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

Bringing Direct Local Control to Interactive Visual Editing of Procedural Buildings

Adria Riu and Gustavo Patow

ViRVIG-UdG
Universitat de Girona, Spain

Abstract

This paper presents a new system to add direct and persistent local control to the visual editing of rules for procedural buildings, avoiding a combinatorial explosion of grammar rules. In this paper we follow the ideas initially proposed by Lipp and co-workers [LWW08]. For this, we have added a few simple new commands, which are added to the artist-provided ruleset in a way completely transparent to the user. The end-user selects the primitives/assets to modify, and the system automatically incorporates these modifications into the ruleset. This change is performed using graph-rewriting techniques, which are both simple to define and control, but also very powerful and practical for these situations.

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

Content creation remains as one of the most important challenges in computer graphics today. One of the most common approaches is to use 3D modeling tools like Autodesk Maya or 3DS Max. However, this is time consuming tedious and repetitive, but gives the designer full control of the whole process. However, this approach scales badly with the number of assets to model, like in the case of a large virtual city. In that case, the currently accepted solution for modeling of large urban landscapes is the use of procedural techniques [WWSR03, MWH*06]. The recent introduction of a visual paradigm that allows the easy editing of the ruleset [Esr12, Epi12, Pat12] has produced a shift in the current trend towards simpler, yet effective, tools.

However, when the user wants to assign a different texture, different window type or different ornamentation rule to a specific window or facade, problems arise as they would have to write several new rules to identify the floor and column of the window and add the corresponding modifications. Later on, the community developed an obvious interest in providing more direct control over the ruleset and the local changes the user might be interested in. This was the path followed by Lipp et al. [LWW08]. However, this seminal work was still based on tools to edit the text-based rules and to provide control by directly modifying the production tree, which are not reflected in the ruleset itself.

So, the motivation behind this paper is to obtain a system that combines the advantages of the above mentioned systems, namely:

- A simple system for local control in building design which blends the tree-view and the graph-view of the system. See Section 2.1.
- A mechanism that allows to store local modifications both in an independent way, as done by Lipp et al. [LWW08], but also in an seamless way integrated with the ruleset itself.
- The implemented mechanism is able to provide persistency.

With the techniques presented in this paper, designers will be able to avoid the cumbersome and sometimes difficult ruleset modifications that result from the change of a simple asset or parameter. Also, the results can be persistently saved, both as an automatically generated (modified) ruleset and as an external independent file. This way, local modifications are now truly integrated into the expressive power of design grammars, bringing procedural modeling even closer to the artistic workflow.

DOI: 10.2312/LocalChapterEvents/CEIG/CEIG12/067-075

2. Previous Work

The current trend in procedural modeling of buildings 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 rule-base: 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 (tagged) shape on the left hand side, called predecessor, and one or multiple shapes and commands on the right hand side, called successors. Commands are macros creating new shapes or commands.

$$predecessor \rightarrow$$
$$SuccessorCommandA;$$
$$SuccessorCommandB;$$

The resulting geometry will also be assigned new tags with the purpose of being further processed. In our implementation, a given pice of geometry carries all tags that this primitive or any ancestor has received during the production process. 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 pre-made assets replacing a current predecessor. 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 1. Visual editing was introduced later, in Esri's City Engine [Esr12] and Epic's UDK [Epi12], and independently described by Patow [Pat12].

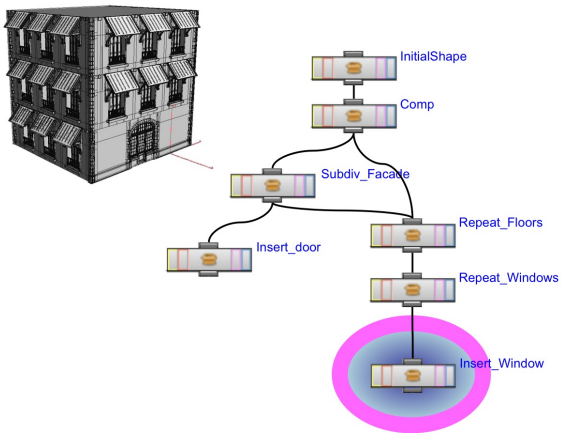


Figure 1: *A visual representation of a simple rule-base. Observe that the rules themselves form a graph.*

