Recompose Grammars for Procedural Architecture

NIKLAUS HOUSKA, Esri R&D Center Zurich, Switzerland
CHERYL LAU, Esri R&D Center Zurich, Switzerland
MATTHIAS SPECHT, Esri R&D Center Zurich, Switzerland

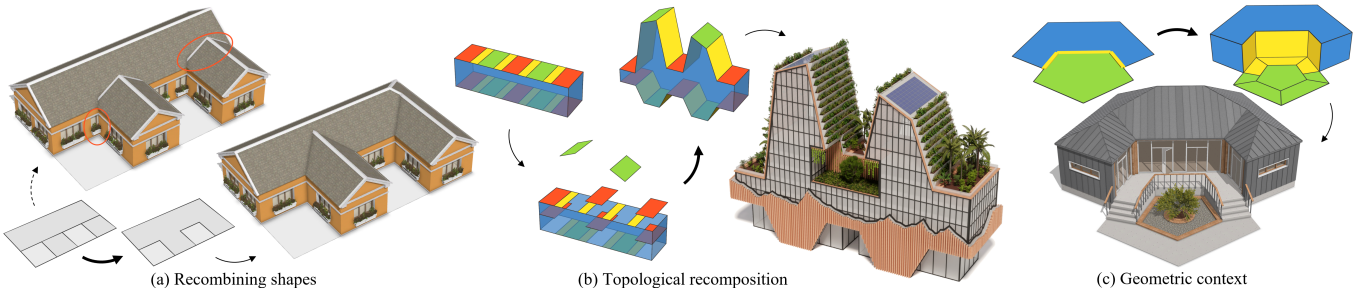

Fig. 1. Our novel grammar language Recomp facilitates many modeling scenarios that are difficult or infeasible in current systems. (a) We provide an integrated solution to recombine subdivided shapes, effectively circumventing numerous multi-shape coordination challenges (red circles). (b) A novel recomposition procedure allows for cohesive local shape modifications, significantly raising the geometric expressiveness. (c) Completed with a powerful tagging system to annotate geometry components, Recomp allows to transport contextual information geometrically and to leverage general topological context like adjacency.

We present the novel grammar language Recomp for the procedural modeling of architecture. In grammar-based approaches, the procedural refinement process is based on shape subdivisions. This process of decomposition results in disconnected subparts, which not only restricts the geometric expressiveness but also limits the control over an appropriate shape granularity needed to coordinate design decisions. Recomp overcomes these limitations by extending grammar languages with the recomposition ability. Fundamental is the concept of rule inlining, allowing for the topological recomposition of edited subparts by collapsing a shape subtree into one single shape on which derivation can continue. This is completed with a versatile geometry tagging system, allowing authors to compile and transport context information at any level of detail and gain full control over the geometry independent of the structure of the shape tree. Through various examples, we demonstrate the power of Recomp in procedural layout and mass modeling, as well as its capabilities in facilitating context-sensitive design.

CCS Concepts: • **Applied computing** → **Computer-aided design**; • **Computing methodologies** → **Mesh geometry models**.

Additional Key Words and Phrases: procedural modeling, grammar language

1 INTRODUCTION

Procedural modeling is an established method for 3D content creation in domains such as film, games, architecture, and urban planning. While some tools like Houdini or Blender provide generic procedural functionality, other applications focus on specific domains, for example SpeedTree (plants), Terragen (landscapes) or CityEngine (buildings and cities).

In the domain of buildings, typically a grammar-based approach is used to describe architecture. Such a description consists of a set of rules defining the hierarchical refinement process (derivation) from a start shape (e.g., a coarse building shell extruded from its footprint) to a detailed model. During the derivation, shapes are iteratively replaced with refined shapes until the desired level of detail is reached. This hierarchical process results in a shape tree whose leaves define the result.

Grammar-based languages, such as CGA shape [Müller et al. 2006] and CGA++ [Schwarz and Müller 2015], achieve stunning results with visually complex details. However, many common modeling scenarios pose challenges, while others remain infeasible due to the following fundamental limitations:

L1. Whenever shapes are subdivided, the parts become independent. Further refinements are then performed locally or require advanced methods, such as events [Schwarz and Müller 2015], for coordination across multiple shapes. Consequently, every occurrence of having multiple shapes representing a whole inherently complicates any subsequent derivation (Fig. 1a).
L2. Shapes cannot be locally modified without breaking the geometric cohesion. For instance, generating slanted surfaces (Fig. 1b) through local transformations is impossible because the modified parts become separate shapes.
L3. Geometric topological context normally cannot be leveraged (e.g., adjacency in the footprint; Fig. 1c), resulting in the loss of spatial relations (e.g., which facades face the courtyard) when the parts are separately derived. Conversely, because context is stored on the shape level, merging shapes also has limited applications, as information about individual parts will be lost.

In this paper we introduce *recompose grammars* that address and overcome these limitations of the hierarchical derivation process. Centered around two complementary key features — the ability to structurally recombine parts of the model and to annotate geometry with tags — we present the following new contributions:

- *Inlined derivation*: Using a new keyword, the derivation of rules and subdivisions can be inlined to collapse the resulting shape subtree back into a single shape on which derivation can continue. The concept is also used to enable Boolean 3D operations.
- *Topological geometry recomposition*: To consolidate the disconnected geometries of the leaf shapes back into one, we track geometry edits throughout the inlined derivation and specify a procedure that restores connectivity and resolves competing edits.
- *Geometry tagging*: We introduce *geometry component tags* and detail the setup of tagged geometry and tag propagation on geometry creation. Powerful selection is enabled through hierarchical grouping, wildcards, and adjacency. Additionally, we enhance shape operations to automatically add semantic tags.

