Advanced Procedural Modeling of Architecture — Supplemental Material

Michael Schwarz Pascal Müller

Esri R&D Center Zurich

Overview

In this document, we first elaborate on some aspects mentioned during the discussion offered in the paper (Sec. 1). Subsequently, we discuss how shape trees are realized in our prototype and what effects the various actions have on such a tree (Sec. 2). A concise reference for CGA++ is then presented in Sec. 3, covering many details omitted in the paper. Finally, definitions for some functions and rules occurring in the paper's examples are given (Sec. 4), exemplifying both more advanced use and the expressive power of CGA++.

1 Further discussion

In the following, we provide more details on some aspects only hinted at in the paper's discussion section. After taking a closer look at means to guide the derivation process, we investigate existing ad-hoc solutions for some specific problems, comment on the technical necessity of the new language features, and assess alternative solution approaches using non-grammar languages.

1.1 Guidance of derivation order

The order in which shapes are refined during the derivation process ultimately affects whether the shapes a contextual query is supposed to consider have already been created when executing this query (and hence can be considered). Consequently, a certain influence on the derivation order is necessary to correctly handle such dependencies. To this end, the original CGA shape [Müller et al. 2006] allows statically assigning priorities to rules (this concept is absent in all (commercial) successor versions). Pursuing a sequential execution model, where one rule is applied at a time, always (one of) the highest-priority rule(s) currently applicable is selected as next rule to execute. Hence, priorities enable structuring the derivation process into global development stages, where all rules belonging to a certain stage are assigned the same priority. This is made more explicit with evaluation phases [Steinberger et al. 2014], which mandate that a rule may only yield symbols corresponding to rules of equal or lower priority and hence enable the concurrent execution of rules assigned to the current phase. A related, more general approach is construction stages [Schwarz and Wonka 2014], where instead of statically assigning each rule to a stage, an operation is provided that signals having reached a certain stage and blocks until all other derivation branches have reached this stage, too. Because the stage is provided as an argument, it may be chosen dynamically, and an arbitrary number of stages is supported.

In CGA++, events provide a more flexible and convenient device for guiding the derivation order. Generally, each active branch of the derivation process can execute its actions independently from all other branches (allowing for parallel execution of multiple branches). A dependency on other branches is explicitly expressed by raising or waiting on an event, and if such an action is encountered, the branch gets suspended until the dependency can be resolved. An event hence serves as synchronization point, indicating having reached a certain development stage. However, because not all derivation branches have to participate in a certain event and event groups further enable restricting events to subtrees, local and hierarchical dependencies can be conveniently expressed. In particular, it is not necessary to impose a global order on how all dependencies occurring during the whole derivation process are to be resolved, as would be required by construction stages (rule priorities further involve duplicating rules whenever they are to be applied in multiple stages), and which easily becomes challenging for the grammar writer. For instance, seemingly simple cases such as where each façade of a building is refined in multiple stages, but only once the front façade has been fully created, the refinement of the other ones can commence (e.g., due to performing queries on the front façade), are cumbersome to handle globally. By contrast, events allow expressing the dependencies within a façade independently from those among façades.

1.2 Ad-hoc solutions for specific tasks

For a few selected tasks that critically depend on information from other shapes, dedicated solutions exist. The functionality offered by them can often be replicated in CGA++ using just its available general language features. Moreover, thanks to its enhanced expressiveness, CGA++ usually enables augmenting the capabilities of such ad-hoc solutions. Note, however, that even with CGA++, performance and efficiency considerations may encourage incorporating dedicated support for addressing frequent use cases, including offering special operations and performing specific optimizations (e.g., using a spatial acceleration structure for certain common queries).

Occlusion The ability to determine whether the current shape is occluded is crucial for many applications. One important aspect is selecting which occluders are considered in such tests (and ensuring their existence), but existing solutions are rather limited (and sometimes surprisingly vague) in this regard. For example, in classical CGA shape [Müller et al. 2006], the query function `Shape.occ` offers several options to this end, such as all other existing shapes, shapes with a certain symbol, and all shapes except the current shape's ancestors (`"noparent"`). However, the respective set of shapes is often not well defined and depends on the concrete derivation order (which often cannot be influenced sufficiently). For instance, assume a split yields multiple shapes, and the rule associated with them first tests for occlusion with `"noparent"` and enlarges the shape if unoccluded, emitting an empty shape otherwise. Then, the order in which these shapes are refined by this rule (which cannot be influenced by priorities) affects which are enlarged and which are replaced by an empty shape (because they are occluded by one of the already processed and hence enlarged parts). Similarly, the PGA system [Steinberger et al. 2014] basically allows querying any shape generated in a previous evaluation phase, but no details are given on how such a shape is identified, both language-wise and implementation-wise (among others, such a shape must be stored persistently in global memory on the GPU, which entails challenges that are not even hinted at). By contrast, the extension by Schwarz and Wonka [2014] supports testing against shapes that exist at a certain construction stage, which is well defined (but slightly limited, though). Esri's CityEngine [Esri 2014] takes a different approach and first constructs a ghost shape tree by evaluating all occlusion tests to false, and then tests against it during the actual derivation process. Consequently, only a test against the shapes in the ghost tree is possible.

In CGA++, arbitrary shapes of the shape tree can be accessed and their existence be enforced. Therefore, more fine-grained occlusion

queries become possible, which may involve spatial query functions such as `overlaps`. In particular, all mentioned solutions could be expressed in CGA++ (even a ghost shape tree, irrespective of how desirable that is).

Alignment Classical CGA shape [Müller et al. 2006] offers snapping as a means to support basic alignment. Special snap shapes (planes and lines) can be emitted, both automatically and via operations, and may then be taken into account by subdivision operations, adjusting split positions to align with such snap shapes. Although helpful for several cases, snapping provides only a limited form of control. By contrast, CGA++ features an expressiveness that allows realizing more complex alignments and exercising full control. In particular, referencing other shapes and coordination via events essentially enable almost unlimited possibilities.

Connecting shapes Targeting complex interconnected structures, the system by Krecklau and Kobbelt [2011] allows collecting (potentially multi-dimensional) lists of source and target rectangles and subsequently creating connections between them. One inherent limitation of their language extension realizing this approach is that it strongly depends on a strictly sequential execution of the grammar, where a rule basically constitutes a subroutine call.

