CEIG - Spanish Computer Graphics Conference (2022)
J. Posada and A. Serrano (Editors)

Reusable procedural building parts

Alejandro Arangua, Gustavo Patow and Gonzalo Besuievsky

ViRVIG, Universitat de Girona, Spain

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 creating a library of reusable procedural parts in a rule-based setting to address this need. No trivial extension exists to perform this action in a way such that the resulting ruleset is ready for production. For procedural reusable parts, we need to handle the rulesets extracted from the source graphs, and later on, merge them with a target graph to obtain a final consistent ruleset. As one of the main contributions of our system, we introduce a library of reusable parts that could be seamlessly glued to other graphs and obtain consistent new procedural buildings. Hence, we focus on intuitive and minimal user interaction, and our editing operations perform interactively to provide immediate feedback.

1. Introduction

A broad range of areas, such as games, movies or urban simulation require virtual 3D city models with detailed geometry. Procedural modeling [MWH*06] has proven to be quite effective, 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 [NSACO05], drag-and-drop mesh tools [Aut12] or the modeling-by-example approach [FKS*04] 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, which is a direct continuation of a previous work [BBP13], we focus on a specific editing application: reusable building parts (or modules) for procedural architectural models. One of the main motivations for choosing this application is to provide a simple but flexible library of building elements, ranging from a simple window or door up to a whole facade, that would allow non-experts to generate new content using pre-existing content. See Figure 1. We present here a complete, end-to-end system for procedural library of building parts, which can be integrated with our previous system that allows selection, storing, loading and compositing.

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 re-configuring old ones. Similar to [LWW08], 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. Our main contributions are: providing an interactive visually-based method for editing models based on procedural architectural and introducing a file format for procedural building parts that can be saved and reuse as an architectural library.

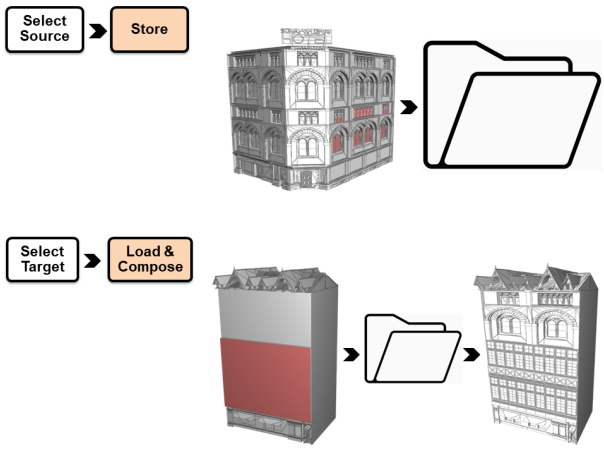

Figure 1: *Reusable procedural architectural parts workflow system. From procedural buildings selected sources, a file is generated containing all the procedural information needed to reconstruct this part (top). At editing time, the artist selects a target, then loads the required parts from the library, and finally a new building is composed (bottom).*

DOI: 10.2312/ceig.20221143

2. Previous Work

The idea of using a library of parts that could be copied & pasted as to modeling tools is not new. However, classical modeling approaches are not 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. Lipp et al. [LWW08] presented a first attempts to improve editing operations for procedural buildings using an interactive visual system, avoiding text editing rules. Later, some approaches tried to bring together direct control and visual procedural languages, resulting in a simple visual traversal of the hierarchy tree plus direct visual assignment of the desired changes [RP12]. However, results only reduce the user options to only a few simple operations [Pat12]. In this paper we provide a completely visual application to store and retrieve procedural building parts to simplify the user's editing tasks.

Our work is based on graph-grammars and graph rewriting techniques. 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 [LSWW11] or the building rulesets [Pat12]. Here we also perform graph-rewriting operations over the rulesets, but in contrast to [Pat12], 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.

This paper extends a previous copy & paste work [BBP13], where a methodology for easy editing procedural buildings was proposed. Here, we include also an intermediate library of procedural architectural elements, which can be saved to disk and later reused in any design project, thus greatly increasing effectively by creating a portfolio or parts that can be designed, created and then reused in any subsequent project.

3. Procedural Modeling

The seminal works by Wonka et al. [WWSR03] and Müller et al. [MWH*06] 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. The resulting geometry is formed by shapes that can be optionally assigned new labels with the purpose of being further processed. In our system, geometry carries all labels that the shape or any ancestor has received during the production process. 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.