The new functionality integrates seamlessly with established shape grammar concepts, offering a simple yet general solution to core limitations without increasing language complexity. Specifically, our approach does not require explicit control over the derivation order. Rules, which hierarchically evolve parts of the model, can be inlined, allowing to return to a unified model at any stage. This provides the flexibility to combine hierarchical and sequential derivation at any level of detail, while avoiding multi-shape coordination challenges (addressing L1.). Our recomposition procedure not only ensures structurally sound geometry in the recomposed shapes, but also enables cohesive local shape modifications, including edits on individual edge and vertex components (L2.). Recomposition also offers a means to apply tags to specific parts of a shape. Tags facilitate organizing the geometry of shapes and leveraging topological relations between components, enabling effective subsequent refinement independent of the geometric complexity (L3.).

We implemented the recomposition functionality (Sec. 3) and geometry tagging (Sec. 4) in our novel grammar language Recomp. In various examples we demonstrate the power of the new features, as well as their efficiency in translating modeling objectives into code (Sec. 5). Notably, the new features enable:

- topologically reconnecting the independent parts of subdivided shapes for general decompose-refine-recompose workflows,
- creating custom forms via local transformations and selective merging of adjacent surfaces, including
- geometric subroutines operating on edge and vertex components,
- organizing components for selection of complex subsets, and
- processing and transporting geometric context using tags.

2 RELATED WORK

Procedural architecture was pioneered by shape grammars [Stiny 1980; Stiny and Gips 1971], which consist of rules that geometrically match and replace one (sub)shape with another. Set grammars [Stiny 1982] simplify matching by treating shapes as symbols instead of geometric bodies, which led to split grammars [Wonka et al. 2003] for modeling building facades. From this foundation, CGA shape [Müller et al. 2006] emerged as a language for symbolic shape grammars for the procedural modeling of buildings. Recomp evolves this approach.

Control over the derivation order is another aspect and generally required to coordinate multiple shapes. Approaches include CGA shape's rule priorities, evaluation phases [Steinberger et al. 2014], and construction stages [Schwarz and Wonka 2014]. Full control over the derivation order is achieved by CGA++ [Schwarz and Müller 2015], providing a generic solution to multi-shape coordination and high expressiveness, but at the cost of increased language complexity. Our approach instead addresses the root problem that leads to many coordination challenges later on.

Other approaches pursue a sequential derivation process, in which the set of existing shapes is always known, thus naturally operating in a global context (overcoming L1.). Approaches include Krecklau and Kobbelt [2011] for interconnected structures, component based modeling [Leblanc et al. 2011], and replacing symbol matching with selection expressions [Jiang et al. 2020]. Similar are group grammars [Carra et al. 2019; Santoni and Pellacini 2016] offering grouping operators based on shape tags. With Recomp, we combine the hierarchical derivation process with advantages of sequential derivation, and generalize geometry component selection.

Notable tangential works that expand CGA shape include: interactive visual editing [Lipp et al. 2008], local editing [Lipp et al. 2019], more classes of shapes [Krecklau et al. 2010], generalized polyhedron scopes [Thaller et al. 2013], shape layers [Jesus et al. 2016] and spherical coordinate systems [Edelsbrunner et al. 2017].

Geometry tagging is used in interactive applications [Michel and Boubekeur 2021; Nazzaro et al. 2021] and tools such as Maya.

Alternative approaches include generative modeling languages [Havemann 2005], volumetric grammars [Willis et al. 2021], and graph grammars [Merrell 2023].

We extend CGA shape's most recent dialect [Esri 2023a,b]. CGA operates on shapes, which are identified by a symbol and have a scope, geometry within the scope, and attributes. The scope represents the shape's oriented bounding box. Geometry is represented by a polygon mesh alongside material information. Rules take the form α-->β, where α is a symbol denoting the rule's name and β comprises the rule's body, containing a sequence of actions (operations and symbols) that output refined shapes. When a shape's symbol matches a rule's left-hand side α, it is replaced by the rule's right-hand side β. Operations modify or subdivide the shape, while a symbol creates a copy of the shape in its current state and assigns it this symbol. If no matching rule is found, the shape becomes a leaf shape, and the union of all leaf shapes comprises the final model. This iterative process of substituting shapes until only leaf shapes remain is called *derivation*. The derivation defines a shape tree, where each shape occurring during the process is a node in this tree.

Two types of operations exist in CGA: shape modifying and subdivision. Shape modifying operations include translating, rotating and resizing the shape's scope, extruding polygons, constructing roofs, or replacing the geometry with an asset. These operations have an immediate effect on the current shape, i.e., its output is input to the following action in the rule's body. Subdivision operations, on the other hand, offer different ways of splitting the shape, resulting in one or more parts for which new actions are specified. Notably are the `comp` operation, which decomposes the geometry into its components such as faces, and the `split` operation, which subdivides the shape along one axis of the scope. Unlike shape modifying operations, subdivision operations create new shapes as successors and have no effect on the current shape.

```
Start --> inline comp(f) { top: A | all= B } Mass
A --> split(x) { 10: t(0,0,6) r(0,20,0) G | 5: Y | 10: t(0,0,-2) R
               | 5: Y | 10: t(0,0,8) r(0,-20,0) G }
Mass --> ...
```

