Hindawi Publishing Corporation
International Journal of Computer Games Technology
Volume 2015, Article ID 808904, 15 pages
http://dx.doi.org/10.1155/2015/808904

Research Article

Procedural Content Graphs for Urban Modeling

Pedro Brandão Silva,[1] Elmar Eisemann,[2] Rafael Bidarra,[2] and António Coelho[1]

[1]*Faculdade de Engenharia/INESC TEC, Universidade do Porto, Rua Dr. Roberto Frias, 4200-465 Porto, Portugal*
[2]*Computer Graphics and Visualization Group, Delft University of Technology, Mekelweg 4, 2628 CD Delft, Netherlands*

Correspondence should be addressed to Pedro Brandão Silva; pedro.brandao.silva@gmail.com

Received 22 April 2015; Accepted 28 May 2015

Academic Editor: Hanqiu Sun

Massive procedural content creation, for example, for virtual urban environments, is a difficult, yet important challenge. While shape grammars are a popular example of effectiveness in architectural modeling, they have clear limitations regarding readability, manageability, and expressive power when addressing a variety of complex structural designs. Moreover, shape grammars aim at geometry specification and do not facilitate integration with other types of content, such as textures or light sources, which could rather accompany the generation process. We present procedural content graphs, a graph-based solution for procedural generation that addresses all these issues in a visual, flexible, and more expressive manner. Besides integrating handling of diverse types of content, this approach introduces collective entity manipulation as lists, seamlessly providing features such as advanced filtering, grouping, merging, ordering, and aggregation, essentially unavailable in shape grammars. Hereby, separated entities can be easily merged or just analyzed together in order to perform a variety of context-based decisions and operations. The advantages of this approach are illustrated via examples of tasks that are either very cumbersome or simply impossible to express with previous grammar approaches.

1. Introduction

Content creation is one of the most expensive factors for many game productions. In particular, urban modeling is an important challenge, as it has applications in various areas from city planning, training, and learning, to simulation, and entertainment. Unfortunately, creating large-scale urban areas by hand is a very complex task that quickly becomes unmanageable in cost and time. Although procedural methods have received much attention in game development, automating urban modeling remains a very difficult process, as it concerns the creation and integration of terrain, vegetation, roads and complex buildings [1], each involving particular representations and content types (meshes, lines, textures, lighting, etc.).

Grammar-based approaches [2–4] have proven useful for the generation of several kinds of pattern structures, yet their formal, textual representation is generally inadequate for artists [5]. A large variety of different rules have to be defined in order to achieve a fine-grained control and, for complex designs, even small changes may require redefining many grammar rules and writing new ones. Such large rule sets also lead to reduced readability and manageability: it becomes hard to find meaningful rule names, rule sequencing becomes hard to maintain, and the data flow becomes hard to follow (see Figure 2).

The initial steps in such grammars are typically top-down, sequentially dividing shapes entities to define local scopes. While each rule can produce multiple shapes, it can only operate on one individual shape at a time. This implies that, once split, separate entities cannot be merged back nor queried anymore as a whole set. This limits the expressiveness of the approach, that is, the range of ideas that can be communicated and represented, such as

(i) *clustering*, for example, to *assemble* buildings into a certain number of neighborhoods featuring different architectural styles or purposes;

(ii) *averaging*, for example, to find the *center* location of a set of buildings to divide them into downtown and peripheral areas;

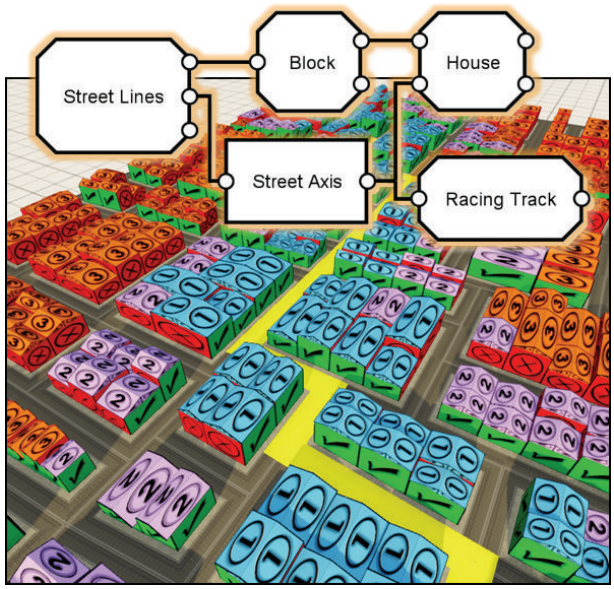

Figure 1: Our graph-based approach introduces context-awareness verifications which can be applied at any step of the generation process. Here, we perform visibility calculations to determine which façades are visible (marked in green) from a highlighted street (marked in yellow) and which are not (marked in red). We also calculate the minimum distance of each house towards that same street and decide on a level of detail (1, 2, or 3, as marked on the roofs) based on the distance. For applications focused on a main path, for example, racing games, this information could be used to guide the procedural content generation and include budget considerations in the construction of the virtual world.

(iii) *ordering*, for example, to create green areas on the blocks *closest* to a particular point of interest or location;

(iv) *finding maximum/minimum*, for example, to create the building garage door on the *largest* façade and the main door on the *smallest* one;

(v) *achieving uniqueness*, for example, to attach a chimney to *only one* of many candidate roof sections;

(vi) *merging*, for example, to merge corner walls of adjacent façades to build continuous balconies;

(vii) *context development*, for example, build the lot gate *right in front* of the building main door;

(viii) *visibility testing*, for example, to determine if a building detail will be seen from a particular location or path (see Figure 1).

Graph-based representations [6] facilitate the understanding of complex rule sets, but in the context of grammars, such solutions still have manageability or flexibility issues [7] and inherited the grammar limitations mentioned above [5].

We introduce *procedural content graphs*, a generic graph-oriented approach to specify content generation procedures. While still retaining the expressive power of grammar-based solutions (such as recursion or natural rule divergence), it extends them by addressing many of their shortcomings. In particular, we make the following contributions:

(i) a collective management of entities as lists or groups, for example, for sorting, advanced filtering, and aggregation operations;

(ii) a flexible and extensible framework featuring parameters, attributes, and so-called *augmentations* to create custom graph nodes with compact manipulation possibilities;

(iii) a visual graph-based framework for procedural content generation, supporting a transparent data flow control to manipulate and combine different data entities within a single pipeline.

We start by reviewing previous work (Section 2) and then present our graph-based approach (Section 3) by describing the specifics of entity representation and data flow management. Next, we present its main content manipulation and integration capabilities (Section 4), providing various examples that would have been cumbersome or impossible to express with grammar-based solutions. We continue with an explanation of our implementation (Section 5), containing further details of the designed framework and achieved flexibility, after which we discuss how our solution compares to other grammar and graph-based approaches (Section 6). Finally, we conclude and give an outlook on future work (Section 7).

2. Previous Work

Grammar-based approaches have proven very useful in the context of plant generation [8] and even extensions have been suggested to integrate environmental influence [9]. Yet, these approaches usually work in a bottom-up fashion, which is less intuitive when working on shapes such as buildings. The application of formal grammars to 2D shapes, so-called *shape grammars*, was introduced by Stiny and Gips [10]. Here, the process is rather top-down and shapes are typically substituted by more complex shapes that are then treated independently. Such an approach is well suited for regular man-made structures, such as façades, as illustrated by split grammars [2], wall grammars [11], and the CGA grammar [3]. The last approach has even been integrated in a commercial software CityEngine [12]. Krecklau et al. [4] introduced a generalization of such grammars, with the possibility to manipulate multiple types of nonterminal objects, as well as to pass nonterminal symbols as parameters into rules using the definition of abstract structure templates. Another grammar extension, presented in [13], introduced procedural scene illumination, defined in terms of lighting goals, luminaire installation sites, and constraints.