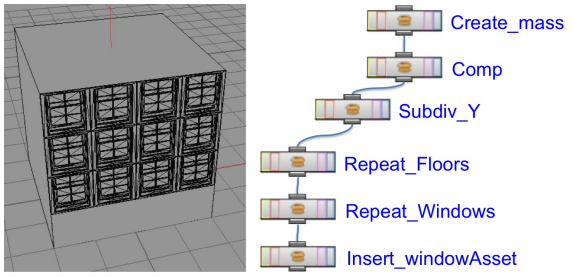
Talton et al. [TLL*11] proposed a Markov Chain Monte Carlo system to generate the variations on a given ruleset to achieve a desired target. Their applications included all king of procedural modeling techniques, from plants and trees (L-systems) to buildings (CGA Shapes) to whole cities (random blocks). 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. Merrell et al. [MSK10] presented a method for automated generation of building layouts for computer graphics applications. In their approach, given a set of high-level requirements, an architectural program is synthesized using a Bayesian network trained on real-world data. The architectural program is realized in a set of floor plans, obtained through stochastic optimization. The floor plans are used to construct a complete three-dimensional building with internal structure. Krecklau and Kobbelt [KK11] presented a system for the easy generation of interconnected structures such as bridges or roller coasters where a functional interaction between rigid and deformable parts of an object is needed. Their approach mainly relies on the top-down decomposition principle of shape grammars to create an arbitrarily complex but well structured layout. None of these works introduce a local control mechanism like we do here.

In their work, Benes et al. [BSMM11] present guided procedural modeling, an approach that allows a high level of top-down control by breaking the system into smaller building blocks that communicate. In their work, the user creates a set of guides that define a region in which a specific procedural model operates. These guides are connected by a set of links that serve for message passing between the procedural models attached to each guide. In this approach, local control is introduced by the building blocks themselves, but further control is left to the the detailed level of procedural systems, thus being completely compatible with the approach proposed here.

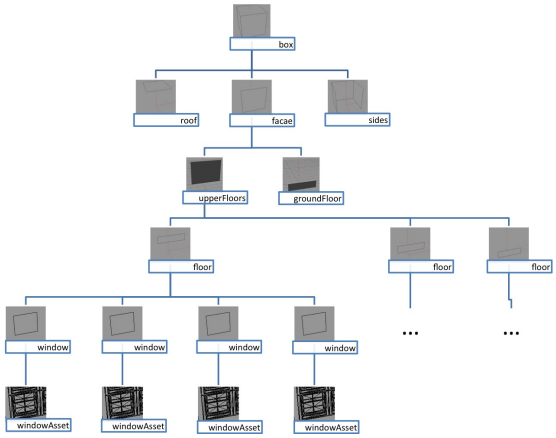
2.1. Direct Local Control, the graph-vision and the tree-vision

As mentioned, traditional implementations [MWH*06] [LWW08] rely on a data structure called configuration of shapes, such that every rule application operates on a configuration and provides as a result a new configuration. In general, this behavior is illustrated as a tree, where each primitive gives as a result a new set of primitives after being processed by an operation [Esr12]. This view of the process actually is of limited help, as it disassociates the primitive tree from the operation that actually produce that tree. even

worse, as the ruleset is expressed as a set of text-based rules, realizing the correspondences between the geometry and the operation that produced it is very difficult. One of the reasons for the introduction of the visual graph-based approach was to break this disassociation and showing in one single framework both approaches [Pat12], as primitives can be seen to "*flow*" from one operation to the next one. Actually, we can informally say that the tree view is the *dual* of the graph-based one.



(a)



(b)

Figure 2: *A very simple building and the two possible visions of its structure: as a graph (top-right) or as a tree (middle). Actually, both visions are complementary, as each node represents an operation that receives some geometry as input and provides the processed geometry as output, as shown in Figure 3.*

A clear example of this paradigm can be seen in Figure 2. In Figure 2(a)-left we can see a trivial example of a simple building, and at its right we can see the simple graph of operations that produced it, while the primitive tree can be seen at Figure 2(b). In the graph view, every operation is represented as a node, while in the tree view every node *is* a primitive. But the most important realization is that both views represent exactly the same process, which is illustrated in Figure 3 for a single node: the *repeat* node called *Repeat_Floors* receives the building upper part, and repeatedly subdivides it into the floors, which are provided as its output.

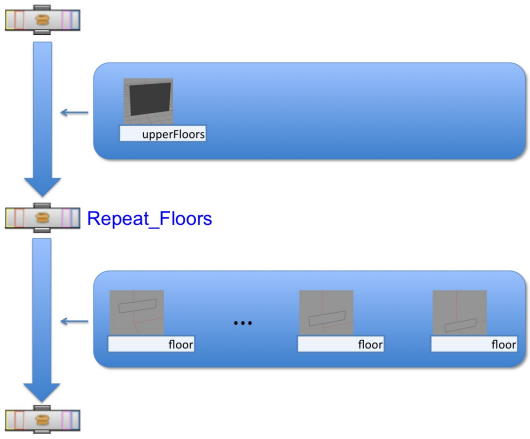


Figure 3: *Each node represents an operation that receives some geometry as input and provides the processed geometry as output.*

For direct modeling, in the visual graph-based paradigm, these two views are well integrated and work harmonically to achieve the desired goal. However, when it comes to exception control, the two views collide and are difficult to integrate.

