Computers & Graphics 37 (2013) 238–246

Contents lists available at SciVerse ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Special Section on Procedural Modeling

Visual copy & paste for procedurally modeled buildings by ruleset rewriting

Santiago Barroso, Gonzalo Besuievsky*, Gustavo Patow

Geometry and Graphics Group, Universitat de Girona, Spain

ARTICLE INFO

Article history:
Received 15 October 2012
Received in revised form
8 January 2013
Accepted 8 January 2013
Available online 26 January 2013

Keywords:
Procedural buildings
Interactive modeling
Graph rewriting

ABSTRACT

With the increase in popularity of procedural urban modeling for film, TV, and interactive entertainment, an urgent need for editing tools to support procedural content creation has become apparent. In this paper we present an end-to-end system for procedural copy & paste in a rule-based setting to address this need. As we show, no trivial extension exists to perform this action in a way such that the resulting ruleset is ready for production. For procedural copy & paste we need to handle the rulesets in both the source and target graphs to obtain a final consistent ruleset. As one of the main contributions of our system, we introduce a graph-rewriting procedure for seamlessly gluing both graphs and obtaining a consistent new procedural building ruleset. Hence, we focus on intuitive and minimal user interaction, and our editing operations perform interactively to provide immediate feedback.

© 2013 Elsevier Ltd. All rights reserved.

1. Introduction

A broad range of areas, such as games, movies or urban simulation require virtual 3D city models with detailed geometry. Procedural modeling [1] has proven to be quite effective for this task [2], offering a potential alternative to the labor-intensive modeling tasks required by traditional 3D modeling techniques for building reconstruction. However, traditional procedural methods are not always a suitable alternative to manual modeling. With this, there is an increasing need for more advanced content creation and editing tools. However, it is not straightforward to extend existing tools, as for example sketch-based interfaces for modeling [3], drag-and-drop mesh tools [4] or the modeling-by-example approach [5] to procedural models because these tools operate on the mesh level, and are not able to preserve the procedural nature (i.e., the ruleset) of the input building.

In this paper we focus on a specific editing application: copy & paste for procedural building models. One of the main motivations for choosing this application is to provide a simple to use, intuitive editing metaphor that would allow non-experts to generate new content using pre-existing rulesets (see Fig. 1). We will present a complete, end-to-end system for procedural copy & paste consisting of the following components: selection, copying, composition (paste) and proportion adjustment. We will also describe our contributions for each component.

With our approach, artists can easily reuse already known building styles to create new content. As a benefit, the tools may shorten significantly the modeling time creation, avoiding designing new rules or reconfiguring old ones. Similar to [6], we used a visual programming paradigm, where the user can construct and modify the building by simply connecting its components on screen by interactive visual inspection. Actually, our approach is complementary to existing modeling techniques, as it produces a building ruleset ready to be used in any production environment.

Contributions To address these challenges, we have developed an end-to-end system for procedural copy & paste with the following additional contributions:

- We provide an interactive and intuitive visually-based method for editing models.
- We introduce a graph-rewriting procedure for seamlessly gluing source and target graphs.
- Our editing operations perform within interactive rates to provide immediate feedback.

2. Previous work

The idea of adding copy & paste operations to modeling tools is not new. The modeling-by-example approach, presented by Funkhouser et al. [5] allowed compositing of different parts from different input models to obtain a desired output. Later, Nealen et al. [3] presented sketch-based interfaces for preserving details

* Corresponding author. Tel.: +34 972418833; fax: +34 972418792.
E-mail address: gonzalo@ima.udg.edu (G. Besuievsky).

0097-8493/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.cag.2013.01.003

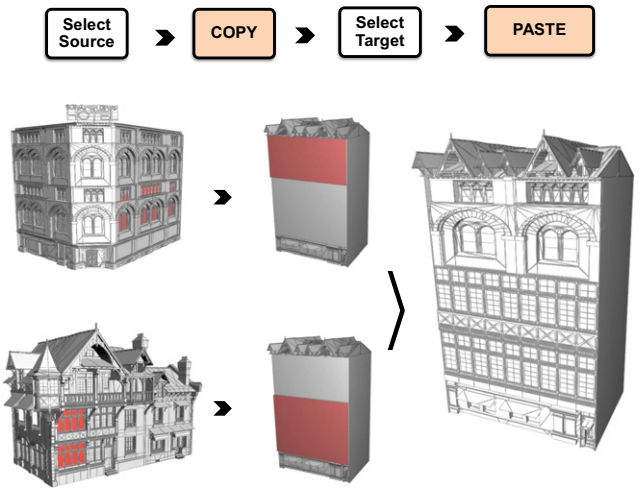

Fig. 1. Copy-Paste workflow system (top). From a procedural buildings selected sources (bottom, left) and a selected target (bottom, middle), a new building is composed (bottom, right).

while mesh editing. There are also drag-and-drop mesh tools, like Autodesk's Meshmixer [4] to manipulate geometry allowing copy & paste operations. However, none of these approaches is suitable to be applied to procedural models because they operate on the mesh level, and are not able to preserve the procedural nature (i.e., the ruleset) of the input building.

Procedural Modeling is a term that describes a family of techniques that generate geometry from a set of rules. These approaches [7,1] have emerged as an elegant solution for the generation of massive urban landscapes from a simple ruleset. Since the earliest works, great improvements have been done in terms of interactivity and ease of use, specially for urban layouts [8,9]. The interested reader is referred to the articles by Vanegas et al. [10] and Watson et al. [2] for a more in-depth review of the current state of the art in this topic.

Lipp et al. [6] presented one of the firsts attempts to improve editing operations for procedural buildings using an interactive visual system, which completely avoids text editing rules. Later, CityEngine [11], Epic's UDK [12] and Patow [13] independently developed visual representations for the rulesets, considerably easing the development process. More recently, some approaches tried to bring together both worlds: direct control and visual procedural languages, resulting in a simple visual traversal of the hierarchy tree plus direct visual assignment of the desired changes [14]. However, tool development has not gone much further in this direction, reducing the user options to only a few simple operations [13]. In this paper we provide a completely visual copy & paste application to simplify the user's editing tasks.

