Procedural Facade Variations from a Single Layout

FAN BAO
Arizona State University
MICHAEL SCHWARZ
Arizona State University and Cornell University
and
PETER WONKA
Arizona State University and KAUST

We introduce a framework to generate many variations of a facade design that look similar to a given facade layout. Starting from an input image, the facade is hierarchically segmented and labeled with a collection of manual and automatic tools. The user can then model constraints that should be maintained in any variation of the input facade design. Subsequently, facade variations are generated for different facade sizes, where multiple variations can be produced for a certain size. Computing such new facade variations has many unique challenges, and we propose a new algorithm based on interleaving heuristic search and quadratic programming. In contrast to most previous work, we focus on the generation of new design variations and not on the automatic analysis of the input's structure. Adding a modeling step with the user in the loop ensures that our results routinely are of high quality.

Categories and Subject Descriptors: I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling

General Terms: Algorithms

Additional Key Words and Phrases: Procedural modeling, facade modeling, design variations

ACM Reference Format:

Bao, F., Schwarz, M., and Wonka, P. 2013. Procedural facade variations from a single layout. ACM Trans. Graph. 32, 1, Article 8 (January 2013), 13 pages.
DOI = 10.1145/2421636.2421644
http://doi.acm.org/10.1145/2421636.2421644

This research was partially funded by the National Science Foundation. M. Schwarz was supported in part by a DAAD postdoctoral fellowship.
Authors' addresses: F. Bao, Arizona State University; M. Schwarz, Arizona State University and Cornell University; P. Wonka (corresponding author), Arizona State University and KAUST; email: pwonka@gmail.com.

DOI 10.1145/2421636.2421644
http://doi.acm.org/10.1145/2421636.2421644

1. INTRODUCTION

Procedural modeling is a useful tool to create large amounts of detailed content. The design process often starts with an image or a drawing of one (or multiple) example(s), and then a grammar is written in textual form to reconstruct one specific input. Afterwards, the grammar is generalized by adding random variations [Watson et al. 2008]. This is one of the successful strategies to model plants using L-systems [Prusinkiewicz and Lindenmayer 1990] or buildings using shape grammars [Müller et al. 2006]. The visual quality of the output and the flexibility of this approach are definite advantages, but the modeling time is often high. This is especially true if different parts of an output model need to communicate and coordinate design choices, as seemingly tiny additions to a design or the specification of constraints may necessitate a significant modification of existing rules and writing of new ones.

An alternative strategy is to infer a grammar directly from a single input object, given in the form of images [Aliaga et al. 2007; Müller et al. 2007] or geometry [Šťava et al. 2010; Bokeloh et al. 2010]. All these previous approaches have in common that they spend most of their effort on image and geometry analysis to understand the structure of the input, with symmetry detection being a major technical ingredient. The main obstacle that we observed is that the structure present in facade layouts is surprisingly complex, and judging what aspects of a layout are important and should be preserved is often subjective. Therefore, it is not surprising that central aspects like alignment (refer to Figure 1) are not captured by grammars like the ones produced in recent work [Šťava et al. 2010; Bokeloh et al. 2010]. While these papers showed promising results, we pursue a different approach to be able to handle challenging layouts.

Our strategy is to combine a semi-automatic solution for structure analysis and an automatic solution for the computation of design variations. First, a user can generate a facade layout from a single image, using semi-automatic tools for hierarchical segmentation and labeling, and assign further attributes like depth to facade elements. The second step consists of the user specifying important relationships in the input layout that should be preserved. Third, design variations are automatically computed by a combination of heuristic search and quadratic programming.

The main contributions of this modeling approach are the following.

—Complementing existing automatic algorithms, we propose a framework that comprises layout modeling from a single input and constrained optimization to compute new design variations. These outputs are of high quality, and while this comes at the cost of additional modeling time, it is essential for most applications in industry.

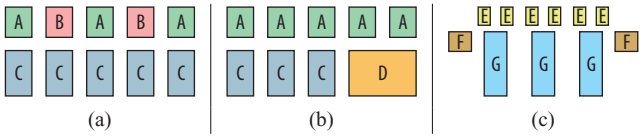


Fig. 1. Examples of interesting alignments present in facade layouts. (a) An alternating sequence of elements ABABA is aligned with a sequence of a single element (C) below. (b) Single elements can be aligned with other single elements (A and C), and multiple smaller elements (A) can be aligned with one larger element (D). (c) Alignments exist between elements of different sizes, and different types of alignment occur; for example, the centers of the elements E are alternatingly aligned to the left or right of elements G.

—Compared to grammar-based modeling, we simplify the modeling process, and we can specify facade layout variations that cannot easily be encoded with existing shape grammars.

The proposed design philosophy of semi-automatic structure analysis and automatic design computation may also lead to interesting work in other areas.

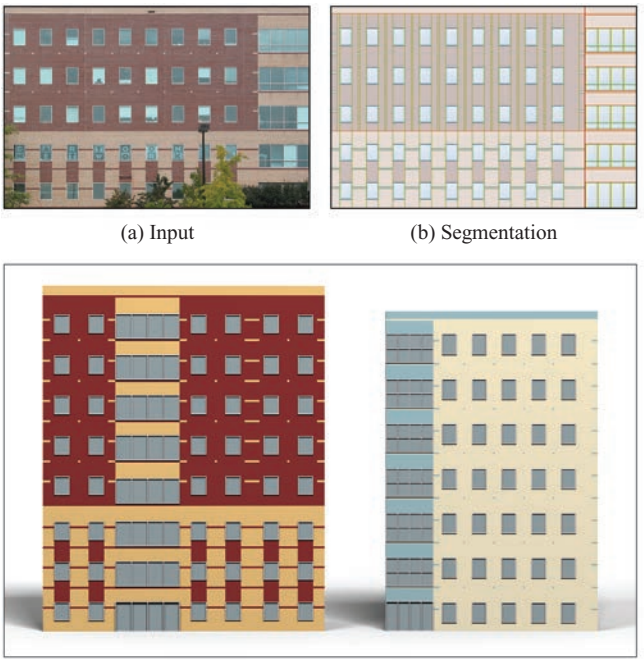
2. RELATED WORK

One popular approach for procedural modeling is to model objects using grammars, such as L-systems [Prusinkiewicz and Lindenmayer 1990] or shape grammars [Müller et al. 2006]. There are various extensions to add additional control to a grammar, such as the ability of a grammar to interact with user-defined shapes [Prusinkiewicz et al. 1994, 2001; Beneš et al. 2011; Talton et al. 2011]. While grammars typically have to be written in a text editor, Lipp et al. [2008] provide ideas how to specify grammars and modify designs with a graphical user interface. In all these approaches, the initial grammar still has to be designed by the user.

A natural question is how to automate this design process. Given a segmented input design as vector graphics, symmetry detection can be used to identify a hierarchical structure in the input and to establish rules that can replicate the input [Šťava et al. 2010]. If the input is an image, symmetry detection and segmentation are significantly more challenging. Therefore, the existing solutions to extract grammars from facade images [Aliaga et al. 2007; Müller et al. 2007] spend most effort on image analysis but not as much on structure analysis. As a result, current approaches only work well for selected facades. Another line of recent work deals with general meshes and point clouds as input [Bokeloh et al. 2010].

In general, grammars have known advantages and disadvantages. The visual quality of the output is typically very high and the examples can be complex. The disadvantages are the high modeling times and the required training in programming or scripting. A major challenge is the coordination among different parts of a design (as shown in the Introduction). We do, however, build on the idea of split operations [Wonka et al. 2003; Müller et al. 2006] used in grammar-based facade modeling, because most facades can be subdivided by splitting rules.

Synthesizing a larger region from a smaller input region has been heavily investigated for textures [Wei et al. 2009], and these methods can also be nicely adapted to architectural geometry [Merrell 2007; Merrell and Manocha 2008]. The most closely related texture synthesis algorithm specializes in facade textures [Lefebvre et al. 2010], and we will compare our results to this work. The regeneration of facade textures can be controlled by a resizing operation on architectural meshes [Cabral et al. 2009].



(a) Input (b) Segmentation

(c) Example variations

Fig. 2. Our framework can generate many variations of a facade design that look similar to a given input facade layout. Starting from an input facade image (a), we semi-automatically create a hierarchical segmentation (b) and model the essential aspects of the layout by specifying important constraints in a user interface. Our relayouting algorithm can then automatically generate many layout variations (c). Due to the semi-automatic modeling step, our procedural variations are all of high visual and structural quality.

Several recent methods in architectural modeling consider annotated examples as input. Impressive results have been recently demonstrated for the problem of floorplan generation [Merrell et al. 2010] and for furniture layout [Merrell et al. 2011; Yu et al. 2011].