With CGA++, compiling lists of source and target shapes is directly supported, and multiple options exist. For instance, all source shapes could be flagged by setting an attribute and then be collected by querying the shape tree, where events can be employed to ensure that all required shapes exist. Complex patterns of which source shape should be connected to which target shape are possible (as in their system), where list and spatial query functions may be utilized for establishing such a pattern. Actually connecting two shapes can be achieved with an according operation; in our prototype, we currently only offer the operation `connectTo` for creating a connecting tube between two polygons. Supporting more advanced connection primitives, such as deformable beams or rigid chains from Krecklau and Kobbelt's system, would be possible, though.

1.3 Technical necessity of new language features

Several new language elements of CGA++ are primarily introduced to allow a convenient syntax and form of expression, while not being required from a purely technical point of view. Apart from syntactical sugar such as the chain operator, one supposedly non-obvious example in this regard is rule values. Although these are unarguably highly useful, they actually can be simulated with already existing language elements; that is, granting first-class citizenship to rules (unlike in the case of shapes) does not extend what can be modeled. Concretely, any anonymous rule can be replaced by an according ordinary (named) rule, turning all captured variables into explicit arguments. Each named rule can be assigned a unique identifier, and thus any rule value can be mapped to such an identifier and a list of arguments. Utilizing a conditional construct with a branch for each possible identifier, the rule corresponding to an identifier may then be invoked with the given arguments. An analogous observation holds in the case of functions.

1.4 Approaches using non-grammar languages

In the quest of achieving an integrated solution to overcome the limitations of current grammar languages, one may consider resorting to a (suitable) more general, non-grammar language. However, while this typically increases the available expressive power significantly with respect to a grammar language, thus (in principle) enabling dealing with the targeted advanced modeling scenarios, it also entails some practical challenges and inconveniences. First, the derivation described by a grammar has to be manually replicated in an appropriate form in this language, which may be complicated by a more limited offer of domain-specific abstractions and involves abandoning at least some characteristics of grammars, such as compact and focused syntax or ease of editing. Moreover, this must be done in a way that transcends the limitations of existing grammar formulations and allows realizing the considered modeling task. Finally, a concrete solution for this task has to be devised.

One potential candidate for pursuing this direction is employing a scripting language in a standard modeling software such as Maya, Houdini, or Blender. Another possibility is using languages for generative modeling, such as GML [Havemann 2005]. Hohmann et al. [2010] even present a library of GML functions that cover a narrow subset of the operations offered in CGA shape and show how rules can be written as functions. However, being a Postscript-like stack-based language, GML affords a syntax and way of expression that deviate considerably from more mainstream languages and is hence challenging for non-programmers. An interesting further option is extending the system presented by Leblanc et al. [2011], where components (shapes) are created and modified by a sequence of statements. It includes support for queries to select components, loops to iterate over a set of components, and operations on components. Basically, each execution of an action during the derivation of a grammar has to be written as a separate statement, with the sequence of statements explicitly encoding the derivation order.

2 Implementation details for shape trees

Using shapes as first-class citizens and offering access to the shape tree in a grammar ultimately requires any concrete implementation to decide on the structure of the shape tree and the modifications of the tree caused by the various kinds of actions. In the following, we provide details on the choices made in our implementation. These take several requirements and considerations into account:

(a) Intermediate results that are assigned a label in a rule body may be referenced not just within this rule body but also using the access operator `::` (with the intermediate result's parent as left-hand operand and the label as right-hand operand). This latter option implies that such a labeled intermediate result must be part of the shape tree. On the other hand, it should not be included in the final model.

(b) Whenever an action, such as a subdivision operation, introduces new independent derivation branches, the order in which these and the current branch are executed should not influence the structure of the resulting shape tree, including the order of any node's children.

(c) Temporary shapes, such as the current one (`this`), the result of a function on a shape, or the tree returned by spawning a new (sub)derivation process on a given shape, have a meaningful parent and hence should allow access to it. This suggests that such shapes are somehow embedded in the parent's shape tree. On the other hand, these shapes need not (and we think should not) be accessible via this parent.

Motivated by the last concern (c), we distinguish between proper children and anonymous children. A node keeps an ordered list of all its proper children; consequently, such children can be directly accessed from their parent. By contrast, a node is estranged from its anonymous children: it may know about their existence but it does not know their identities and hence cannot access them. (Knowing about the existence of anonymous children can be beneficial for memory management purposes.) Thus, each node (except for the root node) has a parent, and it stores a reference to this parent.

Moreover, we associate a visibility with each node to address the concerns (a) and (b). Only visible nodes are considered by the

traversal functions offered, and the final model is defined by the set of all visible nodes with no visible descendants, often referred to as *leaf shapes*. An internal node is always visible if any of its proper children is visible. Invisible nodes are used to represent labeled intermediate results as well as (evolving) structures without any children (yet), such as a group node where no action succeeding its defining `group` operation was executed (so far).

In our implementation, shapes are exposed as being weakly immutable to grammars: no existing data is ever modified but new structural information can be added to a tree. In particular, an invisible node may become visible; this change is recursively propagated upward to the parent unless the node is an anonymous child. Furthermore, child nodes can be added, where proper children may only be appended to their parent's list. Overall, this enables a consistent view on the tree without having to restrict the independent execution of different derivation branches.

Building on this specific concept of a shape tree, we provide an overview of the effects of various actions on the shape tree in the following (see Fig. 1).

Rule execution When an invoked rule is executed for a shape, in a setup step a new derivation branch is created, and a copy of the shape is added as anonymous child to this shape node (Fig. 1 a). It represents the (initial) current state during the execution and is commonly referred to as *current shape* (exposed via `this`). Note that such copies are cheap, as the payload of a node, most notably the shape's geometric data, can be (and actually is whenever possible) shared with other nodes.

Shape instantiation When the current shape is instantiated, for example by a symbol (causing the invocation of the according rule), a copy is made, named by the symbol and its arguments (if any), and added as a visible, proper child to the parent of the current shape's node (Fig. 1 c). If a label is provided for the action, this label is also assigned to the child node, allowing its access by name when navigating the tree.

NIL Analogously, the `NIL` operation adds an empty shape as visible, proper child to the current shape's parent (Fig. 1 d). Among others, it is routinely employed in subdivision operations to remove a part from the final result.

