CEIG - Spanish Computer Graphics Conference (2012)
Isabel Navazo and Gustavo Patow (Editors)

Visual Language Generalization for Procedural Modeling of Buildings

Santiago Barroso and Gustavo Patow

ViRVIG-UdG
Universitat de Girona, Spain

Figure 1: *Some models created with our system: Left: roman bridge. Middle: hermit. Right: hermit inside view.*

Abstract

Procedural modeling has become the accepted standard for the creation of detailed large scenes, in particular urban landscapes. With the introduction of visual languages there has been a huge leap forward in terms of usability, but there is still need of more sophisticated tools to simplify the development process. In this paper we present extensions to the visual modeling of procedural buildings, which adapt concepts from general purpose programming languages, with the objective of providing higher descriptive power. In particular, we present the concepts of visual modules, parameter linking and the possibility to seamlessly add abstract parameter templates to the designer visual toolbox. We base our demonstrations on a new visual language created for volume-based models like historic architectonic structures (aqueducts, churches, cathedrals, etc.), which cannot be modeled as 2D facades because of the intrinsic volumetric structure of these construction (e.g. vaults or arches).

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Computer Graphics—Languages I.3.5 [Computer Graphics]: Computer Graphics—Computational Geometry and Object Modeling I.3.5 [Computer Graphics]: Computer Graphics—Three-Dimensional Graphics and Realism

1. Introduction

The creation of new content always had a great importance in Computer Graphics' applications, and this is becoming even more important as hardware is getting more powerful for handling complex geometry and real time rendering. Now, traditional tools like Autodesk's Maya [Gui05], 3DStudio or Blender, which had been the tool of choice for professional artists up to now, are becoming impractical tools when it comes to model massive highly detailed scenes. Even more, any modification often requires a considerable time, independently of how small it is (e.g. changing the type of windows used in a facade). For these reasons, procedural modeling has emerged as a well established approach for handling these highly complex scenes.

The current trend in procedural building modeling is to use grammar-based techniques that have shown promising results, as shown by Wonka et al. [WWSR03] and by Müller et al. [MWH*06]. Later, Lipp et al. [LWW08] introduced an interactive visual editing paradigm for shape grammars. However, although they realized it was difficult for an artist to see the effects that a rule produced, they still relied on separate workspaces for rule editing and rule execution. Re-

© The Eurographics Association 2012.

DOI: 10.2312/LocalChapterEvents/CEIG/CEIG12/057-066

delivered by
EG EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

cently, these tools have been enhanced with the introduction of non-terminal symbols [KPK10], but this solution still remained inside the text-based paradigm. More recently, Esri's CityEngine [Esr12], Epic Games' UDK [Epi12] and Patow [Pat12] simultaneously introduced very similar visual languages for these shape grammars, where each node is a command and the connection between two nodes represents the flux of geometry between them. However, these visual tools lack the possibility to define higher-level abstraction tools, which is important for large-scale productions.

In this paper we intend to introduce tools to generate and generalize visual languages to simplify the creator's work. We start by presenting a volume-based visual language on which we base our developments, followed by the introduction of the definition of modules, non-terminal symbols, parameters, and links between them within a visual modeling language. As this was already implemented elsewhere [BD04], we do not claim any contribution in this sense, we just offer a new *interpretation* of modules and parameters as functions and its parameters in a programming language sense. So, we can enumerate our contributions in the following four main points:

- A visual language for volume-based architecture, on which we base successive developments (Section 3).
- Higher-level abstractions based on the volume-based visual language (Section 4).
- A mechanism to introduce abstract parameter templates in a visual modeling language (Section 5).
- High-level post-processing tools, which further expand the modeling possibilities (Section 6).

All these contributions enable the final user to achieve not only higher-level abstractions which greatly enhance the artist's pipeline, but also simplify their work. Also, the learning curve of our extensions is smoothly blended into traditional visual modeling approaches, becoming a natural extension of existing techniques. Even non-expert users are now able to generate nontrivial structures and new modules within seconds thereby quickly creating complex scenes.

2. Previous Work

As mentioned, the current trend follows the basic principles of a shape grammar, presented by Müller et al [MWH*06]. The main concept of a shape grammar is based on a rulebase: starting from an initial axiom shape (e.g. a building outline), rules are iteratively applied, replacing shapes with other shapes. A rule has a labelled 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.

predecessor →
  SuccessorCommandA;
  SuccessorCommandB;

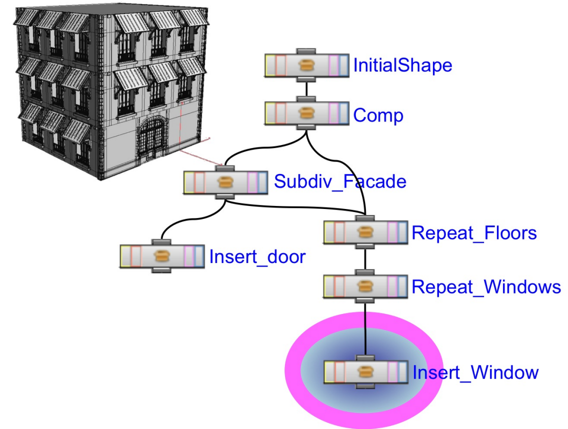

Figure 2: *A visual representation of a simple rulebase. Observe that the rules themselves form a graph.*

Traditionally, four commands were introduced in [MWH*06]: *Split* of the current shape into multiple shapes, *Repeat* of one shape multiple times, *Component Split* (called *Comp*) creating new shapes on components (e.g. faces or edges) of the current shape and *Insert* of premade assets replacing a current predecessor (e.g. traditional triangle-based models). Every rule application creates a new configuration of shapes. Traditionally, during a rule application, a hierarchy of shapes is generated corresponding to a particular instance created by the grammar, by inserting a rule successor shapes as children of the rule predecessor shape [MWH*06] [LWW08]. This production process is executed until only terminal shapes are left. An example rulebase is visualized in Figure 2.

