** Our method allows visual editing without resorting to text files. The only necessary textual entries are naming of new rules and parameters. Drag and drop is used in many places, simplifying rule creation. For example the simple building in Figure 16 corresponds to around 55 lines of CGA shape code. We therefore feel that this method substantially simplifies modeling of grammar-based architecture.

In order to actually evaluate the usability, we introduced artists from our industry partner to our tool. After a short introduction we asked them to create a building, and document the difficulties arising during modeling. As an advantage of our method, they found visual editing of split rules to be much more intuitive than textual editing. Especially the feature to mirror split sizes during editing by pressing a specific key was considered to be very useful.

Two identified shortcomings stem directly from the rule-based nature of CGA shape: First, the dependency between rules is not clearly visible. This can make it hard to understand the rules created by a different artist. The artists proposed to use a node-based display of rules in order to alleviate this. We will incorporate this in future work. Second, artists commented that they would be very exited about alternatives to rule editing by editing buildings using a

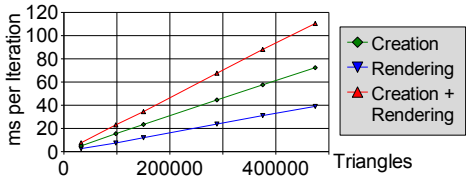


**Figure 17:** *Milliseconds per iteration versus triangles for various modes.*

philosophy of modeling by example.

A number of suggestions was related to software engineering issues and the best way on how to incorporate the software in the existing modeling pipeline. Concerning the expressivity of CGA shape, the artists suggested to add more control to the high-level building shape, mainly by allowing arbitrary ground-plans unique to every floor.

**Performance** One important aspect of a visual editing system that was not mentioned so far is response time. Our visual editor frequently needs to regenerate a whole instance from the rulebase, for example whenever a variable is changed or a constraint like the floorplan is modified through continuous dragging. Thus the performance of the grammar system is as important as the rendering performance itself. Previous work has reported building generations times in the order of a few seconds, which is far too low for interactive (i.e., > 20 updates per second) manipulation. In our implementation, we used several optimizations to ensure interactive generation and rendering times, including custom memory manager, lists, sorting algorithm and random number generator. For the occlusion test required in [Müller et al. 2006] to prevent intersections of shapes, we employed the hardware-accelerated algorithm proposed in [Knott 2003]. This is a combination of stencil buffering and hardware occlusion queries, where a building is interpreted as a shadow volume, and the z-Pass algorithm is used to search for shapes intersecting this volume. This is significantly faster than the octree-based method originally proposed.

In Figure 17 we can see that interactive performance for rendering is achieved for all tested buildings even on a relatively slow PC with Athlon XP 2600 CPU, 1024 MB RAM, and Geforce 6600 graphics card. In combination, creation and rendering is still interactive for a building with about 200,000 triangles. Figure 17 also shows that the performance scales linearly with the building complexity. This is beneficial, as a linear scaling provides a cushion for more complex buildings. For this test, the building in Figure 1 on the middle right was varied using a global height parameter.

**Limitations** We tried to experiment with tree modeling with reasonable success. We believe that our current implementation is mainly suitable for buildings, but we intend to experiment with a larger class of objects in future work. We expect that we would have to extend the rule set, but we believe that many fundamental principles of visual editing of grammars could be reused. Some restrictions in our current implementation stem from choosing CGA shape as underlying production system: CGA shape has no direct support for curved surfaces, making bridges or complex mechanical shapes hard to generate. However, when CGA shape is extended to support these concepts, our approach should be able to handle them seamlessly.

## 6 Related Work

Recent grammar-based procedural methods are based on the shape-grammar formalism that was pioneered in architecture [Stiny and Gips 1972; Stiny 1980], later simplified to set grammars [Stiny 1982]. Specific computer implementations of shape grammars were proposed [Chase 1989; Piazzalunga and Fitzhorn 1998]. It is important to note that those implementations were not targeted at the creation of visually convincing models, instead accurate ground plans were produced.