Shape(-modifying) operations Respecting the weak immutability of nodes, operations modifying the state of the current shape don't actually modify the node representing the current shape but introduce an accordingly modified copy. This is added as an anonymous child, and the execution state is updated such that this new node becomes the current shape (Fig. 1 e). Note that internally, it is safe to (and our implementation hence will) directly modify the shape if the old current shape can no longer be accessed and hence be deleted afterward. In the case that the operation is labeled, also a copy of the resulting new current shape is added as an invisible, accordingly labeled child (Fig. 1 f).

Subdivision operations For subdivision operations, the current shape is copied and added as an invisible, proper child. Subsequently, a node is created for each resulting part and added as a visible, proper child to the operation's node (causing this node to become visible). Each part also introduces a new derivation branch for executing the associated actions: a copy of the part's node is added as an anonymous child and becomes the current shape in this branch (Fig. 1 g).

Scoped execution Putting a sequence of actions in brackets (i.e., `[` *actions* `]`) causes them to leave the state outside the brackets untouched. As these actions can hence be executed independently

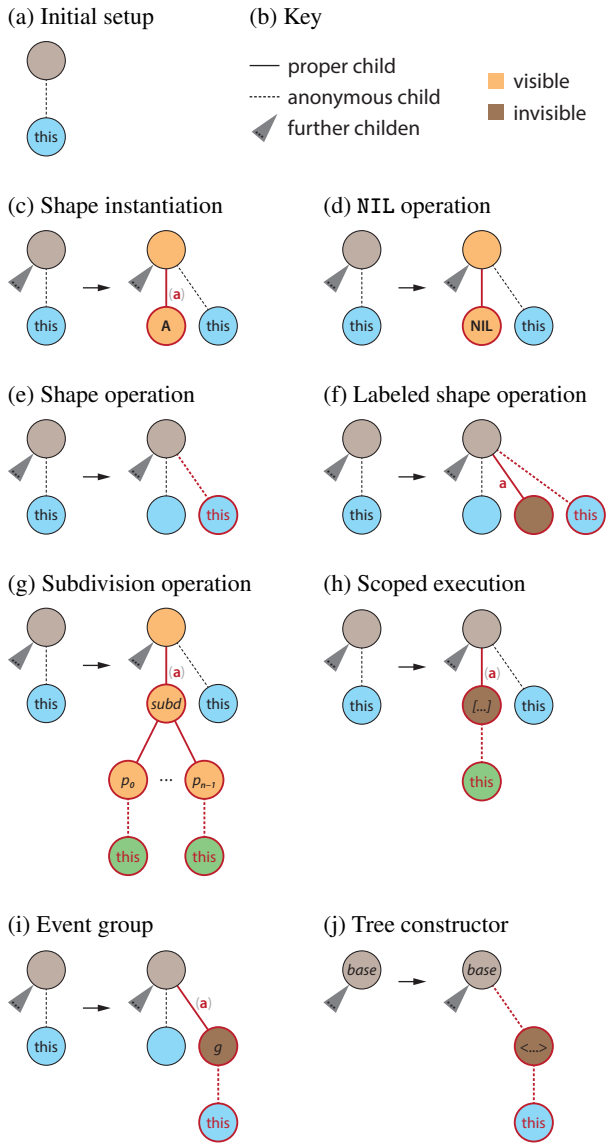

Figure 1: *Effects of actions on the shape tree (new elements are shown in red).*

from the actions following the closing bracket, we create a new derivation branch for these actions. To this end, a copy of the current shape is added as an (initially) invisible, proper child, and a copy of this child is added as anonymous child to it, representing the initial current shape of the new branch (Fig. 1 h).

Event groups The operation `group` introduces a group node by appending a new invisible, proper child, attributed with the group name, to the current shape's parent. Moreover, a copy of the current shape is added to this group node, making it the new current shape (Fig. 1 i). This causes the results of all succeeding actions to become descendants of the group node.

Tree construction When spawning a new derivation process with the construct `<` *actions* `>(`*base*`)`, a copy of the base shape is added as an anonymous child to this shape. It forms the root of the new shape tree and is further set up for executing the actions by adding an anonymous child representing the current shape (Fig. 1 j). Note that the new tree is actually part of the global shape tree, but with its root being an anonymous child, it is not known to and ac-

cessible from the base node; on the other hand, the root has full access to its base node with its ancestors.

Resumable shapes A resumable shape, created with the action `?`*name*`(`arg_0`, ...)`, is simply a copy of the current shape that is added as a visible, proper child and has both *name* and the arguments as special attributes.

3 CGA++ reference

In the following, we provide a concise reference for CGA++, primarily focusing on elements (including specific functions and operations) newly introduced or enhanced with respect to CGA shape. For more details on already existing built-in functions and operations, please refer to the latest CGA shape reference [Esri 2015].

Grammar Generally, a grammar consists of a sequence of (definitions of) rules, functions, events, and constants, potentially augmented with comments.

Expressions Our language supports Booleans, numbers, strings, lists, tuples, shapes, functions, and rules as values. Conceptually, expressions are free of side effects, and values are immutable. In particular, any entity involved in an expression will not be modified by evaluating this expression; however, an accordingly modified copy of it may be included in the result. (Internally, an implementation may of course directly modify the entity if it only occurs in the expression, thus avoiding making a copy and deleting the original (never-used-again) entity. Our prototype system tries to perform such in-place modifications whenever possible.)

Conventions In this reference, an ellipsis succeeding an item (e.g., an argument) indicates that a variable number of such items may be specified (at least one if the first item has an index of 1). Underlined arguments identify expression arguments. Generally, some arguments may be optional and default to specific values, but details are usually omitted in the interest of succinctness.

3.1 Functions

A named function is defined by

`func` *name*`(`$param_0$`, ...) =` *body*

where *name* must be unique, and *body* is an expression that can reference the parameters and also the function itself, allowing recursion. To refer to a function as an object, simply its name is used. This also applies to the built-in functions provided by our system. An anonymous function (instance) is obtained with

`[`$param_0$`, ...]`(*body*)

where any variable from outside the function used in the expression *body* is captured, forming a closure.