The only way to locally control any change in traditional approaches [MWH*06] [Esr12] was to edit the ruleset and rewrite it completely, which is not only tedious, but also very error prone and, more important, can lead to a combinatorial explosion of grammar rules for modifications that should not affect all instances. Later, Patow [Pat12] introduced an explicit *Exception* command, while Krecklau and Kobbelt [KK12] added specific rules selecting a given tag with a given primitive index to apply the different change. Both methods basically are instances of the same rough mechanism for adding local control to the grammar, by introducing commands which allowed to add new specific tags to a given element in the productions. This has the clear advantage of integrating local control into the ruleset with a simple and human-readable mechanism. However, the user was required to know exactly the primitive ID number in order to select and label it. The problem with this approach is its lack of persistence under building modifications, as the primitive ID numbers change when any rule executed before the primitive is changed. Also, bear in mind that this also produces a complete change in the tree, rendering useless any precomputation possibility. Another clear drawback is the impossibility of using more human-friendly selection mechanisms, like allowing the user to select a *specific* window or an *specific* floor.

Lipp et al. [LWW08] provided a direct and persistent local control mechanism over the generated instances, avoid-

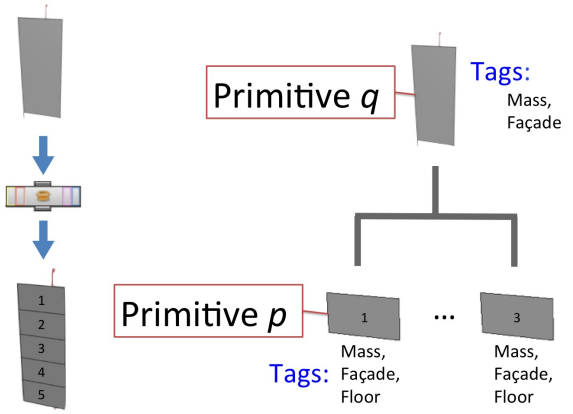


Figure 4: *Parent/children primitives: a primitive q, when processed by an operation, results in a set of primitives p. Any primitive p is called* descendant *of q.*

ing the ruleset explosion mentioned before. However, their method works entirely on the tree-view of the building, in a completely separate way of the ruleset. Because of this, they are forced to keep all local changes independent on a separate file. This approach has some clear benefits, though. For instance, the independence of the local changes and the ruleset allow to keep the original model while still being able to have the changes made afterwards. However, this approach also forces the system to store independent files and have independent processing systems for the building creation and the exceptions introduced through the local control way.

As mentioned above, our objective in this paper is to provide a tool that offers all the advantages of the previous mentioned works, smoothly blending their benefits in a visual editing environment.

3. Visual Local Control

3.1. Direct Visual Selection

The first step is to provide a system that allows to select primitives by a traversal of the primitive-tree, but at the same time keeping track of the operations that produce each primitive, in order to be able to have an accurate positioning of the primitive to control. This clearly implies navigating through the primitive hierarchy, which means tracking which primitives resulted from the operation of a node on any given incoming primitive.

In essence, this can be done by tracking the parent/children relationships between primitives by tracking their unique tag names. The basic verification is whether a given primitive q is ancestor of another one p: If the set of tags associated to q is included in the set of p, both sets not being equal, then we say that q is ancestor of p, or that p is descendant of q. Verification of parenthood between q and p requires both verification of the ancestor relationship and that that the size of the set of tags of p is at most the size of the tag set of q plus one. With this basic functionality, parenthood becomes easy to verify, and the whole hierarchy of primitives can be easily traversed with simple tree-traversal operators. For instance, to know which primitive is the parent from a given one, we simply check all primitives that the current operation (node) receives and perform the parenthood verification for each one. Also, every primitive knows the operation (node) that produced it, bridging the gap between the graph and the tree views.

In our case, we have developed a simple library that bridges the two worlds, called *productTree* that provides this functionality. The library provides a full API, strongly inspired by W3C's Document Object Model [Mar02]: we provide functions to go from a primitive to its parent, its children, and to its following sibling (the operation *nextSibling* applied to the last child primitive returns a *null* value).