One important limitation of all these top-down approaches is that any splitting strategy leads to independent and unrelated parts. Building connections between constructed objects is basically impossible. By passing *attachment points* along [14], some cases, like bridge structures, could be handled. Another alternative was presented in the form of a stack-based programming language [15, 16]. However, in both cases, the approaches proved unattractive to use by designers and nonprogrammers. Building on this observation, simplified interaction schemes have been developed [17]. Here,

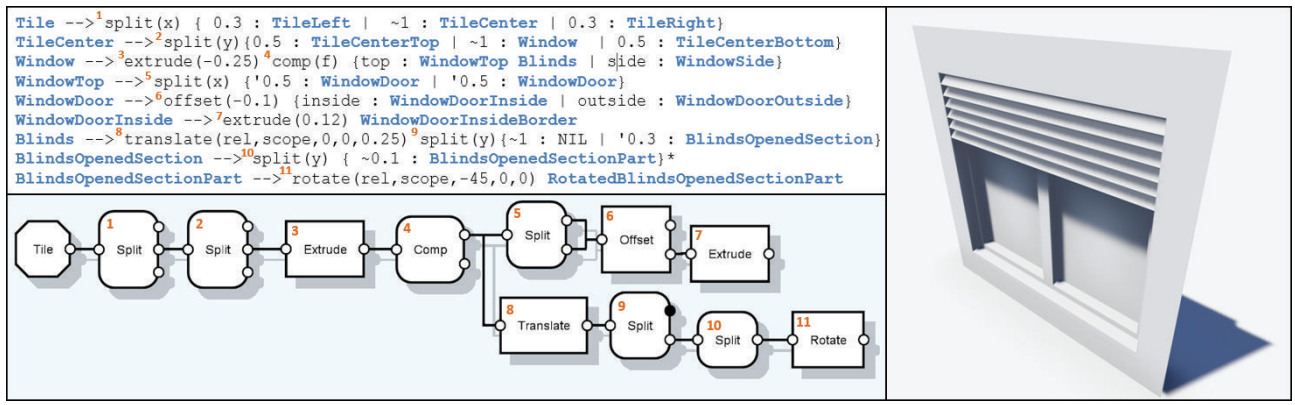

Figure 2: Window example modeled using a CGA shape grammar formulation (upper left; loosely based on [3] and [38]) and our procedural content graphs (lower left), each rule operation corresponding to a node with a matching number. The visual nature of our approach makes the flow of data easier to follow and does not require the user to come up with rule symbols (highlighted in blue) or geometric labels [6, 7], which generally make management difficult and error-prone.

high-level primitives are defined in a "Professional mode," which still requires editing a complex text-based grammar, but interactive customization and combination was possible within a "high-level mode." Interactive design frameworks have also been investigated to simplify grammar definitions [18], but they were still tied to the rule-based structure of the underlying grammar.

A different interactive alternative is the exploration of shape variations, layouts, and components [19, 20] and processing [21]. Grammars can also be derived from existing models [22, 23] and then interactively controlled [24]. However, these approaches have a different focus compared to our work; we provide a framework to guide and support content creation.

A popular alternative for the textual rule definition is the visual programming language paradigm [25], especially the one based on dataflow graphs [26]. Application examples include generation of terrain [27], trees [28, 29], textures/images [30, 31], animations [31, 32], or geometry in general [6, 32], all featuring their own specific manipulation metaphor. A framework that aimed at integrating several types of content was proposed in [33].

The work of Patow [7] is a general system with the goal of making shape grammars more comprehensive by employing a visualization in the form of a node-based system. Nonetheless, the use of the Houdini engine [6] makes it impossible to create cycles and therefore recursive procedures (an essential basis in grammars). In addition, this approach does not allow more than one output port, which could be used to control flow divergence. Instead, it resorts to label filtering, which introduces manageability issues for longer graphs, some of which have been alleviated by [34]. Also, while it is possible to create high-level digital assets, it is not possible to introduce new types of semantic entities and their high-level nodes inherit the same (and other) port limitations. Other node-based systems [5] address these issues, yet they still share the initial limitations of shape grammars. Our work integrates the power of grammar approaches but addresses their shortcomings with the manageability, flexibility, and high expressiveness of visual node-based solutions.

3. Our Graph Approach

The basis of our approach is a *procedural content graph* (PCGR), where nodes and edges describe the procedures and the data flow, respectively. In this section, we will first give an overview of the graph architecture, the handled data, and the execution process. We will then discuss novel concepts such as the control over lists and augmentations and, finally, explain how we can manipulate attributes and encapsulate graphs.

3.1. Graph Structure. A procedural content graph is represented by a directed, cyclic graph $G = (N, E)$ where N are the nodes and E the edges. The graph topology encodes the dataflow process for content generation: edges are the carriers of content, while the nodes are their operators. Nodes represent *procedures* that act (e.g., transform, analyze, or filter) on the incoming data; we will thus use both terms interchangeably.

Different types of data can flow and coexist within the same graph. Their manipulation possibilities depend on the availability of procedures that handle such data. Nodes have typed input/output ports, to which incoming/outgoing edges can connect as long as the connected ports are of compatible data types.

The graph structure is meant to allow the sequencing of procedures, where cycles represent recursive operations. After a procedure is executed, its output data is transmitted via the output ports along the connected edges to the subsequent input ports, where the data is stored in a queue. Procedures are executed in several *rounds* until their data queues are depleted.

We distinguish several node types (to be explained in further sections), which use slightly different representations in our figures. The rectangle, rounded-corner rectangle, and

octagon refer to standard, encapsulated (Section 3.7), and augmentation nodes (Section 3.8), respectively.

3.2. Ports. There are 2 types of input ports: *single* input ports (represented with round outline) consume one data object per round. *Collective* input ports (represented with square outline) will consume the whole object queue, handling it as a list. This introduces the possibility to treat sets of objects instead of individual ones. A node that features at least one collective input port is called a *collective node*, as opposed to a *single node*.

Several edges can connect to one input port (a *convergence* port) and arriving elements are queued for execution. If several edges leave an output port (a *divergence* port), the outgoing data is copied. The result of executing a graph is obtained by collecting all the data emanating from all unconnected output ports or *leaf* ports. Ports can be blocked (represented with dark fill) in order to discard its results from the content pool. Blocking is also applicable to nonleaf ports or even edges, which can be useful to temporarily (for instance, for debugging) or conditionally prevent the flow at a particular location.

Unlike [5, 7], our approach does not pose any restrictions on the number of ports of a node: it can feature zero, one, or more input and output ports. The reasoning is the following: multiple input ports enable handling of several entities at once, which is crucial to combine different data types and perform context-dependent modifications. On the other hand, multiple output ports allow us to apply a filtering operation inside of the node, which is essential for nodes that split entities or simply need to separate the flow based on a certain condition or type of result. In this way, the dataflow filtering is encoded in the graph topology, instead of resorting to rule symbols [3], labels [7], or tags [17]. These previous solutions can make the development more tiresome, less readable, less manageable, and more error-prone, especially for more complex designs (Figure 2).

3.3. Entities. The primary type of information manipulated within a graph, transported through the edges between ports, is organized in form of *entities*. As their name implies, they represent independent and self-contained objects, carrying their own specific semantics.

For instance, to produce and manipulate geometric 3D data, a boundary representation (b-rep) featuring polygonal faces, edges, and vertices is adequate for most modeling operations. Yet a set of nodes for crossings, connected by edges to nodes for streets, is a far more concrete and appropriate data structure for representing street networks. The flexibility to introduce and manipulate any kind of data types means that specific procedures can be developed to optimally deal with them. For instance, by using "terrain" and "street" entities, one can apply algorithms, such as the ones described by Kelly and McCabe [35] or Galin et al. [36], which operate over such specific data representations.

Although each entity is expected to have its specific features, all entities share the possibility to incorporate custom *attributes*. Attributes are represented via a hash map in a key-value fashion and enable us to store properties in an entity dynamically, extending its initial design. This process makes it possible to attach properties, such as "name," "index," and "amount," which may only be relevant and applicable within the context of a particular graph. When an entity instance is used to create new instances (such as a split that cuts one shape into multiple ones), the new ones are called its *descendants*, while the original entity is considered an *ancestor*. Attributes are always copied from the ancestors to their descendants.

Entity types can also derive from others, following an inheritance pattern, which is relevant during the creation of procedures that operate on entity supertypes and not just specific types. For instance, node ports of type "entity" can accept any type of entity data.

An entity can typically aggregate other entities. For instance, "b-rep meshes" contain vertices, edges, and faces, which can be separated and processed individually throughout the graph. Likewise, meshes can also be aggregated within other meshes, recursively. This recursive construction allows us to group elements (e.g., a city entity can contain a list of region entities, which in turn contain houses, all of which are represented by mesh data).

3.4. Execution. The sequence in which nodes should be executed is mainly determined by the topological order of the directed graph, in the sense that the sequence of nodes is based on their dependencies (see Figure 3). Hence, the first nodes to be executed are those featuring no input ports (called *source nodes*). For the ordering of the other nodes, the order is topological, meaning that for every edge NM that connects an output of a node N to an input of a node M, N should be executed before M. After this order has been precalculated, the execution of the graph follows a data-presence protocol. A node can only be executed if each of its input ports has at least one entity in the respective input queue. The execution process runs as follows:

(1) Add the source nodes to a node queue Q (according to an order which can be user-specified).

(2) While Q is not empty:

  (a) Dequeue a node N from Q.

  (b) Execute N.

    (i) Source nodes are executed once.

    (ii) Single nodes will run several times, each time popping one entity from each of their input ports and executing a round with those entities, until at least one input queue is empty.

    (iii) Collective nodes will be executed once, since the list of entities will be emptied in one go. This might lead to some entities being left at single input ports.

