Selection Expressions for Procedural Modeling

Haiyong Jiang, Dong-Ming Yan, Xiaopeng Zhang, and Peter Wonka

Abstract—We introduce a new approach for procedural modeling. Our main idea is to select shapes using selection-expressions instead of simple string matching used in current state-of-the-art grammars like CGA shape and CGA++. A selection-expression specifies how to select a potentially complex subset of shapes from a shape hierarchy, e.g. "select all tall windows in the second floor of the main building facade". This new way of modeling enables us to express modeling ideas in their global context rather than traditional rules that operate only locally. To facilitate selection-based procedural modeling we introduce the procedural modeling language SELEX. An important implication of our work is that enforcing important constraints, such as alignment and same size constraints can be done by construction. Therefore, our procedural descriptions can generate facade and building variations without violating alignment and sizing constraints that plague the current state of the art. While the procedural modeling of architecture is our main application domain, we also demonstrate that our approach nicely extends to other man-made objects.

Index Terms—Procedural modeling, building modeling, selections, grammars

1 INTRODUCTION

Procedural modeling is useful to create a variety of buildings without modeling each building individually, e.g., to synthesize a large environment. A popular approach for procedural building modeling uses grammars, e.g., CGA shape [1]. The main idea of a grammar is to derive a design hierarchically, typically relying on splitting operations. First, a mass model is generated. Then, the side faces of the mass model are extracted as facade polygons. Next, the facade polygons can be split into either columns or floors. After that, floors can be split into tiles and later tiles into walls, windows, or doors. In our work, we would like to improve upon two problems prevalent in this approach.

First, CGA shape provides only limited opportunities to coordinate the different branches of the derivation. For example, a rule could be invoked for a tile somewhere on a building to place a window and a balcony. This rule now needs to decide locally how the window and balcony should be designed such that the design decisions are coordinated with all other elements of a building. It is easy to underestimate how difficult that really is. Specifically, the correct alignment of elements is extremely difficult to model even for buildings of moderate complexity. This issue, among others, was tackled by CGA++ [2]. The proposed solution is to introduce language constructs that enable better communication between the different parts of a design. While this results in noticeable improvements, CGA++ still inherits the limitation of CGA shape that a design is broken down hierarchically with small shapes ultimately trying to make decisions locally (see Fig. 1). In this work, we would like to explore a departure from this traditional grammar-based modeling. Our key idea is to use a global view to describe key design decisions involved in the modeling. For example, instead of windows deciding locally what size, alignment, and type they should have, we would like to write global rules that describe where to put what types of windows. We thereby move from rules of the form: $label \rightarrow actions$ to rules of the form $selection\text{-}expression \rightarrow actions$. In other words, while previous work mainly focused on improvements to the right-hand side of a rule, we propose extensions to the left-hand side of a rule.

Second, the hierarchical splitting approach used by CGA shape and CGA++ has several drawbacks. There are multiple ways to view the same building. For example, looking at a facade one might be interested to express a modeling operation in terms of the floors, in terms of the columns, or for a subset of tiles. The problem is not so much the hierarchical splitting in itself, but the fact that the rule writing forces a building to be split into only one single hierarchy. If the rule writer commits to a floor-based subdivision, it becomes very difficult to express modeling operations that need to coordinate between multiple columns and vice versa. Further, a single hierarchy leads to a larger amount of rules than necessary and to rules that have no semantic. For example, in Fig. 1(c) the facade is first split into a left region, a door column, and a right region. Then the individual regions are split into floors, etc. This structure imposes a hierarchy that has unnecessary intermediate regions that are difficult to name semantically and pose problems for formulating selections (e.g. all the colored regions in Fig. 1(c)). To overcome these limitations, we propose a solution to simultaneously manage multiple hierarchies to support multiple views of the data without splitting the shapes. Fig. 1 illustrates problems with the previous approach and the advantages of our solution on a selected example.

- *H. Jiang is with the National Laboratory of Pattern Recognition (NLPR), Institute of Automation, Chinese Academy of Sciences, Beijing 100190, China, KAUST, Thuwal 23955-6900, Saudi Arabia, and University of Chinese Academy of Sciences, Beijing 100049, China. Email: haiyong.jiang@nlpr.ia.ac.cn.*
- *D.-M. Yan and X. Zhang are with the National Laboratory of Pattern Recognition (NLPR), Institute of Automation, Chinese Academy of Sciences, Beijing 100190, China, and University of Chinese Academy of Sciences, Beijing 100049, China. Email: yandongming@gmail.com, xiaopeng.zhang@ia.ac.cn.*
- *P. Wonka is with KAUST, Thuwal 23955-6900, Saudi Arabia. E-mail: pwonka@gmail.com.*
- *D.-M. Yan is the corresponding author.*

Manuscript received April 19, 2005; revised August 26, 2015.

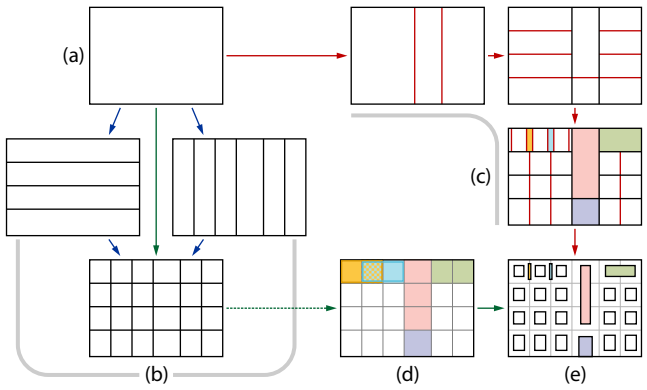


Fig. 1. Illustration of modeling paradigms of SELEX and CGA shape. (a) Empty facade. (b) Grids are used to enable different views of the data, e.g. rows, columns, sub-grids, or individual grid cells. (c) Splitting problems with current approaches, because merging of cells is not possible: if performing splits to establish multi-cell regions, complex and hard-to-maintain sequences of alternating vertical and horizontal splits are required (e.g., first horizontally into three columns, then each column vertically (these are three separate splits, need at least two different split rules), then each row-fragment horizontally again; even in this simple example already hard to keep all splits in sync and (in the case of CGA shape) to find meaningful symbol names). (d) In our approach we can select arbitrary rectangular sub-grids of a grid and place elements relative to these. This enables modeling in a natural and semantically meaningful way. (e) Elements are often arranged according to a grid to simplify alignment, but single elements may span multiple grid cells or be placed in between grid cells. In particular, incorporating elements that straddle two cells, each of them containing a further element, such as the yellow and blue ornaments in the top floor, entails a complex split structure. We support overlapping selections, e.g. a single-cell selection to place each window and a double-cell selection to place ornaments.

In practice, there are two types of modeling scenarios where our approach can provide significant advantages over previous work. First, we can express facade designs in such a way that the alignment between elements will be correctly maintained when resizing a facade or when introducing procedural variations. Second, we can model mid-rise and high-rise buildings that have a fuzzy boundary between mass model and facade. Our procedural framework is the first that provides a reasonable solution to model these buildings.

In summary, we make the following contributions:

- We propose the concept of selection-based procedural modeling to replace traditional grammar derivations where shapes are selected based on labels.
- We can generate procedural facade descriptions that always exhibit correct resizing behavior, i.e. keeping alignments and important constraints, in contrast to previous work.
- We can model the geometry of many mid-rise and high-rise buildings that could not be modeled before in a reasonable fashion.

2 Related Work

Procedural modeling has been successfully employed in a variety of areas, such as plant modeling [3] and street modeling [4], [5]. We refer the reader to a recent survey paper for a review of procedural modeling of virtual worlds [6].

Of particular interest in our review is the way how different production systems match shapes that will be modified. In the context of architecture, the seminal work of Stiny introduced the concept of shape grammars [7]. These grammars work on a configuration of line segments and the underlying selection operation is based on matching a given arrangement of lines to sub-shapes of the current configuration. Stiny later suggested a simplification to set grammars [8] where matching works by identifying an element from a set. Wonka et al. [9] proposed control grammars that can select all tiles of a subgrid of a single regular grid and Lipp et al. [10] used descriptors to encode derivation paths for interactive selections. Compared to previous work, our proposed framework is significantly more powerful and flexible. This enables a user to describe architecture in a more succinct as well as in a more natural way that is semantically meaningful and more similar to how a human would describe a building.

Related to matching is the control of the derivation order in a grammar. Examples of existing approaches are rule priorities [1], evaluation phases [11] and construction stages [12]. The control of derivation order was significantly extended by CGA++ [2]. In general, the expressiveness of CGA++ is very high so that many things can be modeled somehow. However, CGA++ also inherits the limitations of CGA shape that we try to overcome in this paper. A more detailed discussion on grammar-based procedural modeling can be found in the following course notes [13].

An alternative to pure procedural modeling is the combination of optimization with declarative or procedural descriptions. There exist multiple recent frameworks specifically targeted at the modeling of facades and buildings [14], [15], [16], [17], [18], urban layouts [19] and also multiple approaches pitched for more general procedural modeling [20], [21], [22]. This type of work has different types of goals, often it is the simplification of the user experience to make modeling easier for novice users. By contrast, our goal is to push the envelope of what architecture can be described and modeled.