Kelly and Wonka [KW11] proposed a generalization of the classic concept of an extrusion where a set of profiles are used to extrude a floor-plan to generate whole building structures, plus an anchoring system to locate assets like doors, windows or other elements. Simultaneously, Talton et al. [TLL*11] proposed a Markov Chain Monte Carlo system to generate the variations on a given ruleset to achieve a desired target. Their applications included all king of procedural modeling techniques, from plants and trees (L-systems) to buildings (CGA Shapes) to whole cities (random blocks). Lin et al. [LCOZ*11] presented an algorithm for interactive structure-preserving retargeting of irregular 3D architecture models, taking into account their semantics and expected geometric interrelations such as alignments and adjacency. The algorithm performs automatic replication and scaling of these elements while preserving their structures by decomposing the input model into a set of sequences, each of which is a 1D structure that is relatively straightforward to retarget. As the sequences are retargeted in turn, they progressively constrain the retargeting of the remaining sequences. Musialski et al. [MWW12] proposed a novel

© The Eurographics Association 2012.

interactive framework for modeling building facades from images, exploiting partial symmetries across the facade at any level of detail. Their workflow mixes manual interaction with automatic splitting and grouping operations based on unsupervised cluster analysis. Ceylan et al. [CML*12] presented a framework for image-based 3D reconstruction of urban buildings based on symmetry priors: Starting from image-level edges, they generate a sparse and approximate set of consistent 3D lines, which are then used to simultaneously detect symmetric line arrangements while refining the estimated 3D model.

Krecklau et al. [KPK10] introduced a new language (called G^2 for Generalized Grammar) which adapts concepts from general purpose programming languages in order to provide more descriptive power to designers. However, the same authors later recognize [KK12] that their procedural modeling workflow better compares to a programmer writing a piece of code rather than an artist creating a 3D scene. One of the most recent works in the field, done by Krecklau and Kobbelt [KK12], introduces a set of tools to simplify the intuitive and interactive manipulation complex procedural grammars allowing non-programmers to easily modify and combine existing procedural models. However, the creation of these easy-to-use modules still requires high-level programming skills, mainly because its text-based paradigm.

Visual editing was introduced later, in Esri's City Engine [Esr12] and Epic's UDK [Epi12], and independently described by Patow [Pat12], and emerged as a powerful alternative to avoid manually editing rulesets, which is tedious and error-prone. Actually, these works build on the seminal work by [Hae88], and extensions have been thought before for plants [LD98, LD99], or shaders [AW90], and can be classified as *purely visual languages* [BD04].

Cutler et al. [CDM*02] resented a procedural approach to authoring layered, solid models. They defined the internal structure of a volume from one or more input meshes, and sculpting and simulation operators were applied within to shape and modify the model. Their key contribution was a concise procedural approach for seamlessly building and modifying complex solid geometry. On the other hand, Whiting et al. [WOD09] introduced a procedural approach to adjust the constructive parameter of a masonry building to achieve structural soundness. Their approach, based on an optimization loop, was based on a text-based language to describe the input volumes, followed by a simulation step to adjust the parameters that the user defined for the process (e.g. wall thickness, column width, ...). Our approach builds on these approaches by defining a visual language for the volume-modeling operations plus further mechanisms to provide higher expressiveness to the end-user.

3. A visual language for volume-based architecture

In this section we will introduce our volume-based visual language intended for the recreation of historic architectonic structures like stone bridges and churches. To create our structures, we use procedural modeling methods as described by Müller et al. [MWH*06]. Beginning with a coarse volumetric model, production rules iteratively refine the geometry with internal structure and facade details. The procedural system carries semantic information including architectural labels (e.g. arch, wall, column) and rule parameters (e.g. column diameter). However, as mentioned, a difference in our approach to traditional procedural geometry is our use of mass modeling. Müller et al. [MWH*06] consider the building volume as a single solid object with no interior. In contrast, like Whiting et al. [WOD09] did, we model solid objects as interior columns, walls, and other supporting structures. However, we give a step further by introducing a visual language for volume-based procedural modeling to simplify user creation and editing tasks. Therefore, the above mentioned language will be the visual representation of a language for procedural modeling of volumetric buildings, consisting of a set of rules with clearly defined syntax.

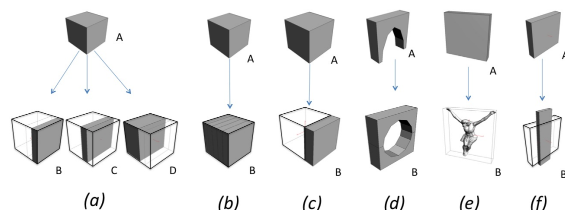

Figure 3: *Operations for volumes. (a) SplitVol (b) RepeatVol (c) ReplaceFaces (d) Symmetric (e) InsertVol (f) TransVol*

Here we provide an easy to manipulate visual equivalent to the commands introduced by Whiting et al. [WOD09]. Visual languages are not new at all in Computer Graphics, as they were used before for plants [LD98, LD99], and Esri's CityEngine [Esr12] and Epic Games' UDK [Epi12] use them to represent the ruleset for building design. This results in a streaming visual node-editing paradigm for shape grammars, in which each node is an operation applied on its incoming geometry stream. A connection wire routes the output from a node to the input of another downstream node. In our implementation, each volume carries its own local coordinate system and scale, called *scope*, which is identical to the one defined by Müller et al. [MWH*06].