Independently of our work, Lin et al. [2011] recently proposed a solution for retargeting a given 3D architectural model to new sizes; we provide a comparison with our approach in Section 9.2.

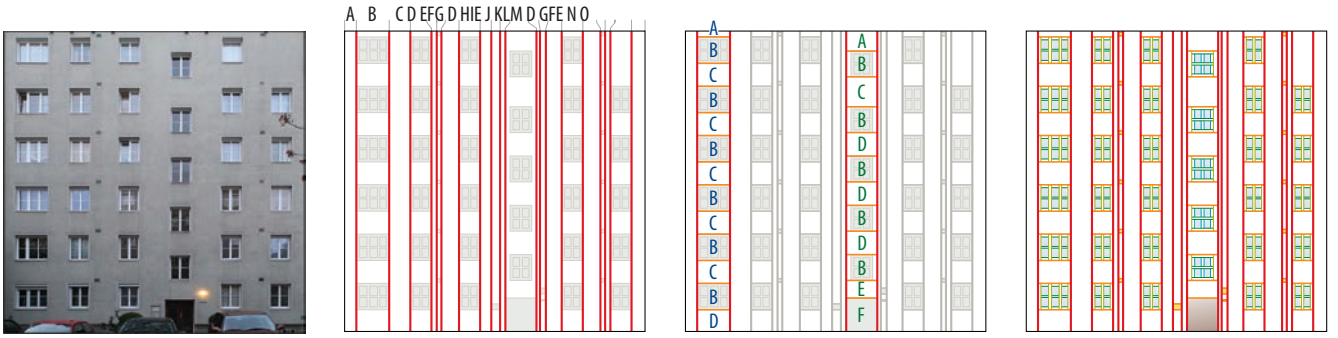
3. OVERVIEW

Our framework has three major components (refer to Figure 2).

Hierarchical segmentation. We take an approximately orthorectified facade image as input and adopt a semi-automatic approach [Musialski et al. 2012] to segment it into a hierarchy of rectangular regions. At this stage, we also provide region labels and approximate depth (Section 4).

Layout modeling. The hierarchical segmentation is further processed by the user to define important aspects of the layout. This step requires higher-level semantic knowledge of the input and, therefore, is also done in a semi-automatic fashion. The user has the ability to specify multiple types of hard and soft constraints that are considered essential to the layout: region-size, frequency, sequence, instance, alignment, same-size, and symmetry constraints (Section 5).

Relayouting. The relayouting algorithm can generate a variation of the input layout for a given target facade size. Layout



(a) Input (b) First level (c) Second level (d) Full hierarchy

Fig. 3. Beginning with a facade orthoimage as input, our semi-automatic approach performs a hierarchical segmentation. Each resulting subregion is assigned a symbol. (For visual clarity, (c) only shows the subregions of first-level regions B and J.)

modeling typically results in a larger number of constraints, and even finding a single layout that satisfies all hard constraints is difficult. Hence, we propose an optimization algorithm that uses a combination of heuristic search and quadratic programming (Section 7).

To evaluate the framework, we show selected facade variations in Section 8. Moreover, we discuss extensions and provide comparisons to existing solutions (Section 9).

4. HIERARCHICAL SEGMENTATION

To obtain a 2.5D geometric representation of the rectified input facade image that makes it amenable for relayouting, we first perform a hierarchical segmentation, adopting the recent, semi-automatic approach of Musialski et al. [2012]. Starting with an initial *region* (a rectangular, axis-aligned area on the facade) that covers the whole input facade, one or more splitting lines of identical direction (horizontal or vertical) that partition the region into self-contained subregions (like floors) are determined. These resulting regions are then split recursively, typically alternating the direction of the splitting lines, until no further splitting lines can be found, ultimately yielding a tree-based hierarchy of regions that segments the input facade (see Figure 3 for an example). A region at the finest level is called *terminal* and corresponds to a leaf node, whereas a *composite* region consists of and is completely covered by nonoverlapping subregions that are either vertically or horizontally arranged.

Generally, splitting lines are chosen according to automatically detected dominant edge features, but as these are not always yielding the decomposition desirable for layout modeling, the user can interactively edit the segmentation. To this end, operations like overriding the automatically inferred splitting direction, adding and removing splitting lines, freely moving a splitting line, or snapping it to an existing splitting line or a detected edge are offered.

For effective relayouting, it is necessary that regions that are supposed to be identically sized in the input actually have the same size in the segmentation; the same holds true for mutual alignments. As factors like noise and imperfect rectification may easily preclude this objective, the user can specify which regions must have the same size and which regions should be aligned; our system then adapts the splitting lines appropriately such that these constraints are met. The user may also indicate that two regions are identical, thus enforcing not only a consistent size but also an identical decomposition.

To enable referencing individual regions during the layout modeling, the subregions within a composite region are sequentially assigned a *symbol* (which we denote by a letter), where two subregions that have been marked identical share the same symbol.

Furthermore, each region can be assigned one or more semantic *labels* by the user, like Window or Door. During modeling, this allows to refer to sets of regions with identical function by using the according label. In our interface, the user first selects one or more regions, where advanced operations, like expanding the selection to include all identical regions or all regions of similar color, are offered. He then can choose from a collection of predefined labels or define a new one and assign it to the selected regions. A region's material and depth constitute two further attributes that can be assigned and modified analogously to labels.

5. LAYOUT MODELING

After the segmentation and labeling, the layout is interactively modeled by providing constraints that have to be respected during automatic relayouting. With them, the user can specify permissible arrangements of adjacent subregions and enforce that aspects of the input facade that he considers to be elementary, like certain alignments, are preserved in a relayout. Equally important is that aspects deemed nonessential are not specified during modeling, as the relayouting engine is then free to ignore them, thus allowing for variations.

5.1 Constraints

Region size. For the size of each region (width or height, depending on the arrangement direction in the encompassing region), an allowed minimum and maximum are maintained, as well as a probability distribution that is sampled to determine the initial size of a region when it is inserted into a new layout. By default, we select a truncated Gaussian distribution centered at the region's input size and choose the bounds at a fixed fraction from the center. The user may change the initial-size distribution, for example, by biasing it towards one of the bounds, and override the minimum and maximum size, entering them either as an absolute value or as a percentage of the input size. Note that by definition, all subregions of a region that share the same symbol also share their size constraints.

Frequency. Moreover, the number of times regions from a given set of regions $\mathcal{R}$ will be inserted within an encompassing region can be constrained, either by giving an absolute range or by specifying a frequency range relative to the encompassing region's input size. The set $\mathcal{R}$ can be specified by a list of symbols (for subregions of the

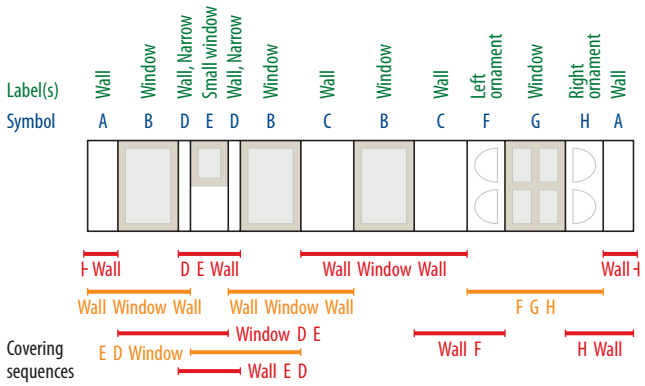
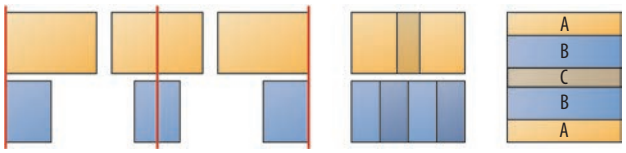


Fig. 4. A relayout of a composite region is described by an arrangement of subregions, each denoted by a symbol and associated with a set of labels; it is only valid if it can be composed by sequences defined during modeling. Example instantiations of such sequences are shown on the bottom, together with the concrete range of subregions covered by each of them.



(a) Alignment (b) Same size (c) Symmetry

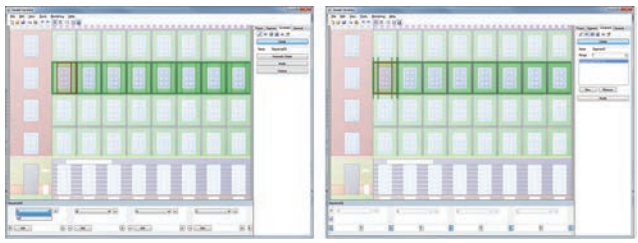
Fig. 5. Supported constraints include (a) inter-region alignment (at minimum, center, or maximum), (b) enforcing the same size for two sequences, and (c) reflective symmetry.

encompassing region) and semantic labels. Set-theoretic operations are also supported, allowing for selections like all regions labeled as *Window* except those additionally labeled as Door-sized.