Another important avenue of recent work is the combination of machine learning and procedural modeling. One goal is inverse procedural modeling, where grammar rules and grammar parameters can be learned from data. One example approach is Bayesian model merging [23] that was adapted by multiple authors for learning grammars [24], [25]. By implementing this approach we noticed that current grammar rules are too difficult to process in machine learning applications, because there are too many dependencies between rules. One motivation of our work is to derive a modeling system that can be easier combined with machine learning techniques such as Bayesian model merging.

Most commercial procedural modeling systems for urban modeling in industry build on the paper by Mueller et al. [1], e.g. [26], [27], and [28]). One important contribution to procedural modeling in industry is the use of a graph-based modeling interface, e.g. Sceelix [27]. VoxelFarm [28] evolves voxel representations. These ideas are orthogonal to the concept of selection-based procedural modeling.

Our system builds on ideas presented in other areas of computer science for specifying selections. Specifically, we evaluated selections in jQuery [29] and XPath [30]. We found that XPath was a more advanced way to specify selections and used it as inspiration for our language. Similar to XPath,

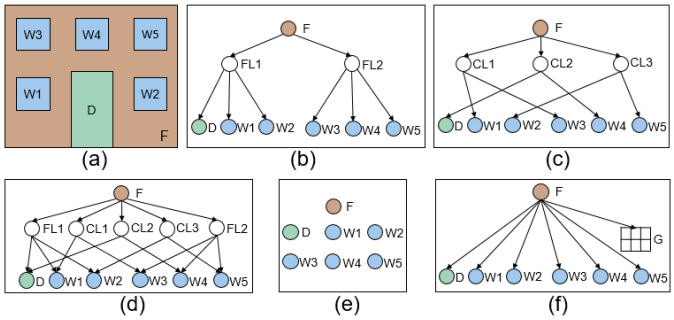


Fig. 2. Different design choices to organize shapes.

we also emphasize the tree structure of the data as each selection navigates down from the root node. Different from XPath, we introduce many extensions for urban modeling, e.g., selections on grids and grouping operations on lists.

3 Selection-based language for procedural modeling

In this section, we describe our language SelEx. First, we will explain the representation that the language evolves, a shape hierarchy consisting of multiple types of shapes (Sec. 3.1). Second, we give a brief description of the language itself in Sec. 3.2. Third, we explain *selection-expressions* in Sec. 3.3. Selection-expressions are the most important concept introduced in this paper. Fourth, we give more details on the modeling operations, called actions, used by the language in Sec. 3.4.

3.1 Shape definitions

Here, we discuss details of our *shape* concept. One important design decision is how to organize the collection of shapes evolved by the language. One design choice is simply to operate on a set of shapes without any hierarchy (See Fig. 2(e)). This is very general, but also quite difficult to realize as it is hard to formalize certain selections. Another possibility is to organize the given shapes in a tree. That is the classical approach used by grammar-based procedural modeling. This is simple to realize, but it requires committing to one particular hierarchy. Especially for facades, multiple hierarchies (or trees) exist at any time and modeling operations are typically expressed in different hierarchies. For example, sometimes windows should be selected based on floors (rows) and sometimes based on columns of a facade (See Fig. 2(b,c)). One possibility to overcome this limitation is to explicitly create and maintain multiple hierarchies in parallel (See Fig. 2(d)). However, this easily leads to consistency problems and there is a very large number of possible intermediate shapes to group other shapes. Our solution to this problem is to use grids as *virtual shapes*. These grids can generate any subregion as auxilliary shape on the fly without explicitly having to generate and manage the subregion (See Fig. 2(f)). Floors and columns in a facade are just special cases of subregions that can be generated. That means that there are two different shape types: *virtual shapes* as explained above and *construction shapes*. We call all other shapes that are not virtual shapes construction shapes. The construction shapes are very similar to shapes in previous work, e.g. CGA shape and CGA++. Virtual shapes can be placed on a 2D construction shape. They typically have multiple rows and columns and therefore consist of multiple cells. These grids guide the placement of other construction shapes, but they are not used to split construction shapes. For a construction shape, multiple virtual shapes can exist which enables us to model complex layouts on a given polygonal shape. Virtual shapes can be used in three ways. First, they can be used to locate a position. In most cases, we place a shape by first selecting a cell of a grid and adding a shape at a location inside the grid cell. Second, virtual shapes facilitate the selection of construction shapes that are contained inside them. For example, to select shapes in the same row or column. Third, virtual shapes help to define the resizing behavior. The grid specification includes information about the spacing of rows and columns and the way rows and columns repeat if enough space is available. As a result, the alignment of shapes will be guided globally instead of locally as in CGA shape.

We use a set of built-in attributes for each shape. *Label* is the name of the shape, e.g. "window". Labels can be unique or shared among multiple shapes. *Type* indicates if the shape is a virtual shape ("virtual"), a construction shape ("construction"), or a cell ("cell") of a virtual shape. *Dim* is a binary variable indicating a 2D or 3D shape. As topology information we store a link to the parent, a list of children, and a list of neighbors. The scope describes an oriented box in 3D space using variables describing a local coordinate frame (xaxis, yaxis, zaxis), a position in R^3 (denoted by xpos, ypos, zpos), and size information (xsize, ysize, zsize).

Our language evolves a hierarchy of shapes and stores them in a tree. Shapes are added to the tree by certain functions in our language. Each shape can only have one parent and only construction shapes are able to have children. In our current version, shapes cannot be deleted, but they can be set to invisible. The root node is a shape with a label "root".

2D shapes are often used to create subdivisions on other 2D shapes, e.g. facades or windows. 3D shapes are typically used to model elements that are extruded from the facade or hanging structures, e.g., a balcony or a window ornament. In our shape hierarchy construction shapes have children that are either *attached shapes* or *contained shapes*. An attached shape is a 3D shape that is linked to a 2D shape. Sometimes the attached shape has a face contained inside the 2D shape, sometimes the shapes do not touch. For example the 3D shape of a balcony could be attached to a 2D shape describing a window position inside a facade. A contained shape is a 2D shape that is contained inside its 2D parent shape or a side face of a 3D parent shape. A *connected shape* is a 2D shape that shares an edge with another 2D shape. Topologically, a connected shape is considered a neighbor in the context of selection-expressions. See Fig. 3 for an illustration.

3.2 Introduction to the language

The proposed procedural modeling language SelEx executes one command at a time. A command can be either a rule or a variable assignment. A rule has the following form:

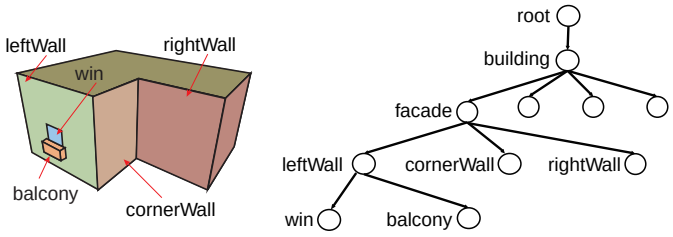


Fig. 3. For the current shape (leftWall), we illustrate a contained shape (win), an attached shape (balcony) and a connected shape (cornerWall).

```
#C1:
facW = 17.6; facH = 12.8;

#C2:
{ <> -> addShape("facade", ...); }

#C3: split facade into cells and set it as the working node
{ <[label=="facade"]> -> createGrid("main", ...); }
{ <[label=="facade"]/[label=="main"]> -> setHeadNode(); }

#C4: add a door touching the ground;
{<cell()[colLabel=="mid"][rowLabel=="gnd"]>
-> addShape("door", ...); }

#C5: add the glass windows above the door
{ <cell()[colLabel=="mid"][rowIdx>1][::groupCols()]>
-> addShape("glass", ...); }

#C6: add ledges at the left side of the top floor.
{ <cell()[colLabel=="left"][rowLabel=="top"][::groupPairs()]>
-> addShape("ledge", ...); }

#C7: add the two-cell window at the right side of the top floor.
{<cell()[colLabel=="right"][rowLabel=="top"][::groupRows()]>
-> addShape("win4", ...); }

#C8: add windows on the ground floor.
{ <cell()[colLabel in ("left", "right")][rowIdx==1]>
-> addShape("win1", ...); }

#C9: add windows on the second and third floor.
{ <cell()[colLabel in ("left", "right")][rowIdx in rowRange(2,-2)]>
-> addShape("win2", ...); }

#C10: add windows in the left side of the top floor.
{ <cell()[colLabel=="left"][rowLabel=="top"]>
-> addShape("win3", ...); }

#C11: generate walls to fill the space.
{ <root()/[label=="facade"]> -> coverShape(); }
```

Listing 1. SELEX code for Fig. 1(e).

```
selection-expression -> actions;
```

where the selection-expression selects a list of shapes from the current shape tree, and actions are commands executed on each shape in the list, e.g. shape refinements. For example, in Listing 1 command $C2$-$C11$ are rules. An assignment is of the form:

```
identifier = expression;
```

where the identifier denotes a variable and expression evaluates to a value. For example, in Listing 1 commands labeled $C1$ are variable assignments. We use an object model to process data of different types: Boolean, float, integer, string, list, pair, shape, and construction-line. A list is a list of other types of objects and we also support lists of lists. A pair consists of two objects. The first object should be comparable, and can be a number, a string, or a Boolean value. The second object can be any kind of object.