For the sake of brevity, we will follow a notation similar to the classic one [MWH*06]:

$$label \rightsquigarrow command(parms)\{productLabels\}$$

These are:

SplitVol This is an operator that is the volumetric counterpart of the two-dimensional *Split* command defined above. See Figure 3(a). Its syntax is:

© The Eurographics Association 2012.

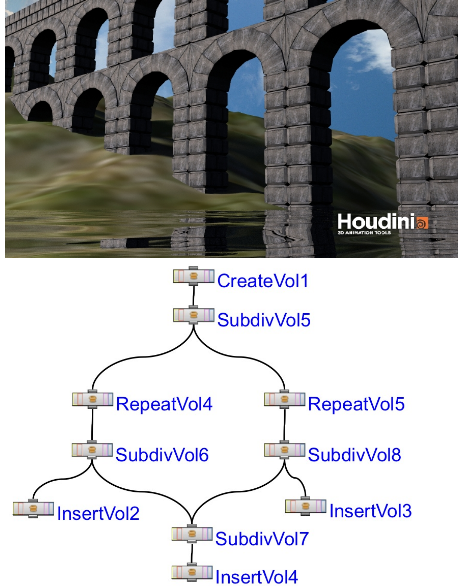

Figure 4: *Aqueduct example. Top: Final model. Bottom: the graph of operations needed to build the aqueduct.*

$$A \rightsquigarrow SplitVol("Z", 0.2, 0.2, 0.6)\{B|C|D\}$$

RepeatVol This is an operator that is the equivalent to the two-dimensional *Repeat* command defined above. See Figure 3(b). Its syntax is:

$$A \rightsquigarrow RepeatVol("X", 0.2)\{B\}$$

ReplaceFaces This is an operator that has no 2D equivalent. It is used to replace (some) faces of the input geometry by new volumes to be later processed. It needs the ID of the face to be replaced and the depth of the new volume. See Figure 3(c). Its syntax is:

$$A \rightsquigarrow ReplaceFaces(1, 0.3)\{B\}$$

Symmetric This is an operator that has no 2D equivalent in standard implementations. Basically, it duplicates and mirrors geometry. The user can specify the distance of the mirror plane along its normal. See Figure 3(d). Its syntax is:

$$A \rightsquigarrow Symmetric("Y")\{B\}$$

InsertVol This operation replaces an input geometry by the geometry provided as parameter. The new geometry will be scaled and translated so its bounding box coincides with the input geometry scope. See Figure 3(e). Its syntax is:

$$A \rightsquigarrow InsertVol("asset.obj")\{B\}$$

TransVol This operation allows to translate, rotate and scale a given pice of geometry. See Figure 3(f). Its syntax is:

$$A \rightsquigarrow TransVol(TransformationMatrix)\{B\}$$

Note that all directional nodes like *SplitVol*, *RepeatVol*, and *Symmetric*, a direction must be specified, which can be X, Y or Z.

Figure 4 shows the Aqueduct example. The corresponding network, also at the figure, consists of only 10 nodes to model the whole structure. The time needed to create this structure from scratch is less than 5 minutes. More examples of the possibilities of this volumetric primitives, and its utilization in conjunction of the other features presented in this paper can be seen in Figure 1.

At Figure 5 we can see the Sainte Chapelle example, which is a Gothic temple located at the Îlle de la Cité, at the center of Paris, France. In this example we have reused the *windows* node previously created, but with the parameters corresponding to the Gothic windows in the temple. More examples can be found elsewhere [BP12].

4. Visual languages, Encapsulation & Parameters

Formally speaking, a spatial arrangement of icons that constitutes a visual sentence is a two-dimensional counterpart of a one dimensional arrangement of tokens in conventional (textual) programming languages, so we can safely establish a correspondence between the tools of traditional text-based languages and the ones of visual-based languages [BD04]. In formalizing visual programming languages, it is customary to distinguish process icons from object icons. The former express computations; the latter identify primitive objects in the language or a spatial arrangement of the elementary object icons. In the languages developed for procedural modeling [LD98, AW90, Pat12], object icons are largely replaced by further operations that create them, resorting only to the use of process icons.

As already mentioned, the original idea of generating higher-level modules was introduce to procedural modeling by Krecklau et al. [KPK10]. Here we extend that idea to visual languages and we seamlessly blend it with a visual-paradigm pipeline [Pat12]. For that, we also use the concept of a *module*. Whenever someone has a part of a network that could be re-used within the pipeline, a module can be created that encapsulates that subnetwork. By reducing several operators into a single new operator, the number of repetitive operations is strongly reduced. Actually, this mechanism is *exactly* the same mechanism of procedure/function definition in programming languages, but here integrated into our visual development environment. Modules defined this way are new custom operator types built from node networks. The author encapsulates the network in a module, and then *promotes* parameters from nodes inside the module up as parameters of the module itself. This way, the functionality of the network is encapsulated in the module, with all the well known re-usability benefits. Observe that module definition is slightly more general than the one defined by Krecklau

© The Eurographics Association 2012.