Sequences. Within a composite region (refer to Section 4), subregions may be arranged in an arbitrary order, but not all arrangements yield a reasonable layout. To determine which arrangements are deemed permissible, the user specifies a set of sequences, where a *sequence* identifies a list of regions that can appear together in the given order as a group in the final arrangement. A sequence is specified by a list of sets of regions and may also contain a leading or trailing special token corresponding to the beginning ($\vdash$) or the end ($\dashv$) of the region, respectively. By replacing each set with one of its members, a concrete instantiation of a sequence is obtained.

During relayouting, the arrangement of subregions in a composite region is only considered valid if it can be completely covered by the given sequences, with adjacent sequence instantiations overlapping in at least one subregion (see Figure 4 for an example). The specified sequence constraints basically define a string grammar, where we found the sequence notation to be most intuitive for modeling and implementation.

Instances. A certain region R, identified by a symbol, may appear multiple times within the relayout of its parent region. By default, all of these instances of R are relayouted identically, that is, they are exact replica. However, the user can define that each instance of R is relayouted independently; he may also provide a range, either absolutely or relative to the parent region's input size, of how many different instance relayouts are desired within the parent region's relayout.



(a) Sequence constraint (b) Alignment constraint

Fig. 6. Example screenshots from our user interface for modeling layout constraints.

Alignment. To capture mutual alignment of facade elements, the user can specify that two regions R_1 and R_2 should be aligned at their minimum, center, or maximum coordinates (refer to Figure 5(a)) if they spatially overlap along the alignment direction (horizontal or vertical). This is determined by the arrangement directions of the respective parent regions, which have to be consistent. By providing a preceding and/or a succeeding sequence for each R_i, the alignment constraint can be restricted to apply only if R_i appears in this context. It is also possible to limit the constraint to a certain encompassing region $\tilde{R}$, making it only apply if both R_i are contained within $\tilde{R}$.

Same size. The user can further require that two sequences have the same size (refer to Figure 5(b)). Again, the arrangement directions of the respective parent regions have to be identical, and the application of the constraint may be restricted by defining a context for each sequence or specifying a required encompassing region. The same-size constraint may also be used to enforce that different instances of a region have the same extent irrespective of their potentially varying relayout.

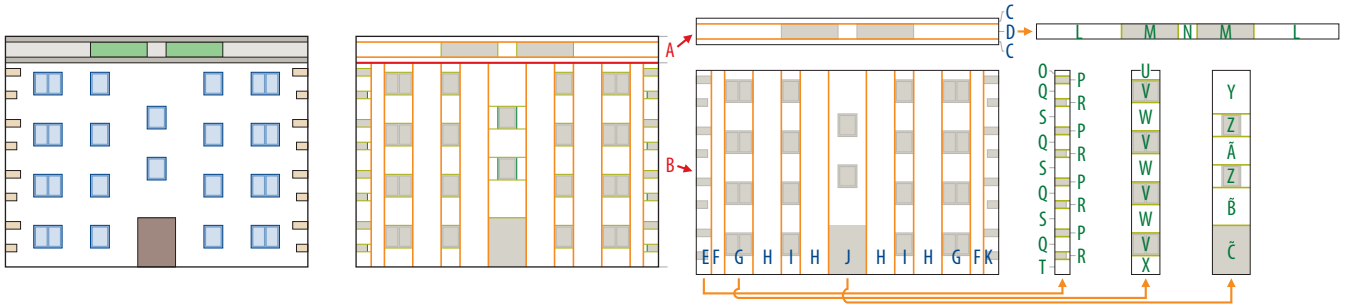
Symmetry. Finally, it can be specified that a composite region is reflectively symmetric (refer to Figure 5(c)), which restricts its subregions such that their symbols form a palindrome. In addition, the user can mark two symbols as mutually symmetric (e.g., E and K in Figure 7) to support more complex symmetries like ABCDCEA, where B and E form a symmetry pair.

5.2 User Interface

We provide a user interface for layout modeling; it shows the input layout in the main window, a sequence editor at the bottom, and various controls in the menu and toolbar on top and in panels on the side (refer to Figure 6). Three important components of the user interface are different ways to make selections (to build sets of regions), to model sequences by example, and to specify constraints using shortcuts based on simple automatic layout analysis; these are described next.

Selecting regions. Initially, an individual terminal region (e.g., a window with symbol C) can be selected with the mouse. Using the mouse wheel or keyboard input, the selection may be navigated up (and back down) the hierarchy of encompassing regions. The selection can also be expanded or reduced by a certain region via modifier keys and the mouse. Additionally, a dialog window can be opened to perform more complex operations, like expanding the selection to other regions that share a label (e.g., choosing the label Window expands the selection to all windows) or employing set operations, such as intersection and union.

Modeling sequences. To build a sequence, the user can select individual elements in succession and generalize each of them using



(a) Input (b) Hierarchical segmentation (c) Second hierarchy level (d) Third hierarchy level

Fig. 7. An exemplary facade with its hierarchical segmentation. Symbols are only shown for the first two levels and some third-level regions; I is decomposed analogously to G, and K is essentially a mirrored version of E. For notational conciseness (avoiding scope expressions), we use a single symbol namespace for all subregions in this example (e.g., we use C instead of A.A and L instead of A.B.A).

selection operations. The current state of the sequence is represented in the sequence editor.

Modeling further constraints. Using these components, region-size, same-size, and frequency constraints are typically straightforward to specify using the controls and dialog boxes. Alignment constraints are slightly more involved because they require more complex selections.

Shortcuts. Several constraint specification tasks are only mechanical, so that we provide several shortcuts to constraint modeling. For instance, sequence patterns can be reused for multiple composite regions if they have identical or similar structure, sequence constraints can be reversed, and constraints can be selected from a list of predefined patterns. One important instance of this latter shortcut is sequence generators.

As most of the time is typically spent on modeling sequence constraints, we use these sequence generators to quickly model the most common types of variations. Some of the generators are more specialized, but the three most important ones are as follows.

(1) *Replication.* The input sequence or a selected subsequence of it is added as a valid sequence. This generator is automatically invoked in case no sequences have been specified for a composite region.

(2) *Repetition.* The user can select regions or subsequences of regions to denote them as either optional or repeating. The generated sequences are typical for retargeting where elements can be replicated or deleted, but the relative order of elements cannot change. If the order is allowed to change, a more general relayouting becomes possible, and we refer to it as shuffling.

(3) *Two-Shuffle.* A subsequence of regions is selected by the user, and then the shuffle generator enumerates all (unique) existing subsequences of length two and their reversed versions. This is particularly useful in case of a sequence of elements separated by identical spacing elements (either with the same symbol or the same label and using this label in the generator input).

The sequence generators just add valid sequences, so that their output can be further edited by the user or combined with the output of another sequence generator.

In our experience, the user interface is sufficient to quickly model common variations and general enough to model all possible constraints in our framework.

6. MODELING EXAMPLE

To illustrate the modeling functionality of the framework with a concrete facade, we consider the example in Figure 7. The hierarchical segmentation follows two design rules: First, we try to capture the natural decomposition of the layout. Second, we do not group facade elements such as windows and doors together with surrounding walls, but try to put splitting lines to separate these elements from wall regions as early as possible. We found it is easier to control the spacing between facade elements this way.

As a next step, the sequence constraints are specified, often with the help of sequence generators. The facade is first split into a top subregion A and a bottom subregion B. By not explicitly specifying any sequence constraints for this composite region, the replication generator is automatically invoked to generate $\vdash$AB$\dashv$ as the only permissible sequence.

The top region A itself consists of a pattern CDC, and by selecting C as optional, the repetition generator creates the sequences $\vdash$CD, $\vdash$D, DC$\dashv$, D$\dashv$. The middle part D has two ornaments M shown in green. In our example design, we want to allow for an arbitrary number of repetitions (including zero) of these ornaments, where the separating wall can either be a long (L) or a short segment (N). To this end, we first group L and N into a region set by assigning a common label Space to them. After that, we use the repetition generator to automatically produce the following sequences: $\vdash$Space, Space M, M Space, Space$\dashv$.

For modeling the bottom region B, many design choices are possible. To enable shuffling (and repetition) of the window and door columns (G, I, J), separated by a wall column (H), we first select the subsequence of nonboundary columns GH$\cdots$HG and invoke the two-shuffle generator, yielding the sequence constraints GH, HG, HI, IH, HJ, JH. Subsequently, we turn to the start and select $\vdash$EFG, generalize G to the set of all window and door columns {G, I, J} (alternatively, we could have assigned a label to them and select this), and generate the corresponding replicating sequence $\vdash$EF{G, I, J}. The end is treated analogously, resulting in the constraint {G, I, J}FK$\dashv$.