3.2 Rules

A named rule is introduced by

name`(`$param_0$`, ...) -->` *body*

where *name* uniquely identifies the rule, and *body* is a sequence of actions that forms the rule's body. To reference such a rule as an object, the rule's name is prefixed with `%`. If the rule has parameters, alternatively a reference that comes with an associated argument list can be obtained via `%`*name*`(`arg_0`, ...)`; the returned rule value captures the argument values and is parameterless. Moreover, anonymous rules are possible: the construct

`%(`$param_0$`, ...)<` *body* `>`

yields a new rule value, where the values of all variables from outside the rule that are referenced by the actions in *body* are captured.

Operations for rules

`invoke(`*rule*`,` arg_0`, ...)`
: Invokes the rule with the provided arguments.

`apply(`*rule*`,` arg_0`, ...)`
: Executes the rule in-place, i.e., the rule's actions are executed directly as part of the currently executed rule body.

3.2.1 Actions

A rule body comprises a sequence of actions, describing how a shape is refined. An action is either a symbol or an operation and may be preceded by a label (*label* `=` *action*) to identify its result.

Elementary actions

symbol`(`arg_0`, ...)`
: Instantiates the current shape and invokes the rule identified by *symbol* with the specified arguments.

`terminal(`*name*`)`
: Instantiates the current shape as a leaf shape. An alternative, concise syntax (for simple names) is simply the name followed by a period (e.g., `A.`).

`?`*name*`(`arg_0`, ...)`
: Instantiates the current shape as a resumable shape with the provided arguments.

`NIL`
: Adds an empty shape as leaf shape.

`stop`
: Aborts the execution of the current rule body, causing all remaining actions to be ignored.

`kill`
: Same as `stop`, but additionally removes all shapes resulting from the current rule body from the final result (by adding an empty shape as successor to each according leaf node).

`nop`
: Dummy operation with no effect.

`[` *actions* `]`
: Executes the actions in a new derivation branch. Therefore, they have no effect on the state outside the brackets.

If the end of a rule body is reached and the last action executed is an operation that only modified the current shape, an according terminal shape is automatically instantiated.

Basic shape-modifying operations (identical to CGA shape)

`t(`Δx`,` Δy`,` Δz`)`
: Translates the current shape's scope. If an argument is prefixed by `'`, it is interpreted as being relative to the scope's size and not absolute.

`r(`α`,` β`,` γ`)`
: Rotates the current shape's scope.

`s(`x`,` y`,` z`)`
: Resizes the current shape's scope.

`i(`*name*`)`
: Loads the specified asset as new geometry of the current shape.

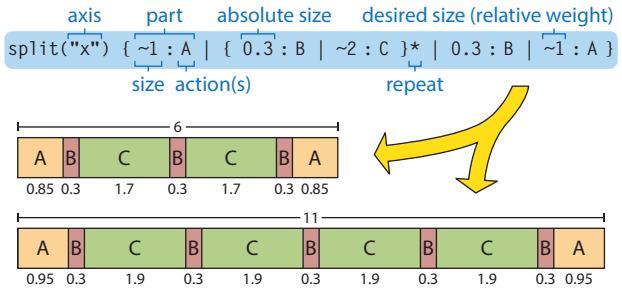

Figure 2: *Example for the* `split` *subdivision operation, showing results for two different input sizes (6 and 11).*

`extrude(`*height*`)`
Extrudes all faces of the current shape by the given amount.

Subdivision operations (enhanced with respect to CGA shape)

`split(`*axis*`)` *pattern*
Splits the current shape along one axis of its scope, where the split pattern is given as a list of *size*`:`*actions* pairs, and executes the specified actions for each resulting part. Syntactically, the list is delimited by braces and uses `|` as separator. Such lists can also be nested and may be repeated, which is indicated by a `*` suffix. Fig. 2 shows a concrete example. This operation provides a unified superset of the functionality offered by the original CGA shape's [Müller et al. 2006] `Subdiv` and `Repeat` operations.

`comp(`*component*`) {` $selector_1$ `:` $actions_1$ `|` ... `}`
Decomposes the current shape into its components (e.g., faces if *component* is `"f"`), and subsequently executes those actions for a particular component whose according $selector_i$ matches. If an `=` separator is used instead of `:`, the matching components are not refined individually but merged and processed as one single shape. In CGA++, $selector_i$ can be an arbitrary predicate expression, where the shape corresponding to the tested component is accessible via `$shape`.

`setback(`*distance*`) {` $selector_1$ `:` $actions_1$ `|` ... `}`
Sets back (offsets inward) those edges of the current shape by *distance* for which any specified $selector_i$ (an arbitrary predicate expression) matches. The edge and the face it belongs to are exposed as temporary shapes via `$edge` and `$face`; their respective indices are available via `$edgeIndex` and `$faceIndex`. Predefined selectors include functions such as `all(`*shape*`)`, where *shape* defaults to `$edge`, as well as the variable `remainder` to identify the shape remaining after all selected edges have been set back. For each edge, the actions of the first matching $selector_i$ are executed on the according edge offset polygon; the remaining shape is processed analogously. If an `=` separator is used instead of `:`, the matching components are merged and processed as one single shape.

3.3 Shapes

Shapes are first-class citizens in CGA++, and the notion of a shape also encompasses the node representing the shape in the shape tree and the subtree rooted in that node. Note that functions modifying an input (sub)tree actually return an accordingly modified new (sub)tree (whose root is an anonymous child, typically of the first input (sub)tree's parent).

Literals

`null`
Null node literal, indicating the absence of a node.

Elementary functions for creation and modification

`<` *actions* `>(`*base*`)`
Constructs a new tree by refining *base* in a new derivation process according to the actions and returning the resulting shape tree.

`tree(`*nodes*`)`
Returns a new tree with copies of the provided nodes (including all their proper descendants) as proper children.

`insert(`*tree*`,` *node*`,` *index*`,` *child*`)`
Returns a copy of *tree* where a copy of *child* has been inserted as *index*-th child of *node* (a node within *tree*).

`remove(`*tree*`,` *node*`)`
Returns a copy of *tree* where the subtree rooted in *node* has been removed.

`replace(`*tree*`,` *node*`,` *replacement*`)`
Returns a copy of *tree* with the subtree identified by *node* being replaced with a copy of *replacement*.

Basic accessor and query functions

`isNull(`*node*`)`
Checks whether *node* is the null node.

`parent(`*node*`)`
Returns the parent of *node* or `null` if none exists.