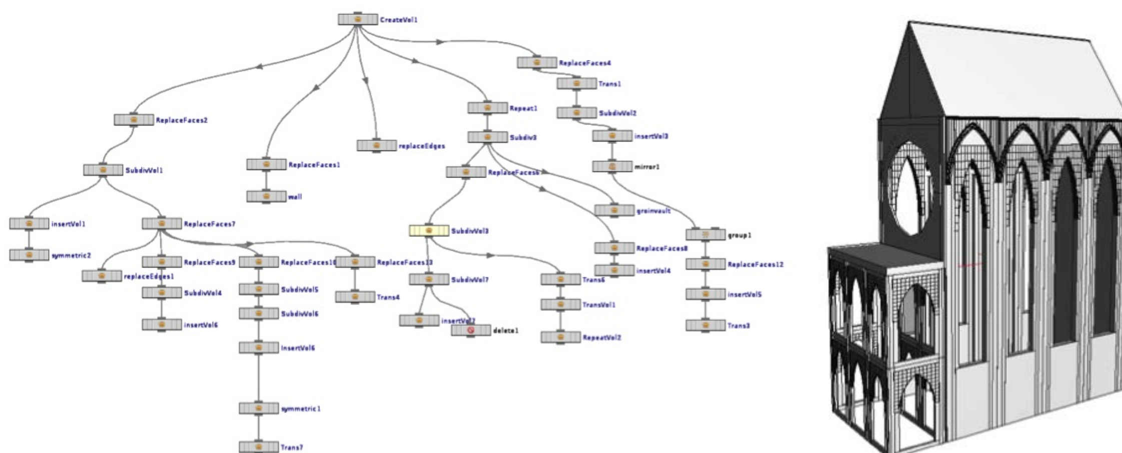

Figure 5: *Sainte Chapelle model. Based on a model courtesy of Emily Whiting.*

et al. [KPK10] mainly because it does not need a distinction between non-terminal symbols and terminal ones: all are treated exactly the same way.

In order to define a new module, the author must create the module contents (the network), define its interface, and store it in the form of a library. In case of need of a more complex functionality, the user is free to add custom scripts that are called either as callback functions when a parameter changes, or when an event external to the module occurs (e.g. a change in the module input geometry). Parameters are only declared in the scope of a module, and thus we avoid dependencies and keep our visual grammar both simple and clear. Each parameter has a unique name within its node, and have predefined types (a float, a 3-component vector, a reference to another node, etc.). At module definition time the user must specify whether a given node parameter is linked to the user interface or not. When parameters are linked this way, they implement a simple direct copy of the values entered in the user interface, but the user is free to, later on, edit the expression stored at each internal node parameter to introduce more complex operations. See Section 4.1.

To simplify subsequent modeling operations by simplifying graph layout and reuse, we created a module to encapsulate the process of repeatedly creating windows of a certain style in a wall. This module is simply called "*windows*". In Figure 6-top we can see its internal implementation, plus its user interface, which is offered to the user/artist. Observe that the user interface provides a number of constructive parameters, as well as references to the window type to be used. Its implementation, shown in Figure 6-bottom, simply repeatedly subdivides the wall in equally sized chunks, and each one is subdivided to generate the space where the actual geometry is created. With the help of our volume library, this meant the use of only 19 operations (nodes).

At Figure 7 we can see the hermit example, where our volumetric library plus our module abstraction mechanism allowed to model the whole building, with only 11 nodes. In the figure observe that the *windows* and *windows1* nodes are both instances of the *windows* module. We have set the *widowType* parameter of the *window* node to the romanesque style, but it can be easily changed with an abstract parameter template, see Section 5. The creation of the whole model takes a few minutes, provided that the *window* non-terminal node was already created.

4.1. Inter-Parameter Relationships

Here we further expand the concept of parameter linking by introducing a mechanism to allow to establish dependencies between parameters. In order to do that, we re-use the concept of the *observer* pattern [GHJV95], in such a way that a parameter, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any value changes. The essence of this mechanism is to define a one-to-many dependency between parameters so that when one object changes its value, all its dependents are notified and updated automatically. Parameter dependencies are basically implemented following this scheme: A parameter registers itself to the observed dependents list by using some explicit function calls. In particular, any reference inside a parameters definition to another parameter automatically triggers this behavior. As an example, the Aqueduct in Figure 4 has the height of its floors linked such that, if there are two floors, the top one is 30% of the bottom one, if there are three the relation is 50%, 25% and 25% (smaller on top), and if there are more, the total height is equally distributed among the floors.

© The Eurographics Association 2012.

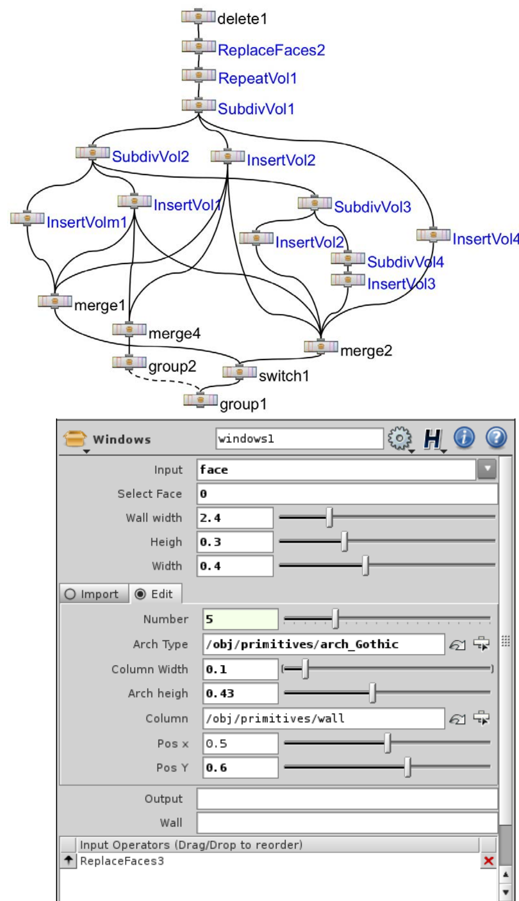

Figure 6: *The* windows *module implementation (top) and its user interface (bottom).*