Our graphical user interface uses this library to allow the user to navigate through the primitive tree. Once the user selected a primitive, either as a result of this navigation through the primitive tree, or by direct point and click, we can precisely locate it both at the tree level and at the graph level. See Figure 5. This allows the introduction of changes at the selected primitives, which can be at any level in the hierarchy. Clearly, traditional tools Autodesk Maya or 3DS Max would require considerable more effort from the user to achieve even a simple change in the building structure. This is where the procedural approach excels: in its ability to manipulate and modify a model created by the end-user.

3.2. Local Control Though Graph Transformation

Once we have an accurate localization mechanism we can rely on, it is time to incorporate the desired user changes into the ruleset. We do this by adding a few new operations that are added to the traditional artistic toolbox, although in our system they are not intended to be directly manipulated by the user.

3.2.1. Operations

We have created four new operations, although only three of them are used for the objective of this paper. The other one, the *tagToAttrib*, is presented here with the sole purpose of completeness. The way these extensions work is by adding an attribute to each primitive, so that each one carries this extra information. Of course, for the whole system to hold, traditional operations should preserve this information. For instance, a *subdiv* node must propagate the attributes of the parent primitive to each children resulting from this operation. Basically, this requirement is the same as the ones for the flag propagation mechanism in the works by Lipp et al. [LWW08] and Krecklau et al. [KPK10].

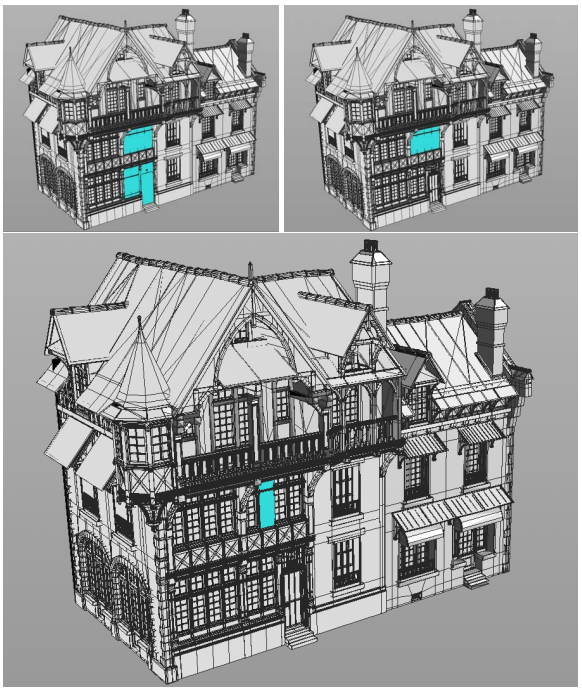


Figure 5: *The selection process: the user browses the primitive tree to find the place to apply the desired changes. In this case, they select a block of windows (top-left), a floor inside the block (top right) or a single window.*

initAttrib : This operation initializes a given attribute to a null value. Basically, it plays the role as variable declaration in traditional programming languages, to avoid querying any primitive for an inexistent attribute (variable). Without loss of generality, in our implementation all attributes are stored as strings, but a simple conversion suffices to have any type needed. In the examples for this paper, we have initialized an attribute labelled "*selection*" with an empty string. This initialization is done just after the *comp* operator that converts 3D geometry into 2D primitives [MWH*06].

addAttrib : This operation sets a given value to one primitive attribute. As logical, it requires the attribute to exist in that primitive, which is done with the previous operation. Then, this operation simply needs the current primitive ID, which can easily be obtained from the primitive itself, and assigns a value for the selected attribute at the chosen primitive. In the examples in this paper, the *addAttrib* simply needs the name of the variable to assign (i.e. *selection*), the primitive ID, and the value to assign to the variable. For the value we have used the concatenation of the string "*select_*" and the string-converted selection number.

attribToTag : This operation simply converts attribute values into tags, as used by the rest of the system. This way, we can have specific tags for each primitive value. This operation works like a *switch... case... default* construct in programming languages. This default tag value is used for all the primitives not carrying any of the required attributes. In the examples shown in Figure 11, as we have applied only two instances of local control, we have used the selection value assigned above and added a small substring (we used "*tag_*") to the tag name. This is equivalent to use the following code:

```
switch( selection ){
  case select_1: assign tag "tag_select_1"
  case select_2: assign tag "tag_select_1"
  default: assign tag "tag_select_default"
}
```

tagToAttrib : This operations represents the inverse of the previous one, assigning an attribute value to any primitive carrying a given tag value. It also has the structure of a *switch... case... default* statement.