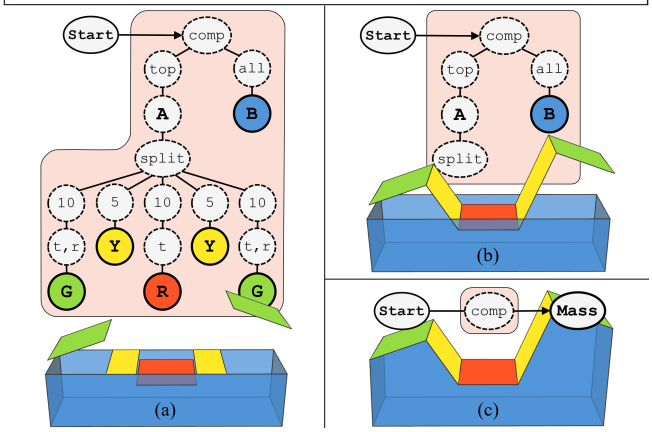

Fig. 2. Example of a grammar showcasing inlining (top). The derivation of the comp operation generates the inlined shape subtree (red) whose leaves define the model (a). The tree is then iteratively collapsed bottom-up (b) and (c), resulting in a single shape with recomposed geometry on which the derivation can continue.

3 INLINING RULES AND SHAPE SUBDIVISIONS

The main step in overcoming the limitations outlined in the Introduction is the ability to inline the derivation of symbols and subdivision operations occurring in a rule's body (Fig. 2). A symbol creates a copy of the current shape and invokes the matching rule in a new derivation branch that defines a shape subtree. When inlined, this subtree is derived immediately and then collapsed, merging its leaf shapes to replace the current shape (Sec. 3.1). In the case of subdivisions, the current shape is split into parts, generating multiple new shapes that are independently refined. Inlining then topologically reconnects the edited geometries of the resulting shapes according to our novel recomposition procedure (Sec. 3.2). This enables decompose-refine-recompose workflows that maintain the geometric cohesion, unlocking the potential for a vastly expanded range of geometrical forms that can be expressed using the available set of operations.

The derivation of a symbol or subdivision operation is inlined by prefixing it with the new `inline` keyword. Importantly, the resulting shape subtree is identical to the subtree generated if it was not inlined, keeping the derivation process consistent. To preserve information on how the geometries of the different leaf shapes relate to each other, we introduce *geometry trackers*. Whenever a subdivision operation occurs during the derivation of an inlined shape tree, such a tracker is initialized for all resulting shapes. During the derivation of a shape with a tracker attached, all changes to its topology relevant for recombining are stored. Finally, the recomposed shape replaces the current one, and the collapsed subtree is removed. Hence, an inlined rule or subdivision operation acts like a regular shape modifying operation and has no effect on the structure of the shape tree.

3.1 Collapsing the Shape Subtree

The collapse of an inlined subtree is an iterative bottom-up process. In each iteration, leaf shapes are registered with their parent. When all children of a shape P are registered, they are combined into a single shape, replacing P. This process continues until only the root is left. An illustration is given in Fig. 3, generated with the following grammar:

```
Start --> inline split(x) { '0.5: Left | '0.5: Right }
Left  --> split(z) { '1/3: G | '1/3: s('0.6,'1.5,'1) Y | '1/3: G }
Right --> split(z) { ~1: NIL | '0.5: split(y) { '0.5: R |
        '0.5: NIL } | ~1: NIL }
```

Process. In case P only has one child, P is simply replaced by that child including its geometry tracker (highlighted in blue in Fig. 3).

In case there are multiple children and P is a subdivision node (highlighted in orange in Fig. 3), each child has a geometry tracker that maps the child's geometry to a distinct part of P's geometry. Our recomposition procedure "stitches" these geometries back together. The result becomes the new geometry of P and P's tracker is updated accordingly. Note that if a child's geometry has been replaced using the insert operation, it will not be reconnected.

If P is not a subdivision node, each child maps to the same parent geometry and we do not recompose. The geometries are simply appended to replace the geometry of P, and P's geometry tracker is cleared, thus it will not be reconnected with other subparts later on.

Further strategies. To accommodate scenarios in which the shapes of the inlined subtree should be merged using Boolean 3D operations instead or not be recomposed at all, an additional parameter can be specified: `inline(union|append)`. In these cases, the leaf shapes of the subtree are directly combined and no trackers are involved.

Shape attributes. In addition to the geometry and the scope, shapes also store attribute values. Attributes are defined as part of the grammar and can be read and set during derivation. We differentiate between attribute values explicitly set for a shape (using the `set` operation) and those inherited from the parent. If a specific attribute value has been set only in one child, this value is transferred to the parent P; otherwise, P retains its current value for that attribute.

3.2 Recomposition Procedure

Given a number of locally edited subparts, the goal of the recomposition procedure is to reestablish their connectivity. The crux of the procedure lies in consolidating subdivisions of shared components, relying solely on tracked information relative to the root topology.

Vertices. The first step of the recomposition procedure is to merge vertices that map to the same original vertex. This mapping is maintained in the *geometry tracker*. The new position is determined by a simple strategy: each vertex contributes a displacement vector, the sum of which is added to the original vertex position. Formally: $v_i' = v_i + \sum_{v \in \mathbf{V}_i} (v - v_i)$, where $\mathbf{V}_i$ is the set of vertices corresponding to the original vertex v_i and that are merged at v_i'. It is worth emphasizing that the simplicity of this "sum of displacements" strategy is essential for ensuring predictable outcomes, making the recomposition procedure a reliable tool for expressing modeling operations.

