

# Node-Based Shape Grammar Representation and Editing

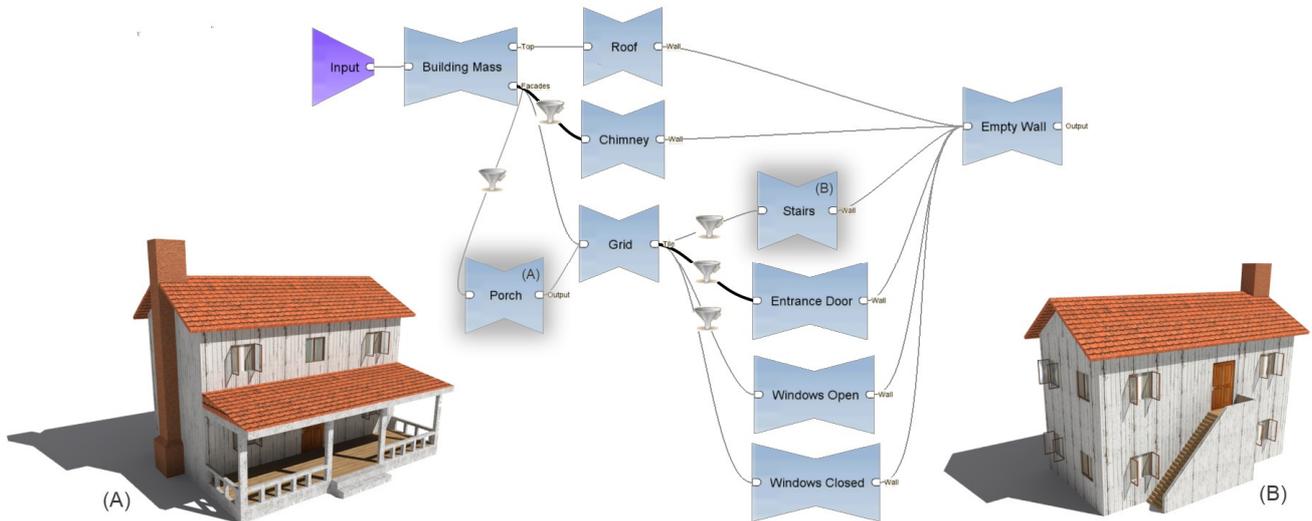
Pedro Brandão Silva  
FEUP/INESC  
ESRI R&D  
TU Delft

pedro.brandao.silva@fe.up.pt

Pascal Müller  
ESRI R&D  
pascal.mueller@esri.com

Rafael Bidarra  
TU Delft  
r.bidarra@tudelft.nl

António Coelho  
FEUP/INESC  
acoelho@fe.up.pt



**Figure 1: A semantic node-based graph featuring the combination of high-level architectural concepts. This representation maps to shape grammars, which in turn generate the displayed models. These two share almost the same graph specification with only slight differences (highlighted nodes and edges). The ‘Porch’ node exists only in the graph of model A, while the ‘Stairs’ only in that of B. Additional local variations are achieved through filters on the edges, e.g. causing every other window to be opened, and the locations of the chimney and entrance door to be changed on the façade.**

## ABSTRACT

Mass content creation is nowadays one of the most important challenges for game artists. This paper presents a high-level architectural modeling solution that combines the full generative power of shape grammars with the ease of use and flexibility of a node-based visual language. Our approach comprises a shape data flow character and introduces some novel features, including recursion, parametric flow, and flow filtering. The main development model consists of encapsulating basic operations into semantically-rich, reusable components that can be more easily assembled using filters. Eventually, this enables users to concentrate on the more intuitive and interactive development layers, while the text-based grammar rules are automatically generated.

## Categories and Subject Descriptors

I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Modeling packages; F.4.2 [Mathematical Logic and Formal Languages]: Grammars and Other Rewriting Systems—

Grammar types (e.g., context-free, context-sensitive);

## General Terms

Design, Languages

## Keywords

shape grammars, node-based design, semantics

## 1. INTRODUCTION

The use of procedural methods in digital games is becoming an increasingly attractive solution for the mass generation of three-dimensional content [1]. Shape grammars are a popular example of procedural modeling effectiveness in the scope of architectural modeling. Using production rules, one can specify a set of geometric replacement processes that progressively increase the detail of a geometric shape, from a basic to a more complex form. This grammar specification can be parameterized and enriched with stochastic techniques, so that it can be used to produce a large variety of designs. When building extensive virtual urban environments, manual approaches tend to be too time-consuming and expensive. Shape grammars, on the other hand, are ideal to handle such extensive, but mostly pattern-repetitive nature.

However, one of the greatest features of shape grammars is also one of their greatest inconveniences: the procedural description of a model is usually done in textual form, as in a sequence of programming instructions, rather than with explicit geometry. This makes it not only unattractive, but also very hard to grasp

and manipulate by artists without a programming or scripting background.

The most common difficulty derives from its decomposition into rules, which makes the procedural process flow hard to understand, maintain and edit. Rules refer to other rules through their named symbols, typically related to the underlying process or entity. Still, not always can rules be named after something semantically relevant, making rule symbols harder to manage and to understand by other users. Ideally, one would rather build, reuse and share shape grammar snippets, yet integrating them requires proper rule symbol association and the resolution of name clashes if no namespace solutions exist.

Parameter passing can become an additional burden, further complicating rule management. Within a rule body, geometric operations are handled with a set of overloaded functions, whose syntax, working and usage is hardly intuitive. Finally, testing and debugging a grammar is also challenging, given its hard control over the sequence of operations scattered throughout the complicated rule chaining.