3.2.2. Ruleset Transformation

Once the user has selected with the mechanism in Section 3.1 a position where to change a given asset or parameter, we need to adapt the ruleset to accommodate the change. For that reason we used the new commands introduced in the last section. The first one to use is the *initAttrib* command. As it is used only for initialization purposes, we add it as soon as we start using 2D primitives. In a standard modeling setting, this is done with a *comp* command, so we locate it and we put the *initAttrib* just after. See Figure 6.

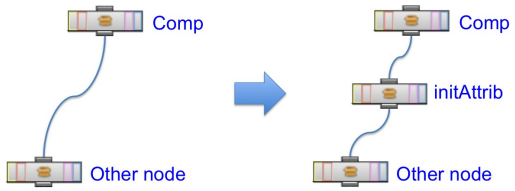


Figure 6: *The introduction of an* initAttrib *node into the graph.*

Once initialized, we have to mark the selected primitive. We do this by setting its attribute with the *addAttrib* command. As the selection mechanism described in Section 3.1 already provides us with the selected primitive, we simply need to query it for its ID and the operation that created it, called *parent* operation. This can be done with two straightforward lines of code. Once we have this information, we simply instantiate a new *addAttrib* operation node, set its parameters, reconnect all outputs from the parent to the *addAttrib* output, and connect this node to the output of the parent node. See Figure 7.

Finally, once we have selected the primitive we want to change, no matter its position at the primitive tree, we need

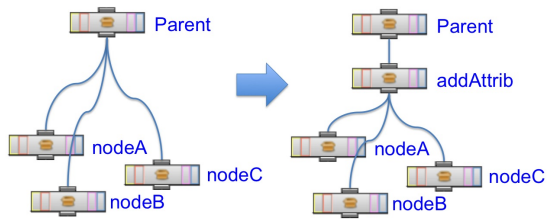


Figure 7: *The introduction of an* addAttrib *node into the graph.*

to add the operations (nodes) that will introduce the actual change at the geometry level. Our system starts by locating all *insert* nodes that process geometry resulting from the selected primitive. This can be simply done by collecting all tags used for the *insert* nodes that process primitives derived from the current one. Then, the user is allowed to select among these options which asset to modify. For instance, if we select the whole ground floor in order to change its windows, the user will probably only need to select between tags "*door*" and "*window*", so this is not a problematic step unless weird names were chosen for the tags. Then, when the user decides to finally apply the change, the algorithm traverses the product tree and locates the *insert* nodes that operate on parts of the selected primitive. For each of these *insert* nodes, the system automatically creates an *attribToTag* node and reconnects all the *insert* inputs as inputs of the new node, and reconnects the *insert* to the *attribToTag* output. The tag selector field of the *insert* is also changed to the *attribToTag* default tag value. Finally, new *insert* nodes are connected to the output of the *attribToTag* operator, and their input selectors are set to their respective tag values. These changes can be seen in figure 8.

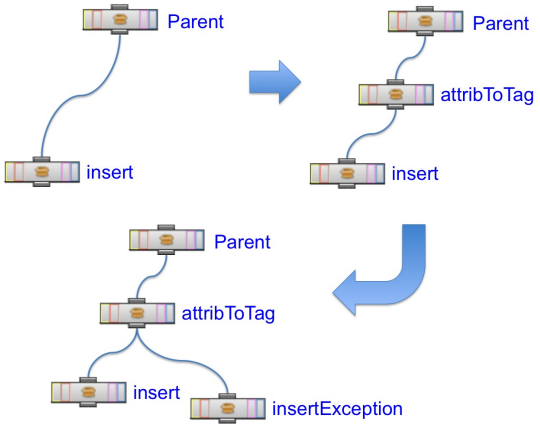


Figure 8: *The introduction of an* attribToTag *node into the graph and the subsequent new insert noes.*

For a more complex example, in Figure 9-top we can see a part of the operations graph corresponding to the left part of the Raccolet House. See Figure 11. Observe that the whole process only requires from the user the selection of the asset to change (or its parameters), which is done with the visual mechanism presented in Section 3.1 plus a couple of parameters indicating the replacement asset and the tag of the target primitive (in case several primitives were selected). Thus, this mechanism completely frees the user from the burden to manually modify the original ruleset to introduce the desired changes.