The ornaments on the side (in E and K) can be modeled by allowing one or more repetitions of the sequence PQR to occur, with two instances being separated by S, and embracing them with O on the top and T on the bottom. Using an advanced repetition generator, we directly obtain the according sequence constraints $\vdash$OP, PQRSP, PQRT$\dashv$. Similarly, a repetition generator can also be employed to model sequences for the window columns G and I,

allowing one or more repetitions of the windows, as well as for the door column J.

The third-level composite regions (R, Z) are not modeled explicitly, causing them to be processed automatically by the replication generator. Therefore, a wall is kept next to the shorter ornaments (R) and each window in the door column is padded on both sides with a wall region (Z).

We then attend to modeling same-size constraints, which in practice is often interleaved with modeling the sequence constraints. First, we enforce that the windows in column I and the wider windows in column G have the same height. Subsequently, we declare that a window in the door column J and a window in column I should have the same size. This is automatically translated to two same-size constraints, one for the height and one for the width. After that, we ensure that floors have the same height. This is more difficult to model because floors consist of multiple elements, forcing the constraints to fix the size of sequences rather than the size of single regions. Concretely, the sequence PQRS in the side ornaments and the sequences VW in the window columns are constrained to have the same height. Furthermore, we enforce that the ornaments on the left side and the right side of the building are symmetrical by applying same-size constraints both to columns E and K and to the respective smaller ornaments within them.

To complete the design, we add a frequency constraint to limit the number of door columns J to one and model two alignment constraints. The first one ensures that the vertical centers of the larger and the smaller ornaments on the side (P, R) are aligned with the top and the bottom, respectively of the windows in a window column (V). The second alignment constraint enforces that the ornaments on top of the facade (M) align with the left, right, or center of a window. This constraint was partially chosen to demonstrate the flexibility of the design system. To allow for more variations, we additionally specify that multiple instances of the ornament M and the spaces L and N around it can be relayouted independently. The alignment of the top ornaments is a great example where the generalization of a single layout requires an active design decision that cannot be done by an automated system, since many equally valid layout rules could be derived in this situation.

Designing this example requires a few dozen clicks and can be done in as few as two to three minutes if the user has a clear goal of what he wants to model. When starting from scratch, however, a more realistic time would be between ten and thirty minutes. This time is mainly spent on experimenting with design choices and thinking about design strategies rather than the actual user interface. This part of the design process should not be eliminated. Two different variations produced with the outlined constraints are shown in Figure 8.

Fig. 8. Two example variations for the facade from Fig. 7.

7. RELAYOUTING

Based on the layout model with its set of constraints, the relayouting algorithm can generate a variation of the input layout for a given target region (given by width and height). This layouting problem is challenging for several reasons. First, a valid layout is a partition of space, requiring the layout's terminal regions to cover the complete facade without overlap. Second, there are a larger number of hard constraints to observe. Third, relayouting necessitates considering continuous variables (region sizes) as well as discrete design choices (e.g., frequency constraints).

While there are several interesting stochastic algorithms that have been applied to architectural layout problems recently, such as simulated annealing [Yu et al. 2011], random jump MCMC [Talton et al. 2011], and Metropolis-Hastings [Merrell et al. 2010], there is no possible simple adaption to our solution. The constraints are simply too restrictive, and stochastically navigating the solution space will not result in a valid solution. For example, Michalek et al. [2002] report that even for moderate floorplan layouting problems comprising about ten independent rooms, simulated annealing may often find no solution, and we have significantly more regions and constraints.

Therefore, we propose a novel algorithm that combines several building blocks. With the constraints being so restrictive, the overall strategy is a heuristic search that draws from planning algorithms targeting constraint satisfaction problems [LaValle 2006]. The goal of this heuristic search is to suggest discrete design choices. These are then handed over to a quadratic programming algorithm that can determine the optimal layout—or fail, indicating that it is impossible to fulfill the constraints.

7.1 Overview

The proposed relayouting algorithm generates a new hierarchical subdivision. Starting with the target region as current region, the algorithm calls a relayouting function *layout*—the essential building block of the optimization—to obtain a partition of the current region into subregions. Each of these subregions is associated with a symbol, relating it to a region in the input segmentation, and an instance identifier. The symbol of the current region determines both the direction along which the subregions are arranged and the constraints for the *layout* function. The algorithm proceeds in a top-down, depth-first manner to recursively split the current region and then subsequently calls the *layout* function for all subregions that are not terminal in randomized order. This strategy effectively transforms a complex 2D layouting problem into a sequence of 1D layouting problems. In the following, we first describe the function *layout* to establish the basic algorithm. Subsequently, we address special cases and improvements to better steer the exploration of possible layouts.

7.2 Layouting a Composite Region

The function *layout* iteratively builds an arrangement of subregions, interleaving a discrete search step to add new subregions with a continuous optimization step to compute the optimal size of these regions. The subregions are added from left to right or top to bottom, respectively. The arrangement corresponds to a list $\mathcal{A}$ of symbols (each with an instance identifier) $A_i, 0 \leq i \leq n$, where A_0 is always the special token $\vdash$, denoting the beginning of the region. Only once list element $A_n = \dashv$, indicating the end of the region, has been inserted, the arrangement sequence is complete.

To append a new symbol to $\mathcal{A}$, we first generate a set $\mathcal{S}$ of valid successor symbols by identifying all sequences defined for the current region that feature a prefix overlapping a postfix of $\mathcal{A}$. Subsequently, several checks are performed to remove elements of

Table I. Statistics for the Example Facade Layouts

Facade	Fig. 2	Fig. 3	Fig. 9	Fig. 10	Fig. 11	Fig. 12	Fig. 13	Fig. 14	Fig. 16
Terminal regions (in input)	568	574	1323	93	12016	953	584	560	1071
Frequency constraints	1	5	5	4	0	0	0	2	0
Sequence constraints	56	70	141	102	275	40	33	297	133
Alignment constraints	64	12	87	19	70	29	23	100	40
Same-size constraints	8	3	2	2	11	0	5	0	7

The complexity of the input facade's hierarchical segmentation is quantified by the number of resulting terminal regions, that is, the number of leaf nodes in the hierarchy. For the number of layout constraints modeled, note that the number of user interactions is typically lower, as a single input may result in multiple constraints.

$\mathcal{S}$ that cannot result in valid layouts. Most notably, we eliminate symbols that violate frequency constraints or whose minimum size is too large to allow them to be placed in the current arrangement.

Assuming that the current incomplete list $\mathcal{A}$ consists of k elements $(A_0, \ldots, A_{k-1})$, we stochastically select an element A_k from $\mathcal{S}$ and sample its initial size $\tilde{x}_k$ according to the corresponding distribution. We then find all newly active constraints to build a quadratic programming problem, which optimizes the size x_i of all elements A_i to make them as close as possible to their desired size.

$$\arg\min_{x_i} \sum_{i=0}^{k} (x_i - \tilde{x}_i)^2$$

The constraints for this problem are set up as follows.

—Region-size constraints yield constraints of the form $\underline{x}_i \leq x_i \leq \overline{x}_i$, where $\underline{x}_i$ and $\overline{x}_i$ are the minimum and maximum of the allowed region size for symbol A_i. If $A_i \in \{\vdash, \dashv\}$, we have $\underline{x}_i = \overline{x}_i = \tilde{x}_i = 0$.

—Alignment constraints for element A_k are translated to $\sum_{i=0}^{k-1} x_i + \lambda x_k = y$, where λ is chosen according to the type of alignment (minimum: 0, center: 0.5, maximum: 1) and y corresponds to the position to align with.

—Same-size constraints where the two affected sequences are both within the current region result in constraints $\sum_{i \in I_1} x_i = \sum_{j \in I_2} x_j$, with the index sets I_1 and I_2 identifying the sequences. If the other sequence is not in the current region and hence has already been placed, the constraint simplifies to $\sum_{i \in I_1} x_i = c$, where c is a constant.

—An additional total-size constraint enforces that the elements in $\mathcal{A}$ do not exceed the current region's size x_{total}. It is of the form $\sum_{i=0}^{k} x_i \leq x_{total}$ if $A_k \neq \dashv$, and $\sum_{i=0}^{k} x_i = x_{total}$ in case $A_k = \dashv$.

This quadratic program can be solved using the Goldfarb-Idnani active set dual method [Goldfarb and Idnani 1983], for which a public implementation is available [Gaspero 2009].

7.3 Additional Considerations

While the described *layout* function is complete, it should be extended as follows. Two of the extensions are necessary to handle all user input, and three extensions aim at optimizing the computation speed.