The idea of the copy & paste operation is intimately related with local control in procedural models. As mentioned, Lipp et al. [6] developed a local control mechanism, but it was external to the ruleset and required the re-evaluation of the local changes for every change in the building structure. Later, Patow [13] introduced an explicit *Exception* command, while Krecklau and Kobbelt [15] added specific rules selecting a given label with a given shape index to apply the different change. None of these approaches achieve the flexibility of the copy & paste editing operation proposed here. In the context of urban models, Lipp et al. [9] also presented a solution for interactive modeling of procedural city layouts that allows to manipulate them intuitively using drag-and-drop operations.

Our work is also based on graph-grammars and graph-rewriting techniques. In Computer Graphics this topic cannot considered to be new: L-systems [16] can be considered as a graph-rewriting system if we consider the processed string as a "language" that is executed to generate a tree, and the rules then are part of a graph-grammar that transforms one string into another. A graph-rewriting system is also presented in [17] for specific 3D furniture model fabrication, but with no user interaction. Even in the context of urban procedural modeling there have been some approaches that take advantage of the graph-like structures that arise in urban layouts [9] or the building rulesets [13]. Here we also perform graph-rewriting operations over the rulesets, but in contrast to [13], where only simple operations to fix some minor design issues in the procedural model where allowed, our editing tool goes far beyond current state of the art techniques, allowing complex editing operations to be performed in a way transparent to the user. There is a large bulk of literature on graph- rewriting techniques, so we refer the interested reader to the excellent work by Heckel [18] and the well-known handbook by Rozenberg [19].

3. Procedural modeling

The seminal works by Wonka et al. [7] and Müller et al. [1] introduced grammar-based procedural modeling for buildings. The main concept of this technique is a shape grammar, which is based on a ruleset: starting from an initial axiom primitive (e.g., a building outline), rules are iteratively applied, replacing shapes with other shapes. A rule has a labeled shape on the left hand side, called predecessor, and one or multiple shapes (also called primitives) and commands on the right hand side, called successor

$$predecessor \rightarrow CommandA, CommandB : labelB;$$
$$labelB \rightarrow CommandC : labelC;$$

The resulting geometry is formed by shapes that can be optionally assigned new labels with the purpose of being further processed. In our system, this geometry carries all labels that the shape or any ancestor has received during the production process. The main commands, the macros that create new shapes in the classic approach, are

- *Subdivision* that performs a subdivision of the current shape into multiple shapes,
- *Repeat* that performs a repeated subdivision of one shape multiple times,
- *Component split* that creates new components shapes (faces or edges) from initial volumes,
- *Insert* command that replaces a pre-made asset on a current predecessor.

Traditionally, during a rule application, a hierarchy of shapes is generated corresponding to a particular instance created by the grammar while inserting rule successor shapes as children of the rule predecessor shape. This production process is executed until only terminal shapes are left.

3.1. Graph-based Procedural modeling

The whole production process described above can be seen as a graph where each node represents an operation applied to its incoming geometry stream and the leaf nodes are the geometry assets [13] (see Fig. 2). A ruleset can be regarded as a *directed acyclic graph* $G=(N,E)$, where N is a set whose elements n_i are called vertices or nodes (i.e., these are commands of the ruleset), and E is a set of ordered pairs of vertices, called edges (i.e., the connections between the rules that represent the flux of geometry). Each command $n_i \in N$ processes its incoming flux of geometry, which is given by all its incoming primitives (i.e., the shapes)

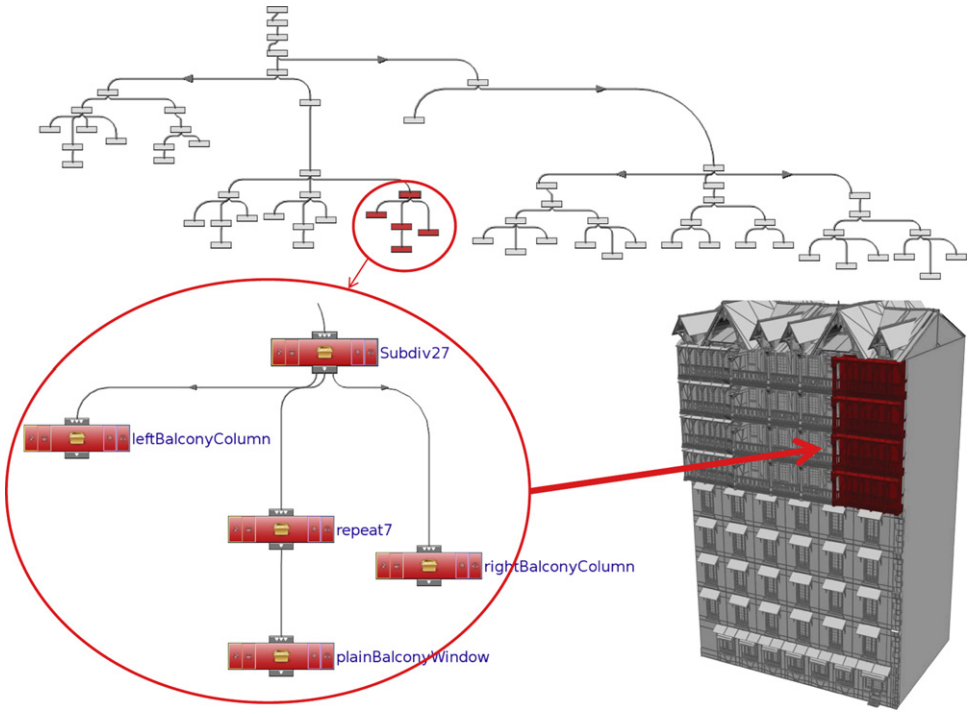

Fig. 2. A full graph-based representation (top) of a procedurally modeled building (bottom, right). The red subgraph (bottom, left) represents the red part of the facade. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this article.)