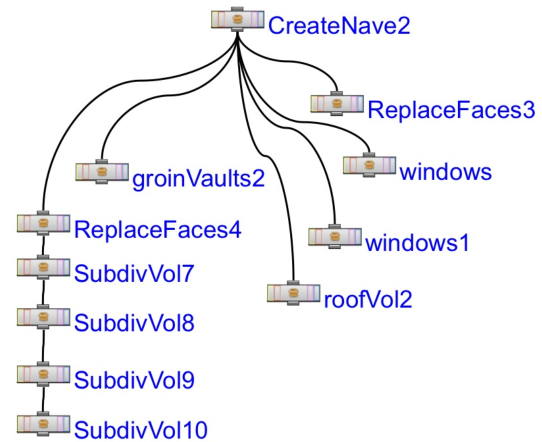

Figure 7: *A hermit and its corresponding graph. Top: the operation's graph. Middle: Inside view. Bottom: Outside view. Left: models before post-processing. Right: models after post-processing*

5. Abstract Parameter Templates

Modules are a good choice for enclosed, dynamic, procedurally generated objects in the scene, but the concept can be further extended by allowing non-terminal symbols as parameters. Basically, this usage of the modules is equivalent as passing a function as parameter to another one, as done in traditional programming languages. This creates abstract structure templates which can be reused at several places in the grammar. Krecklau et al. [KPK10] used the term abstract since the evaluation of such templates does not yield a standalone object. The user has to pass non-terminal symbols for further evaluation. For instance, we can create a module that just creates the grid tiling ignoring of what will be generated within the tiles. Even more, we extended the ideas in that paper by introducing the possibility to have parameters in the modules used as non-terminal symbols.

As a proof of concept, we have re-implemented the architectural examples provided by Krecklau et al. [KPK10] (`http://www.graphics.rwth-aachen.de/index.php?id=314`). That code defines a simple basic facade (See Figure 8-Top) consisting of a first, two middle and a last floor. The floors have ledges as separators, and each ledge corner is defined with a free from deformation algorithm. In general, the facade has 4 entry points where the user can select different methods for the ledge and ledge corner color, and for the windows and the building door. In Figure 8-Top we can see the initial building where all four entries are just geometry-coloring operations. In Figure 8-Middle we can see the replacement of the door and windows by slightly ameliorated versions. These assets basically use a module called *embed* to generate a frame where the actual window is placed. The module to be executed as content of the *embed* module is passed as parameter. In both cases, door and window, in this example the *embed* module receives simple color module as parameter. Finally, Figure 8-Bottom illustrates the behavior when changing the node passed to the *embed* node by a more sophisticated one, which subdivides the window space in smaller areas so simulate smaller

© The Eurographics Association 2012.

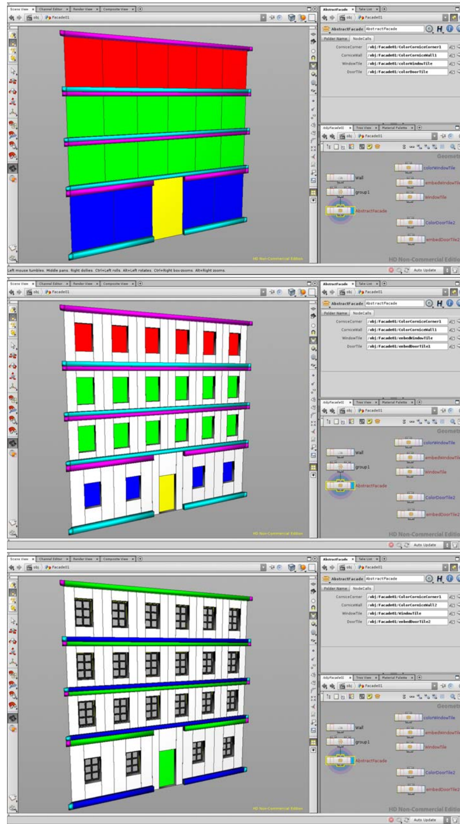

Figure 8: *Generalized grammars applied to a facade design. Top: a simple color-based definition. Middle: the simple color-based assets where replaced by a frame that inside calls, again, the same simple flat color pattern. Bottom: the final model replacing the windows in the middle image by more elaborate procedural assets.*

panels. In this example the nodes where also provided with different coloring modules.

```
1 def changeType():
2     node = current node()
3     allowEditing(node)
4     refNode = node.parm('callNode').eval()
5     type = identify the type of refNode
6     target = locate module's dummy node
7     target.changeNodeType(type)
8     copyParms(refNode, target)
```

Listing 1: *Procedure to change the type inside the* nonTerminalSymbolCall *module.*

As traditional visual modeling languages are not intended for using modules as parameters, only their resulting geometry, we had to use some non-standard techniques to build our prototype. In particular, we developed an abstract node called *nonTerminalSymbolCall* that implements a call to the non terminal symbol passed as parameter. Basically, the module is implemented using the *State* pattern [GHJV95], as this is a clean way for an object to partially change its type at runtime. In our implementation, the module starts encapsulating a simple dummy node which is replaced later on by an instance of the node to be called. this replacement is performed every time the parameter associated with the node to be called from the *nonTerminalSymbolCall* module is changed. Also, to guarantee the node correct behavior, the internal state is verified every time an instance is created or its inputs change. The main procedure in the module is *changeType*, which is shown at Listing 1.

```
1 def copyParms(source, target):
2     for p in source.parms():
3         t = target.parm(p.name())
4         t.set(p.eval())
```