One open problem is the handling of different instances of the same symbol. For example, consider a floor denoted by symbol A. As the layout function stacks different floors, the symbol A can be selected multiple times so that there is a design choice to either force all occurrences of A to be identical or to allow some of them to be different. We therefore require an additional pass over the set $\mathcal{S}$ of potential successor symbols to encode what instances of A are allowed. For instance, if the floor A can have multiple instances, we add a second possible instance A' to $\mathcal{S}$.

A second question is how to enforce reflective symmetry in the list $\mathcal{A}$. Our solution requires minor changes at multiple locations in the layout function. First, while building a list $\mathcal{A}$ from one side of the region, we also construct a reverse list $\overline{\mathcal{A}}$ from the other side. Additional checks are then performed on the set of potential successors $\mathcal{S}$ to make sure that sequence constraints imposed by $\overline{\mathcal{A}}$ are not violated and that frequency constraints are still satisfied when considering $\mathcal{A}$ and $\overline{\mathcal{A}}$ simultaneously. Further modifications are also required to finish a layout, as we are now confronted with two options. The last symbol A_n in $\mathcal{A}$ can either end at the midpoint $\frac{1}{2}x_{total}$ or the center of A_n can be aligned with the midpoint.

If no valid layout can be found for the current region, backtracking is performed. However, backtracking can get stuck figuring out different local configurations, while not being aware of a fundamental error made early on in the layout process, when relayouting an encompassing region. Therefore, we restart the complete layout process and start with a new root region if a maximal number of backtracking steps has been reached (1000 in all examples). Two other improvements are to generalize backtracking to backjumping and to randomly accept or reject new elements A_i based on the error of the quadratic program.

8. RESULTS

In this section, we provide a quantitative and visual evaluation of the framework. Our prototype was implemented in C# and C++, and we used nine facade images to evaluate various aspects of performance. All these examples were segmented interactively.

Layout statistics. A statistical overview of the example facades is given in Table I. For each facade, we list the complexity of the input segmentation and the number of constraints. Note that our layout examples use a significantly larger number of shapes and constraints than recent systems for floorplan [Merrell et al. 2010] and furniture layout [Merrell et al. 2011; Yu et al. 2011].

Variety. The facades were selected to show a variety of results. All outputs are three-dimensional, but we visualize several results in 2D to make the structure better visible. For the facade in Figure 9, the results feature several interacting structures in different form. The windows in the first floor and the upper floors have different glass panel layouts, while they themselves are within a structure of ornamental linear protrusions. The alignment and spacing between windows and doors is preserved, even if ornamental ledges stemming from the top of the facade are mixed into the layout. The result in Figure 10 shows different width and height variations for a small input facade. There are fewer elements in the input, but we can still generate many interesting variations. Our largest example is a skyscraper (see Figure 11), where our algorithm can modify three nested grid levels in different ways while preserving symmetry constraints. One 3D model of the skyscraper comprises over 100K triangles. We selected the example in Figure 12 for its subtle

Fig. 9. Example facade (taken from Müller et al. [2007]; top left) with its hierarchical segmentation (bottom left) and three layout variations (right). The modeled constraints enforce the alignment of windows and doors in the columns and floors of this design, even though the ornamental row can be optionally repeated between floors. The relayouting algorithm also finds creative new layouts for the panels of the door-sized windows in the first floor in all three examples. The shuffling of elements enables the door column to appear multiple times and to occur in different positions. The sign on the first floor and the other ornaments can break the translational symmetry of the columns.

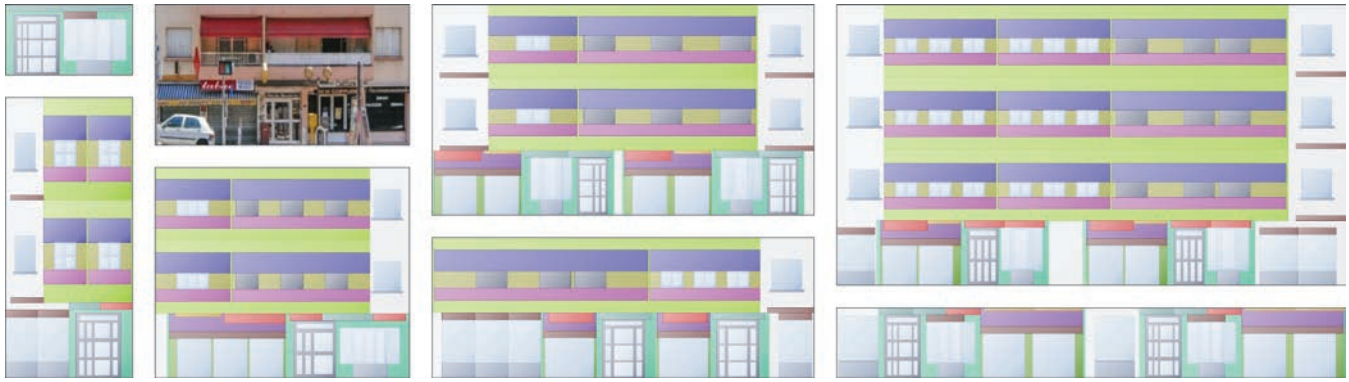


Fig. 10. Several facade variations are shown for the input image in the top row (taken from Lefebvre et al. [2010]), spanning a range of different target facade sizes. Our algorithm can generate variations that are smaller or larger than the input, as well as multiple different variations for a single target facade size.

variation of a seemingly regular layout. There are several variations of ornamental elements between windows that all need to be aligned correctly. The facade in Figure 2 consists of a door column on the right and a grid of windows on the left, which itself comprises a top and a bottom part. In the variations, we allow the door column to move to the interior of the window grid (Figure 2(c), left) and the bottom part of the window grid to be skipped (right), while maintaining the correct alignment of windows and ornaments. A simpler example is demonstrated in Figure 13.

Layout modeling. The user can select how many constraints to specify and how restrictively to model them. While there is no direct correlation between the number of constraints and the number of allowed variations, this is often the case. For example, more alignment constraints typically result in fewer variations and more regular layouts. More importantly, the allowed variations depend on how the constraints are modeled, for example, how long and general the sequence constraints are. In Figure 14, we illustrate the effects of three types of layout modeling. The first example (b: loose) with a random distribution of windows that are often unaligned is contrasted with the third example (d: strict) that only allows for fewer, controlled variations (larger areas are similar to the input and there are more translational symmetries). The second example (c: medium) is an intermediate form.

Modeling times. Most facade layouts in this article can be modeled in about 30 to 60 minutes, including segmentation. Modeling a facade layout with constraints is an iterative process and requires some trial and error. Overall, most of the time is spent analyzing the input layout and experimenting with different design ideas.

Relayouting performance. Our implementation is reasonably fast and can generate one layout in typically tens to hundreds of milliseconds. Average timings for all example facades are given in Table II, using an Intel Core i7 2.67 GHz.

9. DISCUSSION

While we focused exclusively on single facades so far, the applicability of the approach is not limited to this problem domain. In the following, we discuss extensions to mass models and according 3D

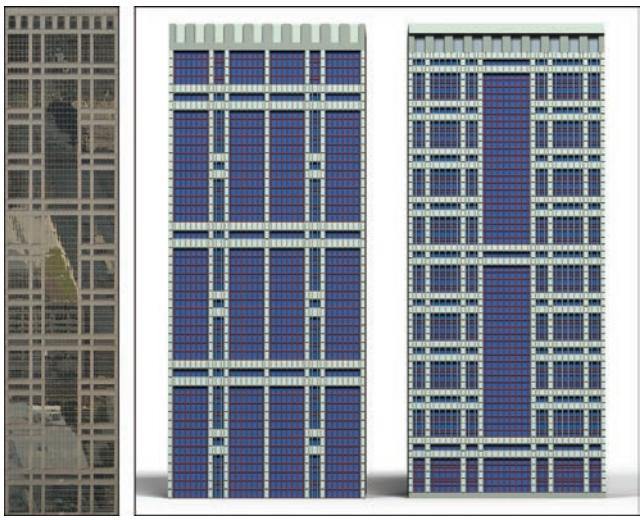


Fig. 11. For the input model on the left, we show two variations of smaller height and larger width. The symmetric design consists of three nested grids and an ornamental structure on top as well as on the thick gray beams.

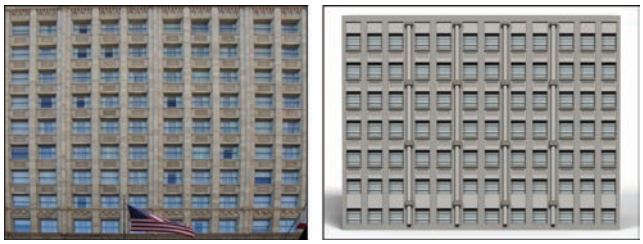


Fig. 12. A facade of a hotel (left) and one generated representative variation (right). Despite its regular appearance, there are several interacting structures that make this layout interesting.

input as well as layout problems beyond facades. Furthermore, we provide a comparison to related existing solutions.