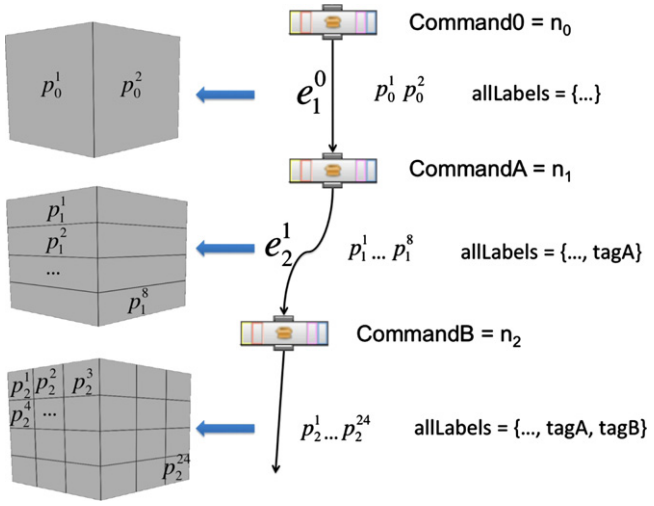

Fig. 3. Rules and products of the graph. Each node operates on the primitive inputs with correct label and outputs a set of tagged primitives.

that have the given predecessor label. Let $e_j^i \in E$ be the edge connecting the output of node n_i to the input of a *downstream* node n_j. Fig. 3 shows node n_1 wired to node n_2, and the wire that connects them is e_2^1. The left part of the figure shows the respective primitives that constitute the geometry flux in e_2^1.

Now, let P_i^{in} be the set of geometric primitives input by n_i through its input wires, and P_i^{out} the ones it produces after its processing. We denote the j-th primitive in the n_i output stream P_i^{out} by p_i^j. Given a node n_j, the set of primitives it receives is given by

$$P_j^{in} = \bigcup_{i:\exists e_j^i} P_i^{out}$$

Each incoming primitive has an associated label, which we denote by $label(p_i^j)$. A command selects the geometry to process by filtering the incoming primitives by their labels (the predecessors). The user must provide the filter for each node n_i as one of the node parameters, which can be recovered through the operation $filter(n_i)$. The commands operate primitive-wise: node n_i operates over each input primitive in P_i^{in} at a time, processing it only if the primitive has the required label, and discarding it otherwise. That is, an input primitive p will be processed if and only if $filter(n_i) = label(p)$. Thus, each node operates only on the primitives with the correct label, resulting in a set of zero, one or more output primitives. This renders the primitives into a tree-like relationship.

As every primitive p is the result of the operation of node n on an input primitive q, we can always track the relationship between p and n by using the operation $node(p) = n$. Also, for any output primitive p, we can track its parent primitive q with the operation $q = parent(p)$. Conversely, the set of children of any input primitive q is retrieved with the operation

$$children(q) = \{p : parent(p) = q\}$$

Any such primitive p is called *child* of q. Note we are assuming that each primitive q is processed by one and only one node n (resulting in zero, one or more primitives p), but it is easy to generalize the notation for the case where more than one node operate on the same primitive.

In our implementation, each primitive carries all labels that were assigned to any of its ancestors or itself. We implemented the operation $parent(p)$ for a primitive p from a node n_i to return the primitive q such that $q \in P_i^{in}$ and

$$\|allLabels(q)\| <= \|allLabels(p)\| + 1$$

where $allLabels(p)$ is an operation that returns the full set of labels attached to primitive p and $\|X\|$ gives the cardinality of a set X (see Fig. 3). The equality case comes from the fact that the user might decide not to add any new label to the generated primitives in some operations. The actual implementation of the operation to retrieve a primitive *most recent* label is

$$label(p) = allLabels(p) - allLabels(parent(p))$$

where the operator "−" is the difference of the label sets and *parent*(p) returns the ancestor primitive of p. Finally, we treat asset insertion separately from the primitive tree. The *Insert* nodes applied to any primitive p can be found with the operation *childInserts*(p).

4. Visual procedural copy & paste

Our visual copy & paste system for procedural buildings allows a user to select objects from several procedural source buildings and composite them into a desired procedural target building. The editing workflow for procedural copy & paste is shown in Fig. 1. Input to the system are procedural building rulesets for the source and target, which where previously created by an artist using tools like CityEngine [11] or Epic's UDK [12]. The system can be divided into four components:

- Selection
- Copying
- Composition (Paste)
- Proportion adjustment

Selection consists of the navigation through the building hierarchy created at rule evaluation, and selecting a location in the hierarchy. Then the users select one or more objects from the source buildings to be copied to any desired location selected in the target building.

The *Copying* operation can be decomposed into identifying the participating primitives, identifying their respective generating operations, and storing this information for later reuse.

Composition consists of consistently pasting the previously copied selection in the target building. It is performed interactively while the user is viewing the resulting composite. Finally, the *Proportion adjustment* component fixes the (possibly) wrong measures that could have been left in the pasted subgraph.

4.1. Selection

The first step is to provide a way to select part of a facade by a traversal of the primitive-tree. This clearly implies navigating through the primitive hierarchy, which means tracking it as described in Section 3.1. We provide two different starting points for this navigation: by selecting the roots of the hierarchy (the 2D primitives produced by the *Component split* operation), or by directly selecting of a leaf by a point-and-click interface. Then, a simple user interface lets the user visit any primitive children, browse through its siblings, and go back to the parent primitive as desired [14].

Once the user selected a primitive p, 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. This is done through the operations *children*(p), *parent*(p) and *node*(p) described in Section 3.1. Fig. 4 shows examples of navigation and selection of different parts of a building.