Listing 2: *Procedure to unlock the current module hierarchy for modification.*

In the code, *callNode* is the name of the parameter in the *nonTerminalSymbolCall* module used to provide the name of the sample node to use. Function *node.changeNodeType*(*nodeType*) is a method that instantiates a node of the required type (*nodeType*) and reconnects all inputs and outputs from the original node. Actually, as this method only changes the reference that to the type the node has, this change simply implies a change in its internal structure, so all references to this node need not to be changed. Also, it is important to remark that we have decided to provide a sample node (pointed by variable *refNode* in the code) instead of a module abstract class name because this allowed us to set parameters in the call. As can be seen in Listing 2, this simply implies to take every single parameter in the referenced node and to copy its contents (values) into the target node.

```
1 def allowEditing(node):
2     while node is not root node:
3         node.allowEditingOfContents()
4         node = node.parent()
```

Listing 3: *Procedure to unlock the current module hierarchy for modification.*

To demonstrate the usefulness of this mechanism, we have extended our previous *windows* node with one parameter template: the kind of arch used. First of all, we have defined two non-terminal nodes, called *RomanesqueArch* and *GothicArch*, which are shown in Figure 9-left. The modified *windows* node with this change allows the user to choose between different types of windows (namely, the between the styles defined above), see Figure 9-right. The final result can be appreciated at Figure 10.

© The Eurographics Association 2012.

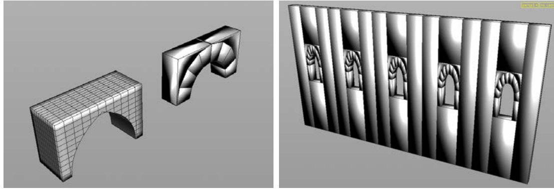

Figure 9: *The two* arch *modules (left) and their usage in the* windows *node (right).*

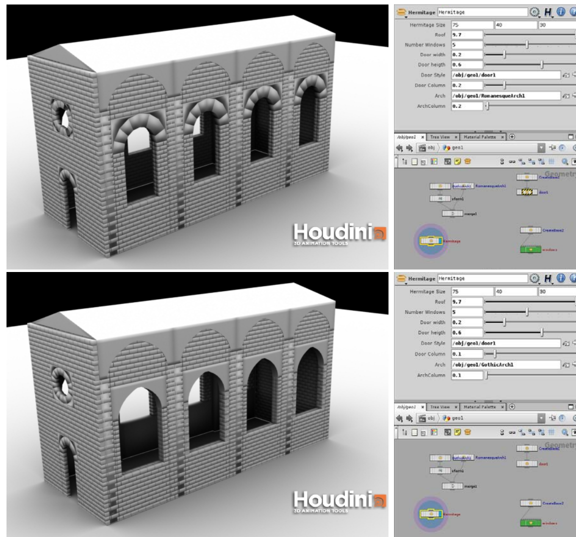

Figure 10: *The* arch *modules used to change the style of the hermit windows. Top: Romanesque. Bottom: Gothic.*

6. Post-processing primitives

In this section we will present two post-processing operators we have created which are seamlessly integrated into our volume-based pipeline and our visual modeling approach. The first of these operations are the transformation of each volumetric primitive into its brick-paved equivalent. Once we model structures as an assemblage of rigid blocks, we are able to analyze the force distributions at the interfaces between adjacent elements and use this for different physical simulations. This step is needed for any further simulation of the building soundness. This was also done by Whiting et al. [WOD09] as an intermediate step before the physical simulation to guide the optimization of the building constructive parameters.

Bricks: Once we have the volumetric-based structure, modeled using the volume-oriented visual language presented in Section 3, we can use a specific node called *toBrick* to transform the input volumes into an assemble or rigid blocks (bricks). This node allows the user to choose the tag of the input geometry to apply the operation to, the size of each

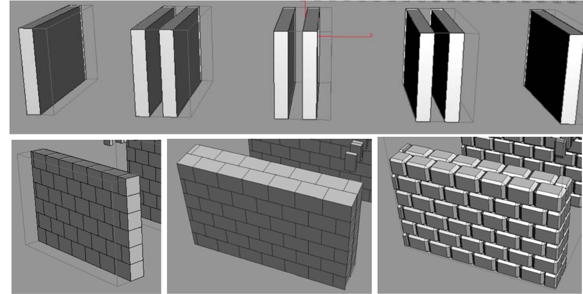

Figure 11: *The* toBrick *operator. top: Its basic construction principles.*

block and the tag to assign to the resulting geometry. This last parameter is needed if the user intends to further process the model.

This module proceeds by first subdividing each input volume into two by the axis of its thinest dimension. This is done because this will be the one that will hold the walls that we build in subsequent steps. See Figure 11-top. Our algorithm iterates over the number of vertical rows of blocks, from bottom to top, shifting each block by half its width. The rationale behind this shift is to guarantee a stronger structure than the one that would result with perfectly aligned blocks. See Figure 11-bottom-left. After this process, the walls have the structure shown in Figure 11-bottom-middle. Finally, we apply an aesthetic rounding to each individual block, with the objective of increasing the realism of the reconstruction. See Figure 11-bottom-right.

Physical Simulation: Once we have subdivided our volume-based model, we can proceed to simulate its structural soundness as done by Whiting et al. [WOD09] work, or simply to simulate its destruction with time (Figure 1) or impacts (Figure 12). This can be done by plugging any simulation engine, like Nvidia's PhysX (`http://developer.nvidia.com/physx`), Bullet (`http://bulletphysics.org/`), Open Dynamics Engine (`http://www.ode.org/`), or even Houdini's own dynamic system. See Figure 13. Please, note that this can be done with traditional tools, too, but here we would like to show the flexibility associated with a volume-based representation. We do not claim a contribution in this respect.