9.1 Extensions

Facades on mass models. Our framework naturally extends to generating facade variations on mass models, like the procedurally generated ones depicted in Figure 15. During modeling, an attribute of the modeled constraints designates whether a constraint is only valid within one face (facade) of the building or if it applies across faces. That way, a single entrance door for the whole building can be enforced, and, as demonstrated by the shown results, floor heights and element sizes can be made coherent across all facades. Because faces are basically just composite regions, such constraints across different faces can be treated analogously to constraints between composite regions on the same facade during relayouting.

3D model input. It is also possible to take a 3D model as input for the facade generation. One option is to extend the layout domain from the 2D facade surface to the whole 3D building volume. This, however, can severely limit the modeling and variation capabilities. For instance, it would restrict us to footprints with two (typically orthogonal) facade orientations, precluding a relayout

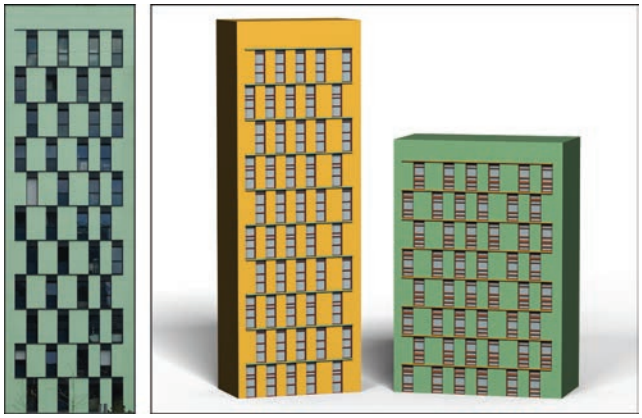


Fig. 13. Two variations (right) of the input facade on the left. The segmentation and layout modeling of such a simpler facade can be performed within a few minutes.

Table II. Average Time Required to Generate a New Variation

Facade in Fig.	2	3	9	10	11	12	13	14	16
Time [ms]	173	70	103	27	3715	16	154	105	262

(a) Input (b) Loose (c) Medium (d) Strict

Fig. 14. For the shown input facade (a), we modeled three different layouts; one selected relayouting example is displayed for each (b–d). The first layout (b) has only a few constraints and the alignment between windows is not enforced. For the second layout (c), a modest number of constraints were modeled so that some alignment and some randomness are present. The third layout (d) has more constraints and allows only for some larger-scale variations. As a result, there are multiple replicated floors, which are additionally similar to the input.

to more interesting shapes, such as a wedge (like the Flatiron Building in New York City). Therefore, we opted for the alternative of considering only the facade shells of a building, that is, the outer geometric layer that contains the facade. These shells are hierarchically decomposed, basically yielding a 2D segmentation, to which our layout modeling framework can be applied directly. After a relayout has been computed, the terminal regions are instantiated with the corresponding 3D content from the facade shell. In Figure 18, we show results obtained with this approach for two building models that we adopted from Google Warehouse.

Applications beyond facades. The current approach can be adapted to further layout problems with regular or semiregular structure, like tiling patterns, carpet patterns, Charbaghs, labyrinths, and furniture layouts. As an example, we applied it to the garden layout

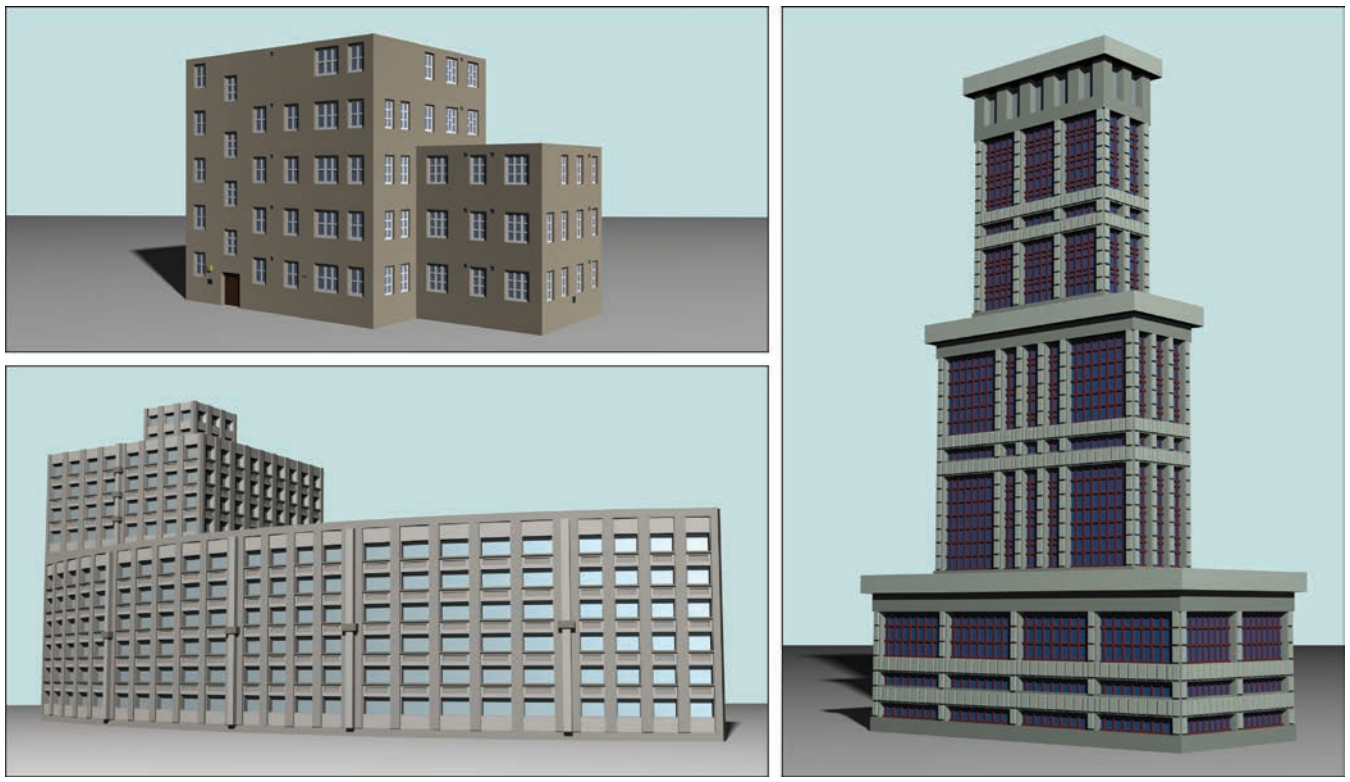


Fig. 15. Three examples illustrate how constraints can be extended to handle multiple faces of a mass model. In the top-left example, using the input facade from Figure 3, we show that all windows are aligned, even though several different window types can be generated throughout the building by relayouting the window frames. The floor heights are the same across the model to make this variation architecturally plausible. The design on the bottom left (input facade from Figure 12) demonstrates that our facade layouts can be mapped to curved footprints, as we do not rely on axis-aligned facades. The model on the right shows a skyscraper (input facade from Figure 11) consisting of three box-shaped masses. The size of the elements is coordinated on all facades of the same mass, and minor variations are allowed between the different masses.

in front of the Taj Mahal to produce some variations, as depicted in Figure 19.

9.2 Comparison to Related Solutions

CGA shape. A formal comparison to CGA shape [Müller et al. 2006] is difficult, because even within CGA shape many different modeling philosophies can be applied to encode facade layouts. We just illustrate two fundamental challenges in Figure 16 that are more difficult to cope with in CGA shape. First, it is easily possible to align facade elements in different floors if they appear in a predictable (e.g., fixed) order and all floors have the same pattern, like all floors creating a layout A{B}*A, where the number of B's can change according to the width of the facade. The symbols may also have a different geometric interpretation in different floors. However, it is not easy to randomly select elements from a set if alignment constraints are in play and to place them in a random order. One strategy would be to compute the start and end positions of the randomly selected elements and pass them to all child shapes in the form of parameters. For complex alignments this leads to a large number of parameters and if statements in the grammar. Second, the random selection creates problems with terminating at a region boundary. Once all elements have been selected, their sizes would have to be consistently adjusted to fill the whole region. Otherwise, there is usually some leftover space that cannot be used well and that hence will be filled with a wall.