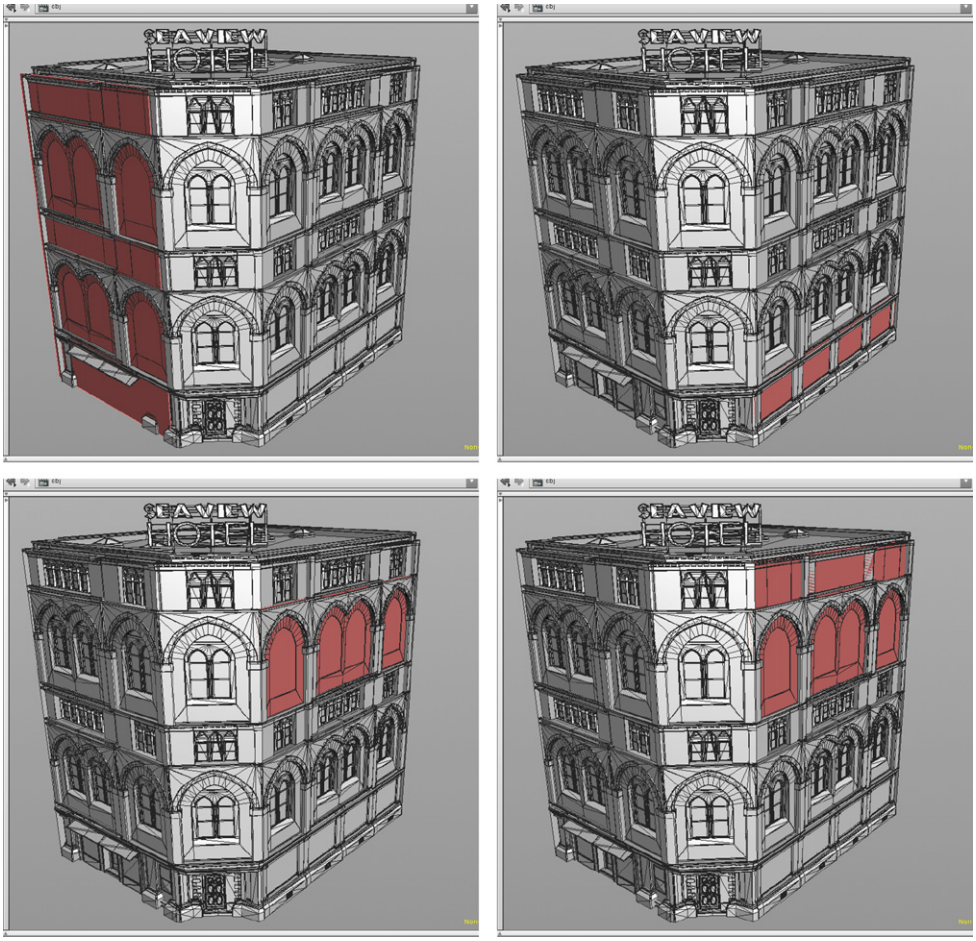

Fig. 4. The selection system with four examples. The visual interface allows to navigate through the primitives of the tree for interactive selection.

4.2. Copy

The next step, once the user has selected a given primitive p, is to collect all nodes in the source graph G^S that operate on p or any of its descendants (see Fig. 5). For that, we iteratively traverse the primitive tree in depth-first order, starting from p, until we reach the leaf primitives. For every descendant d of p, we collect the node $node(d)$ that produced it. This new set of nodes is $N' \subset N^S$, and the node inter-relations are the edges $E' \subset E^S$, forming a subgraph $G' = (N', E')$. We record the labels associated to p with $l = allLabels(p)$. We also keep track of the *Insert* commands associated with each primitive.

Once all primitives have been traversed, and the subset N' has been built, we copy all the nodes belonging to N' and all the edges that connect such nodes into a buffer, which will be later used during composition. Observe that we also select the node n_p that originates p, which should *not* be part of N'. However, keeping this node will be useful later on.

4.3. Composition

Once we have copied the subgraph N', we must select the target area in the composite building. This is done as before, by navigating the primitive hierarchy and selecting a primitive t. Once t is selected, we need to merge the graph $G^T = (N^T, E^T)$ of the target building with the subgraph G' we copied from the source building. This is done following the rules of graph rewriting [19,18]: first, we identify the actions performed on t by G^T (Fig. 6(a)), to then separate t from G^T, as it is going to be processed by G' (Fig. 6(b)). This is done by modifying the ruleset so t is excluded from its regular evaluation. Then, we need to make sure that *only* t is directed towards G' (Fig. 6(c)), and finally, G' must be connected to G^T (Fig. 6(d)). However, the sizes and distances in G' can be very different than those in G^T, so a final adjustment step is needed.

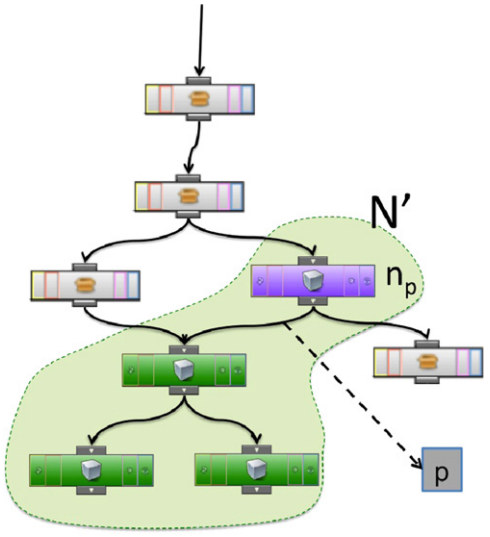

Fig. 5. Copy process. The processes node p (lilac node) and all its descendants (green nodes) are copied into a buffer. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this article.)

A *graph transformation* is defined by a tuple $(c,L,R,glue)$ consisting of a condition c that, if positively evaluated, will activate the rewriting transformation from the left term L into the right term R by means of the gluing mechanism *glue*. In our case, we do not need to evaluate c as the user directly gives us the information where to glue into the target graph by specifying primitive t. Here, $R \equiv G'$ and $L \equiv \{n_t = node(t)\} \cup \{n_i : \exists\, e_i^t \in E^T\}$, i.e., the node that generated t plus all nodes connected downstream to it. In Fig. 6(a) these nodes are shown in blue color. Now, it only remains to define *glue* to transfer the selection to the target building facade.

First, we need to exclude t from the regular processing in the target building. This can be done with an *Exception* node [13], which assigns a user-defined label to a primitive that fulfills a given condition; and a *selector* node, which deletes any primitive that does not have the adequate label. We connect $node(t)$ to the exception node, and then the selector to the exception output. Finally, we reconnect all nodes connected to the outputs of $node(t)$ to be the outputs of the selector node. The parameters for the *exception* node are set to select t. This way, all other primitives produced by $node(t)$ will continue being processed as before, with t set aside of the process. In Fig. 6(b) these new nodes are shown in yellow.