In recent years, various content creation approaches have adopted visual node-based paradigms. With that in mind, we introduce a node-based visual approach to provide a more intuitive interface for shape grammar generation processes, extending graph design concepts from previous works [2–4]. With the same goal, we address the mentioned shape grammar issues by introducing the following contributions:

- We introduce a comprehensive node-based representation for state-of-the-art shape grammar rules that combines the attribute data flow with the shape data flow within cyclic graphs, thereby enabling 3D artists to easily understand and edit complex rule sets.
- For the intuitive authoring of procedural models, we propose a novel visual editing approach that focuses on the encapsulation of shape grammar snippets into reusable, semantically-rich components that can later be easily combined through flow filters and constraints.

This paper is structured as follows: after reviewing some related work, Section 3 will introduce our visual node-based representation of shape grammars. Section 4 will describe the editing mechanics for that representation, while the Section 5 will refer to our implementation, together with some results. Lastly, the conclusions and some future work will be discussed.

## 2. RELATED WORK

The application of shape grammars as a formal approach to architectural design has been first brought up by Stiny and Gips [5], [6]. Building on that concept, Wonka et al. introduced split grammars [7], a new type of parametric set grammar, supported by an attribute matching system steered by a control grammar, offering the flexibility required to model buildings with many different styles. Based on this work, Müller et al. developed the CGA Shape [8], a shape grammar which procedurally generates variations of a model starting from volumetric shapes, and proceeding to the generation of detail consistent with the mass model. The implementation of the CGA Shape is now integrated in the commercial system CityEngine [9]. As a generalization of previous shape object definitions, Krecklau et al. [10] introduced abstract structure templates to support the use of multiple types of non-terminal symbols with domain-specific operators and attributes. By using Python as a language base, they were able to apply their modeling language  $G^2$  to architectural as well as plant

modeling, having expanded it later for the modeling of interconnected structures, such as bridges [11].

The main drawback of these methods lies on their textual, rule-based nature: they allow only indirect control over the final model by editing the underlying text-based grammar, or global control by changing some parameters. An attractive approach to solve this problem consists of using a visual editor as a front-end to automatically generate grammar production rules. Lipp [3] presented a visual editing system which introduced traditional modeling techniques, allowing a more intuitive and powerful control over each grammar aspect. However, they still relied on a rule structure design for editing and execution. Also, despite having introduced instance locators for custom control, the approach is not flexible enough to introduce new types of semantic selectors. Krecklau et al. [4] presented 3D manipulators in order to create a visual representation of editable parameters directly in model space, revealing the influence of a parameter. They also introduced two edition modes: a Professional mode, to create and configure encapsulated modules (called high-level primitives) and a High-Level mode, where these can be combined and customized using an interactive modeling interface. A limitation, though, is that the P-Mode requires manual editing of a textual-based grammar.

Visual programming languages allow the manipulation of program elements graphically rather than textually. Some of its variants represent nodes as operations and edges as conductors of data tokens. Following this paradigm, visual node-based editors have become a standard for a variety of purposes and systems, including material editors (e.g. Autodesk Maya [12]), texture editors (Allegorithmic Substance Designer [13]) script editors (e.g. Autodesk Softimage [14]) and model animation editors (SideFX Houdini [15]). The work of Patow [2] presented a dataflow adaptation for shape grammars using directed acyclic graphs (DAGs), bridging the disassociation between rules. Despite its advantageous possibility of model verification and graph rewriting, the acyclic approach precludes the use of recursion. The paper also mentions the ability to create components, but provides little detail on how the operation nodes are handled or on how the parametric flow can be controlled.

## 3. NODE-BASED REPRESENTATION

In this section we will describe how the shape grammar definition is represented by a visual, node-based workflow, affecting the way that the shapes and their associated attributes and parameters are controlled throughout the graph. We also discuss how the operations are mapped to nodes and how their various configurations can be achieved.

Shape grammar-based procedural methods, such as the ones presented in [7], [8], [10], describe a scene or model as a set of production rules. These follow the form

$$\text{Predecessor} \rightarrow \text{Successor}$$

where the predecessor is a non-terminal symbol to be replaced successively by a set of other non-terminal or terminal symbols (the successor of the rule). The nature of manipulated symbols per method, from geometric constructs enclosing a local cuboid scope to more complex class instances [10]. In short, these approaches perform the successive creation or transformation of geometry by means of operations. Rules are chained by matching non-terminal symbols mentioned in the successor of one rule to that symbol in the predecessor of another rule. Rules may accept parameters and be restricted to the evaluation of a logical expression in order for the rule to be applied.

For exemplification, the used grammar notation throughout this paper loosely follows the one presented by Müller et al. [8] and CityEngine [9], but could be easily adapted for other shape grammar derivations as well. The adaptation of these concepts to a visual paradigm will be explained in the following subsections.

### 3.1 Graph Structure

In our visual representation, the procedural model is a directed cyclic graph  $G(V,E)$ , where the nodes  $V$  represent the modeling process and the edges  $E$  describe the data flow from one node to the next. Connections are established between input and output ports that, being anchored to nodes, characterize the *process interface*. The easy linking and unlinking of nodes provides a more flexible chaining of processes than the standard textual rule paradigm.