Texture synthesis. Lefebvre et al. [2010] proposed an automatic texture synthesis algorithm that is well suited for facade images. We obtained about 100 facade synthesis results for our facade dataset from the authors. The advantage of their approach is that it is automatic and can be applied to other types of architectural textures. However, the careful modeling in our approach generally leads to better layouts. Similar to CGA shape, it is difficult for the texture synthesis algorithm to change the order of elements in a sequence and many facade variations can never be generated. For example, the column with the entrance in Figure 9 that contains the door never changes its place. Also, while many generated layouts are reasonable, the algorithm is prone to generating artifacts (see Figure 17 for some examples), such as misalignment, the generation of elements that are too small (e.g., windows with all glass panels eliminated), an unnatural spacing of windows, broken symmetry within elements, and repetition of elements that should not be repeated.

3D architecture retargeting. Recently, Lin et al. [2011] independently proposed a solution for retargeting architectural models, which is a simpler, restrictive form of relayouting. They focus on complete 3D building models and apply replication and scaling to elements of it to adapt the model to a new extent. Similar to our approach, they rely on a manual hierarchical segmentation

(a) Input (b) CGA shape [Müller et al. 2006] (c) Our approach

Fig. 16. For the input facade on the left (a), we compare a selected CGA-shape variation (b) to one of our variations (c). In the used CGA-shape grammar, each floor is built with a recursive split rule that randomly adds one new element (window, double window, or wall). However, these random decisions cannot be easily coordinated across floors, and hence, the inter-floor alignment of elements is missing in the CGA-shape result. Additionally, when greedily populating each floor with randomly selected elements of varying size, problems arise in terminating cleanly, often necessitating a final, squeezed wall element.

Fig. 17. Several layout constraints are not considered in a texture synthesis approach [Lefebvre et al. 2010]. Red: Windows of the same original size can have different sizes and may be no longer aligned. Purple: The symmetry within architectural elements, such as windows and doors, can be broken. Blue: Too many repetitions of the same element, a ventilation vent, can occur in sequence. Additionally, the vents are no longer aligned. Green: Large gaps can occur between floors.

of the input model into boxes. While we operate in 2D, their decomposition is in 3D, and this has several consequences. On the one hand, this enables Lin et al. to better capture interactions between building facades, as for instance an L-shaped terrace spanning two facades that is cut out of the main mass of the building or a volumetric structure on the corners. On the other hand, this limits them to segment a building into axis-aligned boxes, favoring buildings that can be rotated such that all facades are aligned to an axis. In the presence of curved footprints (refer to Figure 15, bottom left) or facades with arbitrary orientation angles (refer to Figure 18, bottom), complete facades thus have to be enclosed by a single box and can then only be scaled, but not retargeted.

For retargeting, the modeling effort in both systems is comparable. However, the examples shown by Lin et al. [2011] segment buildings into larger boxes that include a whole architectural element like a window or door and its surrounding ornaments, whereas we typically subdivide a layout further into smaller elements roughly the size of window frames or window sills.

Apart from specifying scaling and repetition of elements, we further allow the user to declare undesirable variations by means of constraints (e.g., the alignment of ornaments and windows or windows and windows) and to generate variations via shuffling of elements rather than solely by changing the number of repetitions, thus enabling layouts beyond simple retargeting. For instance, from an input sequence ABB, this makes us easily generate BA, BBA, or BABBAA. If a user of our system chooses to make use of these extended capabilities, the modeling times will of course increase.

In some sense, the philosophy between the two approaches is very different. Lin et al. [2011] aim at a simple user interface for casual users, where only a limited degree of specification is both needed and possible and where most design choices are left to the automatic system. As a consequence, only simple retargeting is supported but not complex relayouting, which requires additional specification. By contrast, we also target applications in industry where professionals typically want to exercise more control over the output, and we thus support relayouting options beyond retargeting and offer additional and more detailed specification possibilities.

9.3 Limitations

There are several limitations in our system. First, we did not model detailed facade elements themselves, but only their placement (for the 3D models in Figure 18, we directly take the input geometry). For example, it would be nice to have a model of the sign (letters) in Figure 9. This is only a limitation in our implementation and not a limitation of the framework. Second, we cannot model layouts on freeform architecture, as we are limited to a rectangular domain. Third, we only support layouts for which a hierarchical, rectangular decomposition exists. One consequence of this is that nonrectangular elements (e.g., circular ornaments) must be approximated by their enclosing rectangles. Furthermore, elements can only be arranged vertically or horizontally but not along arbitrary curves. Forth, the rule modeling process requires some care. It is possible to model constraints that do not have a single feasible layout, not even the original facade. Due to the complexity of the optimization problem, it is in general not possible to determine if a solution exists. While the heuristic search step would ultimately explore the whole solution space if the algorithm were run long enough and thus would find a valid solution if it exists, we abort the process after no solution has been found for some time, as in a typical application of the system, variations should exist and be found quickly. After all, our system aims at enabling a user to generate multiple variations of a layout, and we thus assume that the user will not be interested in generating excessively constrained solutions.

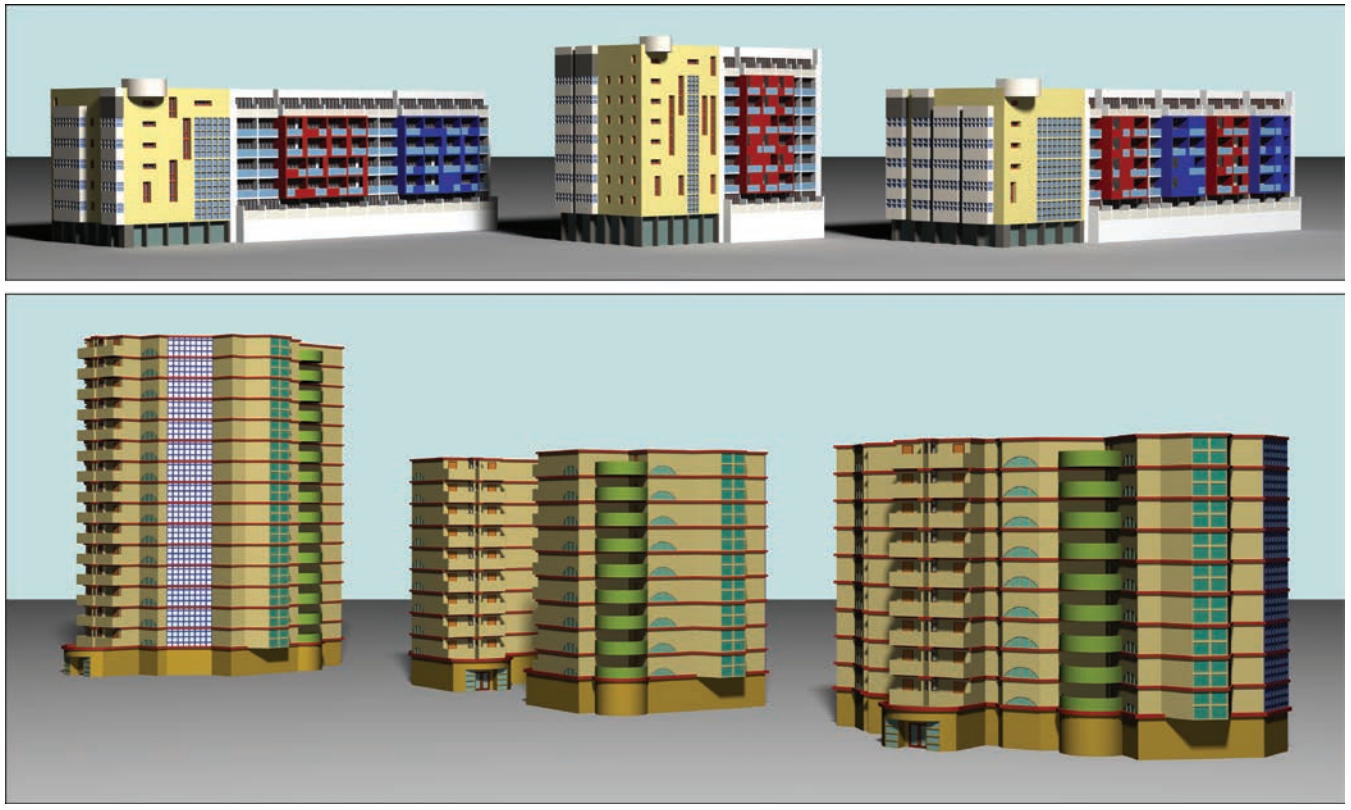
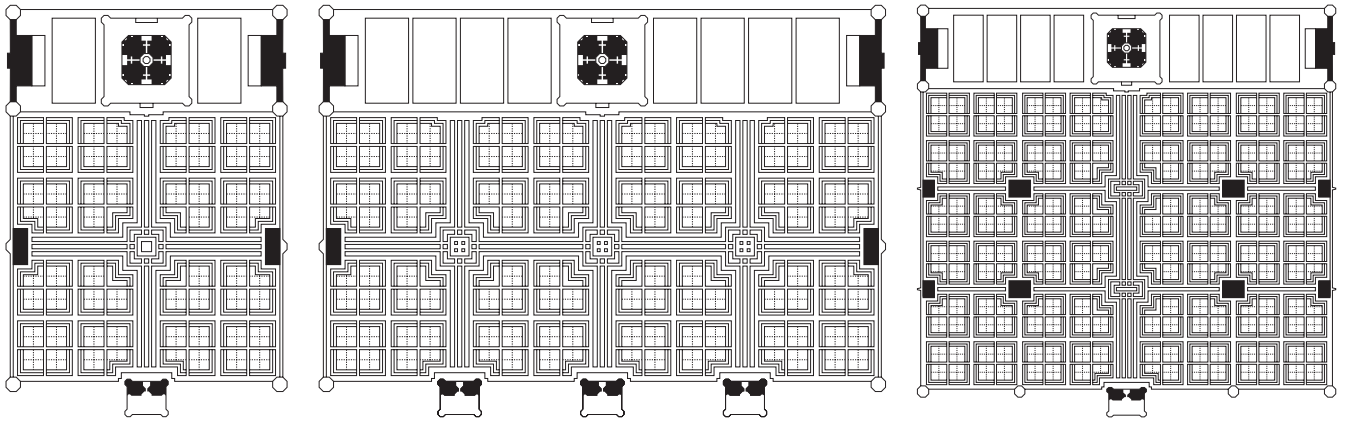