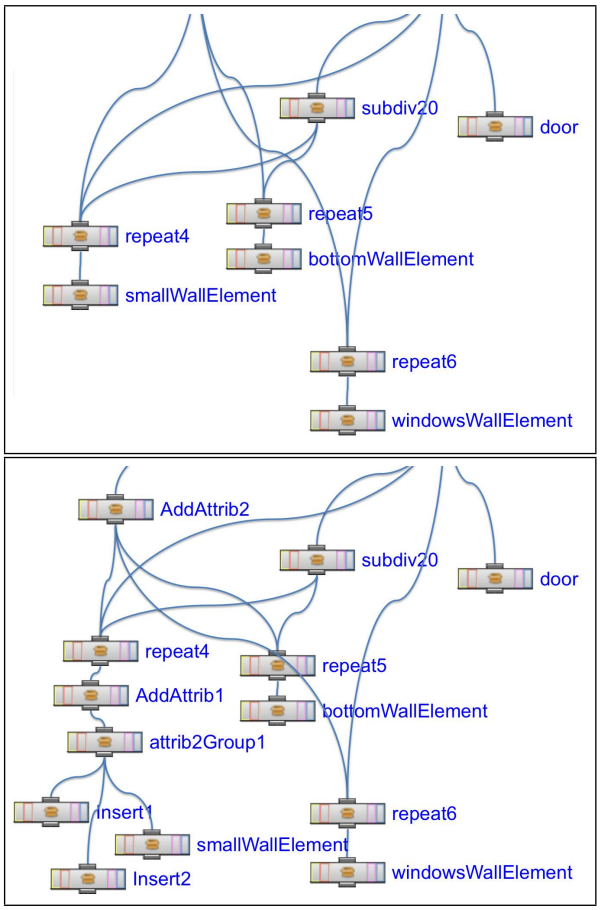


Figure 9: *Two close-up views of the graph transformation process. Top: Graph before the rewriting. Bottom: after. These changes correspond to the bottom image in Figure 11.*

4. Implementation

Our system is implemented over the **building**Engine system, which is part of the **skyline**Engine [RP10] system, which runs on top of SideFX's Houdini [Sid12]. Our selection mechanism, presented in Section 3.1, together with the visual selection interface, were implemented as external Python libraries working on the visual ruleset that defines the operations that construct the building [Pat12]. Each of

the four new operations presented in Section 3.2.1 was created as a visual module using a mixture of Houdini's own nodes and embedded Python scripts. The graph transformation mechanism presented in Section 3.2.2, on the other hand, works a a layer defined through scripts that operate externally to the ruleset definition, modifying it and replacing the "old" ruleset by the "new" one, which can be saved and re-used in a production setting. The code for this paper is available from the project web page, at http://ggg.udg.edu/skylineEngine/.

5. Results

In Figure 10 we can see an example of the navigation through the model primitive hierarchy, performed on the Sea View Hotel model. In Figure 11 we can see our system working on the Raccolet House. The input building is shown in 11-top-left. The user starts by selecting the assets to change, as shown in Figure 5 (top-right and bottom). The user has to further specify the asset to use instead of the original one, and, in the case of a multiple-selection (e.g. Figure 5-top-right), the tag of the assets to change. Then our mechanism modifies the ruleset as shown in Figure 9, resulting in the finished building. This final result can be seen in Figure 11-top-right for single asset and in Figure 11-bottom for a whole row of assets which were changed at once. As mentioned before, this later case required the user to clarify if they wanted to change the columns, the door or the windows, as was done here. The insets in Figure 11 show a zoomed view of the changed area.

6. Discussion

In this paper we have presented a new system to add direct and persistent local control to the visual editing of rules for procedural buildings, avoiding a combinatorial explosion of grammar rules. This system only requires a small an intuitive input from the user: just to point the assets to change, and the system automatically produces a new ruleset that includes the selected modifications. The only change needed for current procedural production systems is the introduction of three new commands, and a graph-rewriting mechanism on top of the whole system. We demonstrate that this suffices to have the same flexibility and simplicity as Lipp et al.'s approach, but seamlessly integrated into any visual pipeline. We honestly think that this greatly enhances the artist toolbox, and even better, this system does so without adding any extra modeling effort. All complex manipulations are automatic an transparent to the end user.

One pice of further research to conduct is, obviously, to assess the system by architects and designers in general. We also believe that the selection system offers a whole set of new possibilities which should be explored, like more abstract selection mechanism, rule-identification and generalization, or even automatic building design by example.

7. Acknowledgments

The Raccolet House and Hotel Sea View were modeled by DAZ 3D software (www.daz3d.com). We want to thank the anonymous reviewers by their constructive comments. This work was partially funded by grant TIN2010-20590-C02-02 from Ministerio de Ciencia e Innovación, Spain.