  (c) Pass on entity data from connected output ports of N to the subsequent input ports of nodes M, according to established edge connections.

  (d) For all nodes M connected to an output port of N, we verify if all their input port queues are

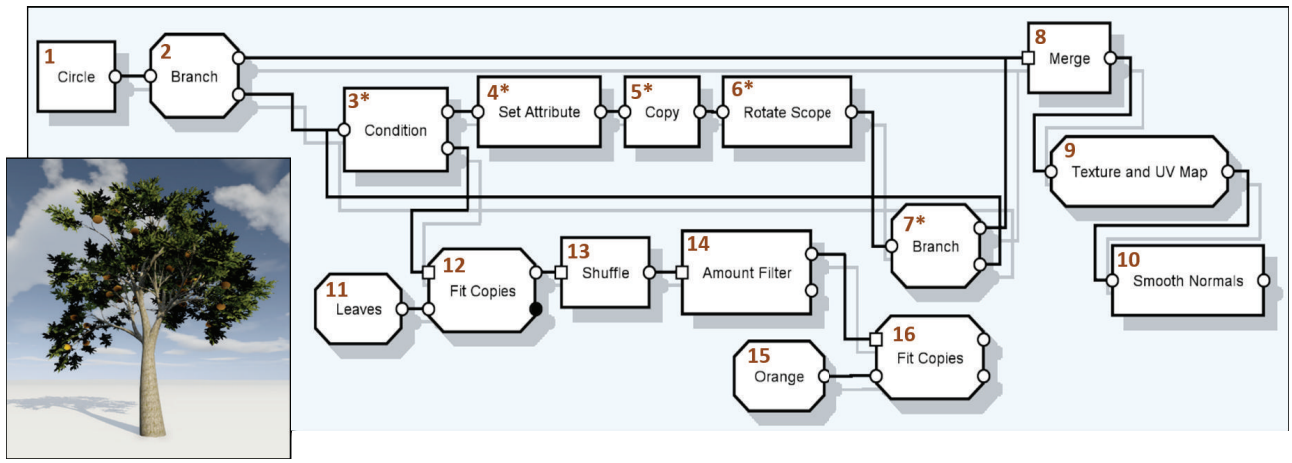

FIGURE 3: Orange fruit tree generated using a cyclic PCGR. The node numbering indicates the topological ordering of the nodes and the ones marked with an asterisk are bound to be executed multiple times, as they are found within a loop.

no longer empty. If all are filled, M is ready to be executed and therefore is added to Q, placed according to the topological order.

(3) Retrieve all entities from nonconnected output ports for storage or display. As mentioned before, entities can also be discarded from the final result by setting the state of their output ports to blocked.

3.5. Recursion. Topological ordering is only possible when graphs are acyclic. This means that, for recursive graphs, cycles have to be identified and some edges are hidden from the topological sorting algorithm. The algorithm to find such edges proceeds as follows:

(1) Mark all edges as "unvisited" and "noncyclic."

(2) For each graph node N, perform a depth-first graph iteration starting with an empty list of visited nodes V:

  (a) Add N to V.

  (b) For each "unvisited" outgoing edge of that node, check the destination node M.

    (i) If M is contained in the list of visited nodes, mark the edge as "cyclic."

    (ii) Otherwise, mark the edge as "visited" and execute recursively at (a) with M and a copy of V.

After these steps the process can proceed as before, with the single exception that, for the topological sorting, the edges marked as "cyclic" are not considered.

However, during the graph execution the edges are considered. Nodes connected to an output port of another node are added all the same to Q even if the edge connecting them is cyclic. Again, the addition to the queue should follow the topological order.

3.6. Procedures, Parameters, and Attributes. The configuration of a procedure's behavior towards the input entities is performed via its *node parameters*. These can be initialized in several ways: with so-called *graph parameters*, which are constant and reusable within the scope of the graph; with a fixed value (a numeric value, a character string); with an attribute of an input entity; or with a more complex expression involving arithmetic operations or function calls (e.g., sin(x), rand( ), length(string), etc.) on any of the former.

In order to employ the aforementioned attributes in a PCGR (see Section 3.3), the user has to define all attributes that entities will carry inside that graph. The attribute declaration (where name, type, and default value are stated) defines a key that is used to access the corresponding value in the entity's attribute hash table. In practice, an explicit storage of the value is not required if the value matches the default value of the attribute. Simply, if the search in the hash table fails, the default attribute value is used. In this way, all entities in the graph can be considered to be always carrying all defined attributes.

Attributes can be read from and written to inside of nodes. As entities flow through nodes or even specific ports, the corresponding value of an entity's attribute key can be accessed and/or changed. Such nodes have their own attribute declarations, whose keys can be mapped to graph attribute keys. For example, the "count" node in Figure 4 takes a list of entities, determines the total number of elements, and assigns the index of each element to the "index" attribute of those entities. By mapping its "index" attribute to the "circle index" attribute of the graph, the value assignment is propagated from the node scope to the graph scope. If one would choose not to perform this mapping, the value of the "index" attribute would be discarded and the "circle index" value would remain unchanged. An inverse attribute propagation, that is, from the graph scope to the node scope, is also possible. This is essential in procedures such as attribute aggregation, where its values must be read in

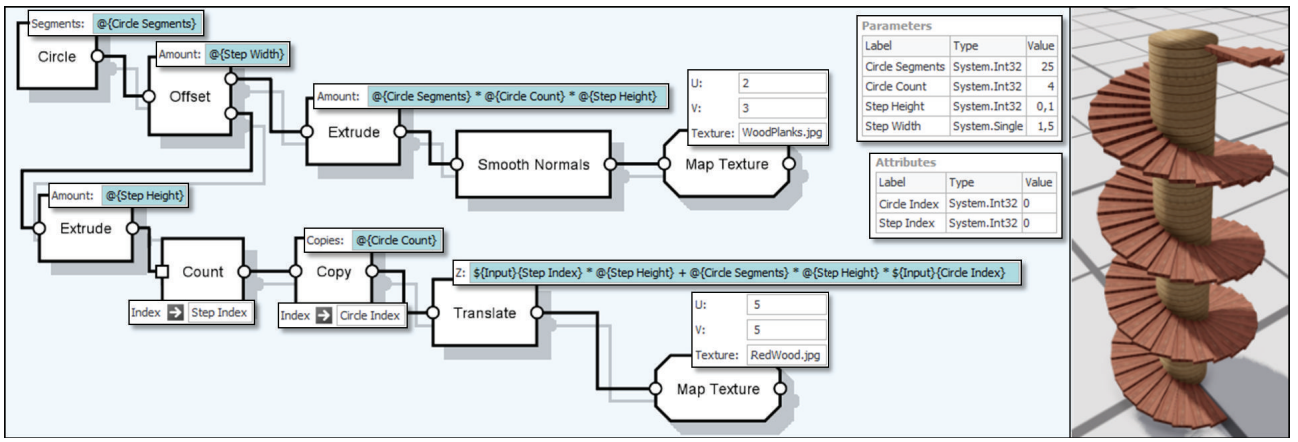

Figure 4: Construction of a stairway using attributes. An offset is applied to a circle to extrude polygonal steps. These are counted and assigned an index attribute. The steps are copied and an additional copy index is assigned. The steps' translation is steered via these attributes. The boxes above the nodes indicate the assigned fixed values (in white) or expressions (in blue). The notation @{*parameter label*} is used to refer to graph parameters, while ${*port label*}{*attribute label*} is used to refer to an attribute of an entity coming from a certain node port. The boxes with arrows under the nodes indicate attribute mapping.

order to perform averaging, minimum, maximum, or other operations.

Although at a first glance it might seem cumbersome to provide attribute definitions for an entire graph, there are several important advantages to it. First, the user keeps an overview of all attributes that entities carry, which is often convenient. Second, it is easier to optimize memory usage, because most intermediate values that nodes could attach to entities might never be used afterwards. Third, it releases the need for namespace handling, as attribute keys are unique and tied to the scopes of nodes and graphs.

3.7. Encapsulation. Certain graphs might often be reused as subgraphs within other graphs, and our approach allows a simple transformation into a node, an *encapsulation*.

The operation to transform a graph G into a new node, N, ready to be used in a supergraph S is relatively straightforward (Figure 5). Any input/output port of a node in G can be marked as a *gate* (usually accompanied with giving it a meaningful name). A gate will serve as a port when this graph is used as a node in a supergraph. Graph parameters of G become node parameters of N by default, yet they can be hidden, if so desired. The same applies to attribute declarations in G. Hereby, the attributes in S can be transferred to G and/or forwarded back from G to the scope of S.