Our language also supports common language constructs, such as random-selection, conditionals, evaluators, and assignments. The most important aspect of SELEX are selection-expressions that can select shapes from a shape tree. These will be described in the next sub-section. The details of the language are described in the additional materials.

3.3 Selection-based procedural modeling

A **selection-expression** selects a list of shapes from the shape tree using selectors interleaved with the operator `"/"`. Each selector takes a list of shapes as input and returns a list of shapes. The implicit input to the first selector is a list containing the root node of the shape tree. The operator `"/"` takes a list of shapes as input and executes the remaining commands for each shape in the list.

Selectors are grouped in selector sequences that consist of specialized selectors that can have three different types: topology-selector (e.g. child, descendant), attribute selector (e.g. `"[label=="window"]"`) and group selector (e.g. `"[::groupRows()]"`). The selectors cannot be arbitrarily mixed within a sequence and they need to occur in the given order. A selection-expression has the following form:

```
< [topoS] [attrS | groupS]* / [topoS] [attrS | groupS]* / ... >
```

That means within each selector sequence, there are zero to one topology-selectors (topoS), zero to many attribute selectors (attrS), and zero to many group selectors (groupS). The topology-selector needs to come first (mainly to improve the performance of our implementation), but the order of the attribute and group selectors can be interleaved. If a selection-expression is empty, it returns the input. When a shape that does not exist is specified, the corresponding selection will return an empty shape and the rule will not be executed. Before going into the details of the individual selectors we give a simple example in Fig. 4 on an abstract graph. In the following, we give a shortened description of the individual selectors. An exhaustive description is given in the supplementary material.

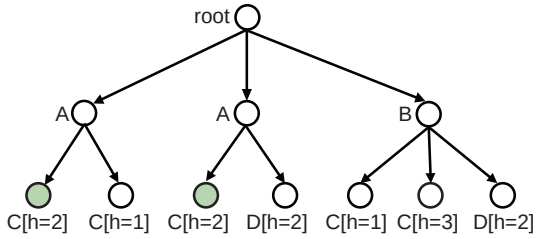


Fig. 4. The nodes selected by the selection-expression `"<[label="A"] / [label="C"][h>=2]>"` are highlighted in green. First, the selection-expression selects the children labeled `"A"` from the root node. For these two nodes labeled `"A"` all children labeled `"C"` with an attribute value $h >= 2$ are selected.

A topology selector takes a list containing a single shape as input, and outputs a list of shapes with the specified topology relation to the input shape. A topology selector has the form [topology-function()] using one of the following functions: `"child()"`, `"descendant()"`, `"parent()"`, `"root()"`, `"neighbor()"`, and

"contained()". The function "contained()" is default for a virtual shape and "child()" is default for a construction shape.

An attribute selector takes a list of shapes as input, and returns a list of shapes whose attributes satisfy some conditions. In its basic form, the selector has the form [attributename comparison value]. The comparison operator is specified as in other programming languages, e.g. ==, <=, >=, ! =, and "in". Examples are "[label = "facade"]" and "[label in ("window_arch", "window_rect")]". Alternatively, in its more general form, an attribute selector is simply a boolean expression and attributes can also be derived online by executing functions on a shape. Examples are "[isEmpty()]", "[numCols() > 4]", or "[toShapeX(0.5) > 2]". The first example selects shapes that do not have child construction shapes, the second example selects shapes that have more than four columns, and the third example selects shapes based on the x-coordinate of the shape center. The function "toShapeX(0.5)" scales the input value 0.5 by the xsize of a shape. An important function is "pattern(regex, pat)" which checks if the pattern character of "regex" at the index position of a shape matches "pat". For example, "pattern("(AB)*", "A")" tests if an input shape is at an odd index position, and "pattern("A(B)*A", "A")" tests if an input shape is at the first or last position of an input list. Also, more complex examples are possible and meaningful, e.g. "pattern("AC(ACCA)*CA", "A")", but regular expressions have inherent ambiguities when multiple repetitions are used. For example, for the case "pattern("A*B*A*"", "A")", we try to keep an equal amount of repetitions. Nested repetitions, e.g. of the form (BA*B)*, are also ambiguous and currently not supported. Functions "isEven()" and "isOdd()" are special cases of the command "pattern(regex, pat)", which check if a shape has an even or odd index in a list of selected shapes. We use "rowIdx" and "colIdx" as the topological position of a cell with respect to the region spanned by input virtual shapes. For example, "rowIdx==1 && colIdx==1" specifies the left bottom cell of a grid.

A group selector takes a list of shapes as input, and applies grouping operations to return a list of combined shapes. A group selector only operates on virtual shapes and regroups subregions, e.g. combines cells of a virtual shape into floors. The unique aspect of the group selector is that it not only selects cells, but also groups cells together to form larger subregions of a grid. Group selectors are implemented using grouping functions. This results in selectors of the following form: [::grouping-function()]. Function "groupRows()" and "groupCols()" merge adjacent virtual shapes (i.e. cells) with the same row or column index. Function "groupRegions()" merges all adjacent virtual shapes, which form one or multiple rectangular regions. We show an example in Fig. 5. Function "groupEach(n)" merges every n adjacent virtual shapes. Function "groupPair()" generates all possible pairings of two subsequent shapes. For example, given "ABCD" it will return "AB", "BC", and "CD". Function "cells()" decomposes a virtual shape as a list of virtual shapes with one cell. Function "sortBy(d, pos, order=1)" sorts the selected shapes by relative position "pos" in the dimension "d" with the increasing (order=1) or decreasing (order=0) order. We show three additional example selections in Fig. 6. For the example in Fig. 1, Listing 1 shows the selection to insert the door in $C4$, the selection to insert the large window above the door (shown in pink) in $C5$, and the selection for the large window on the top right (shown in green) in $C7$. The selection to place dividing elements on the top left between windows is shown in $C6$.

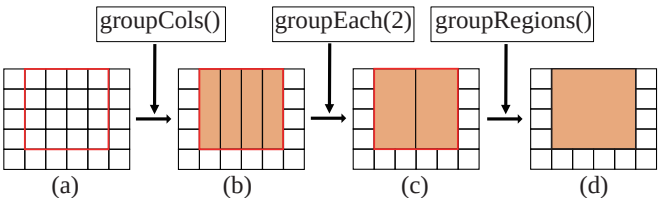


Fig. 5. Given the red selected region of (a), command "groupCols()" groups the cells into columns to create a list of virtual shapes (shown in orange in (b)). Then command "groupEach(2)" groups adjacent columns to yield a list of two regions shown in (c). At last, command "groupRegions()" combines the virtual shapes into a single region shown in (d).

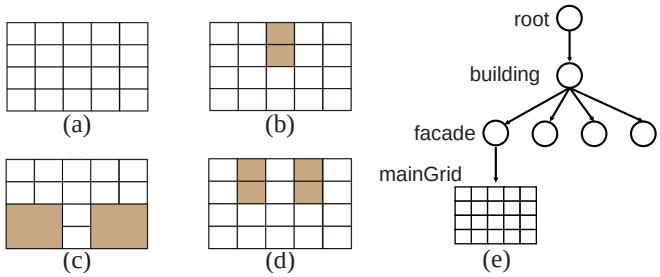


Fig. 6. (a): A grid used to illustrate multiple selections (b): "<descendant()[label=="facade"]/ [label =="mainGrid"]/ [type=="cell"] [rowIdx in (1,2)] [colIdx== 3]>" (c): "<descendant()[label=="facade"]/ [label=="mainGrid"]/ [type=="cell"] [rowIdx in (3,4)] [colIdx in (1,2,4,5)] [::groupRegions()]>" (d): "<descendant()[label=="facade"]/ [label == "mainGrid"]/ [type=="cell"] [rowIdx in (1,2)] [colIdx in (2,4)] [::groupCols()] [::cells()]>".

3.4 Actions

Actions are described by a sequence of functions. In our current implementation, the user can only use built-in functions that are provided by our system. The sequence of functions is executed for each shape in the shape list generated by the selection-expression. These actions mainly follow previous work, so we refer the reader to the supplementary material for a detailed description.

The most important actions are functions used to create new shapes, e.g. "addShape", "attachShape", "coverShape", and "connectShape". These shape commands create new shapes and add them to the current shape hierarchy. The parent of a new shape can be specified explicitely or implicitely using default values. Typically, the parent is the input construction shape, or in case of a virtual shape the first ancestor that is a construction shape. For example, in Fig. 7(a) we show an example of the addShape function. For the example in Fig. 1 the code in Listing 1 makes extensive use of the "addShape" function, since the example is only 2D.

Another aspect of modeling complex residential buildings is that they require a careful control over what geometry is actually being generated. We therefore use commands to create geometry inside shapes that are not covered by other shapes with the "coverShape" function (see Fig. 7(b)). Further, we provide several functions to create virtual shapes (grids). An example is provided in Fig. 7(c).