In the field of computer graphics, L-Systems were applied to generate buildings [Parish and Müller 2001]. The first method to actually generate visually convincing models with high geometric façade detail was *Instant Architecture* [Wonka et al. 2003]. This method introduced split grammars. A *split* was defined as the decomposition of a basic shapes into other shapes. Finally, *CGA shape* was introduced [Müller et al. 2006], extending split grammars in the following ways: They were the first to actually *define the syntax* of split commands. They introduced the *component split*, which allows reducing the dimensionality of the current scope. Additionally, mass modeling was introduced to create more complex buildings shells. Our approach applies to shape grammars in general, but has been implemented using CGA shape [Müller et al. 2006] and greatly enhances its usability, but also its (practical) expressive power through the possibility of local modifications.

Several extensions were proposed to L-Systems, mainly applicable to plants: [Prusinkiewicz et al. 1994] shows how to restrict the growth process of plants by introducing pruning. Later, positional information was used by querying functions depending on the current turtle position [Prusinkiewicz et al. 2001]. This allows determining high-level shapes of plants. Currently such functionality is not present in CGA shape, although it would be interesting to explore how this could be applicable to control high-level façade structures in future work. Also, methods to graphically model plants were introduced: In [Lintermann and Deussen 1999] components specifically targeted at plant modeling are connected in a graph. As those components are very specific to plants, this system can not be directly used for architecture. In [Boudon et al. 2003] a multiscale representation of plants is used in order to minimize the total number of parameters needed in order to specify a plant.

The persistence problem for parametric modeling was described in [Shapiro 2002; Hoffmann and Joan-Arinyo 2002; Havemann 2005], and it was pointed out that there is no general solution known at this time [Shapiro 2002]. Our solution to the persistence problem is specifically targeted at production systems, not parametric modeling in general. We achieve this by exploiting the production hierarchy and semantic informations.

Two methods were proposed that allow creation of procedural architecture without text editing: First, an image-based approach to create CGA shape grammars [Müller et al. 2007], which allows creating rules from building images. Our approach could be well employed to visually enhance or correct the deduced rulebase. Further, a framework for procedural modeling using a visual language was introduced [Ganster and Klein 2007]. Essentially, a mapping of programming constructs to visual symbols is performed. Those symbols can be combined visually. In contrast to our approach, *no* grammar or rulebase is used. Therefore we think this approach is orthogonal to ours and could be combined: The visual language could create some building parts, while our method could visually handle everything rule and grammar related.

A similar argument holds for other techniques *not* based on grammars: [Havemann 2005] introduces modeling using a stack-based programming language. [Birch et al. 2001] was shown to be appli-

cable to building parts. Persistent building interior generation was discussed in [Hahn et al. 2006]. The following areas are adjacent to CGA shape, and are important when whole cities need to be generated: Landscape generation [Fournier et al. 1982], building lot generation [da Silveira and Musse 2006; Laycock and Day 2003], combined street and lot generation [Flack et al. 2001; Parish and Müller 2001].

## 7 Conclusion and Future Work

We present the first real-time visual editing system that allows an artist to visually create a rulebase for shape grammars from scratch. Furthermore, we extend previous shape grammar approaches by providing direct local artist control over the generated instances, avoiding combinatorial explosion of grammar rules for modifications that should not affect all instances. This effectively combines the power of procedural modeling techniques and standard 3D modeling tools. We have described the selection and persistence problem, and provided a solution using so-called instance locators.

While our framework enables real-time editing of individual buildings, there are still some open problems if those buildings should be used in a real-time game. These include: (1) Automatic LOD generation: In order to render large scale scenes, low-detail versions of the building should be created automatically, assisted by the grammar structure. (2) Mesh cleanup: At the moment, the mesh may contain T-Vertices and coplanar polygons, which needs to be resolved for real-time rendering applications. For our visual editor, the next logical step is to extend the concept to whole cities, not just individual building instances. This would include creation of roads, landscapes and distribution of zones in the city.

## 8 Acknowledgements

We thank Stefan Kubicek and Johannes Graf from our industry partner Sproing for helpful suggestions on the user interface. This research was supported by the Austrian FIT-IT Visual Computing initiative, project GAMEWORLD (no. 813387), and by the NSF, contract nos. IIS 0612269, CCF 0643822, and IIS 0757623.

## References

BIRCH, P., BROWNE, S., JENNINGS, V., DAY, A., AND ARNOLD, D. 2001. Rapid procedural-modelling of architectural structures. In *VAST '01: Proc. of the conference on Virtual reality, archeology, and cultural heritage*, ACM Press, NY, USA, 187–196.