The next step is to redirect t to the new subgraph we are going to attach. Again, we do this with another *selector* node that is set to keep t and delete all other primitives. Now, we have to copy the graph G' into the new graph, attaching the nodes in N' to the outputs of the new *selector* node. For that, we instantiate G', including all nodes in N' and their connections. Remember that we have kept n_p, the node in G^S that produced the chosen primitive p, so it will be pasted along with the other nodes in G' (see Fig. 6(c)), where n_p is represented as a lilac node and the other nodes in G' are shown in green.

The final step in the gluing process is to transfer all the *useful* connections from n_p to the last *selector* node created. We say that a connection e_i^p from n_p to n_i is *useful* if $filter(n_i) \in l$ (remember that $l = allLabels(p)$). Finally, we simply delete n_p (see Fig. 6(d)). However, the process still requires a last stage: as mentioned, the whole procedural paradigm is based on the rules processing the primitives with the right labels. As we have pasted parts of a completely different ruleset, there might happen that an interior node in the pasted subgraph G' requires a label that was actually set before, in the part of the source graph G^S that does not contain the subgraph G', but whose nodes produce geometry that is latter processed in G'. This would result in a wrong processing when G' is pasted in the new graph. To prevent this, we use the collected

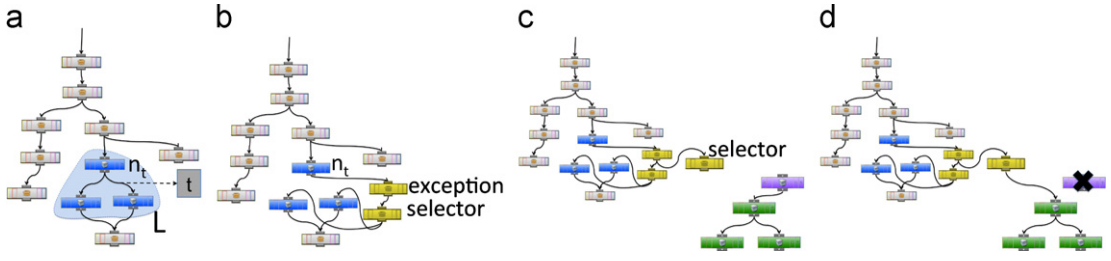

Fig. 6. Composition process. L nodes in blue, G' nodes in green, n_p in lilac, *glue* nodes in yellow. From left to right: (a) selection of the target primitive t and its corresponding nodes in G^T, (b) the isolation of the selected primitive from the regular processing, (c) the subgraph G' is pasted, and (d) the final merging of the copied subgraph G' into the target graph G^T. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this article.)

labels l and apply them to t. To avoid assigning unnecessary labels, before assigning them we check whether they are effectively used in G'. To add a label to t, we simply add a rule

$labelT \rightarrow nop : newLabel;$

where *nop* means *no-operation*.

4.4. Proportion adjustment

Although the above steps have glued the selected ruleset G' into the target graph G^T, special care must be taken to the parameters used in the rules being copied to the new building. As already explained, the method is intended for an automatic replacement of the building part, so, if the sizes of the source and target selected primitives are different, a readjustment must be performed.

If primitive p has a size of $W^p \times H^p$ and primitive t has a size of $W^t \times H^t$, then we change each parameter of the nodes in G' by the respective proportion. If a parameter is defined in a node that operates Y direction (e.g., a *subdiv*{Y} or a *repeat*{Y}) with a value v^p, then we modify it to be $v^p \cdot W^t/W^p$. Conversely, for parameters in nodes operating in the X direction, the new value would be $v^p \cdot H^t/H^p$.

Once this is done, the primitive t, and thus the corresponding part of the new facade, will have the appearance of the original building facade.

5. Results and discussion

Our system is implemented on top of SideFX's Houdini [20]. The implementation is done using embedded Python scripts and external Python methods.

Our visual interface allows to perform all operations iteratively, allowing the user to navigate through the hierarchy and select any primitive at any level (see Fig. 4). Please refer to the accompanying supplemental video showing the interactivity of all the results presented in this section.

Fig. 7 shows a new building with a facade created using three parts from already existing procedural buildings. A typical modeling session using our system (see Fig. 1) has the following sequential steps for each part of the building to be copied: navigate through the source building using the selection tool (see Fig. 4), use the copy button to store in a buffer the interesting building part, navigate through a target building to select the part to paste the copied structure to, and finally apply the paste operation to merge both structures. All these operations took only a couple of minutes for generating the whole model in Fig. 7.

The main contribution in usability of our approach is that it is an end-to-end process that runs without the need of writing rules or even neither changing parameters: everything is done visually.

Our system also allows modifying existing building rulesets (see Fig. 8). In this case we follow the same steps as in the previous example, but the paste operation has to modify the parts affected by the changes. This implies to extract the selected geometry from its previous regular processing, and its processing with the new ruleset. This is done automatically, without the need of user intervention to reconnect or modify the nodes.