`children(`*node*`)`
Returns a list of all visible, proper children of *node*.

`leaves(`*tree*`)`
Returns a list all leaf shapes.

`nodes(`*tree*`,` *traversal*`)`
Returns a list of all visible (proper) nodes in *tree*, where *traversal* allows both for pre-order depth-first (`"dfs"`) and breath-first ordering (`"bfs"`).

`findAll(`*tree*`,` *predicate*`,` *traversal*`)`
Returns a list of all visible (proper) nodes in *tree* that satisfy a certain predicate.

`getNode(`*node*`,` *attribute*`,` *predicate*`)`
Starting from a given node, walks up the tree to determine the closest node that has the specified attribute. If the optional *predicate* is provided, the function further checks whether the attribute's value satisfies the predicate and continues the search if not.

Elementary operations

`include(`*tree*`)`
Embeds the given tree as a sibling of the current shape. Concretely, a copy of *tree* is added as proper child.

`include(`*trees*`)`
Embeds the given trees as siblings of the current shape.

`adopt(`*shape*`)`
Modifies the current shape such that it matches the specified one. To this end, an accordingly modified copy is added as anonymous child and made the new current shape.

Tree rewriting functions

`prune(`*tree*`,` *predicate*`)`
Returns a copy of *tree* where all subtrees rooted in nodes for which the predicate evaluates to true have been removed.

`refine(`*tree*`,` *rule*`)`
Returns a copy of *tree* where all leaf nodes have been refined by invoking the specified rule for them. Note that *rule* is an expression argument and hence evaluated for each leaf node. If an empty rule is provided for a node, no refinement happens for it. All refinements are executed in separate branches of a common new derivation process.

`continue(`*tree*`,` $name_0$ `=` $rule_0$`,...)`
Invokes the rule $rule_i$ for all resumable shapes whose name matches $name_i$, using the arguments associated with the respective resumable shape, and returns an accordingly refined tree.

Shape-modifying and subdivision functions

`t(`*tree*`,` Δx`,` Δy`,` Δz`)`
Returns a copy of *tree* with the scopes of all shape nodes translated by the given offset.

`s(`*tree*`,` x`,` y`,` z`)`
Returns a copy of *tree* where the root shape's scope has been resized to the provided dimensions, and the scopes of all proper descendants have been adjusted accordingly.

`transformScope(`*source*`,` *target*`)`
Returns a copy of *source* where the scopes of all shape nodes are subjected to the transformation that makes the root shape's scope identical to the scope of *target*, thus essentially fitting the source into the target.

`split(`*shape*`,` *axis*`)` *pattern*
Splits the shape along one axis of its scope, where the split pattern is given as a list of sizes (as in the case of the `split` operation but without actions), and returns a list of the resulting part shapes.

`comp(`*shape*`,` *component*`) {` $selector_1$ `|` ... `}`
Decomposes the current shape into its components analogously to the `comp` operation and returns a list whose i-th element is a list of those parts selected by $selector_i$. If $selector_i$ has an `=` suffix, the matching components are merged, forming a single part.

Spatial query functions

`inside(`$shape_1$`,` $shape_2$`)`
Determines whether $shape_1$ is inside $shape_2$.

`overlaps(`$shape_1$`,` $shape_2$`)`
Determines whether $shape_1$ overlaps $shape_2$.

`touches(`$shape_1$`,` $shape_2$`)`
Determines whether $shape_1$ touches $shape_2$.

Multi-shape operations

`union(`*shape*`)`, `union(`*shapes*`)`
Updates the current shape with the union of this shape and the provided shape(s).

`intersect(`*shape*`)`, `intersect(`*shapes*`)`
Intersects the current shape with the provided shape(s).

`minus(`*shape*`)`, `minus(`*shapes*`)`
Subtracts the provided shape(s) from the current shape.

`connectTo(`*shape*`)`
Constructs a connection from the current shape to the specified shape, where each edge of the current shape is joined with a corresponding edge of *shape* via a quadrilateral. The current and the provided shape must each consist of a single face with the same number of vertices.

Attributes Each shape can have a number of attributes. An attribute is identified by a name, and its value can be of any type supported by the language.

`set(`*name*`,` *value*`)`
Operation that sets an attribute of the current shape.

`get(`*shape*`,` *name*`,` *default*`)`
Function for retrieving the value of a shape's attribute. If the attribute does not exist, *default* is returned (or an error occurs if *default* is omitted). If a shape itself does not have an attribute, `get` recursively consults the attribute set of its parent. That is, a shape basically inherits the attributes from its ancestors. Therefore, a shape generally does not maintain a copy of its parent's attributes, which also facilitates locating shape nodes by attributes.

`has(`*shape*`,` *name*`,` *checkAncestors*`)`
Function for checking whether a shape has a certain attribute. The parameter *checkAncestors* specifies whether attributes inherited from the shape's ancestors are considered or not.

The attribute system is also used by several operations, which automatically set some predefined attributes, and further provides an interface to data intrinsic to the shape. Examples of related built-in attributes include (these are identically named in CGA shape):

`scope.tx`, `scope.ty`, `scope.tz`
Position of the scope (x, y, and z coordinates).

`scope.rx`, `scope.ry`, `scope.rz`
Orientation of the scope (rotation about x, y, and z axes in degrees).

`scope.sx`, `scope.sy`, `scope.sz`
Size of the scope (in x, y, and z dimensions).

`split.index`
Index of a part resulting from a `split` operation.

`split.total`
Total number of parts produced by a `split` operation.

`material.color.rgb`
Diffuse color of the shape's material.

Geometric query functions

`area(`*shape*`)`
Returns the surface area of *shape*, where `area(this)` is equivalent to CGA shape's `geometry.area()` function.

`volume(`*shape*`)`
Returns the volume of *shape*, where `volume(this)` is equivalent to CGA shape's `geometry.volume()` function.

3.4 Events

An event is defined and completely specified with the construct

`event` *name*`(`$param_0$`,...) {` *priority* `} =` *handler*

where *name* must be unique, *priority* is an expression that evaluates to a number (and defaults to 0 if omitted), and *handler* is an expression that evaluates to a list of rules. If parameters $param_i$ are provided, a whole family of events is defined; both *priority* and *handler* may refer to the parameters. When handling an event, *handler* will be evaluated to determine the further refinement of the current shapes of all participating derivation branches. The list of these shapes is exposed as variable `$nodes` within *handler*.