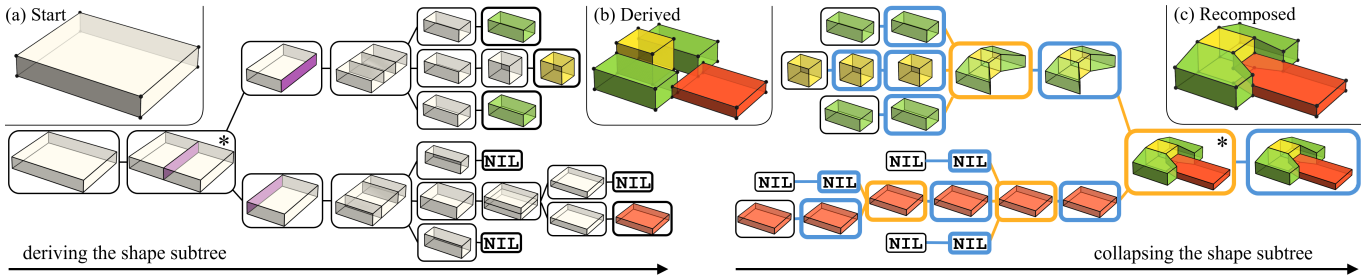

Fig. 3. Example of collapsing a shape subtree. Starting with a cube (a), each shape appearing during the derivation is a node in the shape subtree whose leaves define the derived model (b). From these leaf shapes, the subtree is traversed upwards, replacing predecessors (blue nodes) and recomposing at subdivisions (orange nodes), until only the root remains (c). Fig. 5 illustrates the recomposition at the shared face highlighted in purple (*).

Edges. The second step of the procedure is to reconnect the parts at initially shared edges, which is illustrated in Fig. 4. To accomplish this, edge splits are tracked during the derivation of the parts. Let $e = (v_a, v_b)$ be a shared edge split by the vertex v into $e_1 = (v_a, v)$ and $e_2 = (v, v_b)$, with t as the interpolation value, such that $v = (1-t) \cdot v_a + t \cdot v_b$. The pair (v, t) is associated with e and stored in the geometry tracker. To track further splits, e_1 and e_2 are added to the tracker as well. The edge recomposition method proceeds as follows: For each tracked edge, gather its splits stored in all trackers. Next, sort them based on the interpolation values, forming a chain of new edges. Finally, replace each occurrence of the edge with this chain. Note that the recomposition of edges is purely index-based and does not require the evaluation of any geometric attributes.

Faces. When splitting a volume, new faces are generated at the intersection plane, with each face being shared by two parts. The goal of the last step in the procedure is to reconnect the parts at these shared faces. Fig. 5 demonstrates the process for the shared face highlighted in purple on Fig. 3. Analogous to the approach with edges, shared face splits are tracked during the derivation. A face split is defined by an ordered list of vertex indices, representing the split segments with the first and last vertex lying on the face's boundary. We store interpolation data for each vertex inserted during the split. This involves triangulating the face to get barycentric coordinates for each newly introduced vertex. When recomposing, this enables us to treat all involved vertices as interpolations of the original vertices that define the shared faces, providing us with a planar domain that facilitates the computation of the unified tessellation. Essentially, the only geometric evaluation required during recomposition is the intersection of the face split segments (Fig. 5b).

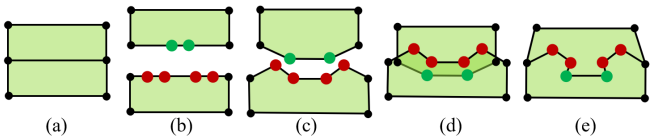

Fig. 4. Example of edge recomposition. The two faces in (a) are refined in different derivation branches: New vertices are inserted by splitting the shared edge (b). These vertices are then translated (c), leading to overlapping geometry (d). The recomposition consolidates the splits (e) based on the relative positions at the point of their insertion in (b).

Each newly formed intersection vertex generates three positions. The first is computed in the planar domain, while the remaining two are calculated on the actual segments. These latter two positions define displacements that, when added to the first position, determine the final vertex location. Ultimately, duplicate faces are removed to complete the process.

Geometry tracker. A geometry tracker maps the geometry $G_C = (V_C, F_C)$ of a shape to the geometry $G_P = (V_P, F_P)$ of its subdivision root and stores splits of shared components. Geometry is represented as a polygon mesh, consisting of a list of vertices V and polygons F, defined by a list of vertex indices. The tracker maintains its own list of vertices, V_T, initialized with V_C, as well as two index maps: $V_C \xrightarrow{I_{CT}} V_T \xrightarrow{I_{TP}} V_P$. Shared edges are stored as pairs of vertex indices connected to a list of edge splits, each comprising a vertex index and the interpolation value. Upon registering an edge split, the resulting edges are added to the data structure as well. Shared faces are stored analogously. Pivotal, all vertex indices point to V_T, allowing to preserve information about component splits even if these components were removed in G_C.

Select operation. When shapes are created from edge or vertex components using `comp(e|v)`, the higher-dimensional components (faces) are not part of the resulting shape subtree and are therefore lost when the operation is inlined. To enable actions on individual edges or vertices while allowing for recomposition into the full geometry, we introduce the `select` operation. It mirrors the syntax of `comp`, automatically collapses the resulting subtree, and includes an additional recomposition step to reintroduce missing components.