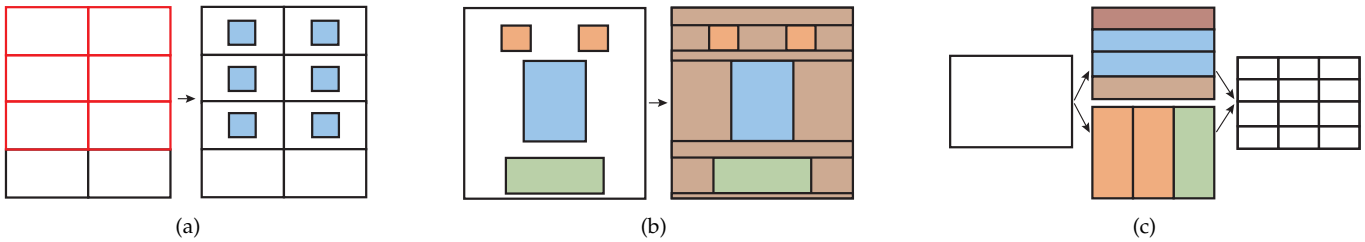


Fig. 7. (a) We show a simple facade where 6 cells were selected. After executing the action *addShape* for each of the cells we obtain the result with new construction shapes added (shown in blue). (b) We show a single shape containing four shapes. After executing the action *coverShape* for the shape, the geometry shown in brown is generated. (c) We show the generation of a single grid as child of a construction shape defined by two sequences of construction lines. After executing the action *addGrid* for the grid cell, a virtual grid shape is generated.

To locally improve the layout, we also provide versions of the addShape action that can specify constraints (e.g., same size, symmetry, distance to the boundary, overlapping and alignment between shapes). The constraints are then resolved locally using quadratic programming with linear constraints. Since these actions are novel, we devote the next subsection to describing them in detail.

3.5 Constraint Functions

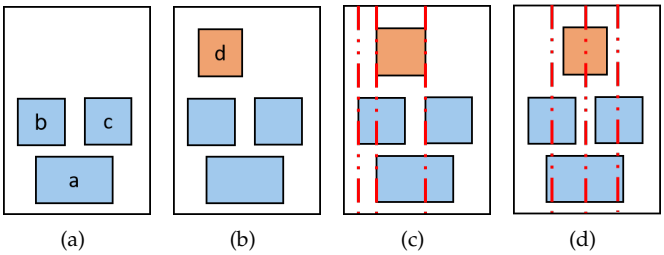


Fig. 8. We compare snapping with and without the use of labels. (a) Given an initial layout with shapes a,b, and c. (b) A new shape labeled d is added. (c) In snapping without labels the shape d snaps to the closest snap lines, e.g. the left of shape a and the left of shape c. (d) Using snapping with labels allows for finer control, e.g. snapping the x-coordinate of the center of shape d to the x-coordinate to the center of shape a.

In general, it is difficult to specify the location of a shape in a stochastic grammar, because we cannot know exactly what shapes have been placed previously and where they are. Our solution is to setup an optimization problem. The variables in the optimization are the lower left corner position (x^*, y^*) and size (w^*, h^*) of a newly inserted shape in conjunction with the two functions `"addShape"` and `"attachShape"`.

Constraint Specification: In SELEX, a sequence of constraints can be specified with the following command:

```
constrain(constraint1, constraint2, ...),
```

We then translate the specified constraints into linear constraints of a mixed integer quadratic programming formulation. The linear constraints we currently support are alignment, symmetry, distance to the boundary, and intersection avoidance. In the following we describe alignment in more detail and leave the description of other constraints to the additional materials.

Alignment: We considered two design choices for this problem based on auxiliary lines (or planes) called snap-lines. The first design choice is to decide if snap-lines should be placed explicitly by a command, or implicitly defined by the boundaries of previously generated shapes. We chose to use the boundaries of previously generated shapes. This is less general, but also creates less overhead for generating snap-lines. The second design choice is to decide if snap-lines should be named with a label or be unlabeled. We chose to use labeled snap lines. We reuse the labels of previously inserted shapes as snap-line labels. New shapes that are generated with snapping enabled only consider snap-lines with a specified label. In Fig. 8, we show an example comparing snapping with and without labels. We observed that labeled snap-lines are useful to make traditional snapping [1] [27] more robust.

Alignment can be specified between two shapes. The input shape and a reference shape specified by a shape label. We support the following types of alignment: `"left"`, `"right"`, `"top"`, `"bottom"`, `"center-x"`, `"center-y"`, `"one2two-x"`, `"one2two-y"`.

```
snap2(shapeLabel1, snapType1, shapeLabel2, snapType2, ...)
```

For example, the function `"constrain( snap2("window1", "left"), snap2("window1", "center-x"))"`, specifies that the input shape should be left and center-x aligned with a shape labeled `"window1"`. We visually illustrate examples of `"left"` and `"one2two-x"` in Fig. 9.

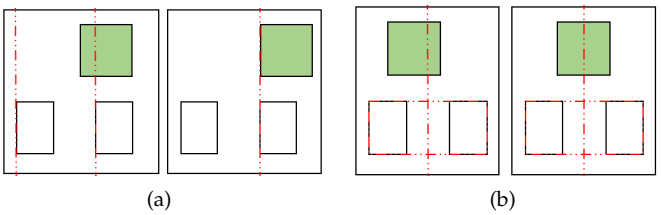


Fig. 9. Two example alignments. In each subfigure, the left side is derived without alignments, while the right side is derived with alignments. (a) Alignment "left" aligns the input shape in green to a reference shape in white. (b) Alignment "one2two-x" aligns the input shape in green to the center of the bounding box of two white reference shapes. The red dashed line denotes the snapping position, while the red bounding box marks the bounding box of two reference shapes.

In the detection step, shapes with the specified label are selected. For example, alignment `"left"` aligns one element to another as shown in Fig. 9(a), but alignment `"one2two-x"`, `"one2two-y"` try to align one element to the bounding

box of two nearby elements with the given label. Thus, the bounding boxes are returned as the selected shapes as shown in Fig. 9(b). Among the selected shapes, the final candidates are shapes that satisfy the specified alignments to input shape within a threshold (half of the width or height of the input shape in our experiments). For example, left alignment will test if the difference between left edges of the selected shape and the input shape is within the threshold.

In the second step, snapping position s_i is calculated. For example, left alignment will use the nearest edge of the selected shape with respect to the left edge of the input shape as illustrated in Fig. 9(a). For alignment `"one2two-x"`, `"one2two-y"`, the nearest horizontal or vertical center position relative to the horizontal or vertical center position of the input shape will be used, as illustrated in Fig. 9(b).

At last, alignment can be achieved by adding the alignment constraints to an optimization. Assuming we would like to align to a position s_i, the constraint is formulated as: $x^* + \alpha_i * w^* = s_i$, where α_i equals to $-0.5, 0.0, 0.5$ for left, center-x, and right alignment, respectively.

If multiple alignments are specified within a `snap2` function, one of these alignments should be enforced. For example, the function `"constrain(snap2("window1", "left", "window1", "center-x"))"`, specifies that the input shape should be either left or center-x aligned with a shape labeled `"window1"`.

Selecting one constraint from n equality constraints of a form $x^* + \alpha_i * w^* = s_i$ can be reformulated as a set of linear constraints as follows:

$$\begin{aligned} x^* + \alpha_i * w^* - s_i + M * b_i \geq 0, &\forall i \in [1, n], \\ x^* + \alpha_i * w^* - s_i - M * b_i \leq 0, &\forall i \in [1, n], \\ \sum_j^n b_j = n - 1, & \\ b_i \in \{0, 1\}, &\forall i \in [1, n], \end{aligned} \tag{1}$$

where M is set to a large constant (10000 in our code).

The specified constraints may be compatible or not. To tackle potential conflicts in the constraints, we incrementally check the compatibility. If no conflict is detected, we just add the constraint to the constraint set. Incompatible constraints are dropped. That means, that constraints specified first implicitly have a higher priority. At last, an optimizer will enforce the selected constraints to obtain optimal shape parameters. The constraints are only checked once when a shape is added.

The optimization uses a quadratic objective function with linear constraints. The variables in the optimization are the (final) position (x^*, y^*) and size (w^*, h^*) of a newly inserted 2D shape. The objective function encodes that the final position and size should be close to the approximate specification (x, y, w, h) in a least squares sense:

$$(x^* - x)^2 + (y^* - y)^2 + (w^* - w)^2 + (h^* - h)^2, \tag{2}$$

Since the variables can be floating point or integer, the problem is a Mixed Integer Quadratic Programming (MIQP) problem. In our implementation, Gurobi [31] is utilized as a solver.

4 Modeling Example

The example shown in Figure 10 illustrates virtual shapes and selection-expressions. Details about this example are given in the figure caption. This illustration should give the reader a good intuition about the capabilities of our approach. In addition, we highlight two of the modeling steps in Figs. 11(a) and 11(b). The figures show the current 3D model to the left and the current shape tree to the right.

5 Results and Discussion

We present experimental results of our procedural modeling system, using modeling examples of 2D facades, 3D buildings, chairs, tables, shelves, and parking lots. Our system is implemented using C++ and Python. All experiments shown in this paper are conducted on a computer with dual-core 2.70 GHz Intel Xeon CPUs and 64GB RAM.