Operations

`event(`*name*`,` *group*`)`
Raises an event and suspends the current branch. Once the event instance got handled, the branch is resumed, initially executing the actions of the rule determined by the event's handler. If *group* is specified, the event is restricted to the subtree identified by the first group node with a matching name that is found when traversing the tree upward.

`group(`*name*`)`
Opens a new group with the given name by creating an according group node. All actions succeeding this operation belong to this group.

`wait(`*event*`)`
Suspends the execution until the specified (global) event gets handled (for the first time).

`wait(`*root*`,` *event*`,` *group*`)`
Suspends the execution until the specified event gets handled in the subtree identified by *root* (for the first time).

Event handler functions Several functions are provided to facilitate writing an event handler expression and to provide a concise syntax for common cases.

`pass(`*shapes*`)`
Loops over all shapes and returns `%<>` (i.e., a rule with no actions) for each in a list, which has the same size as *shapes*.

`foreach(`*shapes*`) { `*actions*` }`
Returns a list of rules compiled by looping over all shapes and evaluating `%<` *actions* `>` for each.

`forall(`*shapes*`,` *operation*`,` *toKeep*`) { `*actions*` }`
Returns a list of rules, featuring `%<` *operation*`(`*shapes*`)` *actions* `>` for the *toKeep*-th shape and `%< kill >` for all others. For example, `forall` offers a concise way to perform a Boolean operation (e.g., *operation* = `"union"`) involving all shapes and only continue at one shape with the result, abandoning all others.

`select(`*shapes*`) { `$selector_1$ ` : ` $actions_1$
` | `$selector_2$ ` = `$handler_2$ ` | ... }`
Loops over all shapes and checks for each which $selector_i$ (an arbitrary predicate expression) matches; it eventually returns a list of rules in the correct order. If no selector matches, `%<>` is returned for this shape. Otherwise, if the matching selector is followed by `:`, `%<` $actions_i$ `>` is returned. Conversely, in the case of an `=` separator, $handler_i$ is evaluated for the list of all shapes matching the selector (accessible via `$groupNodes`). This conveniently allows for nesting event handler functions.

`partitionByPred(`*shapes*`,` *predicate*`,` *groupHandler*`)`
First, partitions the provided list of shapes analogously to `groupByPred` (cf. Sec. 3.7), putting two elements into the same group if *predicate* evaluates to true for them (accessible via `$a` and `$b`). For each resulting group of shapes, the expression *groupHandler*, which can access the group's shapes via `$groupNodes` and should yield a list of according rules, is then evaluated. Finally, the obtained lists of rules are merged such that the partitioning gets reversed, i.e., the *i*-th rule corresponds to the result for the *i*-th shape.

`partitionByNumber(`*shapes*`,` *partition*`,` *groupHandler*`)`
Identical to `partitionByPred`, but partitions *shapes* analogously to `groupByNumber` (cf. Sec. 3.7), putting all elements for which *partition* evaluates to identical values into the same group.

3.5 Constants

Global constants can be defined with

`const` *name* `=` *value*

where *name* must be unique and *value* is an expression that can be evaluated at the beginning of the derivation process.

3.6 Control constructs

Selective evaluation or execution The following constructs can occur anywhere in a rule body and thus generalize the concept of conditional and stochastic rules.

- Conditional evaluation (within an expression) or execution (within a sequence of actions):

  `case { `$condition_1$`: `$result_1$ ` | ... | else: `*default*` }`

  The conditions $condition_i$ are sequentially evaluated until one yields true. In that case, $result_i$ constitutes the result (an expression or a sequence of actions, respectively), otherwise *default*.

- Stochastic evaluation or execution:

  `prob { `$probability_1$`: `$result_1$ ` | ... | else: `*default*` }`

  The result is determined randomly based on the provided probabilities (`else` $= 1 - \sum_i probability_i$).

Auxiliary variables Values can be assigned to variables and subsequently be used in an expression or within the arguments of actions via the following construct:

`with(`var_1 ` = `$value_1$`, ...,` *expression*`)`
`with(`var_1 ` = `$value_1$`, ...) { `*actions*` }`

3.7 Lists

Lists are ordered sequences of values of identical type. Many functions exist for working with lists; note that if these involve a modification of a provided list, conceptually, an accordingly modified new copy of it is created.

Elementary functions for creation and modification

`list(`$value_0$`, ...)`
Creates a new list with the given elements.

`append(`*list*`,` *value*`)`
Appends *value* to *list*.

`insert(`*list*`,` *index*`,` *value*`)`
Inserts *value* into *list* at position *index*.

`concat(`$list_1$`,` $list_2$`)`
Concatenates two lists.

`remove(`*list*`,` *index*`)`
Removes the *index*-th element from the list.

`removeFirst(`*list*`,` *value*`)`
Removes the first occurrence of *value* from the list.

`removeAll(`*list*`,` *value*`)`
Removes all occurrences of *value* from the list.

`sublist(`*list*`,` *first*`,` *size*`)`
Returns the sublist of *size* consecutive elements starting at position *first* as a new list. If *size* is omitted, the sublist extends to the end of *list*.

Basic accessor and query functions

`size(`*list*`)`
Returns the number of elements in a list.

`first(`*list*`)`, `last(`*list*`)`
Return the first and last element, respectively, of *list*.

`get(`*list*`,` *index*`)` or *list*`[`*index*`]`
Returns the *index*-th element of *list*.

`index(`*list*`,` *value*`)`
Determines the index of the first occurrence of a value. If no such element exists, −1 is returned.

`find(`*list*`,` *predicate*`)`
Determines the index of the first element for which *predicate* evaluates to true. If no such element exists, −1 is returned.

Transformation functions

`reverse(`*list*`)`
Returns a reversed copy of *list*.

`shuffle(`*list*`)`
Returns a randomly permuted copy of *list*.

`makeSet(`*list*`)`
Returns a duplicate-free copy of *list*.

`filter(`*list*`,` *predicate*`)`
Returns a copy of *list* where all elements for which *predicate* evaluates to false have been removed.

`map(`*list*`,` *expr*`)`
Evaluates the expression argument *expr* for each element of *list* and returns a list of the resulting values. Within *expr*, the respective element's value is accessible via `$value` and its index via `$index`.