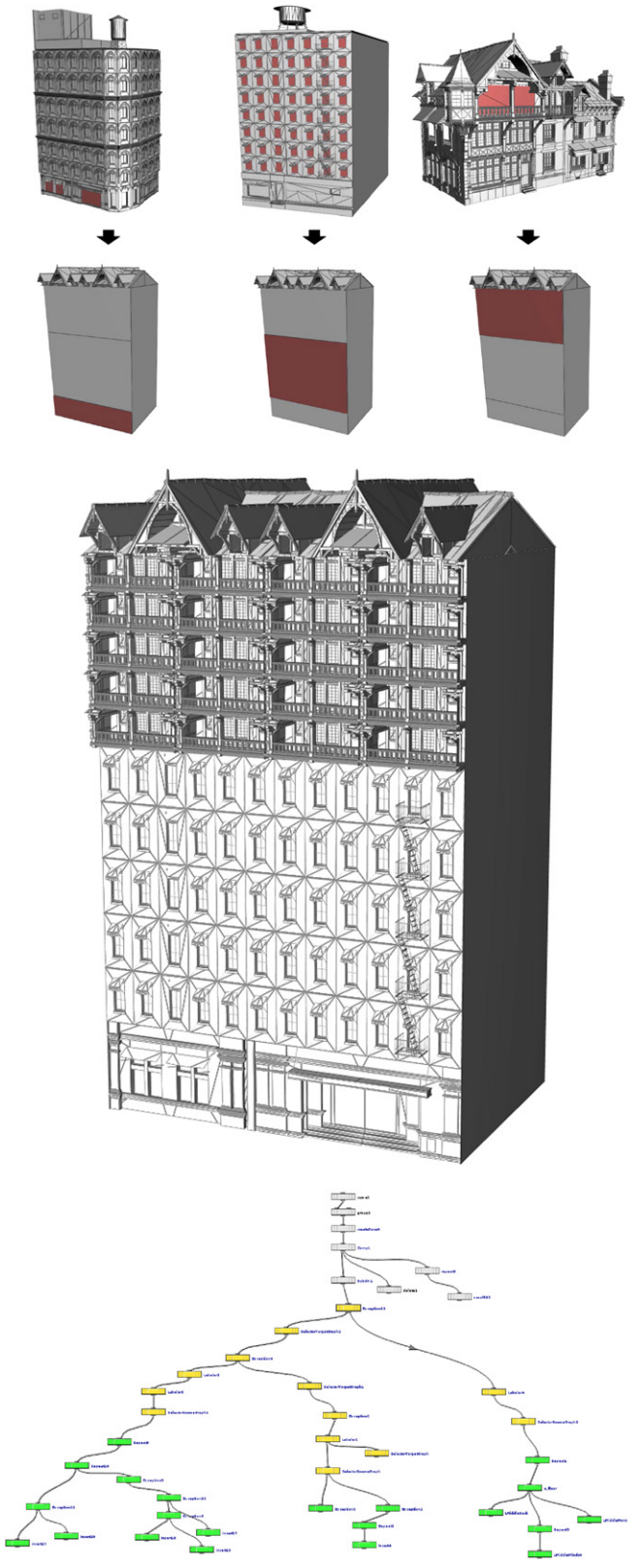

Fig. 7. From three selected parts of existing buildings (top), a new building is created (middle). In the resulting graph (bottom), the green nodes represent the subgraph G', whereas the *glue* ones are shown in yellow. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this article.)

5.1. Discussion

Comparing our system to existing solutions like the one proposed by Esri's CityEngine [11] or Epic's UDK [12], we observe that in their system the only way to copy & paste the ruleset operating on a building part is by manually selecting the needed rules, pasting them in the target ruleset, and then manually merging the rules to obtain the final ruleset.

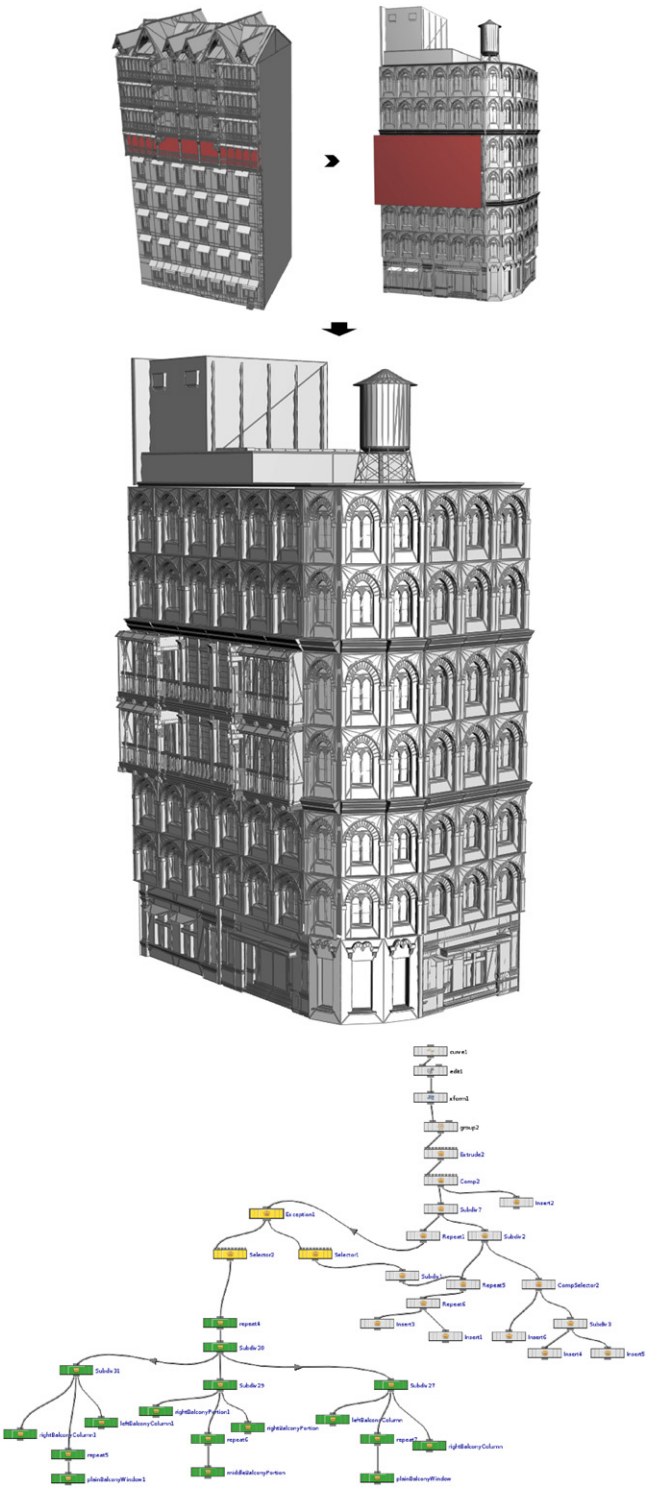

Fig. 8. From an existing building (top, right), a new building is created (middle) using a peace of facade as source of modification(top, left). The resulting graph (bottom) is shown with the same color representation of the previous example. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this article.)