We distinguish several types of graph nodes, each describing either flow-operational essentials or geometric operations. Their type is identified by both shape and color. Shape defines node class and characterizes its general behavior within the graph, while the color is used for differentiating existing categories within a class (see Figure 2).

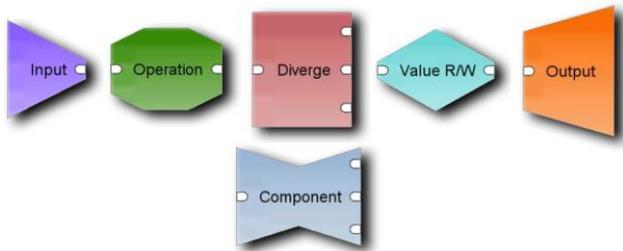


Figure 2: Available node classes

Here, we identify the following classes:

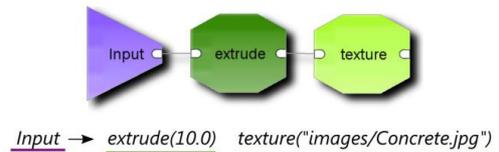
- **Input Node:** the starting point of every graph. As such, it is the location where global graph properties can be defined, such as starting shape, tags or other metadata. Input nodes have no input ports and only one output port. Each graph must have exactly one and only one input node.
- **Operation Node:** the most common node, featuring multiple categories, according to the type of performed operation, such as geometry manipulation, coloring or texturing. Each type of operation features its own distinct node configuration possibilities. Operation nodes have only one input node and one output node, meaning that they take one single shape as input at a time, perform some kind of transformation on it and then return also a single shape.
- **Diverge Node:** takes one single shape as input, but, contrary to the operation node, can output multiple ones. Functionally, it performs a transformation, selection or partition, resulting in various shapes, which can be handled in different ways. These nodes have one input port and at least one output port. Shape splits are examples of operations fitting this class.
- **Value Read/Write Node:** takes one single shape as input at a time. Reads or writes some attribute data associated to it and then forwards the shape to the output port.
- **Output Node:** defines a data stream exit. The use of output nodes is optional, serving mainly as a means to label and identify the resulting leaf shapes (see section 3.5).

- **Component Node:** encapsulates a full graph as a single complex operation. Components are discussed in more detail in section 3.5.

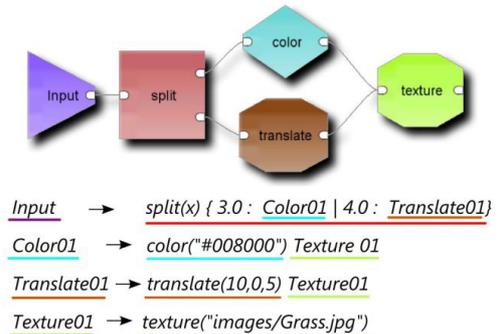
### 3.2 Geometric Data Flow

Nodes are handlers of shape data: in general, nodes receive shapes as input, perform a transformation, analysis or filter and output one or more resulting shapes. Two nodes can be connected by drawing an edge from an output port of one node to the input port of the other (see Figure 3a). When several nodes are chained through this approach, a *shape data flow* is defined. For each resulting shape, we can trace back the sequence of nodes and ports that led to its creation. This is called the *development path* of a shape.

(a) Operation Sequencing



(b) Path Divergence and Path Convergence



(c) Shape Copy

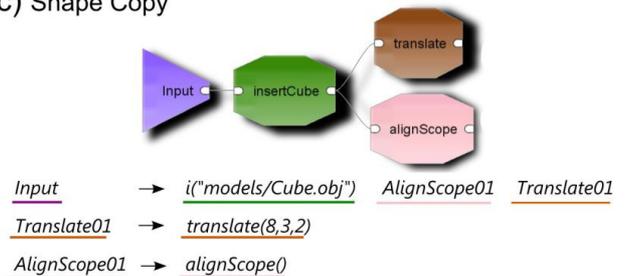


Figure 3: Examples of visual graphs and their translation to shape grammar rules.

Each node can have only one input port, but accepts multiple incoming edges, allowing the reuse of a development path (path convergence – see Figure 3b). On the other hand, some nodes may contain multiple output ports, as a means to allow the separation of development paths (path divergence – see Figure 3b). Also, output ports can also be the source of multiple outgoing edges, each producing a shape copy (see Figure 3c).

Sometimes, one may need to subject a shape to the same procedural paths more than once, following a *recursive* scheme. It is therefore possible to create cycles within the graph, connecting

an output port of a child node to the input port of a parent node (see Figure 4). Naturally, such recursive paths must provide at least one possible exit condition so as to avoid infinite loops. Such cases can be easily found and avoided using graph-transversal algorithms.

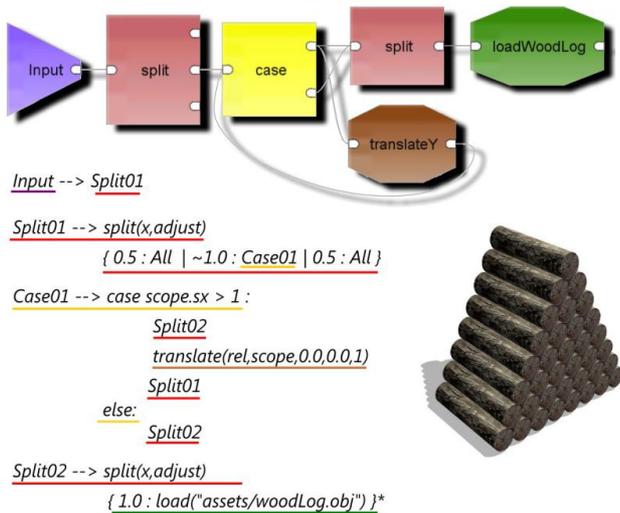


Figure 4 - Recursive example for creating a stacked pile of wooden logs.