5.1 Comparison to traditional splitting rules

We compare our selection-based procedural modeling strategy with traditional split-based modeling. The main point of the comparison is to demonstrate that traditional split-based modeling leads to invalid designs in more complex cases. The idea of split-based modeling is to hierarchically split a design by the rules of the grammar. This modeling strategy is employed by the original version of CGA shape and it can also be used in other modeling systems, e.g. [27] and [28]. We create results using CGA shape as the representative system for split-based modeling.

We use a small dataset of 10 facades. The goal of the data set was to highlight difficult shape configurations and alignments, see Fig. 12. For example, the alignments between elements of different sizes in $F027$, $F030$ and $F032$, and the alternating ornament styles in $F004$, $F006$.

We first designed a deterministic size-independent procedural description using SELEX and CGA shape that takes a rectangle of arbitrary size as input. We tried to ensure that the competing CGA shape description is created in a reasonable way. Therefore, we use three human users to model the facades using CGA shape in addition to an automatic method proposed by Wu et al. [32]. We then select the best design for the comparison. We compare the results on an input rectangle of the same size as the given reference image. We can observe that SELEX as well as CGA shape can replicate the input layouts, but the description length and the resulting shape complexity varies. We use the number of shape operations and the number of commands (in SELEX) / number of rules in CGA shape as an approximate measure for the complexity of the procedural description. From the results in Table 1 we can see that our description uses fewer commands and operations than the CGA shape description. Currently, our commands are a bit longer than CGA shape rules therefore the actual file size is comparable. Further, we can see that the shapes generated / managed by our procedural description are fewer than the shapes generated by CGA shape. Only after subdividing all shapes for rendering, our description yields the same number of shapes (polygons) as CGA shape. This is a key advantage of our method. While the higher number of shapes produced by the CGA shape description is not a problem in itself, the number is indicative of how many auxiliary splits are encoded in the CGA shape grammar that do not carry semantics (see Fig. 13 for an example). These splits will

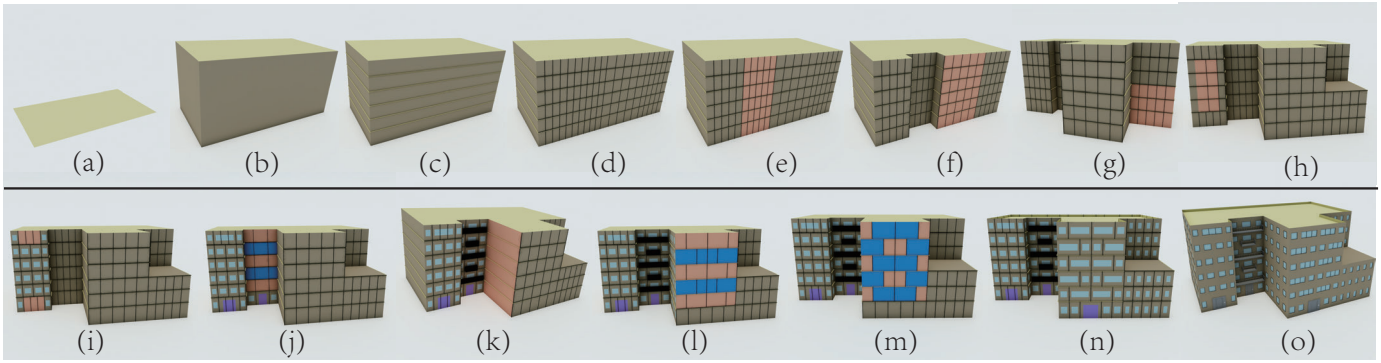


Fig. 10. A modeling example. (a) User-specified footprint polygon. (b) The footprint is extruded into a building. (c) All facades of the building are split into floors by adding a grid as a virtual shape. This grid works similar to construction lines in technical drawing and does not actually split the building geometry. (d) Each facade inherits the floor information and is split into a finer grid by specifying columns. The columns are labeled with "colLeft", "colMidLeft", "colMidRight", "colRight". (e,f,g) The columns are selected by label "colMidLeft", "colMidRight", "rowDown" and push/pull operations are applied to form the mass of the building. (h) A subregion on the left side is selected and a sub-grid is added. (i) Multiple-cells in the main grid are selected to add shapes spanning across several cells. (j) Even/odd rows in a sub-region from the second row to the last row are selected. (k) A connecting shape is selected. (l) Even/odd rows are selected and different sub-grids are added. (m) First the wide and then the narrow columns are selected to add wide and narrow windows respectively. (n) Additional windows and doors are added. (o) Assets are added.

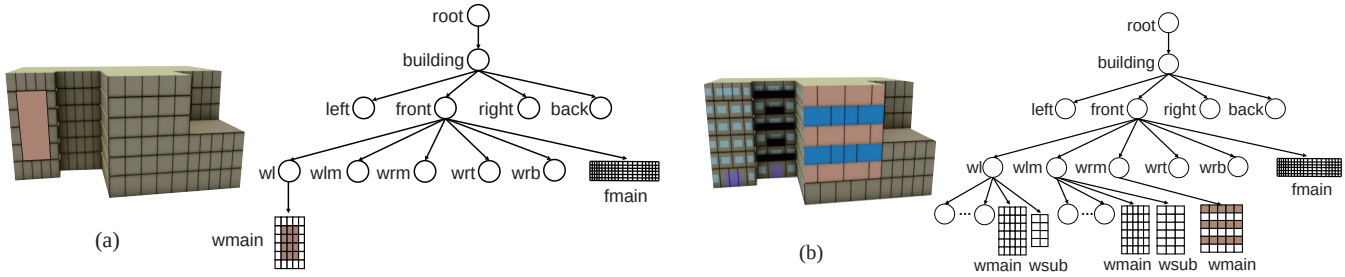


Fig. 11. Two examples with shape tree. (a) For the given model on the left with the shape tree on the right, we select shapes using the following selection expression: "<descendant()[label=="wl"] / [label=="wmain"] / [type=="cell"] [rowIdx in rowRange(2,-2)] [colIdx in colRange(2,-2)] [::groupRegions()]>". The selected shapes are highlighted in red. (b) For the given model on the left with the shape tree on the right, we select shapes using the following selection expression:"<descendant()[label=="wrm"] / [label=="wmain"] / [type=="cell"] [rowIdx in rowRange(2,-1)] [::groupRows()] [pattern("(ab)*", "a")]>". The selected shapes are highlighted in red.

Fig. 12. Ten facades used in our evaluation and comparison. The facades were selected to exhibit various forms of alignment and constraints.

partially contribute to the problems CGA shape has when resizing layouts as described in the next experiment.

Second, we compared the results on input rectangles of different size. For each layout we decided on a set of constraints that are essential to the layout and that should be preserved during resizing. These constraints are same-size constraints, alignment constraints (top, bottom, left, right, middle), alignments of one element to multiple other

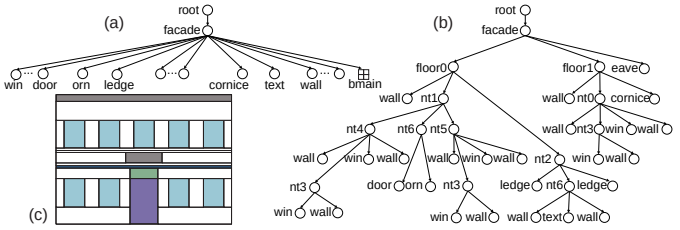


Fig. 13. Different shape trees are generated by split grammars such as CGA shape and SELEX. We show the input layout (c), the shape tree of CGA shape (b), and the shape tree of SELEX (a). Basically, SELEX has a flat structure, resulting in far fewer intermediate shapes. For example, there is no shape for floors in SELEX, but it is easy to access floors and columns by querying the virtual grid shape.

TABLE 1
Comparison of ten facades modeled with CGA shape and SELEX. #rule is the number of rules in CGA shape, and the number of commands in ours. #op denotes the number of shape operations, which are the split and repeat operations in CGA shape, and actions in SELEX. #finalShapes gives the number of shapes in the final SELEX model and the number of terminal shapes in CGA shape. #allShapes counts the number of shapes managed. These are terminal and non-terminal shapes in CGA shape and contruction and virtual shapes in SELEX. The difference between #finalShapes and #allShapes indicates that the structure of SELEX is flat compared to CGA shape.