BOUDON, F., PRUSINKIEWICZ, P., FEDERL, P., GODIN, C., AND KARWOWSKI, R. 2003. Interactive design of bonsai tree models. In *CG Forum: Proc. of Eurographics*, EG, vol. 22, 591–599.

CHASE, S. 1989. Shapes and shape grammars: from mathematical model to computer implementation. *Environment and Planning B: Planning and Design 16*, 2, 215–242.

DA SILVEIRA, L. G., AND MUSSE, S. 2006. Real-time generation of populated virtual cities. In *VRST '06: Proc. of the ACM symposium on Virtual reality software and technology*, ACM Press, NY, USA, 155–164.

FLACK, P., WILLMOTT, J., BROWNE, S., ARNOLD, D., AND DAY, A. 2001. Scene assembly for large scale urban reconstructions. In *VAST '01: Proc. of the conference on Virtual reality, archeology, and cultural heritage*, ACM Press, NY, USA, 227–234.

FOURNIER, A., FUSSELL, D., AND CARPENTER, L. 1982. Computer rendering of stochastic models. *Commun. ACM 25*, 6, 371–384.

GANSTER, B., AND KLEIN, R. 2007. An integrated framework for procedural modeling. In *SCCG '07*, Comenius University, Bratislava, M. Sbert, Ed., 150–157.

HAHN, E., BOSE, P., AND WHITEHEAD, A. 2006. Persistent realtime building interior generation. In *sandbox '06: Proc. of the ACM SIGGRAPH symposium on Videogames*, ACM, NY, USA, 179–186.

HAVEMANN, S. 2005. *Generative Mesh Modeling. PhD thesis*. TU Braunschweig.

HOFFMANN, C., AND JOAN-ARINYO, R. 2002. *Handbook of Computer Aided Geometric Design*. Elsevier, ch. 21: Parametric modeling, 519–541.

KNOTT, D. 2003. *CInDeR Collision and Interference detection in real time using graphics hardware*. Master's thesis, UBC.

LAYCOCK, R. G., AND DAY, A. M. 2003. Automatically generating large urban environments based on the footprint data of buildings. In *SM '03: Proc. of the ACM symposium on Solid modeling and applications*, ACM Press, NY, USA, 346–351.

LINTERMANN, B., AND DEUSSEN, O. 1999. Interactive modeling of plants. *IEEE CG Appl. 19*, 1, 56–65.

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

MÜLLER, P., ZENG, G., WONKA, P., AND GOOL, L. V. 2007. Image-based procedural modeling of facades. *ACM Trans. Graph. 26*, 3, 85, 1–9.

PARISH, Y., AND MÜLLER, P. 2001. Procedural modeling of cities. In *SIGGRAPH '01: Proc. of the 28th annual conference on CG and interactive techniques*, ACM Press, NY, USA, 301–308.

PIAZZALUNGA, U., AND FITZHORN, P. 1998. Note on a three-dimensional shape grammar interpreter. *Environment and Planning B: Planning and Design 25*, 1, 11–30.

PRUSINKIEWICZ, P., M.J., AND MÊCH, R. 1994. Synthetic topiary. In *SIGGRAPH '94: Proc. of the 21st annual conference on CG and interactive techniques*, ACM Press, NY, USA, 351–358.

PRUSINKIEWICZ, P., MÜNDERMANN, L., KARWOWSKI, R., AND LANE, B. 2001. The use of positional information in the modeling of plants. In *SIGGRAPH '01: Proc. of the 28th annual conference on CG and interactive techniques*, ACM Press, NY, USA, 289–300.

SHAPIRO, V. 2002. *Handbook of Computer Aided Geometric Design*. Elsevier, ch. 20: Solid modeling, 473–518.

STINY, G., AND GIPS, J. 1972. Shape grammars and the generative specification of painting and sculpture. *Inf. Proc. 71*, 1460–1465.

STINY, G. 1980. Introduction to shape and shape grammars. *Environment and Planning B: Planning and Design 7*, 3, 343–351.

STINY, G. 1982. Spatial relations and grammars. *Environment and Planning B: Planning and Design 9*, 1, 113–114.

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