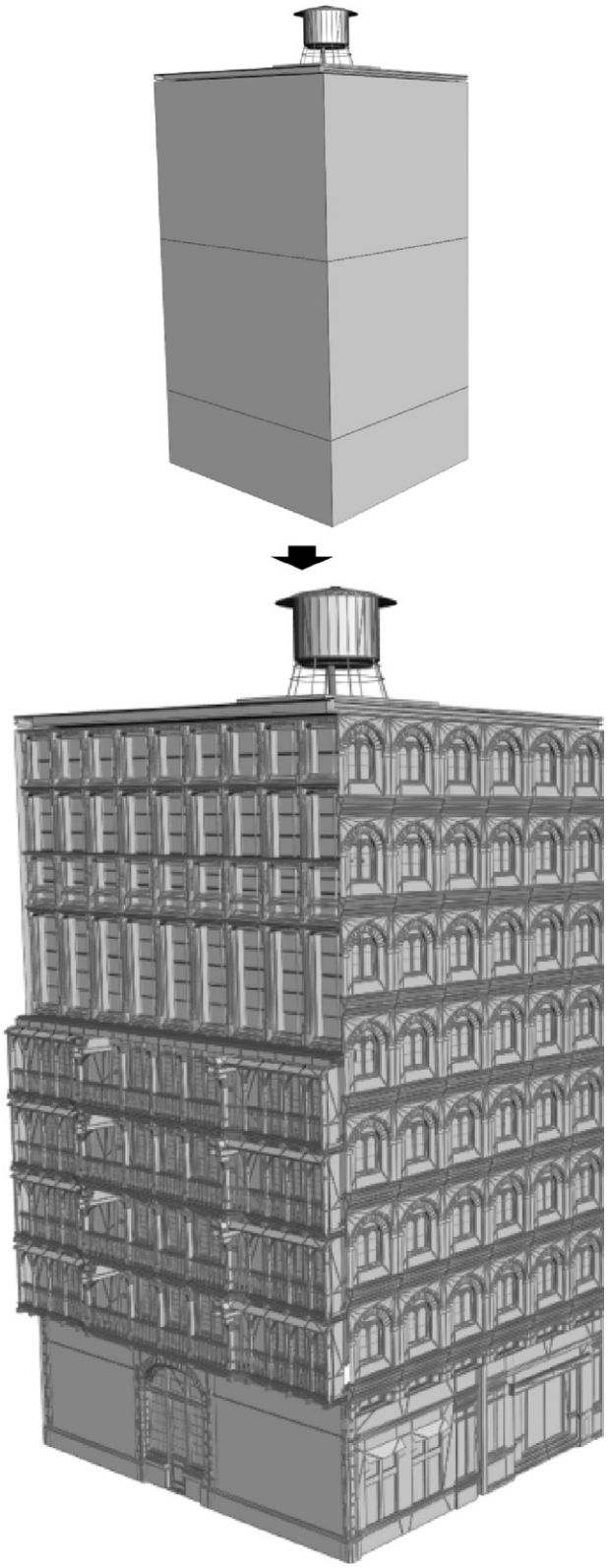

Fig. 9. Final building (bottom) used for counting the number of internal operations performed in Table 1. The model is generated from a naked building (left) using different parts of previous models.

As we stated before, our technique shortens modeling time. In order to evaluate the improvement with respect to a manual approach, we measured the number of operations we perform, which is equivalent to the number of manual operations required. For this test we used the model shown in Fig. 9, built from a naked building using selected parts from the four input models shown in Figs. 1,7 and 8. Table 1 shows the number of operations for creating and deleting nodes, connecting and disconnecting nodes, and changing parameters. A total of 226 operations were

Table 1
Number of paste operations for generating the building of Fig. 9. We measure the following number of operations: creating/deleting nodes, connecting/disconnecting nodes and changing parameters.

Operations	Node+	Node−	Connect+	Connect−	Ch.Param	Total
Top left	4	1	8	1	17	31
Middle left	4	1	8	1	27	41
Bottom left	4	1	8	1	20	34
Right facade	18	1	35	1	65	120

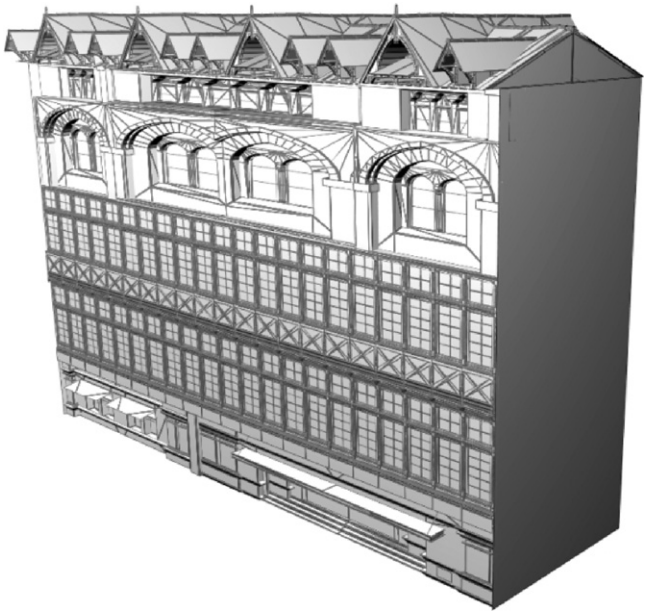

Fig. 10. Resulting model for applying an unwanted operation, the top inserted windows does not fit well with the rest of the building. The user can both manually readjust them changing rules or undo the operation.

needed for the whole copy & paste process (copying implies no changes in the source graph). A simple analysis leads to the conclusion that the computational time required by our algorithm is approximately linear in the number of pasted nodes. From the user perspective, we could compare this figures with the number of mouse clicks required in our system for the same operation. Of course, this will depend mainly on the number of model inputs. For this test model, a total of 16 mouse clicks were required, four for each input part. We can conclude that, for this test, we reduced work by a factor of 14 times, computed as the ratio of manual changes in the ruleset compared to the effort required with our technique (*num.op./num.clicks*). However, this comparison is somewhat incorrect in the sense that it is difficult to measure the time needed to *understand* a given ruleset in order to be able to select nodes, and then to *understand* the target ruleset to be able to marge the result. In our system, no understanding of the ruleset is required, just a simple visual tree traversal.