		F001	F002	F003	F004	F006	F027	F030	F032	F060	F062
CGA	#rule	14	34	15	28	22	29	31	27	21	24
	#op	14	34	15	28	22	29	31	27	21	24
	#finalShapes	63	94	36	106	94	254	152	104	68	60
	#allShapes	99	148	56	161	136	348	218	170	113	96
Ours	#rule	**7**	**10**	**6**	**13**	**11**	**19**	**14**	**22**	**12**	**14**
	#op	**7**	**16**	**9**	**20**	**14**	**22**	**23**	**24**	**14**	**16**
	#finalShapes	64	**87**	**32**	**102**	**78**	258	**119**	108	75	62
	#allShapes	**66**	**89**	**34**	**104**	**80**	**262**	**121**	**114**	**77**	**65**

elements, and the minimal and maximal empty space to the boundary of the layout. We implemented a constraint specification language and an automatic constraint checking algorithm for this purpose. A more detailed description of the modeled constraints is given in the additional materials. Distinguishing important from accidental constraints is a modeling problem in itself, but several constraints are generally accepted as important. For example, if windows are aligned across floors in the input, then this is an essential aspect of the design. We only encoded such essential constraints for our tests. We evaluated the modeling quality by generating 12 resized versions of each facade and automatically checking the preservation of the expected constraints. The results in Fig. 14 show that CGA shape often violates important constraints. Even a few violated constraints can mean that the resulting layout has undesirably low quality. We illustrate this using examples shown in Figs. 15 and 16. In general, here are some reasons why CGA shape fails in the resizing comparison:

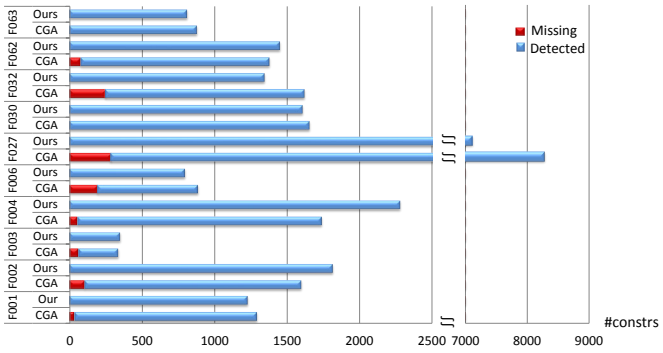


Fig. 14. Comparison with CGA shape on the alignment constraints. We produce 12 resizing results for each facade. The whole table can be found in the appendix. Our method can keep all alignments, while CGA shape fails to keep all alignments on eight out of ten facades. The total number of constraints is different because the resizing behavior of CGA shape and SELEX differs (see Fig. 15).

Fig. 15. Illustration of the resizing problem. From left to right: input layouts, results of CGA, and our results. Important constraints are violated using CGA shape. For both facades CGA shape splits first into floors and then each floor into walls and windows. In the top row we show the worst case result. There is a slight difference in the width of the wall shape in the second floor on the left and the right of the facade. After resizing, the repeat rule for the windows on the left and the right produces a different number of repetitions. This is due to the fact that the shape on the left side is a bit smaller. On the bottom we show the common case of misalignments. The windows in floors two to four are aligned in the input layout. However, CGA shape destroys the alignment between floors three and four. This problem is inherent when modeling with a single splitting hierarchy and impossible to fix in a reasonable manner.

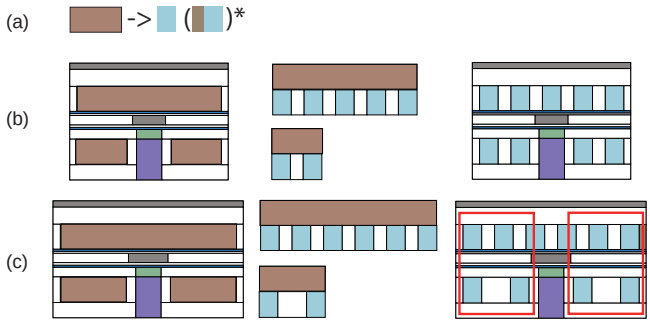


Fig. 16. Alignments collapse because of the inconsistent scaling of walls when repeating some shapes in the example. (a) The derivation of non-terminals is shown, where * means repetition of some shapes. (b, c) Two rows illustrate the derivation for different sizes of a layout, respectively. The first column shows intermediate results when deriving a grammar. The middle column givens the derivation of the non-terminals. The right column is the final result. The misalignments are highlighted in the red bounding boxes.

- CGA shape needs to commit to vertical or horizontal splits. This typically means that the alignment of elements will be correct in the chosen splitting direction and incorrect in the other direction after resizing.
- CGA shape splits into too many auxiliary regions. Auxiliary regions require CGA shape to commit to a distribution of the available space early on. This distri-

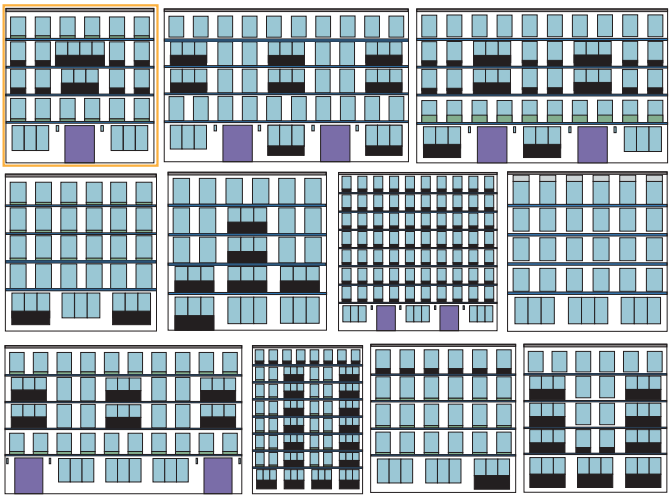


Fig. 17. Facade variations generated by our system. The input layout is highlighted in the yellow box. Other results are sampled variations for facades of different size.

bution conflicts with the alignment of shapes created further down in the derivation tree.

- CGA shape has difficulty modeling layouts that need multiple overlapping hierarchies or that place elements in muliple "cells".
- CGA shape has difficulty modeling patterns that need global control, e.g. $A(BA)*$ ornaments on top of identical windows.

Third, we generated stochastic procedural descriptions using SELEX. Since CGA shape already fails on the simpler task of creating size-independent descriptions, this problem will only get worse when introducing procedural variations. In Fig. 17 we show several variations of a single facade generated by our system. We can observe that all alignments are correctly maintained.

5.2 Comparison to selection-based modeling

Alternatively, it is possible to use existing software to mimic our proposed selection-based modeling. This is possible for example in CGA shape by quering indices that are automatically generated and also in CGA++. In addition, that requires a lot of pre-calculation and flow control statements in the rules. Since a prototype of CGA++ is not easily available, we also chose to compare to CGA shape for this comparison. The main purpose of the comparison is to show that using selection-based procedural modeling with correct alignment is not easily feasible in current modeling systems. We originally wanted to use the 10 facades from the previous comparison, but we found it a bit too difficult to correctly implement the alignment in CGA shape. We therefore generated three new stochastic facade models that can be resized and that can produce procedural variations to illustrate the problem. One model is very simple and two models have medium complexity. The results are shown in Fig. 22. Not only are the descriptions in CGA shape longer, they are also very difficult to generate, because there is no easy way to encode the alignment correctly. The full code is provided in the supplementary materials.

5.3 Residential building modeling

Here we provide some examples of residential buildings modeled with our system. We believe that these buildings are generally too difficult to model with existing procedural modeling software. We took photographs or renderings of existing buildings stemming from an internet search for *residential building*. We selected the buildings in such a fashion, that their complexity exceeds the modeling capabilities of CGA shape. Specifically, we were looking for a non-trivial interplay between the facade structure and the mass model. For these buildings it is no longer possible to model the facade as a tapestry on extruded shapes. We first took these images as reference and encoded these buildings using SELEX. Then we generalized the description to make the buildings size independent. We show the recreated buildings and selected resized versions in Figs. 18, 19, and 20. Subsequently, we generated more complex procedural variations of these buildings. We show selected variations in Fig. 18 right and Fig. 21. In these examples, keeping alignments across different facade regions is difficult. This difficulty increases with 1) the number of regions a facade has (e.g. because the mass model is not simply an extruded polygon, but has many faces). 2) the number of possible structural variations through resizing, stochastic parameters, or stochastic rule selection. 3) the complexity of the patterns or element arrangements.

5.4 Modeling furniture

Our current implementation is suitable to model a wide range of man-made objects. Here, we show the application of our framework to modeling furniture. Since most furniture is built using symmetry, alignment, and a regular arrangements of parts, our framework is ideally suited to model arrangements of individual furniture parts. We generated stochastic grammars that can generate many variations of desks, chairs, tables, shelves, and beds and show selected models in Fig. 23. The main point of these examples is to highlight the part arrangements. Therefore, we only use boxes as assests for the individual shape parts.

5.5 Modeling parking lots

Our framework is also suitable to model layouts, e.g. shopping malls, floor plans, parks, and parking lots. We selected parking lots as representative example and implemented a stochastic grammar to generate variations of parking lots (see Fig. 24 for a generate model). Parking lots typically consist of strips of parking lots (single or double) that exhibit interesting variations in spacing between dividers and parking lots, irregular spacing due to handicapped lots, and alignment between lots and objects such as trees and lamps. In addition, there is the alignment and arrangement of the individual strips that has to be handled by the grammar.