Fig. 18. Our framework can also take 3D models as input (shown on the left). In both examples, we demonstrate the possibility of shuffling columns in the design. In the top example, the input design has two regions with protruding balconies (that are red and blue). The first variation only uses red balconies, and the second variation uses four balcony regions alternating in color. The thin orange windows on the yellow facade can be relayouted to create new interesting design variations. The bottom example shows results for different complex footprints.



(a) Input (b) Variation 1 (c) Variation 2

Fig. 19. The Taj Mahal Mughal garden layout (a) is used to generate two variations (b, c). Our design forbids the replication of the Taj Mahal on the top as it is a unique design, but the number and style of garden elements, entries (bottom), water canals, and the separating elements between the gardens can be varied. We used this example to create a link to the pioneers of shape grammars [Stiny and Mitchell 1980], who encoded this garden design using thirty-nine rules.

10. CONCLUSION AND FUTURE WORK

We have presented a framework that given an input facade can produce many different facade layouts. Each offers a variation of the input that captures the essence of the input's design, like certain alignments, as specified by the user in a modeling step.

Based on the experience gained from this project, we believe that the following problems are especially interesting for future work.

First, we conjecture that the regularization of a noisy input layout during the initial hierarchical segmentation can be automated. That is, regions that are only almost aligned or almost identically sized due to noise and imprecise rectification can be detected and adjusted to yield exact alignments and consistent sizes. Second, we would like to combine our approach with an architectural reshaping framework, like the one proposed by Cabral et al. [2009]. Third, mixing multiple input facade designs to create new, composite designs is an interesting idea that could significantly increase the number of attainable variations. Fourth, it would be exciting to investigate adapting and evolving our approach to work with more general 3D shapes, possibly by accordingly enriching and modifying existing shape editing systems, like the one by Bokeloh et al. [2012], which are currently restricted to basic retargeting operations. Finally, we believe that the relayouting of plant models will pose additional challenges that are worth pursuing.

ACKNOWLEDGMENTS

We would like to thank Przem Musialski for his help with the segmentation stage and Sylvain Lefebvre for promptly generating and providing a large number of results with his texture synthesis method that helped us with the comparison.

REFERENCES

Aliaga, D. G., Rosen, P. A., and Bekins, D. R. 2007. Style grammars for interactive visualization of architecture. *IEEE Trans. Vis. Comput. Graph. 13,* 4, 786–797.

Beneš, B., Šťava, O., Měch, R., and Miller, G. 2011. Guided procedural modeling. *Comput. Graph. Forum 30,* 2, 325–334.

Bokeloh, M., Wand, M., and Seidel, H.-P. 2010. A connection between partial symmetry and inverse procedural modeling. *ACM Trans. Graph. 29,* 4, 104:1–104:10.

Bokeloh, M., Wand, M., Seidel, H.-P., and Koltun, V. 2012. An algebraic model for parameterized shape editing. *ACM Trans. Graph. 31,* 4, 78:1–78:10.

Cabral, M., Lefebvre, S., Dachsbacher, C., and Drettakis, G. 2009. Structure-preserving reshape for textured architectural scenes. *Comput. Graph. Forum 28,* 2, 469–480.

Gaspero, L. D. 2009. QuadProg++ 1.2. http://sourceforge.net/projects/quadprog/.

Goldfarb, D. and Idnani, A. 1983. A numerically stable dual method for solving strictly convex quadratic programs. *Math. Program. 27*, 1–33.

LaValle, S. M. 2006. *Planning Algorithms*. Cambridge University Press, New York.

Lefebvre, S., Hornus, S., and Lasram, A. 2010. By-example synthesis of architectural textures. *ACM Trans. Graph. 29,* 4, 84:1–84:8.

Lin, J., Cohen-Or, D., Zhang, H., Liang, C., Sharf, A., Deussen, O., and Chen, B. 2011. Structure-preserving retargeting of irregular 3D architecture. *ACM Trans. Graph. 30,* 6, 183:1–183:10.

Lipp, M., Wonka, P., and Wimmer, M. 2008. Interactive visual editing of grammars for procedural architecture. *ACM Trans. Graph. 27,* 3, 102:1–102:10.

Merrell, P. 2007. Example-based model synthesis. In *Proceedings of the Symposium on Interactive 3D Graphics and Games.* 105–112.

Merrell, P. and Manocha, D. 2008. Continuous model synthesis. *ACM Trans. Graph. 27,* 5, 158:1–158:7.

Merrell, P., Schkufza, E., and Koltun, V. 2010. Computer-generated residential building layouts. *ACM Trans. Graph. 29,* 6, 181:1–181:12.

Merrell, P., Schkufza, E., Li, Z., Agrawala, M., and Koltun, V. 2011. Interactive furniture layout using interior design guidelines. *ACM Trans. Graph. 30,* 4, 87:1–87:9.

Michalek, J. J., Choudhary, R., and Papalambros, P. Y. 2002. Architectural layout design optimization. *Engin. Optim. 34,* 5, 461–484.

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

Müller, P., Zeng, G., Wonka, P., and Gool, L. V. 2007. Image-based procedural modeling of facades. *ACM Trans. Graph. 26,* 3, 85:1–85:9.

Musialski, P., Wimmer, M., and Wonka, P. 2012. Interactive coherence-based façade modeling. *Comput. Graph. Forum 31,* 2, 661–670.

Prusinkiewicz, P., James, M., and Měch, R. 1994. Synthetic topiary. In *Proceedings of SIGGRAPH 94*. 351–358.

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

Prusinkiewicz, P., Mündermann, L., Karwowski, R., and Lane, B. 2001. The use of positional information in the modeling of plants. In *Proceedings of SIGGRAPH 2001*. 289–300.

Šťava, O., Beneš, B., Měch, R., Aliaga, D. G., and Krištof, P. 2010. Inverse procedural modeling by automatic generation of L-systems. *Comput. Graph. Forum 29,* 2, 665–674.

Stiny, G. and Mitchell, W. J. 1980. The grammar of paradise: on the generation of Mughal gardens. *Environ. Plan. B: Plan. Des. 7,* 2, 209–226.

Talton, J. O., Lou, Y., Lesser, S., Duke, J., Měch, R., and Koltun, V. 2011. Metropolis procedural modeling. *ACM Trans. Graph. 30,* 2, 11:1–11:14.

Watson, B., Müller, P., Wonka, P., Sexton, C., Veryovka, O., and Fuller, A. 2008. Procedural urban modeling in practice. *IEEE Comput. Graph. Appl. 28,* 3, 18–26.

Wei, L.-Y., Lefebvre, S., Kwatra, V., and Turk, G. 2009. State of the art in example-based texture synthesis. In *Eurographics 2009 Annex (State of The Art Reports)*. 93–117.

Wonka, P., Wimmer, M., Sillion, F. X., and Ribarsky, W. 2003. Instant architecture. *ACM Trans. Graph. 22,* 3, 669–677.

Yu, L.-F., Yeung, S.-K., Tang, C.-K., Terzopoulos, D., Chan, T. F., and Osher, S. 2011. Make it home: Automatic optimization of furniture arrangement. *ACM Trans. Graph. 30,* 4, 86:1–86:11.

Received November 2011; revised July 2012; accepted July 2012