### 3.3 Parametric Data Flow

In addition to the geometric data flow, our visual paradigm represents as well the parametric data flow across the graph nodes, which is fundamental to steer the generation processes. In textual shape grammars, this has been achieved either through attribute declarations, which exist on a global scope, or through parameter passing from one rule to the next rule. While this difference is obvious and necessary for programmers, it is not always easy to grasp by designers with little programming background. With this in mind, the two concepts have here been merged into a simple global attribute manipulation.

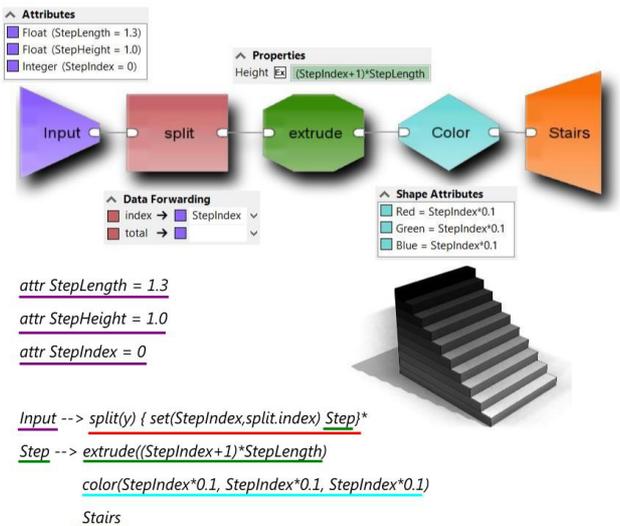


Figure 5: Parametric flow: for each shape, the *StepIndex* variable is updated in the split node and used afterwards to create the sized step, colored with a gray shade.

In our visual representation, global attributes are defined right on the input node, which, being the starting point in every graph, makes them accessible to all following nodes. Attributes can be read and written anywhere in the course of a development path, yet this change is only propagated to its descendant shapes (i.e. each shape has its own copy of the attributes, which is passed on to derived shapes). This procedure follows the functional programming paradigm (where the function arguments are always passed by value) but without requiring an explicit declaration.

Attribute data can be explicitly changed through specific nodes, e.g. a node that explicitly assigns a new value. In addition, some nodes can provide useful data related to the performed operation (e.g. in a split, the total number and the index of each resulting shape). This data may be used by assigning it to a global attribute, a process which is called *data forwarding* (see Figure 5). When shapes “flow” through such nodes, new data is assigned, prevailing throughout the rest of the development path, unless it is expressly changed again. Doing so reduces the need to deal with local attributes or explicit references to nodes.

### 3.4 Node Configuration

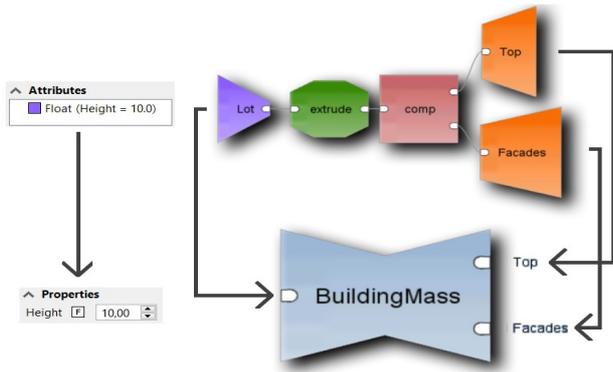
Nodes perform the actual shape operations. In textual grammars, this is achieved by using functions and their arguments. Since each operation may work in different ways, functions may be overloaded, allowing for multiple combinations of arguments. Getting to know and use such overloads, however, constitutes another difficulty, especially for those with less programming background. In order to address this, our visual representation uses the concept of node *properties*: strongly-typed fields, which can be explicitly changed, but that assume certain values by default. This has become a rather common approach in visual applications, due to its intuitiveness and ease of use. Properties may be deactivated or grouped into distinct property sets, called node *styles*. For instance, an operation for roof creation may incorporate various types of parameters, depending on the type of roof, justifying the definition of styles such as “shed”, “hip”, “gable”, etc. Properties are usually assigned fixed values, but support mathematical expressions and function calls that operate on basic parametric data as well as global attributes (see Figure 5).

In addition to their own properties, some nodes may incorporate an enumeration of configurable elements, here generically called *parts*. In a split operation node, for example, a part would correspond to a slice, whose size can be configured. In a conditional check node, a part would correspond to a single condition in a case-else control structure. In some cases, the part creation is linked to a part creation, since each part leads to a distinct development path.

### 3.5 Encapsulating Graphs into Components

Once assembled, full graphs can be encapsulated into *components* to be used as nodes in other graphs. The input node and output nodes of the source graph are mapped to the input and output ports of the component node, respectively, defining its interface (see Figure 6). Graph attributes of the component become its configurable properties, while certain selected attributes become data forwarding fields. In short, a component becomes another operation that provides a more elaborate procedure.

Output nodes are especially important to identify shapes meant to be further derived. If no output nodes are defined, the resulting shapes coming from empty output ports (ports without outgoing edges) will still be present in the final model, but simply assigned an “All” symbol, and cannot be distinguished.