5.6 Time performance

The derivation of the procedural buildings is fairly fast, since the optimization to enforce the alignments and other linear constraints is only performed locally. The buildings shown in this paper are generated in 1 to 23 seconds. See Table 2 for the timings and the complexity of the models

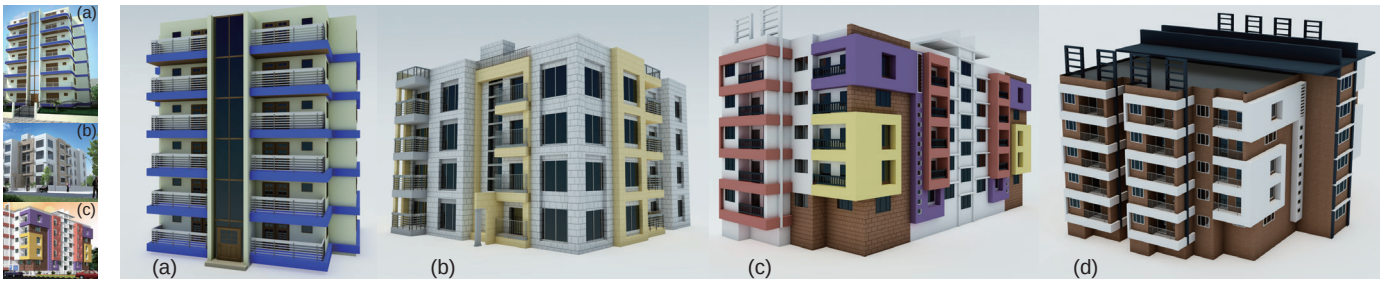


Fig. 18. Selection-based procedural modeling enables us to design complex residential buildings. We created procedural models based on reference images shown on the left. Our corresponding procedural models are show in the middle (a,b,c). The advantage of procedural models is that they can be used to generate many variations. In (d) we show one such variation stemming from the building shown in (c). The difficulty of these examples is the interplay between facades and mass model.

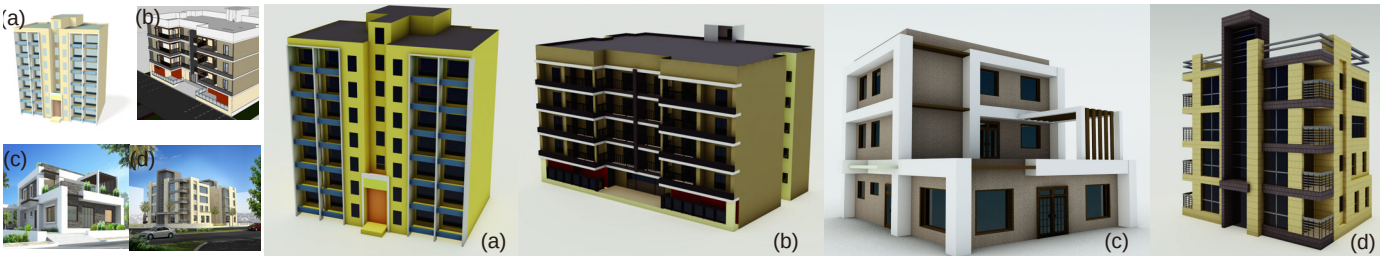


Fig. 19. Modeling examples of 3D buildings and their corresponding reference images.

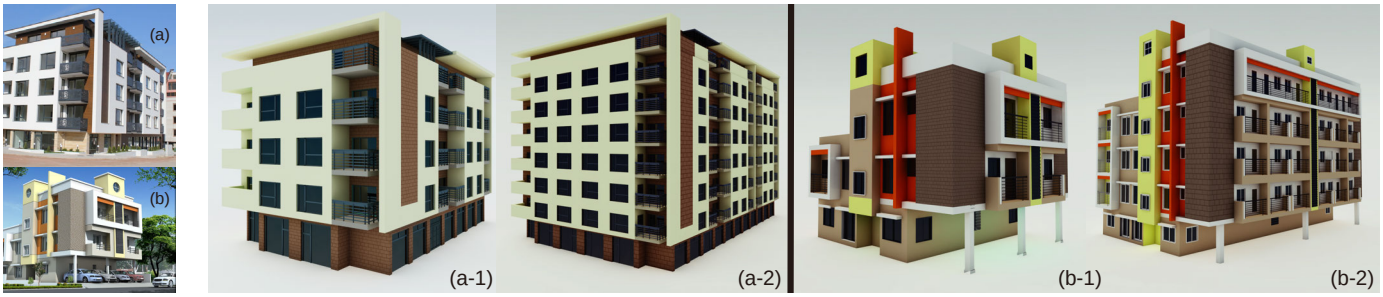


Fig. 20. The resizing results of selected 3D buildings. We show the reference images (a,b), the modeling results recreating the reference images (a-1, b-1) and one selected resizing result for each building (a-2, b-2).

Fig. 21. For two reference images, we show our procedural reconstruction and four variations.

shown in this paper. For the timings we measure the generation of the buildings in their original size.

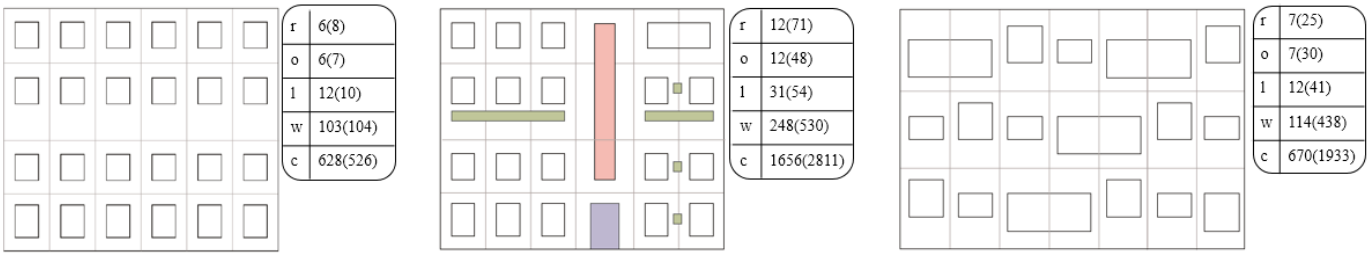


Fig. 22. Three stochastic facade models. For each model we show statistics for SELEX and then for CGA shape in brackets. We show the number of rules *r*, the number of operations *o*, the number of lines *l*, the number of words *w*, and the number of characters *c*.

Fig. 23. Modeling examples of SELEX for indoor scenes, including chairs, shelves, beds and tables.

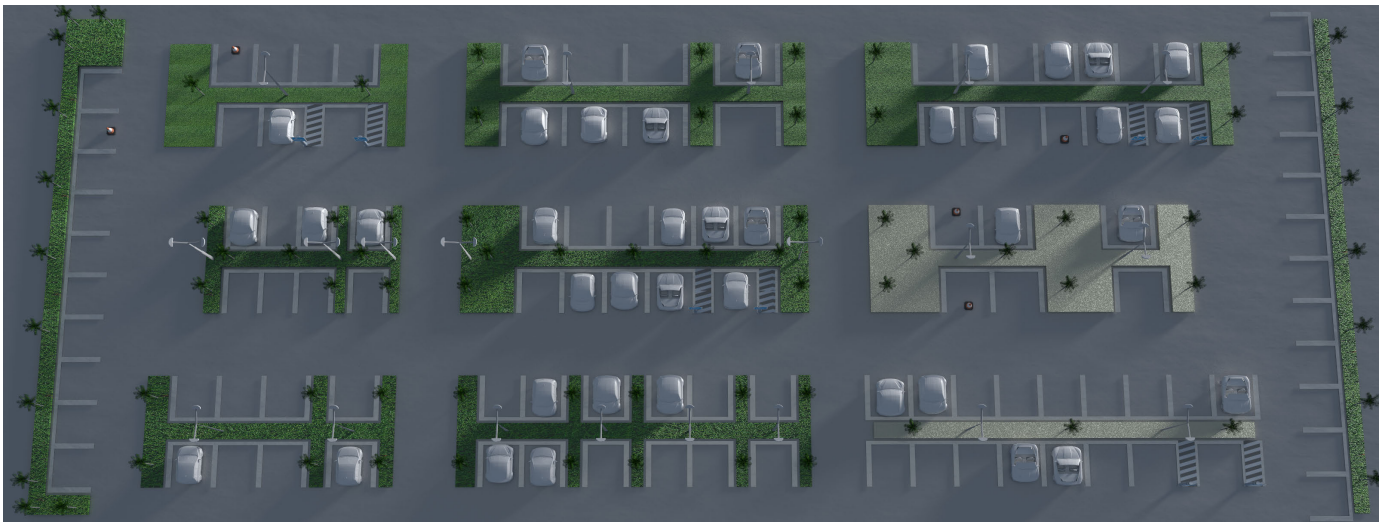


Fig. 24. Modeling example of SELEX for a parking lot.

TABLE 2
Statistics for 3D building modeling. The average time for building generation is 6.24s.

	B001	B002	B003	B004	B005	B006	B007	B008	B009	B010
#rules	31	79	72	78	81	80	99	45	80	86
#ops	55	110	89	123	119	123	189	72	151	138
#allShapes	270	370	409	734	672	764	268	329	3716	996
#finalShapes	194	265	355	590	546	653	202	288	2080	789
Time(s)	1.11	2.47	2.52	5.08	5.43	8.38	3.38	1.85	23.19	8.99

5.7 Discussion on the shape hierarchy

A fundamental design choice of implementing selection-based procedural modeling is how to create and update the shape hierarchy. The solution proposed in the paper explicitly models the hierarchy as new shapes are inserted as children into the shape tree, based on the commands that are used to create new shapes.

An alternative method would add shapes to an unstructured set of shapes and then automatically compute one or multiple shape hierarchies. Our method is significantly more efficient and simpler to implement. However, we consider the second alternative to be an interesting challenge for research combining procedural modeling with machine learning.