Regarding execution, a subgraph will always be executed as long as possible and, only when its execution queues are empty, will the supergraph continue execution. As a result, this complete local execution of encapsulated graphs can also be helpful to control the execution order, in cases where this is needed. Once the encapsulated graph finishes execution, all attributes that have been defined within this subgraph (but not those mapped to the supergraph) are removed from all entities before proceeding to the supergraph.

Encapsulating a graph and properly organizing its parameters, attributes, and ports to define more complex structures facilitates the definition of models with richer semantics. By successively encapsulating graphs, one can achieve a higher level of control where the manipulated nodes can represent increasingly complex architectural structures (pillars, windows, balconies, doors, etc.) instead of low-level operations. This concept is well discussed by Silva et al. [5] and well advertised by side effects [6]. However, their restriction on a certain amount of ports per node (the former has a one input port limit and the later has a one output port limit) reduces the expressiveness and semantic potential of the node, given that it is at the port level that one can distinguish which kinds of structures are accepted, as well as output.

3.8. Augmentations. By default, procedures have a static *signature*; that is, they offer a fixed set of available control parameters, attributes, and ports. Yet, an operation such as the split requires the enumeration of several slices and the customization of each one. In CGA shape grammars, each slice carries information about size, flexibility, and output rule symbol. Only by having all such information at once can the procedure precalculate the remaining available splitting size and adjust each slice size accordingly.

The generic way for PCGRs to deal with such flexible design annotation consists of *augmentations*, which can be seen as extensions to the nodes. An augmentation is a node structure that aggregates parameters, attributes, or ports. Again, for the split node example, an augmentation is the structure that holds the information about the slice and contains the output port where to send the result. Any number of such augmentations can be added to a node, and doing so affects the number of its output ports.

Generally speaking, augmentations are useful to indicate *lists* of constraints or guidelines that a node should consider in

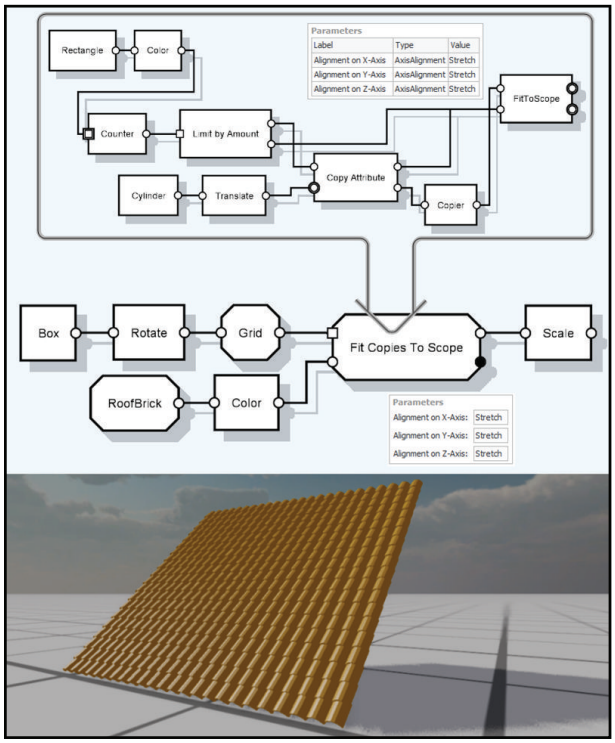

Figure 5: Encapsulation of a graph that extends the "fit-to-scope" operation to lists. The thick-line ports indicate the gates of the graph, which are mapped to the encapsulated node's ports. Graph parameters are also mapped to the node's parameters. The supergraph uses this encapsulated graph to easily copy and fit a brick rooftop into each cell of a rectangular grid (both brick and grid are also encapsulations). The result is a detailed roof surface.

its operation. In most cases, the order of the listed augmentations defines priorities in the node configuration and changes may yield different results. Consider the following examples, for which augmentations are especially useful:

(i) *Merging and Grouping*. Enumerate criteria for merging and grouping, for example, based on attributes of the entity (size, material, etc.).

(ii) *Ordering*. Enumerate criteria for sorting the input data in a descending or ascending order, for example, based on attributes of the entity (size, material, etc.).

(iii) *Splitting/Decomposition*. Define the size/criteria of split slices and provide additional ports to output them.

(iv) *Stochastic/Conditional decision*. Decide upon control flow according to sets of probabilistic or attribute-based conditions.

(v) *Creation*. Enumerate vertices of a polygon (or mesh) to be assembled.

(vi) *Custom Import/Export*. Load/save custom attributes that may be associated to the geometric data from/to a database, for example, geospatial data.

By their nature, augmentations cannot be reduced to a simple sequence of nodes or encapsulation. This is easier to understand, for example, for operations that require some kind of precalculation (e.g., split), evaluation (e.g., condition), or any other "atomic" organization (e.g., grouping or sorting).

4. List Manipulation and Data Integration

Having introduced the basic functional aspects of PCGRs, this section will focus on several content manipulation and integration possibilities that have hardly been addressed, if at all, in previous approaches.

4.1. Grouping, Merging, Unification, and Clustering. Shape grammar approaches use a top-down approach, transforming one simple entity into many complex ones, but there are many cases where joining back entities into single ones is not only convenient, but also necessary to achieve certain designs. In a PCGR, data is organized into entities that can be assembled or decomposed as manipulation needs arise. In this sense, the gather nodes are the key to the accumulation and congregation of entities. Hereby we define *grouping* as the process of collapsing various entities into a single container entity, according to certain criteria, such as spatial distribution, matching properties, or common attributes. Grouping works primarily as a means to organize and structure entities, building a hierarchy, but one which can be ungrouped at any point; hereby, the original entities are recoverable.

In the geometric domain, we additionally distinguish the concept of *merging* and *unification*. Merging is the process of gathering all faces, edges, and points of different shapes into a single shape entity, while unification does the additional step of finding and connecting common faces, vertices, and edges. A useful application of these operations is portrayed in Figure 6, featuring balconies stretching across corners, a recurring issue hardly achievable using shape grammars. The façades are split into a grid of separate tiles, each containing information about their X-Y index within the grid of the respective façade. These indices are used to conditionally filter the leftmost and rightmost tile from each façade, for nonground floors. The "group" node then assembles all these tiles by floor. Within each group, tiles are merged and unified if they have overlapping edges (using the "adjacency merge" node). Having the corner faces together within the same shape, the extrusion for corner vertices can be done according to the sum of the faces' normal, creating the seamless result intended for such balconies.

Spatial clustering is another form of grouping that joins elements according to spatial proximity, as illustrated in Figure 7. Given lots of the peripheral area of the generated city, 6 random lots were selected as initial centroids and, using a k-means algorithm, the remaining lots were sorted according to their proximity to these cluster centroids. The obtained clusters were then used to construct alternating industrial and residential neighborhoods, with green areas located around some cluster centers.

4.2. Aggregation. One of the possibilities that emerge from the control over sets of objects is the application of

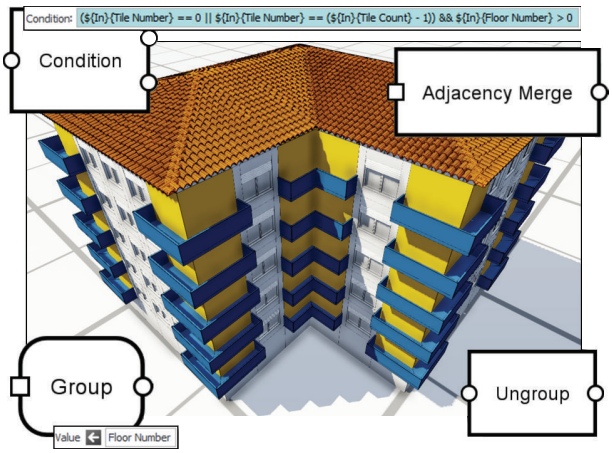

Figure 6: Building featuring balconies that stretch across façade corners, an example that cannot be naturally achieved using shape grammars.

aggregation functions—minimum, maximum, average, and sum—over entity properties or attributes. They are frequently used with condition or grouping nodes to filter and organize entity sets.

Figure 7 displays a complex example featuring an extensive urban environment with thousands of buildings. The starting point was a list of street blocks, each defined as a shape entity. For each one, the geometric centroid was calculated and stored as an attribute. Using the "aggregator" node, the whole set of blocks was gathered, the centroid attribute taken as a value to average, and the result stored as the "city centroid," the city center. The distance calculation to the center is performed for each block individually but, in order to calculate the relative distance (a value between 0.0 and 1.0), the maximum distance was first determined using again an aggregation node. City zoning was then determined by this relative measure: ≤0.3 for the downtown area, ≤0.45 for the commercial area, and the rest for the peripheral area. The height and style of buildings are random within the range acceptable for the assigned zone.

After the aforementioned clustering operation for the peripheral area was applied, the center of each cluster was calculated using the same aggregation method.