**Figure 6: Mapping of a graph to a component node**

Creating components consists of encapsulating shape grammar snippets. When they are imported to new graphs, an important step consists of merging their rules, connecting the intended grammar rules while avoiding name clashes. For that reason, imported components must be given a unique id within the containing graph, in order to create a namespace and overcome such conflicts.

After being conceived and employed as a component, its underlying graph may again be subject of change. This fact introduces a management difficulty, since greater changes on an interface level may affect the node's connectivity. On the other hand, changes may have unexpected effects in already finalized models. To help addressing these issues, we have devised two import methods:

- **Import by Copy:** The underlying grammar logic of the component node is copied to the graph. Subsequent changes on the original component graph are not refreshed. This mode is ideal to avoid unexpected changes on the underlying model.
- **Import by Reference:** A reference to the original graph is kept, and the component node is refactored whenever underlying changes occur, possibly breaking connectivity within the new graph, if the interface is changed.

## 4. VISUAL SEMANTIC EDITING

We introduce a new editing methodology consisting of the combination of semantically-rich component nodes. After introducing the concept of filters, we provide insight on how they can be used to enhance component nodes. Finally, we explain how they can be used to create building models using a vocabulary that is more familiar to designers.

### 4.1 Flow Filters

The graph-based approach is based on the paradigm shape and parametric data flow. For this, we introduce *filters*, which are flow control elements that block certain shapes while letting others through. Filters perform based on the features of the passing shapes, which are simply called *shape properties SP*. They may refer not only to their directly contained attributes, but also to the result of an expression or function on the shape. A filter consists of a combination of logical statements, called *criteria CR*, joined using conjunctive (“and”) and disjunctive (“or”) logical operators. When the criteria hold, the filter decides positively upon the shape flow and negatively otherwise. Each criterion consists of a check on a shape property, through a specific operator or function, hereby called the *check function CF*.

A criterion can be formally described as:

$$CR = CF (SP, CP_1, CP_2...CP_n)$$

where  $CP_n$  refers to the parameters of the check function.

Examples of criteria would be:

$$\mathbf{scope.tx} > 3.5 \quad (1)$$

$$\mathbf{isInRange}(\mathbf{height}, 20,50) \quad (2)$$

$$\mathbf{getFacingDirection}() == \text{“North”} \quad (3)$$

$$\mathbf{isConcave}() \quad (4)$$

The elements stressed in bold in the above conditions are the shape properties, which are read from the shape, at the time of the criteria verification. Properties are verified through either functions (2) or operators (1,3) that take further criterion parameters. If the property is itself a Boolean (4), no check function is required.

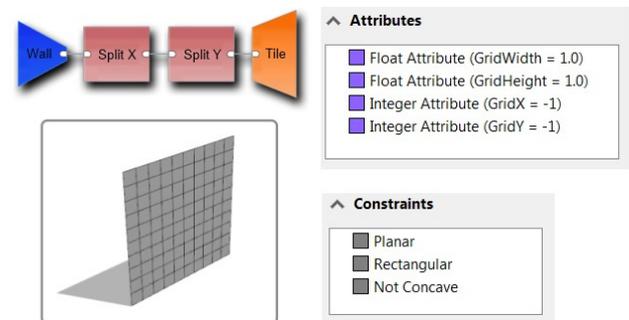
One of the most fundamental features behind the construction of these criteria is the decoupled composition of their members: Shape Property, Check Function and Criterion Parameters. Each of these can encompass a virtually unlimited amount of variants, which are meant to be easily extensible by the programmer, the user or the system itself.

## 4.2 Designing Semantic Components

While components can be used to simply encapsulate small routines without specific meaning, they can also be used to build architecturally significant elements, such as building parts, façade styles or windows. The way these elements are meant to be combined, however, does follow certain architectural design standards. Embedding component nodes with information about their meaning, usage and typical restrictions can contribute to the creation of a high-level vocabulary for the design of building models.

Semantic information is present in components first of all in their external interface, comprising the expected kind of input shape, the returned output shapes and the global attributes. The type of expected input shapes is defined in the input node. Likewise, output nodes identify the type of shapes that the graph produces. Global attributes are, as mentioned, defined within the input node and can be marked for data forwarding. For each of these elements, using appropriate nomenclature is essential to guarantee proper understanding and use of the component.

Figure 7 shows a basic component that splits a façade into a grid, providing X and Y index information as data forwarding attributes of the resulting tiles.



**Figure 7: Component Creation and Configuration**

Data flow on a graph can also be configured with *input constrains*, which are flow filters that impose restrictions on the input shapes. Examples of useful constraint types on actual component instances include:

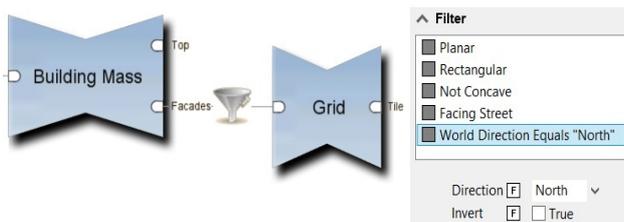
- **Direction**, e.g. to restrict windows from facing west;
- **Occlusion**, e.g. to avoid placing a balcony on a façade occluded by other;
- **Number**, e.g. to avoid building more than one main entry door;
- **Geometry**, e.g. to ensure the construction of certain structures on planar or rectangular faces (see Figure 7);
- **Size**, e.g. to avoid unrealistic constructions with mismatching proportions.