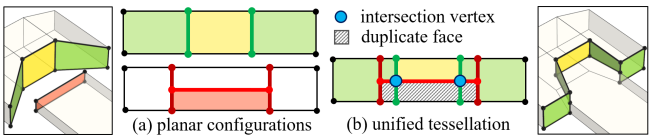

Fig. 5. Example of face recomposition. The tracked face splits are planarized onto the original face (a), highlighted in purple in Fig. 3. Then, they are combined via intersection, resulting in a unified tessellation (b). Finally, the faces are deplanarized, and duplicate faces are removed.

```
Parcel --> inline F
           cleanupGeometry(edges) // merges faces with same tags
           select(e) { tagged("A") && tagged("B"): tag("X") }
           comp(f) { tagged("A"): extrude(h) Mass }

F --> split(x) { w: tag("A")
               | ~1: split(z) { ~w: tag("B") | w: tag("A") }* }

Mass --> comp(f) { tagged("X"): FacA | side: FacB | top: Roof }
```

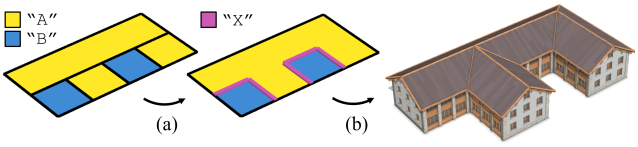

Fig. 6. Example of tags and inlining used in conjunction to create a unified F-building layout, constructing a joint roof covering the whole building, and leveraging face adjacency to distinguish facades facing the courtyards.

3.3 Boolean 3D Operations

Rule inlining allows for using one other shape as operand, offering an efficient solution for integrating multi-shape operations, like Boolean 3D operations, into shape grammars. The current shape serves as the first operand, while the inlined subtree represents the second. Both are input to a Boolean operation, the result of which replaces the current shape. In RECOMP, the operations `union`, `subtract`, and `intersect` are implemented as keywords (Fig. 8).

4 ANNOTATED GEOMETRY

In grammar-based procedural modeling, the derived model is organized in a shape tree, where each part is identified by the symbol of the shape it is contained in. When shapes are merged, i.e., using Boolean operations or through recomposition, the individual shape symbols are lost. This necessitates means of identifying geometry components to distinguish the different parts of the model and take advantage of the connectivity restored between them. Traditionally, the `comp` (component split) operation is used for this purpose, taking a list of *selector:actions* pairs, each of which identifies a set of components and specifies actions for their further refinement. In CGA languages, the *selector* primarily relies on evaluating the geometric properties of the respective component, such as its orientation, to determine a match. This does not provide sufficient control to meaningfully decompose more complex shapes.

We transcend this limitation with *tags*. Tags are string attributes stored directly on mesh components, and every face, edge, and vertex may have an arbitrary number of tags.

Applying tags. The new `tag(`*name*`)` operation assigns *name* as a tag to the highest-dimensional components available, typically faces. The `select` operation is key in enabling tags to be set on specific components, crucially edges and vertices. It allows to specify the desired components using a *selector*, apply tags in the *actions* sequence, and then return to the full geometry through recomposition. An example using face and edge tags is demonstrated in Fig. 6.

Tags are hierarchically grouped using the name separator `.` (dot), for instance `"Fac.A"`. We refer to the dot-separated parts of a tag as *subtags*. The `deleteTags` operation clears tags, while currently

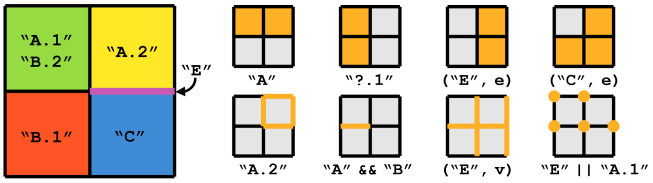

Fig. 7. Example illustrating some tag-based selection options (left: tagged shape; top right: face selection; bottom: edge/vertex selection).

applied tags are queried using the `getTags` and `hasTags` functions. Each can be provided with a query to only consider specific tags.

Tag propagation. We enhanced all geometry-modifying operations to preserve tags on existing components and define rules for their propagation to new geometry, ensuring contextual information is carried through the modeling process. Operations like `extrude`, `offset`, and all roof operations consistently propagate tags: new faces inherit the source face's tags, while new faces emanating from an edge – such as the side faces in `extrude` or the border faces in `offset` (Fig. 1c) – also inherit their source edges' tags. The same applies for edges emanating from a vertex. The `cleanupGeometry` operation maintains tagged edges and edges shared by faces with different tags, enabling the important functionality of selectively merging adjacent surfaces (Fig. 6a). On geometry subdivision, split faces and edges retain their tags, while new components at the intersection plane have none.

Component selection. The new `tagged(`*query*`)` function offers tag-based selection (Fig. 7) and can be used in all *selector* expressions. The *query* may include the wildcards `?` (question mark) to match one subtag, and `*` (asterisk) to match one or more. Due to the name grouping, a tag query like `"Fac"` also matches tags with additional subtags like `"Fac.A"`. By default, tags stored on a component can also be used to select its sub-components (i.e., to get the edges of a tagged face). To select components based on tags of sub-components or adjacent components, an additional argument is used.