References

[BSMM11] BENES B., STAVA O., MECH R., MILLER G.: Guided procedural modeling. *Comput. Graph. Forum 30*, 2 (2011), 325–334. 2

[Epi12] EPICGAMES: Unreal development kit (udk), 2012. http://udk.com. 1, 2

[Esr12] ESRI: Cityengine, 2012. http://www.esril.com/software/cityengine/index.html. 1, 2, 3

[KK11] KRECKLAU L., KOBBELT L.: Procedural modeling of interconnected structures. *Comput. Graph. Forum 30*, 2 (2011), 335–344. 2

[KK12] KRECKLAU L., KOBBELT L.: Interactive modeling by procedural high-level primitives. *Computers & Graphics*, 0 (2012), –. 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. 4

[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. 2

[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, 3, 4

[Mar02] MARINI J.: *Document Object Model*, 1 ed. McGraw-Hill, Inc., New York, NY, USA, 2002. 4

[MSK10] MERRELL P., SCHKUFZA E., KOLTUN V.: Computer-generated residential building layouts. *ACM Trans. Graph. 29*, 6 (Dec. 2010), 181:1–181:12. 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, 5

[Pat12] PATOW G.: User-friendly graph editing for procedural modeling of buildings. *IEEE Computer Graphics and Applications 32* (2012), 66–75. 1, 2, 3, 6

[RP10] RIDORSA R., PATOW G.: The skylineengine system. In *XX Congreso Español De Informática Gráfica, CEIG2010* (2010), pp. 207–216. 6