Inputs, outputs and constraints help specify how components are intended to be used, but should not restrict it. For example, although in most cases windows are meant to be built on planar walls, it would not be wise to impose such a regulation, since it might need to be overridden e.g. on a fictional building or when addressing a more modern architectural style. Their application must therefore provide the necessary flexibility for the users to employ the components as they see fit.

### 4.3 Semantic Flow Control

We introduce a new graph creation paradigm for the combination of the presented semantic components into *semantic graphs*. Considering that each node and port has a certain architectural meaning, the whole graph is a connection of *semantic concepts*, instead of mere geometric operations. This way, even untrained users can read, understand and assemble such graphs without specific knowledge of shape grammars and geometric derivation.

Semantic graph editing can be aided through flow filters to introduce restrictions on concept connections. Filters can be placed on the edges, thus establishing conditions on the shape flow. When two semantic components are connected, existing constraints of the destination node are integrated with the edge filter (see Figure 8).

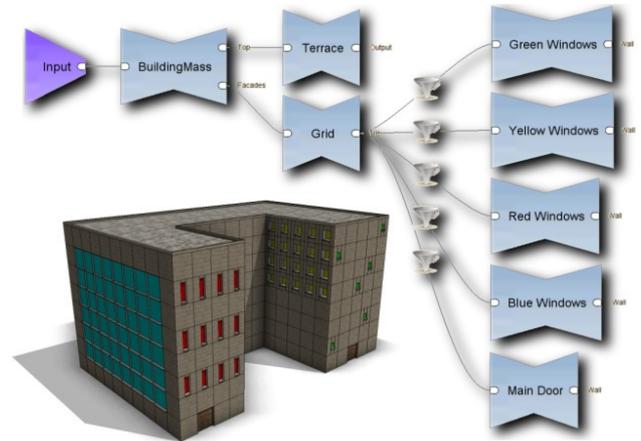


**Figure 8: Flow filter on an edge, featuring several criteria. The first 3 criteria correspond to the constraints defined on the grid component (see Figure 6).**

The purpose of this filter approach is to provide a simple and straightforward method for introducing local changes. The work by Patow [2] presented a similar level of control using exception nodes. Our solution is more flexible and compact.

Figure 9 shows a semantic graph and its corresponding model outcome: one component node creates the main building block, another builds a terrace on the top face and a last one creates a grid-like structure on the façades. From this last component, 4

different window styles and 2 doors are introduced according to certain patterns.



**Figure 9: Semantic Assembly of components, featuring distinct applications of flow filters**

As mentioned before, filters are sets of conditional criteria, based on shape properties, such as direction, geometry, size, occlusion state, etc. It is possible to introduce new shape properties in the following ways:

- By defining global attributes in the input node of the semantic graph;
- By using components that feature data forwarding attributes.

Using a database of check functions, it is possible to generate matching criteria. In the example above, all windows have been placed according to the (x,y) coordinates of the façade grid (see Figure 7), which was a shape property provided by the grid component itself. The generated criteria are listed in Figure 10.

| Categories       | Items  |
|------------------|--|
| *Connector*      | GridX(GridLayouter) Equals ...               |
| Component Data   | GridX(GridLayouter) Is Multiple of ...       |
| Custom           | GridX(GridLayouter) Is Greater Or Equals ... |
| Direction        | GridX(GridLayouter) Is Lesser Or Equals ...  |
| Geometric        | GridY(GridLayouter) Equals ...               |
| Global Attribute | GridY(GridLayouter) Is Multiple of ...       |
| Indexing         | GridY(GridLayouter) Is Greater Or Equals ... |
| Pivot            | GridY(GridLayouter) Is Lesser Or Equals ...  |
| Scope            |  |

**Figure 10: Criteria generation for grid coordinate properties.**

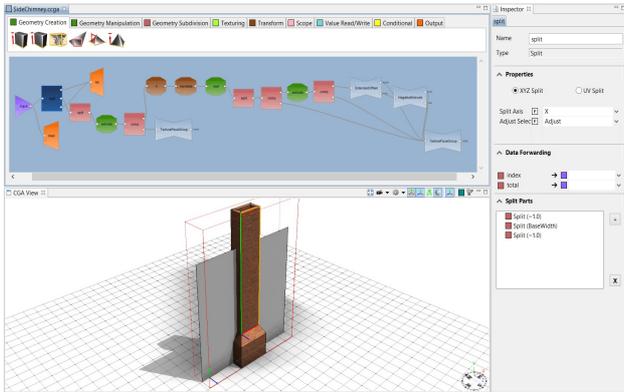
This level of control based on a grid structure is comparable to that of Lipp et al. [3] through *semantic locators*, which allow the control over individual columns, rows or particular coordinates. Our criteria play a similar role, but are more flexible, since they are generated from shape properties. This enhances the level of local control extensibility of this semantic assembling solution.

## 5. IMPLEMENTATION AND RESULTS

In order to test the applicability and usability of such a visual programming definition, we have developed an editor that follows the mentioned methodological approach to address the generative power of the CGA Shape Grammar [8]. This choice was motivated not only by the grammar's popularity in the field, but also by the availability of the authoring tool *CityEngine* [9]. The editor has been implemented in Java using the Eclipse RCP, a rich client platform for general purpose applications.

## 5.1 Graphical User Interface

The graphical user interface (GUI) is split into multiple dockable windows, as shown in Figure 11.