Automatic semantic tagging. We further enhance established shape operations to automatically apply predefined tags that identify semantically different components of the output geometry (Fig. 8). These *auto-tags* persist only until the next call of their respective operation, at which point they are replaced with new ones.

```
Parcel --> extrude(40)
           subtract Shifted
           Mass
Shifted --> t(20,-15,-15) r(15,12,25)
Mass --> comp(f) { tagged("bool.A") ... }
```

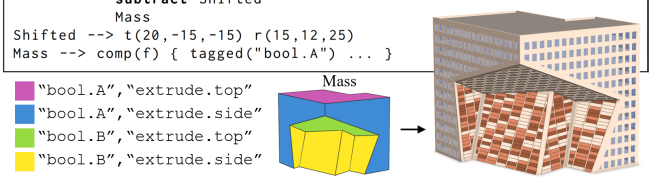

Fig. 8. Example of auto-tags applied by the `extrude` and Boolean 3D (`subtract`) operations while deriving the building mass. It uses these auto-tags to generate different facades and roofs.

5 RESULTS AND DISCUSSION

Transcending the limitations of the strict hierarchical derivation process, Recomp enables novel procedural modeling applications and increases the inherent geometric expressiveness (Fig. 9) and context-sensitivity (Fig. 10) of grammar-based approaches. Code snippets for all examples are provided in the supplemental materials.

5.1 Geometric expressiveness

Shape recomposition allows the model to be iteratively evolved at any level of detail and enables new geometric forms. While our method (Sec. 3.2) handles general recomposition cases, modeling objectives are typically expressed through a series of simple and intuitive steps. This is illustrated by the four examples in Fig 9.

Twisted building (Fig. 9 a-d). This example demonstrates the creation of a custom building mass through recomposition. The extruded footprint is first split into sections (a), and each section's top face is rotated and scaled to produce a twist-and-taper effect (b). Inlining the `split` operation then recomposes these sections into a single connected mass (c), automatically removing inner faces duplicated by the top and bottom of adjacent sections. The auto-tags, initially applied by the `splitAndSetbackPerimeter` operation to create insets, are propagated throughout this process (green and yellow), providing context for further refinement (d). The grammar:

```
Lot --> splitAndSetbackPerimeter('Offset)
      { '0.5: { Length: Depth: NIL | ~1: 0: NIL } }*
      { remainder: Footprint }

Footprint -->
    extrude(Height)
    inline split(y) { '1/Vertical_Res: YStep(split.index) }*
    comp(f) { tagged("setback.back")= GreenFacade
            | tagged("setback.side")= ConcreteFacade
            | all= Mass }

YStep(ind) --> select(f) { top: YStepPlate(ind+1) }
YStepPlate(ind) with (
    a := (ind-1) / (Vertical_Res-1)
    f := (1-a) + a*Taper_Factor
) --> r(0, 0, ind*Twist_Angle/Vertical_Res) s('f, 'f, 0) center(xy)

Mass --> split(y) { ~Floor_Height: Floor | Slab_Height: Slab }*
```

One World Trade Center (Fig. 9 e-h). The modeling steps required to transform the extruded mass (e) into an antiprism are possible, and directly translate into code, with inlining: We select the top face, tag its vertices, rotate the scope, and subdivide it with two splits (f). Then, we select the tagged vertices and move them down (g), shaping the antiprism that is further refined into the iconic landmark (h):

```
Tower -->
    extrude(Height)
    select(f) { top: SplitFace }
    select(v) { tagged("Corner"): translate(world, 0, -Height, 0) }
    cleanupGeometry(vertices) // merges collapsed edges
    MakeDetails

SplitFace with ( w := sqrt(scope.sx*scope.sx/2) ) -->
    select(v) { all: tag("Corner") }
    rotateScope(0, 0, 45)
    inline split(x) { ~1: X. | w: X. | ~1: X. }
    inline split(y) { ~1: X. | w: X. | ~1: X. }
```

Warped window facade (Fig. 9 i-m). This example illustrates a custom facade layout. The surface is initially split into independent wall and tagged window tiles (i). The window geometry is then

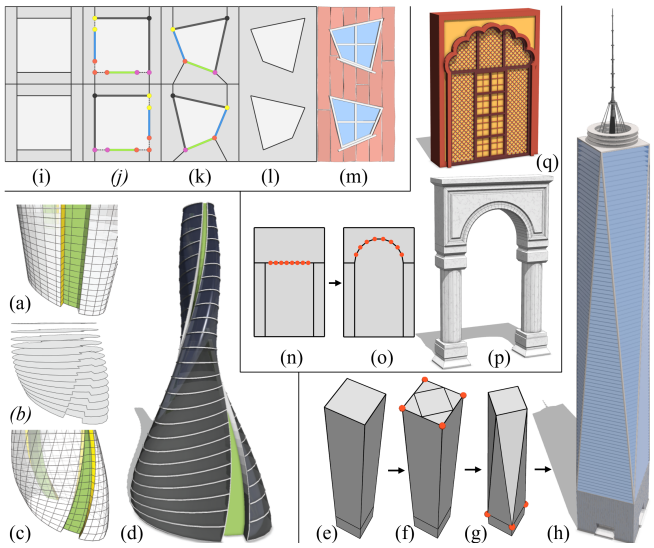

Fig. 9. Layout and mass modeling examples with pursued modeling strategies and final results. *(b)* and *(j)* show the model before recomposition.

locally deformed by selecting edges and resizing them (j). Inlining the facade rule applies these edits to the entire geometry (k) and also enables subsequent geometry cleanup to merge the individual wall pieces back into a single surface with window holes (l). Further refinements can then again operate on the whole facade, such as generating the wooden slats that span across the entire surface (m).

Procedural arches (Fig. 9 n-q). Recomp enables rules to function as subroutines for typical geometric procedures. In this example, a rule is designed to process a single edge. The edge is split and the new vertices moved into a circle using trigonometric calculations. Using `select`, this rule is applied to the middle edge in (n), creating the layout (o) for a Roman arch (p). The rule is simply applied twice to create the recursive pattern on a multifoil arch (q).