4.3. Amount Filtering and Counting. A very important flow control feature, unaddressed by shape grammars, is the ability to determine the amount of entities flowing through a particular point of the graph. This measure is especially important for filtering entities according to absolute or relative quantities. It is also required when a sequential numbering or alphabetization of entities is to be achieved.

For the example in Figure 7, exactly 10 blocks per cluster were picked to build green areas. The "amount filter" node introduces this possibility, as it gathers entities as a list and isolates the first n entities, starting at a given index i, and forwards them to its first output port, while the remaining ones are sent to the second port.

The need to isolate one entity from a set occurs often when a particular detail is to be introduced at a given point, for example, placing the main door at one of many candidate façades of a house. This is illustrated in Figure 8, where a common rule has to be found to address the various possible building shapes. The main door should face the street, yet many options exist (red, yellow, and blue arrows). Deciding upon the smallest façades (yellow and blue arrows) reduces the number of possibilities, but to ensure that only one door is created, the "amount filter" node is required, so as to pick the first element from the list.

4.4. Ordering, Reversing, and Shuffling. The order in which entities flow throughout the graph may be determinant to achieve a specific result. The "cluster" procedure (Figure 7), for instance, automatically picks the first n entities as source for the initial cluster centroids. As such, shuffling the city blocks beforehand ensures some randomness in the clustering process.

The "order by" augmentation node can sort entities by given attribute combinations. When used in conjunction with "amount filter," it can be used to filter the n smallest façades (Figure 8) or the n closest blocks to the cluster center (used for green area determination in Figure 7). Likewise, obtaining the largest façades or most distance objects could be obtained through the "reverse" node that inverts the entity list order.

4.5. Context-Sensitive Design. The generation process should depend not only on the properties of each entity, but also on its context. In PCGRs, this is best addressed using several input ports, each carrying data according to a specific meaning. Figure 1 portrays an environment built around this concept, where the level of detail of each building is determined by its distance to the main path (using a similar approach as Section 4.2). On the other hand, the decision on whether to design each façade depends on its visibility from the same path, as façades that will not be seen are best left out to reduce the memory and rendering overhead. The operation facilitating this verification, also employed in Figure 8, hereby simply named "is oriented to," accepts a list of shapes and a list of streets and calculates, for a given angle tolerance, if the shape is visible from any of the streets, without being occluded by another shape entity. The result is assigned to an attribute, but that distinction could also be done using several outputs. One important consideration is that, by gathering all the shapes and streets at once, this node can organize and optimize the verification internally using spatial data structures, such as quad- and octrees.

Another example node depicted in Figure 8 that interrelates entities is the "ray cast" node. For a given shape arriving at the first port, it casts a ray, following on the scope direction, and searches for the first intersection with the shapes arriving at the second port. The original object is returned on the first output port, while the separation of hit and nonhit shapes is performed to the second and third output ports. The actual hit location is stored to an attribute, which can later be used as a reference point for splitting or other operations.

4.6. Enforcing Execution Scopes. As mentioned in Section 3.2, collective nodes attempt to gather as many entities as possible

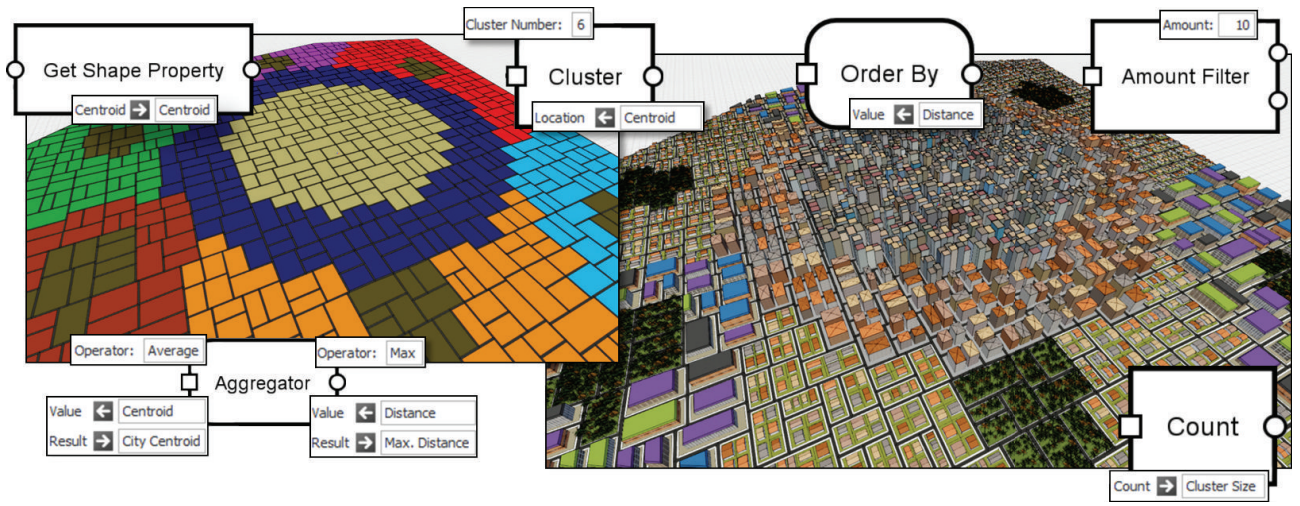

FIGURE 7: Organization of city blocks for construction. After randomly generating about a thousand city blocks, the city center was determined in order to build a downtown area, a surrounding commercial area, and a peripheral area. The latter was divided into 6 clusters to define alternating industrial and residential neighborhoods. Again, the center point of each cluster was then calculated and the 10 closest blocks were reserved for green areas. This process involved several aggregation, sorting, clustering, grouping, and filtering operations.

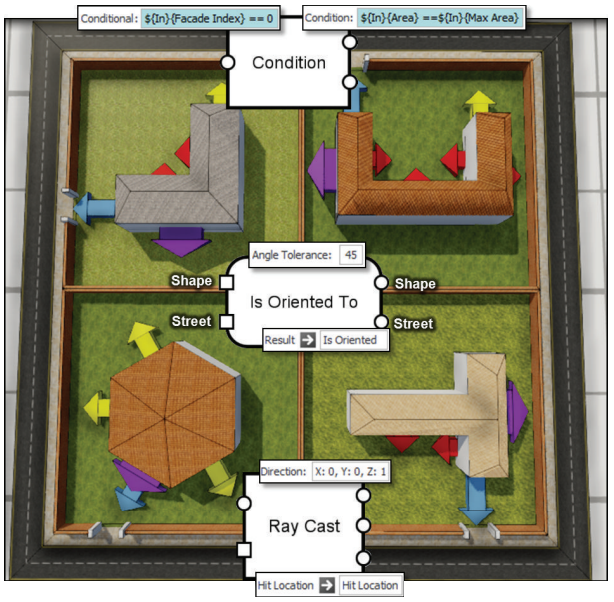

FIGURE 8: Selection of the main door façade for several building footprint shapes. Purple arrows indicate a selection by a given façade index, very common in systems such as [6], which do not always guarantee plausible door locations. All other arrows refer to façades that are street oriented (as in [12]). From these options, the smallest façades were chosen using aggregation (yellow and blue arrows), but, to guarantee uniqueness, amount filtering was employed (blue arrows). The selected façade was fed to the "RayCast" node, which returned the front-facing wall and pointed to the right entity and location for the lot entrance to be built.

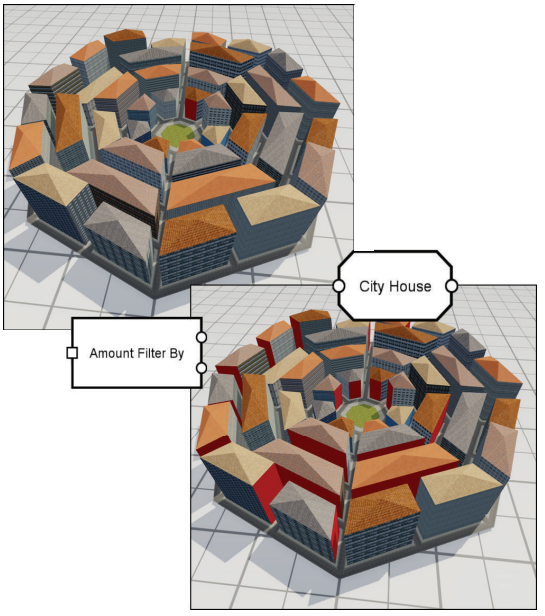

FIGURE 9: Example of the need to enforce execution scopes. If all the facades are collected without differentiation, only one facade of the whole building set would be selected, instead of one per building.

before executing its operation. This may, in some situations, become an issue if one would rather have it executed for subsets of the collected entity list. In this sense, encapsulation does come as a means not only to organize graph procedures, but also to provide an improved control over the data flow.