**Figure 11: Graphical user interface of the visual editor**

The graph editor area (see Figure 11, top) contains the necessary tools for the construction of graphs. Adding nodes to a graph is either achieved by dragging-and-dropping from a toolbar located on the top or through a mouse context menu. Connections between the nodes are built by drawing edges between ports.

The Inspector pane (see Figure 11, right) provides information concerning the currently selected node or edge, as well as the necessary controls (e.g. sliders, textboxes) to alter its properties.

Finally, the 3D Rendering view (see Figure 11, bottom) displays the geometry model of the current graph. Among the usual viewport rendering options, it allows the visualization of the shape scopes, pivots and bounding boxes, which are of extreme relevance to the modeling control, but often a source of frustration by many users.

## 5.2 Editing Graphs

Our graph design canvas allows the combination of CGA Shape operations. Node labels reflect the operation or component name by default, but can be renamed to improve readability without affecting the chaining of the underlying rule symbols. Input nodes

allow the choice of the starting shape (e.g. horizontal and vertical rectangles, concave shapes, volumes), which is used to create the preview model in the 3D rendering area.

To allow easy debugging, edges can be temporarily deactivated by the user, interrupting the data flow but without actually breaking existing graph connections. This option is extremely useful when a user wants to focus on a specific path, while temporarily hiding the remaining ones. Similarly, nodes can also be disabled, which is useful to understand the consequence of certain operations.

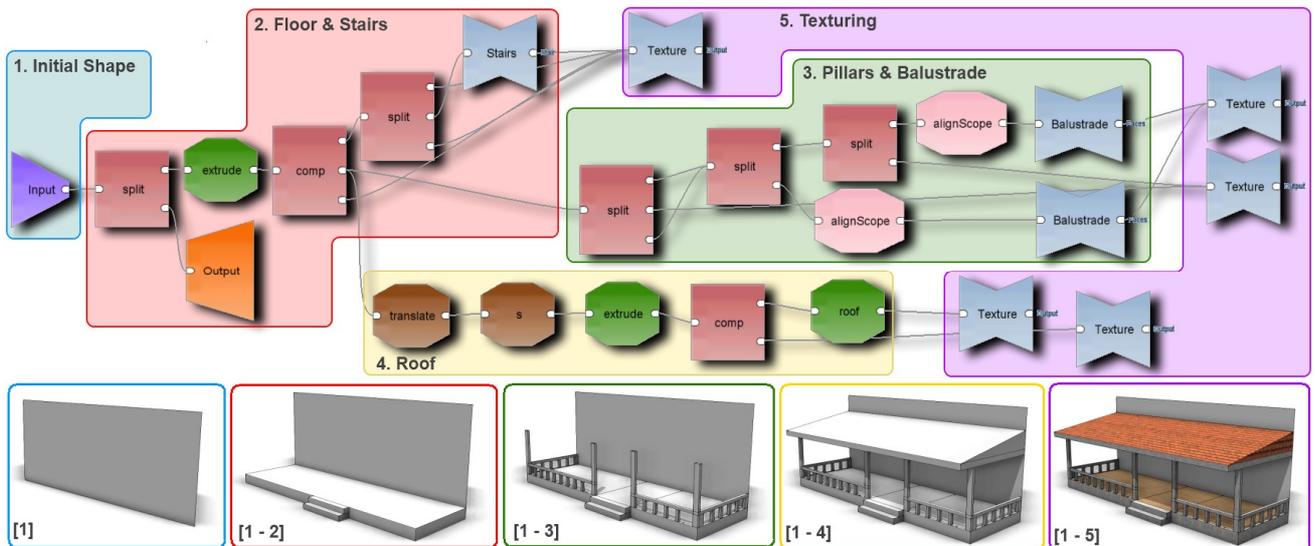
We show in Figure 12 the stepwise creation of a porch component suitable to be applied on a façade shape. The graph features further component nodes that refer either to previously created subroutines ('Texture') or to more complex architectural elements ('Stairs', 'Balustrade'). The various steps show the impact of each graph section, which was achieved through the addition of a small number of nodes and connections. In practice, this example corresponds to over 200 lines of CGA rules scattered among various files. In contrast, our representation clearly introduces far more readability, maintainability and usability than its textual counterpart.

Figure 1 and 9 depict simpler semantic graphs created using our editor. The use of filters allows for a complete model overhaul with only a small number of changes and facilitates the creation of multiple building variations with very little effort.

We introduced our visual editors to artists with a CGA Shape Grammar background. Their reaction was very positive: they were already familiar with the node-based paradigm for procedural design and appreciated its adaptation to the shape grammar specificities. They became especially interested in the possibility of creating semantic components. However, when assembling them, they pointed out that it was not always easy to choose the right filter on a first try and suggested using an interactive picking system that would find the filters based on the user selection on the model.

## 5.3 Exporting and Importing CGA Shape

Besides the obvious increase in user productivity, the automatic generation of the grammar rules from the node representation is



**Figure 12: Construction of the porch component featured in Figure 1. The colored overlays indicate how each graph section contributes to the respective model states shown on the bottom. The corresponding CGA spans over 200 lines of grammar rules.**

beneficial in the sense that it can be tweaked in order to produce better structured and more optimized rules. The code generation process starts by building the header of the file and writing the global attributes defined in the input node. From there, the algorithm recursively analyzes the chaining of nodes, declaring new rules for each of the following cases: (i) if it is an input node; (ii) for each port of a divergence node; (iii) for each node port featuring shape copy; (iv) if the input port of a node has several incoming edges, i.e. for recursion and path convergence cases; (v) when referring and joining component nodes. If a node does not fit any of these cases, it is simply chained within the leading rule.