5.2 Context-sensitivity

The topological connectivity restored by our recomposition procedure, together with tags, provide direct access to contextual information required to drive design decisions. We illustrate this with the construction of a parameterized low-poly house in Fig. 10.

We start by unioning three tagged planes, representing the building base and extension, and a designated area for a tree (a). Then, we extrude both building sections to the shared height, and further extrude the higher section (if any), tagging each level accordingly. This results in one connected model contained within a single shape (b), where tags group the surfaces in two ways: once per layout section and once per floor level. Our design aims to transform the lower roof into a terrace if accessible from the upper level, or construct a joint flat roof if both roofs are adjacent. Moreover, large windows should face the tree area and the terrace. Using the applied tags, as well as geometric and topological properties of the model, we can reliably derive the required information (c). For instance, edges connecting the lower roof and the upper level represent the terrace

entrance and are tagged as such. Based on this tag, we can then identify the large window and terrace surfaces to continue separate derivation (d). Extract from the grammar:

```
Mass with (
    h1 := min(Base_NFloors,Extension_NFloors)*Floor_Height
    h2 := abs(Base_NFloors-Extension_NFloors)*Floor_Height
    highPart := case Base_NFloors > Extension_NFloors: "Bldg.Base"
                case Base_NFLoors < Extension_NFloors: "Bldg.Ext"
                else:                                  "None"
) -->
    select(f) { tagged("Bldg")= extrude(h1) }
    select(f) { top && tagged(highPart)= extrude(h2) tag("Lvl.2")
              | tagged("Bldg"): tag("Lvl.1") }

    select(f) { top && tag("Lvl.1"): tag("Roof.Low")
              | top && tag("Lvl.2"): tag("Roof.High") }

    select(e) { tagged("Roof.Low") && tagged("Lvl.2"):
                    tag("TerraceEntr")
              | tagged("Roof") && tagged("Bldg.Base")
                    && tagged("Bldg.Ext"): tag("Roof.FlatMarker") }
    DecomposeMass

DecomposeMass -->
    comp(f) { side && tagged("TreeArea", e)= LargeWindows
            | side && tagged("TerraceEntr", e)= LargeWindows
            | side= Facades
            | tagged("Roof.Low") && tagged("TerraceEntr", e)=
                  Terrace
            | tagged("Roof")= Roof
            | tagged("TreeArea"): Tree }

Terrace --> TerrFloor comp(e) { !tagged("TerraceEntr")= Railings }

Roof-->
    case hasTags("Roof.FlatMarker"):
        deleteTags() cleanupGeometry(edges) FlatRoof
    else: GableRoof
```

The preserved connectivity in the mass model not only facilitates the overall design process, but also simplifies subsequent modeling tasks. First, through geometry cleanup, adjacent surfaces are merged, enabling a split pattern across different building sections for the large windows (e), and removing inner borders on the flat roof. Second, the edge tags allow to avoid placing terrace railings that obstruct facades (f). Third, by having all facade pieces together in one shape, we can use functions to query the index of the largest front-facing one to place a single door. Without recomposition, these tasks are typical examples of coordination challenges, requiring access to other shapes via synchronization of the derivation process. The information lost through subdivision must then be inferred via spatial queries, while relying on Boolean operations to geometrically merge shapes, which are not guaranteed to restore the topology.

5.3 Comparison to Related Methods

Numerous prior works deal with grammar-based procedural modeling. With Recomp, we are the first to present a comprehensive solution to recombining parts of a hierarchically derived model.

CGA++ [Schwarz and Müller 2015] introduces events to synchronize derivation and grants first-class citizenship to shapes and shape trees. This provides great support to coordinate refinement decisions across multiple shapes. However, CGA++ inherits the limitations of hierarchical derivation and does not effectively address scenarios where independent shapes ultimately attempt to form a whole. This and many coordination tasks are subsumed by Recomp's ability to return to a unified and tagged model.

To organize and select shapes in a sequential derivation process, Selex [Jiang et al. 2020] uses virtual grids and selection expressions,

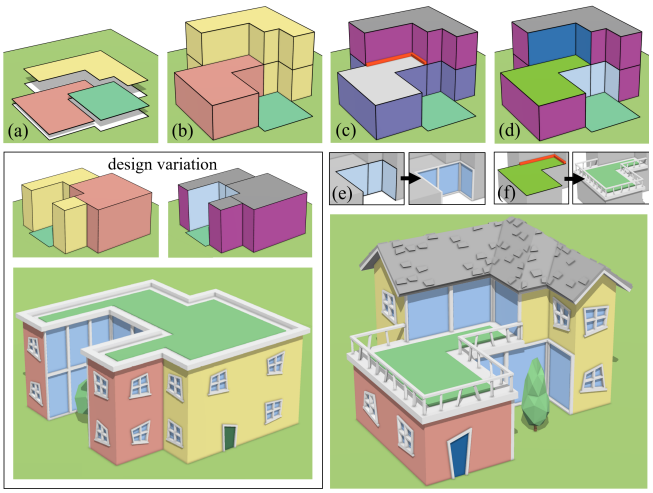

Fig. 10. Low-poly building design, showing two variations. (a-c) Pursued modeling strategy with colors visualizing tags. (d) Decomposed mass for separate further refinement. (e,f) Modeling details. (bottom) Results.

while group grammars [Carra et al. 2019; Santoni and Pellacini 2016] use shape tags and grouping operators. In contrast, Recomp allows to sequentialize hierarchical derivation steps through recomposition and uses tags to organize and select geometry inside a single shape.