Figure 9 reflects a situation where several houses are generated using a simple graph, supposing one example where one would try to select exactly one facade of each building to create the main door. If all the facades are collected without differentiation and queued at the amount filter node, only one facade of the whole building set would be selected, instead of one per building, as desired.

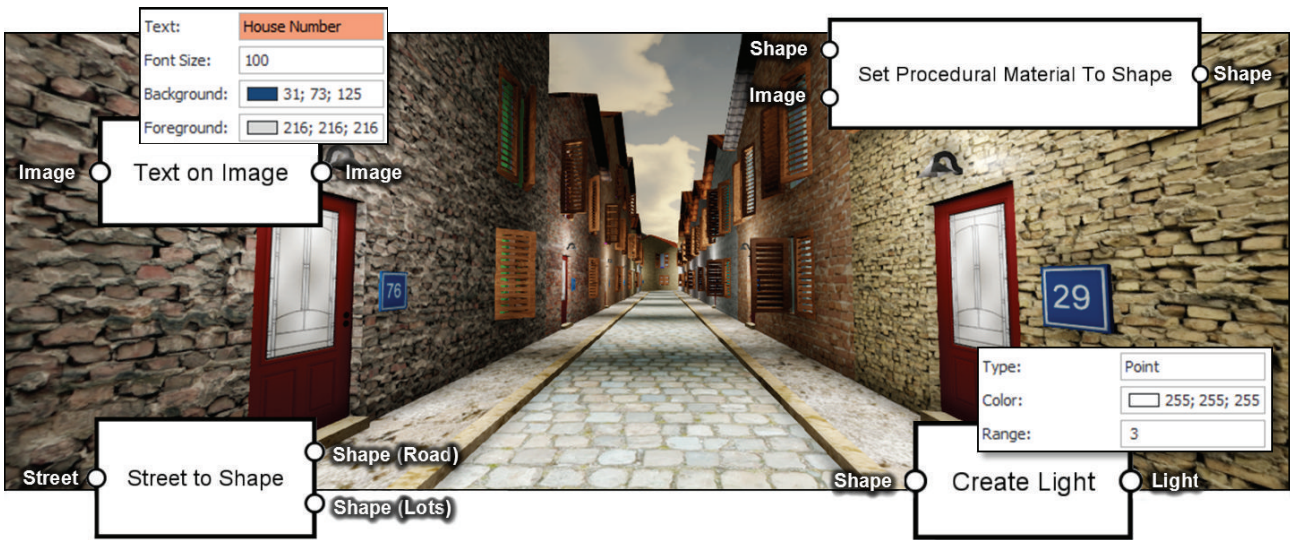

Figure 10: Manipulation of different entity types within the same scene. A street entity led to the generation of several buildings, represented as shapes. Each building was given a number which was then used to produce a custom texture to attach next to each door. Light sources were then instanced on the location of each street lamp, just above the doors. The generation of each entity type therefore greatly benefits when processed in sequence, in a single pipeline, instead of in separate environments.

This can be solved using an "amount filter by," which performs that separation internally using attributes, enumerated via node augmentations as grouping criteria (see Section 3.8). This approach can quickly become impractical, if such grouping would have to be performed, for example, per neighborhood, then the building, and then per floor. On the other hand, this requires all collective node operations to support such criteria analysis, as well as to have differentiating attributes configured every time.

Another alternative consists in isolating data using encapsulation so as to enforce *execution scopes*. The idea is that the operation over each lot is handled independently. For each input lot, the sequence of graphs, including collective nodes, is performed, meaning that the entity collection of the "merge" and "amount filter" operations are executed only within the scope of each individual building. This is guaranteed by the fact that encapsulated nodes, as any other procedures, work isolatedly on its executed queues, before allowing the supergraph to proceed with its execution.

4.7. Integrating Different Entity Types. The ability to manipulate several types of entities lies on the existence of nodes capable of creating, analyzing, or transforming them. This possibility is reflected in the typing of node ports, which provides the means to understand the node's purpose and compatibility. Different entities can coexist within the same graph and, in some cases, even share the same nodes, ports, or edges, if they share the same ancestor type.

Figure 10 portrays an example where several entity types are manipulated within the same graph. Using a street network entity as the starting point, roads/sidewalks and lots were derived, as shape entities, using a "street to shape" node. The lots were developed into whole buildings and the light-source entities generated at the location of the lamps above each building's main door. As for the door numbering, individual textures were generated using the "text on image" node and then placed on a plate next to the door using the node "set procedural material to shape," which combines texture and shape, as a custom mesh material. The result is a complete urban scenery, which would otherwise require multiple environments for development and whose integration would require additional manual or scripting efforts.

5. Implementation

We implemented the procedural content graph approach in a prototype system called *construct*, which consists of a framework and a visual editor. In this section, we provide details on its architecture, functionality, and possibilities.

5.1. Framework. The framework consists of a set of assemblies developed using C#. The core contains the graph representation; basic entities, attribute types and procedures; execution algorithms; and the means to load and integrate procedure libraries. The definition of different entities (geometric meshes, surfaces, streets, images, lights, etc.) is split into different libraries, each incorporating specific procedures for manipulation and interoperation. By using C#'s reflection abilities, both library data and documentation are automatically extracted, all with the purpose of facilitating the development of the system.

Currently, construct implements: procedures for common operations in shape grammars (e.g., extrusion, splitting, translation, and rotation); texture synthesis (e.g., invert, add, subtract, etc.); surface manipulation (e.g., noise, smoothing, and leveling); street generation (e.g., buffer, translation, texturing, etc.); and light manipulation (e.g., instantiation, transformation). More importantly, it also provides many

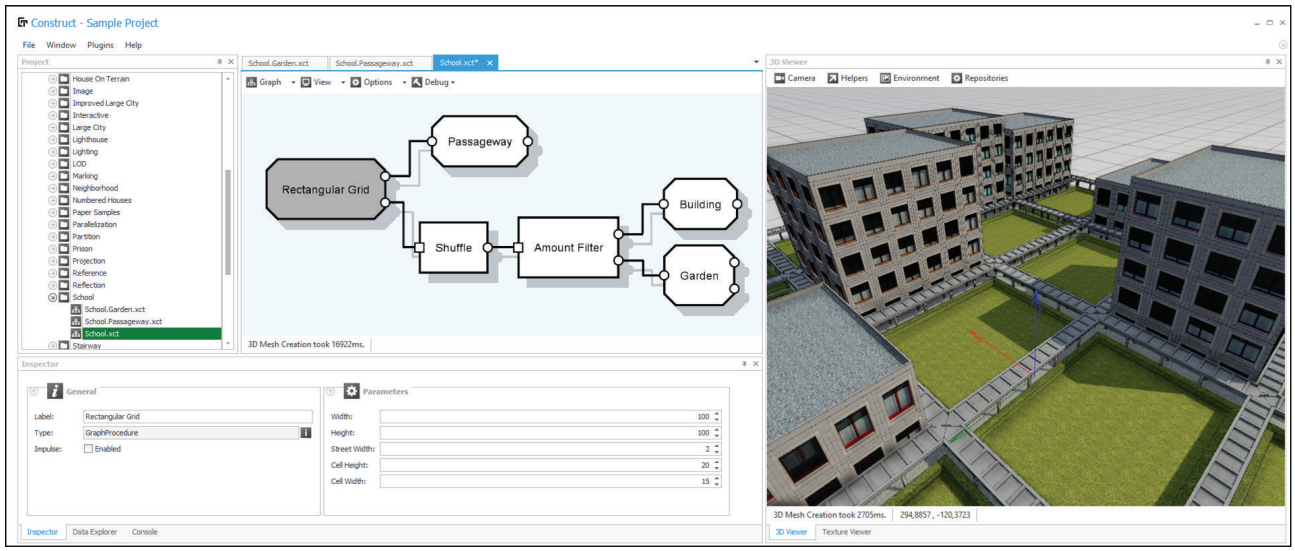

FIGURE 11: Graphical user interface of the Construct Visual Editor.

operations focusing on list processing, context-based querying, and type integration, discussed in Section 4.

Extending the framework by developing whole new low-level nodes and entities requires some programming experience. We have made the framework available for a variety of projects and confirmed that it was easy for experienced programmers to add such new low-level procedures for very disparate purposes. We can therefore expect that the amount of available operations will further increase when construct is publicly released.

5.2. Visual Interface. In order to design graphs with our approach, we have developed a visual interface (Figure 11). The editor is plugin-based, providing the possibility to add custom file editors and docking windows. The main plugin for graph manipulation provides interactive tools for the creation and editing of graphs, as well as for the visualization of their output.

To manage graph, node, and port definitions, a separate inspector window is available, showing details of the currently selected element. When a node is selected, the user can edit its parameters (which features a parser of simple mathematical expressions and function calls), attributes, and augmentations. When a port is selected, the user can change its state to "blocked" (see Section 3.2) or to "gate" (see Section 3.7). A separate option to edit graph details is also available, letting the user define the graph parameters and attributes, as well as some metadata (name, description, etc.).