Simulation of physical interaction can also be observed at the right images at Figure 7, where the building was post-processed with a simulation that produces building degradation. The resulting model was rendered in a dusty environment to complete the ambiance of the scene.

7. Implementation

Our system is implemented over the publicly available **building**Engine system, which is part of the **skyline**Engine

© The Eurographics Association 2012.

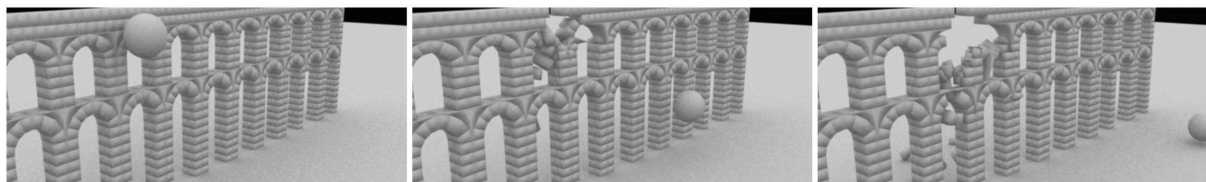

Figure 13: *Physical impact of a large ball onto the Aqueduct model seen in Figure 4.*

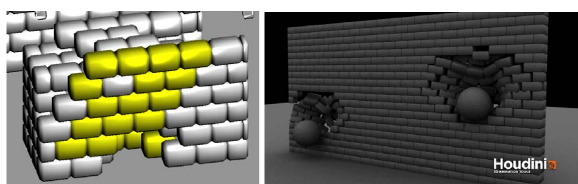

Figure 12: *Once the volumes are replaced by an assemblage of rigid blocks, we can select some for an artist-controlled destruction process, or for a simple impact simulation.*

[RP10] system, which runs on top of SideFX's Houdini [Sid12]. Now, we will present some implementation details of our system. The interested reader can download the full source code for our demos and libraries from http://ggg.udg.edu/skylineEngine/.

Modules: For the implementation, we have used the *Houdini Digital Assets* (HDA). A digital asset defines a node type and exists inside an *Object Template Library* (OTL) file. The node type is implemented in terms of other nodes wired together inside a node called a *subnet*. These nodes inside the subnet are called the definition's contents. A module's algorithm is determined by the nodes inside it. To edit those nodes the user has to create an instance of the digital asset, unlock it, modify the contents, and save the definition. New digital asset instances are normally locked, meaning that they are read-only, and they automatically update when the asset's definition changes. An unlocked instance is editable, does not update when the definition changes, and you can save its contents to change the definition [Sid12].

Parameter dependencies: We have used Houdini's own linking mechanism through so called *channels*. Houdini offers several ways to define these links, but in the end they all sum up to an explicit reference to a node's parameter and its current value. Parameters are implemented as instances of the *Parameter* class, and this basic procedure is reflected by a call like $parm(path).eval()$, where $parm()$ is a command that returns the parameter referenced by the $path$, followed by an evaluation request. Observe that in Houdini parameters are strongly typed, so the types of the observed parameters and the types of the dependent parameters should coincide.

8. Discussion and Future Work

We have introduced a set of visual tools that will help current production pipelines based on visual procedural languages. These tools include a volume-based visual language, and the visual counterparts for modules, non terminal symbols, parameter linking, and abstract parameter templates. Our extensions blend smoothly into traditional visual modeling approaches, becoming a natural extension of existing techniques. As mentioned, all these contributions enable the final user to achieve not only higher-level abstractions which greatly enhance the artist's pipeline, but also simplify their work. We believe that even non-expert users are now able to generate nontrivial structures and new modules within seconds thereby quickly creating complex scenes, but a thorough user study is left as future work. Finally, once designers have finished, the resulting models can be incorporated into any application, like 3D GIS or other content-production pipelines.

Our abstract template prototype is not free of drawbacks, though. In particular, Houdini is not intended for this sort of dynamic node-changing manipulations, which forced us to find some workarounds for them. For instance, an HDA can be either locked, so changes in the module definition are propagated automatically; or it can be unlocked, so it may have a current implementation that is different from any other instance of the same node. The problem is that the contents of an HDA, when it is locked, cannot be changed nor altered in any way. As a consequence, module instances cannot be locked, as this blocks the *changeNodeType* operation, in fact raising an error message. To allow the change to happen, we need to unlock the HDA instance. But this unlocking should be propagated upwards to any module using this unlocked one. The code in Listing 3 implements this simple self-plus-upwards-unlocking mechanism. We would like to emphasize that this does not represent any problem for end-user development. However, problems appear if one needs to debug something *inside* the module or inside any module this one uses, as changes would be local only to the current node. To propagate changes to all other module instances, one has to *manually* lock back each one, and then they will update their contents next time they are called. This process can be cumbersome for large structures with many nested module calls.

© The Eurographics Association 2012.

Acknowledgments

We would like to thank Krecklau et al. [KPK10] for making publicly available the code for their prototype of the Generalized Grammar G^2. Sainte Chapelle in Figure 5 based on a model created by Emily Whiting. This work was partially funded by grant TIN2010-20590-C02-02 from Ministerio de Ciencia e Innovación, Spain.