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 [Pat12] (see Figure 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),

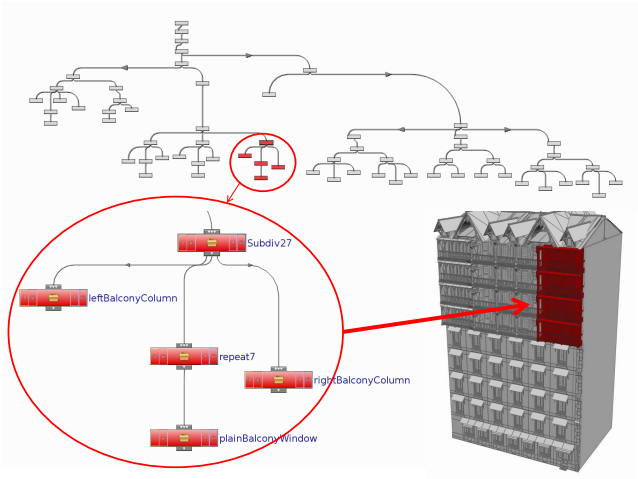

Figure 2: *A full graph-based representation (top) of a procedurally modeled building (bottom, right). The red subgraph (bottom, left) represents the red part of the facade.*

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) that have the given predecessor label. Let $e^i_j \in E$ be the edge connecting the output of node n_i to the input of a downstream node n_j.

4. A Library of Reusable Procedural Parts

Our system for saving procedural building parts allows a user to select objects from several procedural source buildings and save them to an asset library. The full workflow is shown in Figure 1. Input to the system is roughly the user interacting with tools like CityEngine [Esr12] or SideFX's Houdini [Sid12]. The system can be divided into two stages: Selecting and saving the procedural selected part into a file in XML format; and loading a file for the final composition.

4.1. Select and Store

The first step is to select a part of a facade, which is identified by traversing the primitive hierarchy. We provide two different starting points for this navigation: by defining a selection window with the mouse over the geometry the user wants to select, or by directly selecting a leaf by a point-and-click action. Then, the user can visit any primitive children, browse through its siblings, and go back to the parent primitive as desired [RP12]. Some selection examples can be seen in Figure 3.

Once the user selected a set of primitives $\{p\}$, either as a result of this navigation through the primitive tree or by direct point and click, we can precisely locate the common ancestor n_p, both at the tree level and at the graph level. For this, all the possible paths to n_p are stored, each in a separate stack. Then, the algorithm starts to traverse the graph from the root, updating each stack for each iteration with the labels of the nodes it is traversing. Being all the

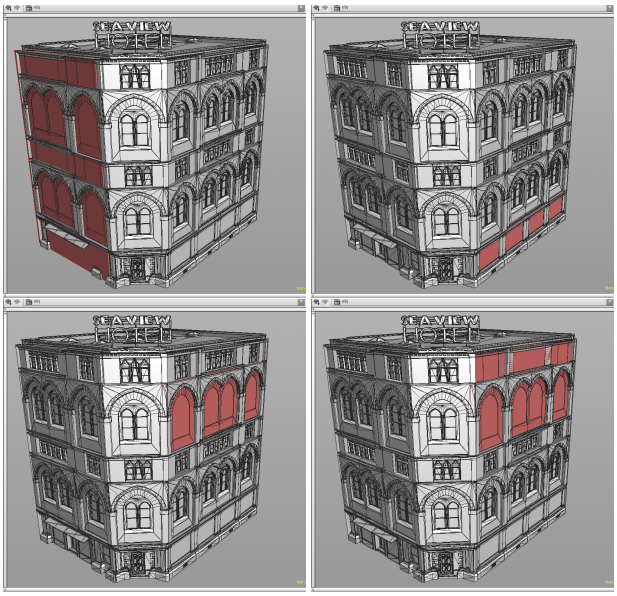

Figure 3: *The selection system with four examples. The visual interface allows to navigate through the primitives of the tree for interactive selection, and then save the selected elements to disk.*

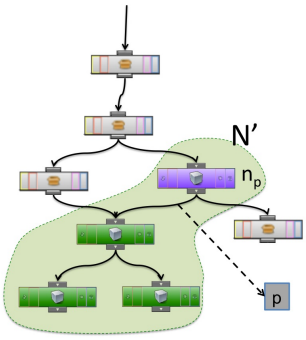

Figure 4: *Selection process. The selected node n_p (lilac node) and all its descendants (green nodes) are saved to disk, together with the necessary connecting information.*

paths updated, it performs a set operation on the labels of the current level, to remove the repeated entries. The algorithm stops when the set operation returns an empty set. Note that, with this method, there can be several common ancestors at the same level.