To view the output of PCGRs, a rendering window is available, which includes various visualization aids that allow users to better perceive mesh scopes (similar to [3]), such as edges or bounding boxes, among other guides.

The editor also features a debugging window to display log information produced by the nodes. When in debug mode, it can also provide information about the flow sequence, procedure execution times, and the amount of produced data. The *Live-Execution* mode, on the other hand, will automatically reexecute the PCGR for every change, helping the user understand the impact of modifications on the graph output. Since the whole graph is executed on a different thread, interaction remains smooth.

The visual interface facilitates intuitive graph manipulation. For example, the insertion of nodes is done through a quick search window that lists all available procedures. Once the node is added, edges can be drawn by dragging the mouse between two ports. Since ports can be of different types, visual guides on ports (changing their color and size) indicate which connections are allowed.

Encapsulation of graphs is also very much facilitated. To include a graph in a supergraph, the user simply has to drag the graph's file into the supergraph's editor canvas and the encapsulated node will be instantiated. The user can still set the subgraph's parameters and gates afterwards and the changes will be reflected in the supergraph immediately. Conversely, a user can select a subset of nodes and edges and choose the option to encapsulate that selection into a new node file. Referenced control parameters/attributes will automatically be built into graph parameters/attributes, while connected ports will automatically be converted to gates and added to the encapsulated node's signature.

6. Discussion

In this section, we discuss how procedural content graphs compare to existing approaches, namely, grammars and graph-based design tools. We also briefly discuss performance and feedback received on the use of our system.

6.1. Comparison with Grammar Approaches. As shown in Section 4, procedural content graphs can represent and manipulate content as lists or groups of entities, enabling operations like, for example, aggregation, sorting, and advanced

filtering, which, to the best of our knowledge, are either not achievable or very contrived using a grammar approach.

We now turn to the issue of the generality of PCGR, explaining how existing grammar specifications can be equally represented using procedural content graphs, often with the advantage of clarity and conciseness.

Generative grammars, such as the ones presented by Wonka, Müller, and Krecklau [2–4, 14], are defined typically as a set of production rules *Predecessor* → *Successor*, where the predecessor is a nonterminal symbol and the successor is a set of one or more nonterminal or terminal symbols. Given a starting nonterminal symbol, the process consists of successively replacing symbols that match the predecessor with the ones indicated in the successor. For instance, in shape grammars, geometric shapes are the symbols that are created or successively transformed by means of operations.

That same rule structure is captured in the graph connectivity of a PCGR: entities (the nonterminal symbols) emanating from a given output port (the predecessor) flow through established edge connections to input ports of other nodes (the successor), thereby further transforming them. The possibility to create any number of connections between any pair of output-input ports of compatible types induces that the same possibilities for rule convergence, divergence, and recursion still apply. As for other grammar features, they are easily reproduced in a PCGR as follows:

(i) *Parametric Rules* [3]. Parameter passing is achieved by means of attribute handling (Section 3.6).

(ii) *Conditional Rules* [3]. They are achieved through condition nodes, featuring one input, 2 outputs, and a boolean parameter/expression. If they are evaluated to be true, the entity received as input is sent to the first output port, otherwise, to the second.

(iii) *Scope Rules* [3]. Pushing and popping scope states are achievable through an attribute of type list/stack, which is written to and read from, just like any other attribute.

(iv) *Split Rules* [3]. A split is an augmented node, which allows a flexible definition of the split sizes and the introduction of a separate output port for each split. Snap shapes can also be introduced dynamically through specific augmentation types (Section 3.8).

(v) *Occlusion Query Tests* [3]. An occlusion query node uses its gather port to receive all the shapes, organize them into an octree, and test for occlusions (within a certain distance, defined in the node parameters). The occluded and nonoccluded shapes are returned via different output ports.

(vi) *Generalization of Nonterminal Objects* [4]. It is supported naturally through the typed nature of procedural content graphs (Section 3.3).

(vii) *Connecting Structures* [14]. They are supported naturally through the possibility to define multiple input ports or gather ports (Figure 8).

(viii) *Accessing and Creating Containers* [14]. They are supported by the merging and grouping abilities offered by gather ports (Section 4).

6.2. Comparison with Existing Systems and Tools. When compared to text-based alternatives such as grammars, PCGRs offer a visual solution that makes data flow easier to understand, follow, and manage, especially for complex constructions. That goal has been addressed before [5] but inherited the limitations from the underlying shape grammars, such as the impossibility to aggregate and manipulate entity lists. The Houdini-based approaches [6, 7] somehow address this issue as well but impose restrictions on recursion and encapsulation (Section 3.7). In addition, they rely on labels to filter and control the flow in an implicit form (rather than an explicit representation via edge and port connectivity), which introduces a management issue shared by grammar-based systems. Moreover, since symbols, like labels, are not always easily given semantically relevant names, it is hard to manage and maintain a project, especially as it gets larger.

As to the implementation of our framework and visual editor, construct is a proof of concept, not meant to be compared directly to commercial tools such as CityEngine [12] or Houdini [6], certainly not regarding design features (such as the sophistication of the available procedures) or interface accessibility. However, our framework has been designed with extensibility in mind and users are able to easily expand it with their own components. In particular, the flexibility of supporting custom entities, augmentations, and encapsulations brings about the integration of all components in one unifying system.

6.3. Performance. In general, the execution of PCGRs is fast and on par with tools as Houdini and CityEngine, for the same degree of geometric complexity. Nonetheless, we have not made an exhaustive performance comparison, as it would go beyond the scope of this paper. Also, performance depends considerably on the type of processing algorithms, some of which are, to the best of our knowledge, actually unfeasible with other approaches, as stated in Section 4. Most important of all is the fact that the overhead time for node sequencing and data transport is negligible. This means that the graph representation and data flow execution algorithm itself does not constitute a performance issue.

For the various examples used here, the total generation times were at most a few seconds, mostly just a few milliseconds, on an Intel Core i7-2670QM, 2.2 Ghz Laptop, featuring 16 GB RAM, Nvidia GeForce GTX 560M. Consequently, interactive graph modifications are possible and the user receives quick feedback. For larger or more detailed examples, such as the one displayed in Figures 6 and 12, encapsulation and parametrization significantly help reducing the generation scope (to focus on smaller number of blocks or buildings at a time instead of a vast area), therefore ensuring a smooth interactive experience.

The use of Microsoft XNA as a rendering engine has restricted the current construct visual interface to a 32-bit process. As a result, memory allocation is limited, making

Figure 12: Detailed, close-up view on the city from Figure 7.

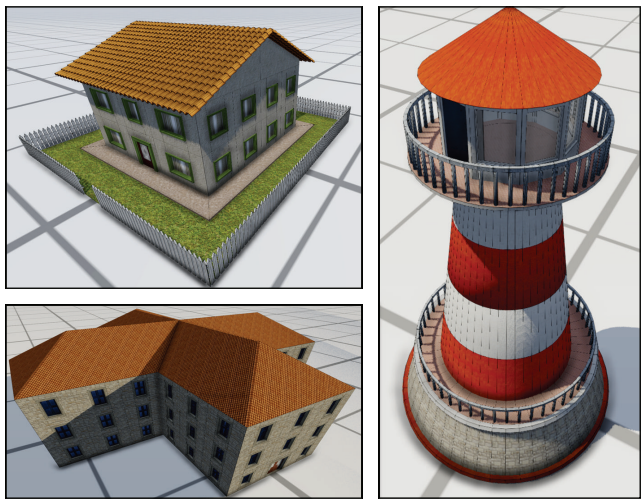

Figure 13: Examples of user-designed buildings using our system.

the design of both very large and very detailed scenes impossible to sustain and visualize. We intend to address this issue in the future to better test the limits of our system.

6.4. User Feedback. We let four users test our system and performed active demo sessions with another five users, most of them having had some contact or actual experience with CGA shape grammars (through CityEngine) or Houdini. Due to their experience with shape grammars and with graph-based design tools, these participants were swiftly able to build basic structures and had simple results within a very short time (Figure 13).

All users were given only a few example graphs to examine and had to figure out for themselves how the system worked. Their responses led us to conclude that the simple graph topology, node nomenclature, and reduced graph size (achieved through successive encapsulation of meaningful operations) are very positive and valued factors. To all questioned users, there was a clear preference of such visual representation over the text-based ones. Examples that show control and merging of lists with geometric entities were considered most exciting to shape grammar users. They understood and indicated that this new possibility would allow them to execute operations based on relationships between entities, considering this feature to be very useful.