References

[AW90] ABRAM G. D., WHITTED T.: Building block shaders. *SIGGRAPH Comput. Graph. 24*, 4 (1990), 283–288. doi:http://doi.acm.org/10.1145/97880.97910. 3, 4

[BD04] BOSHERNITSAN M., DOWNES M. S.: *Visual Programming Languages: a Survey*. Tech. Rep. UCB/CSD-04-1368, EECS Department, University of California, Berkeley, Dec 2004. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2004/6201.html. 2, 3, 4

[BP12] BARROSO S., PATOW G.: Recreación de estructuras arquitectónicas mediante modelaje procedural basado en volúmenes. In *Arqueologica 2.0* (2012), p. in press. 4

[CDM*02] CUTLER B., DORSEY J., MCMILLAN L., MÜLLER M., JAGNOW R.: A procedural approach to authoring solid models. *ACM Trans. Graph. 21*, 3 (July 2002), 302–311. URL: http://doi.acm.org/10.1145/566654.566581, doi:10.1145/566654.566581. 3

[CML*12] CEYLAN D., MITRA N. J., LI H., WEISE T., PAULY M.: Factored facade acquisition using symmetric line arrangements. *Computer Graphics Forum (Proc. EG'12) 31*, 1 (May 2012). 3

[Epi12] EPICGAMES: Unreal development kit (udk), 2012. http://udk.com. 2, 3

[Esr12] ESRI: Cityengine, 2012. http://www.esril.com/software/cityengine/index.html. 2, 3

[GHJV95] GAMMA E., HELM R., JOHNSON R., VLISSIDES J.: *Design Patterns*. Addison-Wesley, Boston, MA, January 1995. 5, 7

[Gui05] GUINDON M.-A.: *Learning Maya: Foundation*. Alias LearningTools, 2005. 1

[Hae88] HAEBERLI P. E.: Conman: a visual programming language for interactive graphics. *SIGGRAPH Comput. Graph. 22*, 4 (1988), 103–111. 3

[KK12] KRECKLAU L., KOBBELT L.: Interactive modeling by procedural high-level primitives. *Computers & Graphics*, 0 (2012), –. URL: http://www.sciencedirect.com/science/article/pii/S0097849312000672?v=s5, doi:10.1016/j.cag.2012.03.028. 3

[KPK10] KRECKLAU L., PAVIC D., KOBBELT L.: Generalized use of non-terminal symbols for procedural modeling. *Comput. Graph. Forum 29*, 8 (2010), 2291–2303. doi:http://dx.doi.org/10.1111/j.1467-8659.2010.01714.x. 2, 3, 4, 5, 6, 10

[KW11] KELLY T., WONKA P.: Interactive architectural modeling with procedural extrusions. *ACM Trans. Graph. 30*, 2 (Apr. 2011), 14:1–14:15. URL: http://doi.acm.org/10.1145/1944846.1944854, doi:10.1145/1944846.1944854. 2

[LCOZ*11] LIN J., COHEN-OR D., ZHANG H., LIANG C., SHARF A., DEUSSEN O., CHEN B.: Structure-preserving retargeting of irregular 3d architecture. *ACM Trans. Graph. 30*, 6 (Dec. 2011), 183:1–183:10. URL: http://doi.acm.org/10.1145/2070781.2024217, doi:10.1145/2070781.2024217. 2

[LD98] LINTERMANN B., DEUSSEN O.: A modelling method and user interface for creating plants. *Comput. Graph. Forum 17*, 1 (1998), 73–82. 3, 4

[LD99] LINTERMANN B., DEUSSEN O.: Interactive modeling of plants. *IEEE Comput. Graph. Appl. 19*, 1 (1999), 56–65. doi:http://dx.doi.org/10.1109/38.736469. 3

[LWW08] LIPP M., WONKA P., WIMMER M.: Interactive visual editing of grammars for procedural architecture. *ACM Transactions on Graphics 27*, 3 (Aug. 2008), 102:1–10. 1, 2

[MWH*06] MÜLLER P., WONKA P., HAEGLER S., ULMER A., VAN GOOL L.: Procedural modeling of buildings. *ACM Trans. Graph. 25*, 3 (2006), 614–623. 1, 2, 3

[MWW12] MUSIALSKI P., WIMMER M., WONKA P.: Interactive Coherence-Based Façade Modeling. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2012) 31*, 2 (May 2012), 661–670. URL: http://www.cg.tuwien.ac.at/~pm/Facade2012/index.html. 2

[Pat12] PATOW G.: User-friendly graph editing for procedural modeling of buildings. *IEEE Computer Graphics and Applications 32* (2012), 66–75. doi:http://doi.ieeecomputersociety.org/10.1109/MCG.2010.104. 2, 3, 4

[RP10] RIDORSA R., PATOW G.: The skylineengine system. In *XX Congreso Español De Informática Gráfica, CEIG2010* (2010), pp. 207–216. 9

[Sid12] SIDEFX: Houdini 12, 2012. http://www.sidefx.com. 9

[TLL*11] TALTON J. O., LOU Y., LESSER S., DUKE J., MĚCH R., KOLTUN V.: Metropolis procedural modeling. *ACM Trans. Graph. 30*, 2 (Apr. 2011), 11:1–11:14. URL: http://doi.acm.org/10.1145/1944846.1944851, doi:10.1145/1944846.1944851. 2

[WOD09] WHITING E., OCHSENDORF J., DURAND F.: Procedural modeling of structurally-sound masonry buildings. *ACM Trans. Graph. 28* (December 2009), 112:1–112:9. URL: http://doi.acm.org/10.1145/1618452.1618458, doi:http://doi.acm.org/10.1145/1618452.1618458. 3, 8

[WWSR03] WONKA P., WIMMER M., SILLION F., RIBARSKY W.: Instant architecture. *ACM Transaction on Graphics 22*, 3 (July 2003), 669–677. Proceedings ACM SIGGRAPH 2003. 1

© The Eurographics Association 2012.