After n_p is found, we collect all nodes in the source graph G^S that operate on n_p or any of its descendants. See Figure 4. For that, we traverse the primitive tree in depth-first order, starting from n_p, until we reach the leaf primitives. For every descendant d of n_p, we collect the node $node(p)$ 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 $G' = (N', E')$. All the information is stored in a XML file, containing all the nodes N', edges E', the proportions meta-data, together with other attributes and actions in G'.

4.2. Load and Compose

In a later stage, the user selects where some loaded geometry should be placed on the target building, using the same selection tool. After the selection, the common ancestor algorithm is used to find the primitive n_t and the graph is traversed from that node to get the $G^T = (N^T, E^T)$ on the target building.

To merge the graph G^T of the target building with the subgraph G' from the library file, we follow the rules of graph rewriting [Roz97, Hec06]: first, we identify the actions performed on n_t by G^T, and then separate n_t from G^T, as it is now going to be processed by G'. This can be done with an Exception node E [Pat12], which assigns a user-defined label to a primitive that fulfills a given condition; and a Selector node S, which deletes any primitive that does not have the adequate label. We connect $node(n_t)$ to E, and then S to E's output. Finally, we reconnect all nodes connected to the outputs of $node(n_t)$ to be the outputs of S. The parameters for E node are set to select n_t descendants. This way, all other primitives produced by $node(n_t)$ will continue being processed as before, with n_t set aside of the process.

The next step it to redirect t to the loaded subgraph G'. Again, we do this with another selector node S that is set to keep t and delete all other primitives. Now, we have to load and integrate the graph G' into the new graph, attaching the nodes in N' to the outputs of S. For that, we instantiate G', including all nodes in N' and their connections. Remember that we have kept information about n_p, the node in G^S that produced the chosen primitives p, so it will be composed along with the other nodes in G'.

The final step in the gluing process is to transfer all the useful connections from n_p to the last selector node created, S. 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. As we have stored the original proportions and sizes in the XML file, there is no need to manually adjust them to have a consistent result.

5. Results and Conclusions

Our system is implemented on top of SideFX's Houdini [Sid12]. using the buildingEngine system [RP10, Pat12]. Given that the implementation is done using embedded Python scripts and external Python methods, the presented framework is easily reproducible almost straightforwardly.

Figure 5 shows new buildings with facades created (or modified) using parts from an already existing procedural building. A typical modeling session using our system (see Figure 1) has the following sequential steps for each part of the building to be reused: In a first stage, the user navigates through the source building using the selection tool (see Figure 3), use the select button to store in an XML file. Then, in a posterior session, the user can navigate through a target building to select the part where to integrate one of the saved structures, and finally applies the compose operation to load and merge both structures.

The main contribution in usability of our approach is that it is an end-to-end process that runs without needing to write any rules or even neither changing any parameters: everything is done visually. We have explored new alternatives for storing and visual editing