Although we designed our system with the idea of freeing the user from manual intervention at the ruleset level during the copy & paste operations, in some cases the user might want to manually adjust the resulting ruleset. In general, in our experiments we found that there was no need to further edit the resulting ruleset, as the implemented graph-rewriting steps already glued correctly the two graphs. However, the automatic proportion adjustment for the parameters described in the previous section may need some further parameter tweaking, which is to be expected: it can happen that the measures and distances designed for a building to look right might not fit well to a target building and could require some adjustment. For instance, an asset could require an offset to be positioned correctly in a given building, but the same offset, even if corrected as described in our system, might lead to a separation between parts of the building coming from different rulesets, resulting in a lower geometric quality as the model would not be watertight. In any case, these small adjustments are quite simple to perform. All the examples in this paper and the companion supplemental video were rendered *without* any manual user intervention.

Another related aspect that we would like to emphasize is that the copy & paste system does not free the user from any responsibility during the editing process. For instance, if the user copies a floor and pastes it into a whole facade, the result would be a very stretched floor, which probably is not what the user had in mind, but is the logical consequence of the choices made. As an example, Fig. 10 shows a resulting model where the top windows were copied from a *Subdiv* rule and pasted into a large part of the facade. For these cases two kinds of strategies can be used. One of them is, as explained before, to require manual intervention to change the rules and parameters (add a *Repeat* rule in this example). The other workaround that we came up with would be to implement both a generalized *undo* operation, plus adding the possibility for the system to add, upon user-confirmation, *repeat* nodes in either the horizontal direction, the vertical direction, or both. We leave this for further research, as guessing user intentions is beyond the scope of this paper.

6. Conclusions and future work

We have explored new alternatives for visual editing procedural buildings. In particular, we have developed a copy & paste mechanism that would allow non-technical users to reuse whole rulesets from existing ones, without the burden of any manual intervention. Up to now, users cannot seamlessly and transparently perform these actions. However, our copy & paste application can be smoothly integrated in any of these systems: Our current implementation is a few hundred lines long and can be immediately incorporated into any rule-based system like the commercial ones mentioned before. Our system only requires a small and intuitive input from the user: just to select the area to copy, the corresponding target area, and the system automatically produces a new ruleset that includes the selected modifications. We believe that this greatly enhances the artist toolbox, who is now able to obtain elaborated procedural models in only few minutes and without requiring to program rules. All complex manipulations are automatic and transparent to the end user.

As mentioned above, trying to infer the user's intention for any copy & paste operation is not a trivial task, but is worthwhile exploring, as it would increase the friendliness and usability of the whole system. Also, after the initial step done is here, it is possible to devise a ruleset-merging mechanism for buildings much in the same line as Lipp et al. [9] did for whole cities.

Acknowledgments

This work was partially funded by the TIN2010-20590-C02-02 project from Ministerio de Ciencia e Innovación, Spain. Our buildings are based on the Raccolet house and Urban Sprawl models from Daz3D (http://www.daz3d.com/).

Appendix A. Supplementary material

Supplementary data associated with this article can be found in the online version of http://dx.doi.org/10.1016/j.cag.2013.01.003.

References

[1] Müller P, Wonka P, Haegler S, Ulmer A, Van Gool L. Procedural modeling of buildings. ACM Trans Graph 2006;25(3):614–23.
[2] Watson B, Müller P, Veryovka O, Fuller A, Wonka P, Sexton C. Procedural urban modeling in practice. IEEE Comput Graph 2008;28:18–26.
[3] Nealen A, Sorkine O, Alexa M, Cohen-Or D. A sketch-based interface for detail-preserving mesh editing. ACM Trans Graph 2005;24(3):1142–7.
[4] Autodesk, Meshmixer, 2012. 〈http://www.meshmixer.com〉.
[5] Funkhouser T, Kazhdan M, Shilane P, Min P, Kiefer W, Tal A, et al. Modeling by example. In: ACM SIGGRAPH 2004 Papers. SIGGRAPH '04; 2004, p. 652–63.
[6] Lipp M, Wonka P, Wimmer M. Interactive visual editing of grammars for procedural architecture. ACM Trans Graph 2008;27(3) 102:1–10.
[7] Wonka P, Wimmer M, Sillion F, Ribarsky W. Instant architecture. ACM Trans Graph 2003;22(3):669–77 (Proceedings ACM SIGGRAPH 2003).
[8] Vanegas CA, Aliaga DG, Beneš B. Interactive design of urban spaces using geometrical and behavioral modeling. ACM Trans Graph 2009;28(5): 111:1–10.
[9] Lipp M, Scherzer D, Wonka P, Wimmer M. Interactive modeling of city layouts using layers of procedural content. Comput Graph Forum 2011;30(2):345–54 (Proceedings EG 2011).
[10] Vanegas CA, Aliaga DG, Wonka P, Müller P, Waddell P, Watson B. Modelling the appearance and behaviour of urban spaces. Comput Graph Forum 2010;29(1):25–42.
[11] Esri. Cityengine. 2012. URL 〈http://www.esril.com/software/cityengine〉.
[12] EpicGames. Unreal development kit (udk). 2012. URL 〈http://udk.com〉.
[13] Patow G. User-friendly graph editing for procedural modeling of buildings. IEEE Comput Graph 2012;32:66–75.
[14] Riu A, Patow G. Bringing direct local control to interactive visual editing of procedural buildings. In: XXII Spanish Computer Graphics Conference. 2012. p. 67–75.
[15] Krecklau L, Kobbelt L. Interactive modeling by procedural high-level primitives. Comput Graph 2012;36(5):376–86.
[16] Prusinkiewicz P, Lindenmayer A. The algorithmic beauty of plants. New York, NY USA: Springer-Verlag New York Inc; 1996 0-387-94676-4.
[17] Lau M, Ohgawara A, Mitani J, Igarashi T. Converting 3d furniture models to fabricatable parts and connectors. ACM Trans Graph 2011;30(4):1–6.
[18] Heckel R. Graph transformation in a nutshell. In: Electronic Notes in Theoretical Computer Science, Elsevier; 2006. p. 187–98.
[19] Rozenberg G., editor. Handbook of graph grammars and computing by graph transformation: volume I. foundations. River Edge, NJ, USA: World Scientific Publishing Co., Inc.; 1997. ISBN 98-102288-48.
[20] SideFX. Houdini 12. 2012. URL 〈http://www.sidefx.com〉.