`map(`$list_1$`,` $list_2$`,` *expr*`)`
Evaluates the expression argument *expr* for each element of $list_1$ and the according element in $list_2$, and returns a list of the resulting values.

`sort(`*list*`,` *before*`)`
Returns a sorted copy of *list* where the expression argument *before* indicates whether one element (`$a`) goes before another one (`$b`) or not; e.g., `$a < $b` results in numbers being ordered ascendingly.

`scan(`*init*`,` *list*`,` *combine*`,` *inclusive*`)`
Performs a scan of a list, sequentially combining the elements via *combine*, starting with *init*, and putting the results for each step in a new list. For example, `scan(1, list(2, 3, 5), $a * $b, true)` yields a list with elements 2, 6, 30.

Reduction functions

`countIf(`*list*`,` *predicate*`)`
Returns the number of elements in *list* for which *predicate* evaluates to true.

`any(`*list*`,` *predicate*`)`
Returns whether any element exists in *list* for which *predicate* evaluates to true.

`sum(`*list*`)`, `min(`*list*`)`, `max(`*list*`)`
Return the sum, the minimum value, and the maximum value, respectively, of a list of numbers.

`flatten(`*list*`)`
Merges a list of lists into a single list.

`reduce(`*list*`,` *combine*`)`
Performs a list reduction, where *combine* specifies how two elements (`$a` and `$b`) can be combined to a new element. For example, `sum(x)` ≡ `reduce(x, $a + $b)` and `flatten(x)` ≡ `reduce(x, concat)`.

Grouping functions

`groupByPred(`*list*`,` *predicate*`)`
Groups the elements of *list* such that if *predicate* evaluates to true for a pair of them (accessible via `$a` and `$b`), these two elements are put into the same group, and returns a list of the groups (each represented as a list of the group elements).

`groupByNumber(`*list*`,` *partition*`)`
Groups the elements of *list* such that all elements for which *partition* evaluates to identical values are put into the same group, and returns a list of the resulting groups.

3.8 Tuples

Tuples are fixed-size sequences of values of possibly different types. Built-in functions for using tuples include:

`tuple(`$value_0$`, ...)`
Creates a new tuple with the given components.

`size(`*tuple*`)`
Returns the number of components in a tuple.

`get(`*tuple*`,` *index*`)` or *tuple*`[`*index*`]`
Returns the *index*-th component of *tuple*.

3.9 Built-in support functions

Many common support functions exist. In the following, only functions used in the examples are listed.

Mathematical functions

`min(`*x*`,` *y*`)`, `max(`*x*`,` *y*`)`
Return the minimum value and the maximum value, respectively, of two numbers.

`floor(`*x*`)`, `ceil(`*x*`)`
Return the value of a number rounded downward and upward, respectively, to the nearest integral value.

`rint(`*x*`)`
Returns the rounded integral value of a number.

Random numbers

`p(`*probability*`)`
Returns true with the given probability and false otherwise.

`rand(`*min*`,` *max*`)`
Returns a uniformly distributed pseudo-random number from the range [*min*, *max*].

4 Examples

For selected examples, we provide some of the grammar definitions that are omitted in the paper in the interest of space.

4.1 Event handling function `subtractLargerOnes`

The handler `subtractLargerOnes` employed in snippet S1 of the paper yields rules that subtract from each shape all shapes with a larger area:

```
1  func subtractLargerOnes(shapes) =
2    with(byArea = sort(shapes, area($a) > area($b)),
3      isLargest = [s](index(byArea, s) == 0),
4      largerOnes = [s](sublist(byArea, 0, index(byArea, s))),
5      select(s : shapes) {
6        !isLargest(s): minus(largerOnes(s))
7      } )
```

First, the shapes are sorted by area (line 2). Using this sorted list, the function **isLargest** checks whether a shape is the largest one, and the function **largerOnes** returns for a shape the list of shapes that are larger (i.e., precede it in the area-sorted list). Utilizing these auxiliary functions, the list of rules for the further refinement of the input shapes is determined and returned (lines 5–7). The largest shape will be left unchanged, while each other shape will subtract all shapes that are larger than itself.

4.2 Façades

In the example presented in Sec. 6.3 of the paper, the following constants are used in addition to **upperFloorElems**:

```
1  const minWallW = 0.5
2  const maxWallW = 3.0
3  const minDoorW = 1.0
4  const maxDoorW = 1.4
5  const firstFloorElems = list(
6    tuple(%WindowCell(%SingleWindow), 0.6, 1.0),
7    tuple(%WindowCell(%WideWindow),   1.5, 3.0))
```

The façade elements (and wall pieces) are generated with a set of simple rules:

```
8  WindowCell(w) -->
9    split("y") { ~2: Wall | 1.1: invoke(w) | ~1: Wall }
10 DoorCell --> split("y") { 2.0: Door | ~1: Wall }
11 SingleWindow --> ...
12 DoubleWindow --> ...
13 WideWindow --> ...
14 Door --> ...
15 Wall --> ...
```

Note that **WindowCell** takes another rule as parameter. The omitted actions in lines 11–15 are mainly responsible for setting the material color and loading geometry from asset files (via the **i** operation).

To make some expressions more succinct, the following auxiliary functions are introduced:

```
16 func randomElem(l) =
17   with(n = size(l), get(l, min(floor(rand(0, n)), n - 1)))
18 func sublist2(l, first) =
19   case { first >= size(l): list()
20        | else:             sublist(l, first) }
21 func sublist3(l, first, last) = case {
22   first >= 0 && first < last: sublist(l, first, last-first)
23 | first >= 0 && last < 0: sublist(l, first)
24 | else: list() }
```

The function **getCells** recursively traverses a subtree to find all shape nodes that have been marked with the attribute **"floorCell"** (it does not consider descendants of such nodes):

```
25 func getCells(node) =
26   case { has(node, "floorCell", false):
27            list(node)
28        | else:
29            flatten(map(c : children(node), getCells(c))) }
```

These shapes, referred to as "cells", are then investigated by calling **legalDoorPlacements** to compile a list of placements of a door satisfying the constraints imposed by the design's objectives:

```
30 func legalDoorPlacements(f, cells) = with(
31   c_x0 = map(c : cells, get(c, "scope.tx")),
32   c_w  = map(c : cells, get(c, "scope.sx")),
33   f_x0 = get(f, "scope.tx"),
34   f_x1 = f_x0 + get(f, "scope.sx"),
35   flatten(map(cells, testCellForDoor(cells, c_x0, c_w,
36                                      f_x0, f_x1, $index))))
```