[Sid12] SIDEFX: Houdini 12, 2012. http://www.sidefx.com. 6

[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. 2

[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

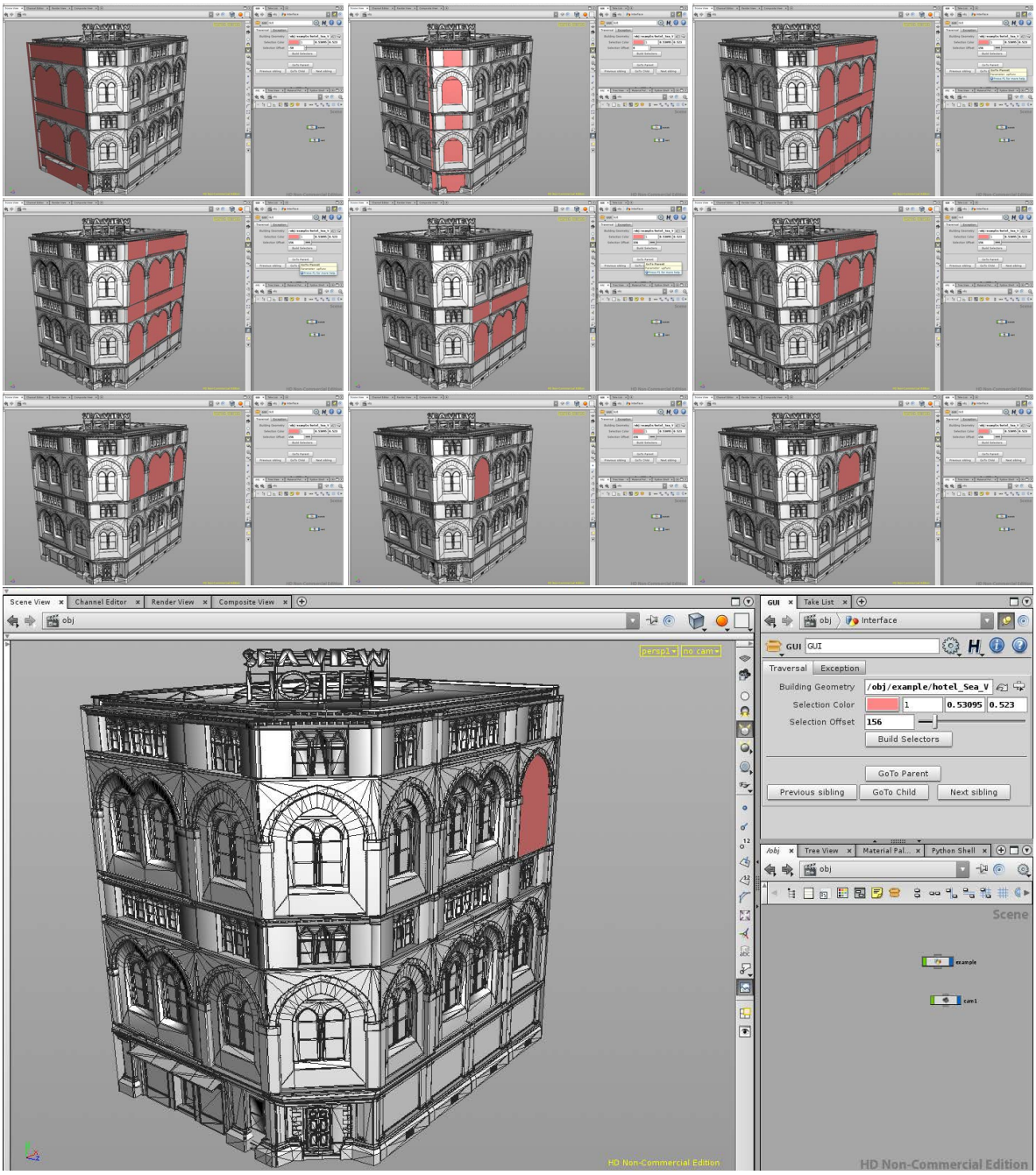


Figure 10: *Another example of selection for local control. .*

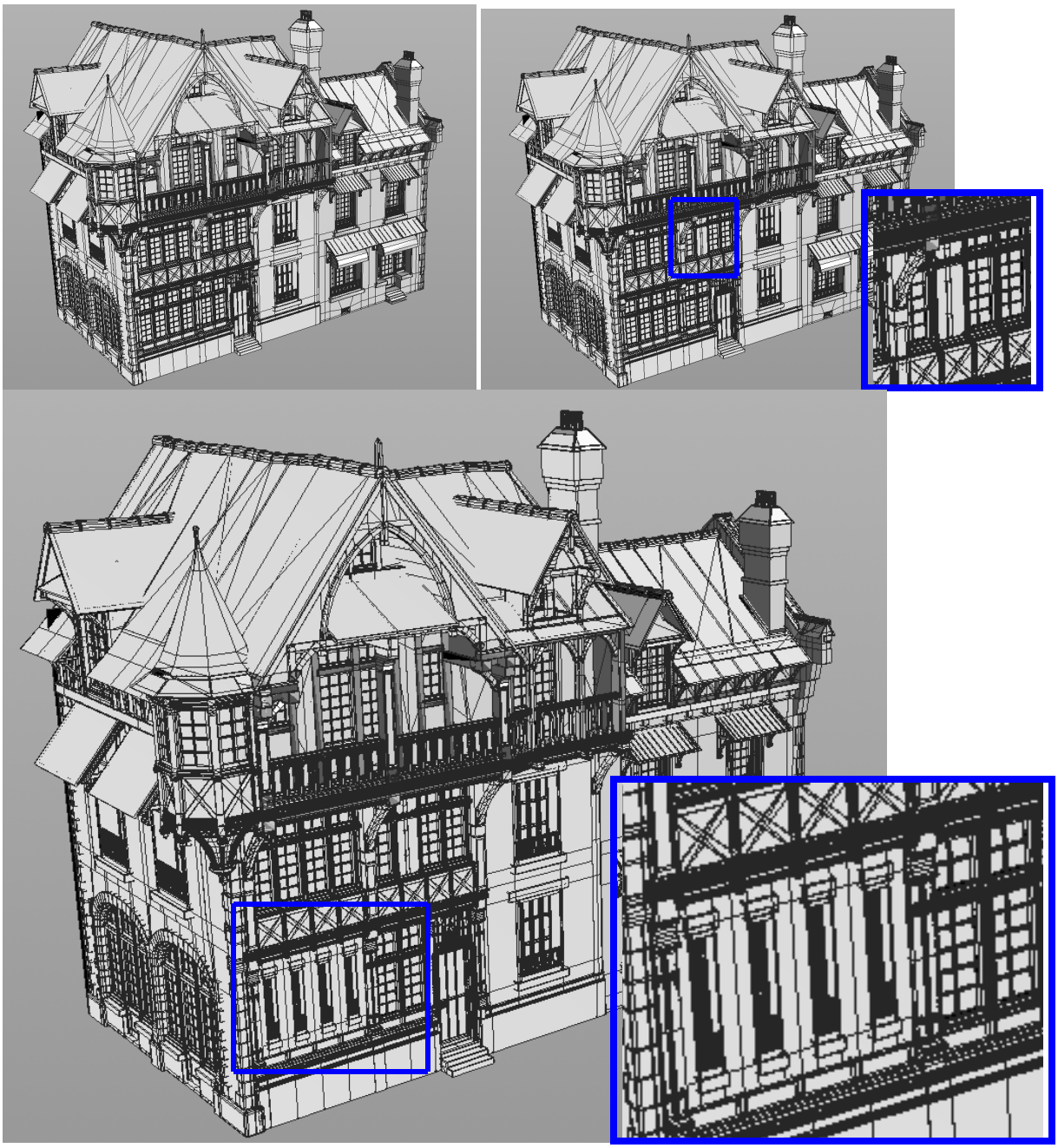


Figure 11: *Results: The user decides to change a single window from its original type (top-left) to a new one (top-right), and finally changes a whole set at once (bottom).*