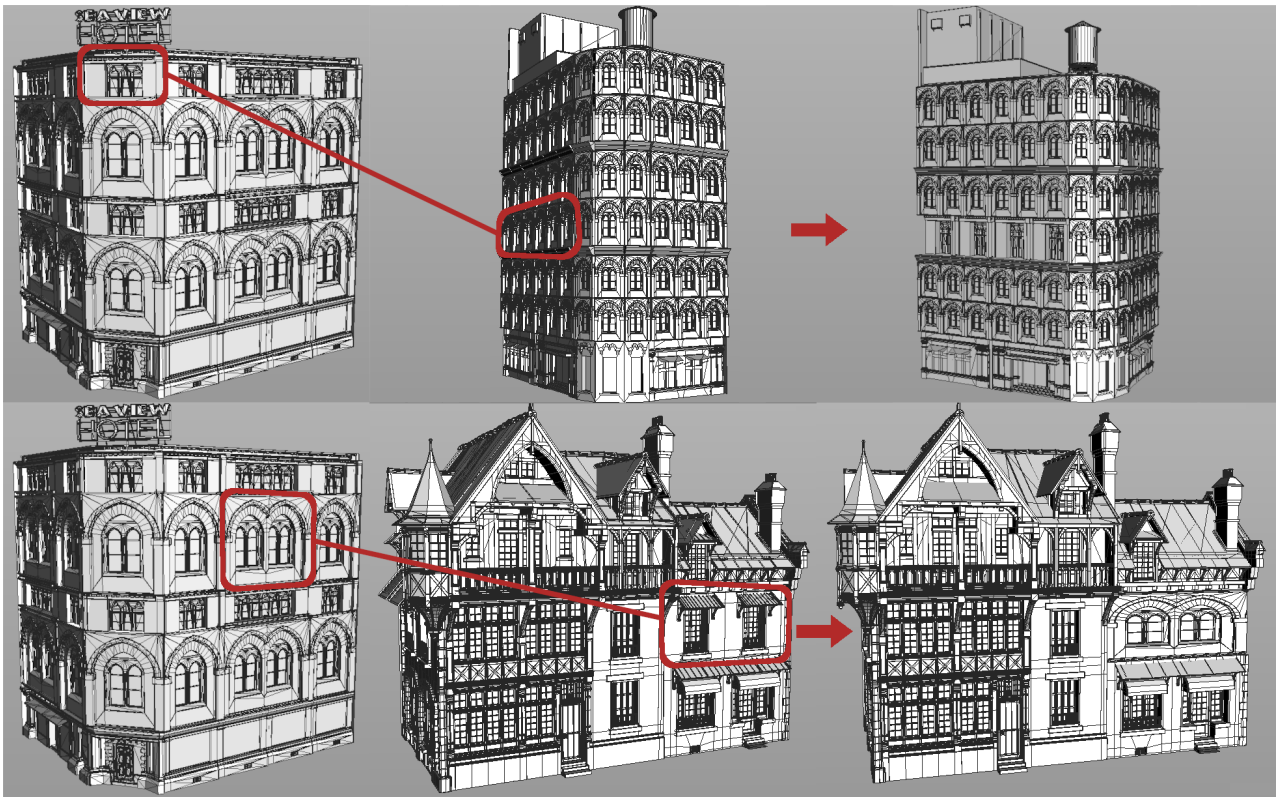

Figure 5: *Two examples where different parts from an existing building are saved to disk and later inserted into procedural buildings. Above: into an office building. Below: into the Raccolette house.*

procedural buildings, by developing a library of reusable procedural parts that would allow non-technical users to reuse whole rulesets from existing ones, without the burden of any manual intervention nor knowing any grammar.

Although we designed our system with the idea of freeing the user from tedious manual intervention at the ruleset level during the storing or loading and composing 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 were rendered without any manual user intervention.

Acknowledgements

This work was partially funded by project TIN2017-88515-C2-2-R from Ministerio de Ciencia, Innovación y Universidades, Spain. Our buildings are based on the Raccolet house and Urban Sprawl models from Daz3D (http://www.daz3d.com/).

References

[Aut12] AUTODESK: Meshmixer, 2012. URL: http://www.meshmixer.com. 1

[BBP13] BARROSO S., BESUIEVSKY G., PATOW G.: Visual copy & paste for procedurally modeled buildings by ruleset rewriting. Computers & Graphics 37, 4 (2013), 238–246. Special session on procedural modeling. doi:10.1016/j.cag.2013.01.003. 1, 2

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

[FKS*04] FUNKHOUSER T., KAZHDAN M., SHILANE P., MIN P., KIEFER W., TAL A., RUSINKIEWICZ S., DOBKIN D.: Modeling by example. In ACM SIGGRAPH 2004 Papers (New York, NY, USA, 2004), SIGGRAPH '04, ACM, pp. 652–663. 1

[Hec06] HECKEL R.: Graph transformation in a nutshell. In Electr. Notes Theor. Comput. Sci (2006), Elsevier, pp. 187–198. 3

[LSWW11] LIPP M., SCHERZER D., WONKA P., WIMMER M.: Interactive modeling of city layouts using layers of procedural content. Computer Graphics Forum (Proceedings EG 2011) 30, 2 (Apr. 2011), 345–354. 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

[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

[NSACO05] NEALEN A., SORKINE O., ALEXA M., COHEN-OR D.: A sketch-based interface for detail-preserving mesh editing. ACM Trans. Graph. 24, 3 (July 2005), 1142–1147. 1

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

[Roz97] ROZENBERG G. (Ed.): Handbook of graph grammars and computing by graph transformation: volume I. foundations. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997. 3

[RP10] RIDORSA R., PATOW G.: The skylineengine system. In XX Congreso Español De Informática Gráfica, CEIG2010 (2010), pp. 207–216. 3

[RP12] RIU A., PATOW G.: Bringing Direct Local Control to Interactive Visual Editing of Procedural Buildings. In XXII Spanish Computer Graphics Conference (Jaen, Spain, 2012), Eurographics Association, pp. 67–75. 2

[Sid12] SIDEFX: Houdini 12, 2012. URL: http://www.sidefx.com. 2, 3

[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. 2