After determining the left x coordinates and the widths of all cells, each cell is investigated by the following function:

```
37 func testCellForDoor(cells, c_x0, c_w, f_x0, f_x1, i) =
38   with(
39     x1_prev   = case { i > 0: c_x0[i-1] + c_w[i-1]
40                      | else:  f_x0 },
41     x0        = c_x0[i],
42     x1        = x0 + c_w[i],
43     x0_next   = case { i + 1 == size(cells): f_x1
44                      | else:                 c_x0[i+1] },
45     wallL_w   = x0 - x1_prev,
46     cellW     = x1 - x0,
47     wallR_w   = x0_next - x1,
48     wallSpace = max(0, min(wallL_w, wallR_w) - minWallW),
49     candidatesWallL = case {
50       i > 0 && wallL_w >= minDoorW + 2 * minWallW:
51         list(tuple((x1_prev + x0) / 2,
52                    min(maxDoorW, wallL_w - 2 * minWallW)))
53     | else:
54         list() },
55     candidatesCell = case {
56       cellW + 2 * wallSpace >= minDoorW:
57         list(tuple((x0 + x1) / 2,
58                    min(maxDoorW, cellW + 2 * wallSpace)))
59     | else:
60         list() },
61     concat(candidatesWallL, candidatesCell))
```

At first, the x coordinates of the preceding cell's right side (or the façade's right boundary in case of the first cell), the considered cell's left and right side, and the succeeding cell's (or façade's) left side (or boundary) are determined and used to compute the widths of the wall piece on the cell's left, of the cell, and of the wall piece on the cell's right. Subsequently, it is checked whether centering the door with respect to the left wall piece would leave enough wall space on both sides of that piece (line 50), and if so, an according placement candidate is determined (lines 51–52). Center-aligning the door with the cell itself is considered next (lines 55–60); note that the door may be wider than the cell as long as sufficient wall space remains on both sides. Finally, a list of the identified candidates for door placement is returned.

The list of the cells of an upper floor is also used when filling the façade parts on the left and the right side of the placed door with elements from **firstFloorElems**:

```
62 FillFirstFloor(cells) --> with(
63   x0 = [s](get(s, "scope.tx")),
64   x1 = [s](get(s, "scope.tx") + get(s, "scope.sx")),
65   firstCell = find(c : cells, x0(c) >  x0(this)),
66   lastCell  = find(c : cells, x1(c) >= x1(this)),
67   relevantCells = sublist3(cells, firstCell, lastCell),
68   snapPositions = flatten(map(c : relevantCells,
69                               list(x0(c), x1(c)))))
70 { FillFirstFloor2(snapPositions) }
```

First, the subset of the cells that cover the currently considered façade part is determined (lines 65–67), using the functions **x0** and **x1** (lines 63–64) for a compact formulation. From the left and right sides of those cells, a list of snap positions is compiled (lines 68–69); these represent all positions that result in a vertical alignment across floors meeting the design's constraints. The further processing is then delegated to the following rule:

```
71 FillFirstFloor2(snapPositions) -->
72   case { size(snapPositions) > 1:
73            FillFirstFloor3(snapPositions)
74        | else:
75            Wall }
```

If less than two snap positions remain, a wall piece is created; otherwise, the following rule is used (it is not merged into `FillFirstFloor2` merely to more effectively use the limited column width in this document):

```
76 FillFirstFloor3(snapPs) --> with(
77   widths = map(p : sublist(snapPs, 1), p - snapPs[0]),
78   elems = filter(e : firstFloorElems,
79                  find(s : widths, e[1]<=s && s<=e[2]) >= 0))
80 { case {
81     size(elems) > 0:
82       with(e = randomElem(elems),
83            w = randomElem(filter(s : widths,
84                                  e[1] <= s && s <= e[2])),
85            skip = index(widths, w) + 2,
86            wallW = snapPs[0] - get("scope.tx"))
87       { split("x") {
88           wallW: Wall
89         |  w: invoke(e[0])
90         | ~1: FillFirstFloor2(sublist2(snapPs, skip)) } }
91   | else:
92       FillFirstFloor2(sublist(snapPs, 1)) } }
```

Considering a placement with the left side at the first snap position, a list of allowed element widths is determined from the other snap positions (line 77), and the elements from `firstFloorElems` with a compatible permissible width range are selected. If no such element exists, the next snap position is considered for placement (line 92). Otherwise, a permissible element and an allowed width are randomly chosen. Similar to `FillUpperFloor`, a wall piece and this element are split off (lines 87–90), and the remaining façade part is filled recursively (beginning at the next snap position to the right of the placed element).

References

Esri, 2014. Esri CityEngine 2014.1.

Esri, 2015. CGA shape grammar reference. http://cehelp.esri.com/help/topic/com.procedural.cityengine.help/html/cgareference/cgaindex.html.

Havemann, S. 2005. *Generative Mesh Modeling*. PhD thesis, TU Braunschweig.

Hohmann, B., Havemann, S., Krispel, U., and Fellner, D. 2010. A GML shape grammar for semantically enriched 3D building models. *Computers & Graphics 34*, 4, 322–334.

Krecklau, L., and Kobbelt, L. 2011. Procedural modeling of interconnected structures. *Computer Graphics Forum 30*, 2, 335–344.

Leblanc, L., Houle, J., and Poulin, P. 2011. Component-based modeling of complete buildings. In *Proceedings of Graphics Interface 2011*, 87–94.

Müller, P., Wonka, P., Haegler, S., Ulmer, A., and Gool, L. V. 2006. Procedural modeling of buildings. *ACM Transactions on Graphics 25*, 3, 614–623.

Schwarz, M., and Wonka, P. 2014. Procedural design of exterior lighting for buildings with complex constraints. *ACM Transactions on Graphics 33*, 5, 166:1–166:16.

Steinberger, M., Kenzel, M., Kainz, B., Müller, J., Wonka, P., and Schmalstieg, D. 2014. Parallel generation of architecture on the GPU. *Computer Graphics Forum 33*, 2, 73–82.