On a negative side, users commented on the difficulty to find low-level nodes that they needed among the list of existing ones, an issue we plan to address by introducing a categorization of the existing procedures in the future. As expected, many other pointed issues lied on pure GUI-based deficiencies, such as the lack of keyboard shortcuts, poor autocompletion for the expression editor, and the need for long mouse trips between graph canvas and inspector window. Therefore, these features will have to be addressed before more formal and complete usability studies can be performed.

At this stage, construct has already been used in numerous research projects on (or using) procedural content generation [37]. Besides the ease-of-use, researchers point out the great benefit of being able to easily define their custom procedures and interactive tools.

7. Conclusion

We have introduced procedural content graphs (PCGRs), a generic approach for the specification of content generation procedures, which retains the advantages of grammar-based solutions and addresses most of their limitations. In particular, we demonstrated that a variety of list-based operations, as, for example, merging, ordering, aggregation, and clustering, strongly contribute to a richer and more expressive design specification than that offered by other grammar and graph-based approaches. This is also patent in the support of custom entity types, which contribute to a more intuitive description of the content generation procedures. Moreover, the introduction of augmentations, encapsulation and attributes, significantly helps keeping content design and development compact and flexible.

In the future, we would like to perform a formal user study to quantitatively investigate user friendliness. So far, our experience is that even people unfamiliar with other procedural design methods are able to grasp its functioning and to rapidly build their first graphs.

We believe that the generality of procedural content graphs provides an ideal ground for context-dependent development, towards a unified pipeline for comprehensive procedural content generation. Even more, this approach has the potential to be integrated with real-time simulation or game engines as well, in order to support adaptive generation of large virtual environments. Because the approach was designed with extensibility in sight, we can expect that it will give rise to an active community contributing new features, from new procedures and entities, to more high-level editing features, ultimately leading to an increasing deployment of procedural content graphs.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This work has been supported by the Portuguese government, through the National Foundation for Science and Technology (FCT) (Fundação para a Ciência e a Tecnologia) through the Ph.D. Scholarship SFRH/BD/73607/2010.

References

[1] R. M. Smelik, T. Tutenel, R. Bidarra, and B. Benes, "A survey on procedural modelling for virtual worlds," *Computer Graphics Forum*, vol. 33, no. 6, pp. 31–50, 2014.

[2] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky, "Instant architecture," in *Proceedings of the 30th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '03)*, pp. 669–677, ACM, San Diego, Calif, USA, July 2003.

[3] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. van Gool, "Procedural modeling of buildings," in *Proceedings of the 33rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '06)*, pp. 614–623, ACM, Boston, Mass, USA, August 2006.

[4] L. Krecklau, D. Pavic, and L. Kobbelt, "Generalized use of non-terminal symbols for procedural modeling," *Computer Graphics Forum*, vol. 29, no. 8, pp. 2291–2303, 2010.

[5] P. B. Silva, P. Müller, R. Bidarra, and A. Coelho, "Node-based shape grammar representation and editing," in *Proceedings of the Workshop on Procedural Content Generation in Games (PCG '13)*, May 2013.

[6] Side Effects Software, Houdini, 2015, http://www.sidefx.com/.

[7] G. Patow, "User-friendly graph editing for procedural modeling of buildings," *IEEE Computer Graphics and Applications*, vol. 32, no. 2, pp. 66–75, 2012.

[8] P. Prusinkiewicz, M. Hammel, J. Hanan, and R. Mech, "L-systems: from the theory to visual models of plants," in *Proceedings of the 2nd CSIRO Symposium on Computational Challenges in Life Sciences*, vol. 3, pp. 1–32, Victoria, Australia, 1996.

[9] R. Mech and P. Prusinkiewicz, "Visual models of plants interacting with their environment," in *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, pp. 397–410, August 1996.

[10] G. Stiny and J. Gips, "Shape grammars and the generative specification of painting and sculpture," in *Information Processing '71*, C. V. Friedman, Ed., pp. 1460–1465, North-Holland, New York, NY, USA, 1972.

[11] M. Larive and V. Gaildrat, "Wall grammar for building generation," in *Proceedings of the 4th International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia (GRAPHITE '06)*, pp. 429–437, ACM, December 2006.

[12] Esri, "Esri cityengine—3d modelling software for urban environments," 2013, http://www.esri.com/software/cityengine/.

[13] M. Schwarz and P. Wonka, "Procedural design of exterior lighting for buildings with complex constraints," *ACM Transactions on Graphics*, vol. 33, no. 5, article 166, pp. 1–16, 2014.

[14] L. Krecklau and L. Kobbelt, "Procedural modeling of interconnected structures," *Computer Graphics Forum*, vol. 30, no. 2, pp. 335–344, 2011.

[15] S. Havemann, *Generative mesh modeling [Ph.D. thesis]*, University of Braunschweig, Institute of Technology, 2005.

[16] B. Hohmann, S. Havemann, U. Krispel, and D. Fellner, "A GML shape grammar for semantically enriched 3D building models," *Computers & Graphics*, vol. 34, no. 4, pp. 322–334, 2010.

[17] L. Krecklau and L. Kobbelt, "Interactive modeling by procedural high-level primitives," *Computers & Graphics*, vol. 36, no. 5, pp. 376–386, 2012.

[18] M. Lipp, P. Wonka, and M. Wimmer, "Interactive visual editing of grammars for procedural architecture," *ACM Transactions on Graphics*, vol. 27, no. 3, article 102, 2008.

[19] E. Hale and N. Long, "Enumerating a diverse set of building designs using discrete optimization," in *Proceedings of the 4th National Conference of IBPSA-USA*, pp. 77–84, New York, NY, USA, August 2010.

[20] L. Leblanc, J. Houle, and P. Poulin, "Component-based modeling of complete buildings," in *Proceedings of Graphics Interface (GI '11)*, pp. 87–94, ACM, May 2011.

[21] N. J. Mitra, M. Wand, H. R. Zhang, D. Cohen-Or, V. Kim, and Q. X. Huang, "Structure-aware shape processing," in *Proceedings of the SIGGRAPH Asia Courses (SA '13)*, pp. 1–20, 2013.

[22] M. Bokeloh, M. Wand, and H. P. Seidel, "A connection between partial symmetry and inverse procedural modeling," in *Proceedings of the ACM SIGGRAPH 2010 Papers (SIGGRAPH '10)*, pp. 104:1–104:10, ACM, New York, NY, USA, 2010.

[23] J. O. Talton, Y. Lou, S. Lesser, J. Duke, R. MěCh, and V. Koltun, "Metropolis procedural modeling," *ACM Transactions on Graphics*, vol. 30, no. 2, article 11, 2011.

[24] A. Milliez, M. Wand, M. P. Cani, and H. P. Seidel, "Mutable elastic models for sculpting structured shapes," *Computer Graphics Forum*, vol. 32, no. 2, pp. 21–30, 2013.

[25] P. E. Haeberli, "Conman: a visual programming language for interactive graphics," *ACM SIGGRAPH Computer Graphics*, vol. 22, no. 4, pp. 103–111, 1988.

[26] W. M. Johnston, J. R. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Computing Surveys*, vol. 36, no. 1, pp. 1–34, 2004.

[27] W. Machine, 2015, http://www.world-machine.com/.

[28] B. Lintermann and O. Deussen, "Interactive modeling of plants," *IEEE Computer Graphics and Applications*, vol. 19, no. 1, pp. 56–65, 1999.

[29] Speedtree, 2015, http://www.speedtree.com/.

[30] Allegorithmic, Allegorithmic substance designer, 2015, http://www.allegorithmic.com/products/substance-designer.

[31] Nodebox, 2015, http://www.nodebox.net/.

[32] Grasshopper, 2015, http://www.grasshopper3d.com/.

[33] B. Ganster and R. Klein, "An integrated framework for procedural modeling," in *Proceedings of the 23rd Spring Conference on Computer Graphics (SCCG '07)*, M. Sbert, Ed., pp. 150–157, Comenius University, Bratislava, Slovakia, 2007.

[34] S. Barroso, G. Besuievsky, and G. Patow, "Visual copy & paste for procedurally modeled buildings by ruleset rewriting," *Computers & Graphics (Pergamon)*, vol. 37, no. 4, pp. 238–246, 2013.

[35] G. Kelly and H. McCabe, "Citygen: an interactive system for procedural city generation," in *Proceedings of the 5th Annual International Conference in Computer Game Design and Technology (GDTW '07)*, pp. 8–16, Liverpool, UK, 2007.

[36] E. Galin, A. Peytavie, E. Guérin, and B. Beneš, "Authoring hierarchical road networks," *Computer Graphics Forum*, vol. 30, no. 7, pp. 2021–2030, 2011.

[37] P. B. Silva, *Improving expressiveness, integration and manageability in procedural content generation [Ph.D. thesis]*, Faculdade de Engenharia da Universidade do Porto, 2015.

[38] Esri, CityEngine Help, 2015, http://cehelp.esri.com/.