Visual CGA files are stored in their own format and the CGA are created on demand. Older CGA files can be translated to this interactive paradigm, too. Importing requires a first parsing step to identify the rule symbols, attributes, parameters and operations. Afterwards, operations within rules are turned into nodes and the connections between them are extracted from rule references and from the operation sequencing within rules. Existing rule parameters are replaced with global attributes. By default, a single component is created from the CGA file, although it is also possible to create multiple components from selected subsets of rules.

## 6. CONCLUSIONS AND FUTURE WORK

We presented a visual node-based methodology to address the numerous shape grammar challenges that arise from its textual representation. On a representation level, we extended existing geometric flow approaches [2] with the possibility to define recursive procedures. We addressed the parameter handling with a simple concept of attribute data flow and demonstrated how function overloads can be mapped to node properties. On an editing level, we proposed an intuitive authoring technique that focuses on the encapsulation of grammar snippets into reusable components. When enhanced with semantic data, they provide a high level architectural vocabulary that can be combined using filters, yielding a much more flexible approach than the creation of semantic locators [3].

Future work will focus on further developing semantic graph manipulation and expressive power. Our current approach still requires users to search and edit filters directly, but this could be assisted through interactive techniques to find filters based on an explicit selection on the model. This is a first of many steps towards the integration of manual editing and procedural modeling techniques [16]. We believe that node-based systems are instrumental to harness the power of procedural modeling methods and that properly supporting both semantics and manual editing will be essential to bring those methods to mainstream content production workflow.

## ACKNOWLEDGMENTS

We would like to thank all our colleagues at the Esri R&D for their precious advices, background of node-based systems, implementation support and testing feedback.

This work is partially supported by the Portuguese government, through the National Foundation for Science and Technology - FCT (Fundação para a Ciência e a Tecnologia) and the European Union (COMPETE, QREN and FEDER) through the project PTDC/EIA- EIA/114868/2009 entitled "ERAS - Expeditious Reconstruction of Virtual Cultural Heritage Sites - PTDC/EIA- EIA/114868/2009" and through the Ph.D. Scholarship SFRH/BD/73607/2010.

## REFERENCES

- [1] R. M. Smelik, T. Tutenel, K. J. de Kraker, and R. Bidarra, "A declarative approach to procedural modeling of virtual worlds," *Computers & Graphics*, vol. 35, no. 2, pp. 352–363, Apr. 2011.
- [2] G. Patow, "User-Friendly Graph Editing for Procedural Modeling of Buildings," *IEEE Computer Graphics and Applications*, no. April, 2012.
- [3] M. Lipp, P. Wonka, and M. Wimmer, "Interactive visual editing of grammars for procedural architecture," *ACM Transactions on Graphics*, vol. 27, no. 3, p. 1, Aug. 2008.
- [4] L. Krecklau and L. Kobbelt, "Interactive modeling by procedural high-level primitives," *Computers & Graphics*, vol. 36, no. 5, pp. 376–386, Aug. 2012.
- [5] G. Stiny, "Introduction to shape and shape grammars," *Environment and Planning B*, vol. 7, no. 3, pp. 343–351, 1980.
- [6] G. Stiny and J. Gips, "Shape Grammars and the Generative Specification of Painting and Sculpture," in *Information Processing '71*, 1972, pp. 1460–1465.
- [7] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky, "Instant architecture," in *ACM SIGGRAPH 2003 Papers*, 2003, vol. 22, no. 3, pp. 669–677.
- [8] P. Müller, P. Wonka, S. Haegler, A. Ulmer, L. Van Gool, and L. Van Gool, *Procedural Modeling of Buildings*, vol. 25, no. 3. Boston, Massachusetts: ACM, 2006, pp. 614–623.
- [9] Esri, "Esri CityEngine | 3D Modelling Software for Urban Environments," 2013. [Online]. Available: <http://www.esri.com/software/cityengine/>. [Accessed: 21-Feb-2013].
- [10] L. Krecklau, D. Pavic, and L. Kobbelt, "Generalized Use of Non-Terminal Symbols for Procedural Modeling," *Computer Graphics Forum*, vol. 29, no. 8, pp. 2291–2303, Dec. 2010.
- [11] L. Krecklau and L. Kobbelt, "Procedural Modeling of Interconnected Structures," *Computer Graphics Forum*, vol. 30, no. 2, pp. 335–344, Apr. 2011.
- [12] Autodesk Inc., "Autodesk Maya," 2013. [Online]. Available: <http://usa.autodesk.com/maya/>. [Accessed: 21-Feb-2013].
- [13] Allegorithmic, "Allegorithmic Substance Designer," 2012. [Online]. Available: <http://www.allegorithmic.com/products/substance-designer>. [Accessed: 21-Feb-2013].
- [14] Autodesk Inc., "Autodesk Softimage," 2013. [Online]. Available: <http://autodesk.com/softimage>. [Accessed: 21-Feb-2013].
- [15] Side Effects Software, "Houdini," 2013. [Online]. Available: <http://www.sidefx.com/>. [Accessed: 21-Feb-2013].
- [16] R. Smelik, T. Tutenel, K. J. de Kraker, and R. Bidarra, "Integrating procedural generation and manual editing of virtual worlds," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games - PCG '10*, 2010, pp. 1–8.