5.8 Limitations

In our current implementation, we do not model curved shapes directly, but only import them as assets. Therefore

(a) (b)

Fig. 25. Two examples that are beyond the modeling capacity of SELEX. (a) A building with curved surfaces. (b) Processing complex polygons, e.g. polygons with holes.

we cannot model most curved facades (see Fig. 25(a)). We also did not implement operations to process complex polygons, e.g. splitting operations for polygons with an arbitrary number of vertices or polygons with holes (see Fig. 25(b)). Finally, we did not spend any time to optimize the syntax of our modeling language, because that makes the examples harder to read. As a result, the descriptions are probably quite a bit longer than they have to be.

6 Conclusions and Future Work

We presented a novel approach to procedural modeling using expressive selections instead of the simple matching of labels in current grammars. To this end, we introduce a procedural modeling language to encode procedural objects using selection expressions that enables us to model with a global view of the data. Our results show that our procedural description can generate stochastic variations correctly, e.g. correct resizing behavior, in contrast to the current state of the art. As challenging examples we demonstrated models of mid- and high-rise buildings that have no reasonable description in other procedural modeling languages like CGA shape and CGA++. In future work, we would like to combine SELEX with machine learning techniques to learn a procedural shape space from a large set of input models.

Acknowledgements

We would like to thank Michael Schwarz for developing an initial version of the language and procedural modeling system with us in 2015/2016. He proposed the concepts of virtual, attached, and contained shapes and contributed to the development of the navigation-based selection and constraint handling. He also created Figure 1 and suggested the term selection expression.

We also had multiple helpful discussions with Peter Rautek and Liangliang Nan about SELEX. Fuzhang Wu helped with the comparison to CGA shape. Further, we would like to acknowledge funding from the Visual Computing Center (VCC) at KAUST through the CARF program and the National Natural Science Foundation of China (61620106003, 61802362, 61772523, and 61331018).

References

[1] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. Van Gool, "Procedural modeling of buildings," *ACM Transactions on Graphics*, vol. 25, no. 3, pp. 614–623, 2006.

[2] M. Schwarz and P. Müller, "Advanced procedural modeling of architecture," *ACM Transactions on Graphics*, vol. 34, no. 4, pp. 107:1–107:12, 2015.

[3] P. Prusinkiewicz and A. Lindenmayer, *The Algorithmic Beauty of Plants*. New York: Springer-Verlag, 1990.

[4] Y. I. H. Parish and P. Müller, "Procedural modeling of cities," in *Proceedings of SIGGRAPH 2001*, 2001, pp. 301–308.

[5] G. Chen, G. Esch, P. Wonka, P. Müller, and E. Zhang, "Interactive procedural street modeling," *ACM Transactions on Graphics*, vol. 27, no. 3, pp. 103:1–103:10, 2008.

[6] 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.

[7] G. Stiny, "Introduction to shape and shape grammars," *Environment and Planning B*, vol. 7, no. 3, pp. 343–351, 1980.

[8] ——, "Spatial relations and grammars," *Environment and Planning B*, vol. 9, no. 1, pp. 113–114, 1982.

[9] P. Wonka, M. Wimmer, F. X. Sillion, and W. Ribarsky, "Instant architecture," *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 669–677, 2003.

[10] M. Lipp, P. Wonka, and M. Wimmer, "Interactive visual editing of grammars for procedural architecture," *ACM Transactions on Graphics*, vol. 27, no. 3, pp. 102:1–102:10, 2008.

[11] M. Steinberger, M. Kenzel, B. Kainz, J. Müller, P. Wonka, and D. Schmalstieg, "Parallel generation of architecture on the GPU," *Computer Graphics Forum*, vol. 33, no. 2, pp. 73–82, 2014.

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

[13] ——, "Practical grammar-based procedural modeling of architecture," in *SIGGRAPH Asia 2015 Courses*, 2015.

[14] M. Bokeloh, M. Wand, H.-P. Seidel, and V. Koltun, "An algebraic model for parameterized shape editing," *ACM Transactions on Graphics*, vol. 31, no. 4, pp. 78:1–78:10, 2012.

[15] J. Lin, D. Cohen-Or, H. Zhang, C. Liang, A. Sharf, O. Deussen, and B. Chen, "Structure-preserving retargeting of irregular 3D architecture," *ACM Transactions on Graphics*, vol. 30, no. 6, pp. 183:1–183:10, 2011.

[16] F. Bao, M. Schwarz, and P. Wonka, "Procedural facade variations from a single layout," *ACM Transactions on Graphics*, vol. 32, no. 1, pp. 8:1–8:13, 2013.

[17] M. Ilčík, P. Musialski, T. Auzinger, and M. Wimmer, "Layer-based procedural design of façades," *Computer Graphics Forum*, vol. 34, no. 2, pp. 205–216, 2015.

[18] M. Dang, D. Ceylan, B. Neubert, and M. Pauly, "SAFE: Structure-aware facade editing," *Computer Graphics Forum*, vol. 33, no. 2, pp. 83–93, 2014.

[19] C. A. Vanegas, I. Garcia-Dorado, D. G. Aliaga, B. Benes, and P. Waddell, "Inverse design of urban procedural models," *ACM Transactions on Graphics*, vol. 31, no. 6, pp. 168:1–168:11, 2012.

[20] 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, pp. 11:1–11:14, 2011.

[21] Y.-T. Yeh, K. Breeden, L. Yang, M. Fisher, and P. Hanrahan, "Synthesis of tiled patterns using factor graphs," *ACM Transactions on Graphics*, vol. 32, no. 1, pp. 3:1–3:13, 2013.

[22] D. Ritchie, B. Mildenhall, N. D. Goodman, and P. Hanrahan, "Controlling procedural modeling programs with stochastically-ordered sequential Monte Carlo," *ACM Transactions on Graphics*, vol. 34, no. 4, pp. 105:1–105:11, 2015.

[23] A. Stolcke and S. Omohundro, "Inducing probabilistic grammars by Bayesian model merging," in *Proceedings of ICGI-94*, 1994, pp. 106–118.

[24] J. O. Talton, L. Yang, R. Kumar, M. Lim, N. D. Goodman, and R. Měch, "Learning design patterns with Bayesian grammar induction," in *Proceedings of UIST '12*, 2012, pp. 63–74.

[25] A. Martinović and L. Van Gool, "Bayesian grammar learning for inverse procedural modeling," in *Proceedings of CVPR 2013*, 2013, pp. 201–208.

[26] Esri, "Esri CityEngine 2015.2," 2015.

[27] Sceelix, "Sceelix," https://www.sceelix.com/, 2016.

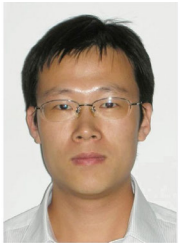
[28] Voxel Farm Inc., "Voxelfarm," http://www.voxelfarm.com/, 2016.

[29] D. Janzen and K. De Volder, "Navigating and querying code without getting lost," in *Proceedings of the 2nd international conference on Aspect-oriented software development*. ACM, 2003, pp. 178–187.

[30] J. Clark, S. DeRose *et al.*, "Xml path language (xpath) version 1.0," 1999.

[31] I. Gurobi Optimization, "Gurobi optimizer reference manual," 2016. [Online]. Available: http://www.gurobi.com
[32] F. Wu, D.-M. Yan, W. Dong, X. Zhang, and P. Wonka, "Inverse procedural modeling of facade layouts," *ACM Transactions on Graphics*, vol. 33, no. 4, pp. 121:1–121:10, Jul. 2014.

Haiyong Jiang received his Ph.D. degree from the National Laboratory of Pattern Recognition of the Institute of Automation, Chinese Academy of Sciences (CAS) in 2017 and his Bachelor's degrees from University of Science and Technology Beijing in 2012. His research interests lie in computer graphics and computer vision.



Dong-Ming Yan is an associate professor at the National Laboratory of Pattern Recognition of the Institute of Automation, Chinese Academy of Sciences (CAS). He received his Ph.D. from Hong Kong University in 2010 and his Master's and Bachelor's degrees from Tsinghua University in 2005 and 2002, respectively. His research interests include computer graphics, geometric processing and visualization.

Xiaopeng Zhang is a professor at the National Laboratory of Pattern Recognition of the Institute of Automation, Chinese Academy of Sciences (CAS). He received his Ph.D. degree in Computer Science from the Institute of Software, CAS in 1999. He received the National Scientific and Technological Progress Prize (second class) in 2004. His main research interests include computer graphics and image processing.

Peter Wonka is Full Professor in Computer Science at King Abdullah University of Science and Technology (KAUST) and Associate Director of the Visual Computing Center (VCC). Peter Wonka received his doctorate from the Technical University of Vienna in computer science. Additionally, he received a Masters of Science in Urban Planning from the same institution. After his PhD, Dr. Wonka worked as postdoctoral researcher at the Georgia Institute of Technology and as faculty at Arizona State University. His research interests include various topics in computer graphics, computer vision, remote sensing, image processing, visualization, machine learning, and data mining. He currently serves as Associate Editor for ACM Transactions on Graphics, IEEE Computer Graphics and Applications, and IEEE Transactions on Graphics and Visualization.