Generalizations like more classes of shapes [Krecklau et al. 2010], polyhedron scopes [Thaller et al. 2013], and spherical coordinate systems [Edelsbrunner et al. 2017] allow rules to operate in extended domains, which would integrate well with Recomp's approach.

Geometry component tags exist in other tools (e.g., Maya) where they can be interactively set in a GUI. With Recomp, we introduce tags to procedural modeling and define the domain-specific functionality. Related concepts are semantic tags [Schwarz and Wonka 2014], which resemble our auto-tags, and CityEngine's limited edge attributes [Esri 2023b] to bring in external context, such as street adjacency of the initial shape. Our tags allow such information to be processed and reliably propagated throughout the derivation.

Limitations. While Recomp allows shapes to be recombined at their common root, it does not provide general access to other shapes. Our procedure does not prevent invalid geometry (e.g., a user could move a vertex to create a self-intersecting surface). Besides that, control over the recomposition process (i.e., vertex merging) and evolving tags into key-value pairs would allow for more functionality. This could be explored in future work.

6 CONCLUSION

We have presented Recomp, a novel grammar language for procedural architecture that turns the strict decompose-refine workflow of established languages into a flexible decompose-refine-recompose workflow. The cornerstone is a novel topological recomposition procedure for cohesive shape recombination, complemented with geometry component tags for effective further refinement. As demonstrated in various examples, the new features integrate seamlessly with existing functionality, and overcome important limitations of

the hierarchical derivation, enabling many new applications and solutions to previously unattainable modeling tasks.

ACKNOWLEDGMENTS

We thank Pascal Müller and Olga Sorkine-Hornung for enabling this research and providing valuable advice. We thank Mark Quinn for supporting us with high-quality renderings of our examples and thank Stefan Lienhard for reviewing and providing suggestions for improving the paper.

REFERENCES

Edoardo Carra, Christian Santoni, and Fabio Pellacini. 2019. Grammar-based procedural animations for motion graphics. *Computers & Graphics* 78 (2019), 97–107. https://doi.org/10.1016/j.cag.2018.11.007

Johannes Edelsbrunner, Sven Havemann, Alexei Sourin, and Dieter W. Fellner. 2017. Procedural modeling of architecture with round geometry. *Computers & Graphics* 64 (2017), 14–25. https://doi.org/10.1016/j.cag.2017.01.004 Cyberworlds 2016.

Esri. 2023a. *ArcGIS CityEngine*. https://www.esri.com/en-us/arcgis/products/arcgis-cityengine/overview

Esri. 2023b. *CGA shape grammar reference*. https://doc.arcgis.com/en/cityengine/2023.0/cga/cityengine-cga-introduction.htm

Sven Havemann. 2005. *Generative Mesh Modeling*. Ph. D. Dissertation. TU Braunschweig.

Diego Jesus, António Coelho, and António Sousa. 2016. Layered shape grammars for procedural modelling of buildings. *The Visual Computer* 32 (06 2016). https://doi.org/10.1007/s00371-016-1254-8

Haiyong Jiang, Dong-Ming Yan, Xiaopeng Zhang, and Peter Wonka. 2020. Selection Expressions for Procedural Modeling. *IEEE Transactions on Visualization and Computer Graphics* 26, 4 (2020), 1775–1788. https://doi.org/10.1109/TVCG.2018.2877614

Lars Krecklau and Leif Kobbelt. 2011. Procedural Modeling of Interconnected Structures. *Computer Graphics Forum* 30 (04 2011), 335–344. https://doi.org/10.1111/j.1467-8659.2011.01864.x

Lars Krecklau, Darko Pavic, and Leif Kobbelt. 2010. Generalized Use of Non-Terminal Symbols for Procedural Modeling. *Computer Graphics Forum* 29 (08 2010), 2291 – 2303. https://doi.org/10.1111/j.1467-8659.2010.01714.x

Luc Leblanc, Jocelyn Houle, and Pierre Poulin. 2011. Component-Based Modeling of Complete Buildings. In *Proceedings of Graphics Interface 2011* (St. John's, Newfoundland, Canada) *(GI '11)*. Canadian Human-Computer Communications Society, Waterloo, CAN, 87–94.

Markus Lipp, Matthias Specht, Cheryl Lau, Peter Wonka, and Pascal Müller. 2019. Local Editing of Procedural Models. *Computer Graphics Forum* 38, 2 (2019), 13–25. https://doi.org/10.1111/cgf.13615

Markus Lipp, Peter Wonka, and Michael Wimmer. 2008. Interactive Visual Editing of Grammars for Procedural Architecture. In *ACM SIGGRAPH 2008 Papers* (Los Angeles, California) *(SIGGRAPH '08)*. Association for Computing Machinery, New York, NY, USA, Article 102, 10 pages. https://doi.org/10.1145/1399504.1360701

Paul Merrell. 2023. Example-Based Procedural Modeling Using Graph Grammars. *ACM Trans. Graph.* 42, 4, Article 60 (jul 2023), 16 pages. https://doi.org/10.1145/3592119

Élie Michel and Tamy Boubekeur. 2021. DAG amendment for inverse control of parametric shapes. *ACM Trans. Graph.* 40, 4, Article 173 (jul 2021), 14 pages. https://doi.org/10.1145/3450626.3459823

Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. 2006. Procedural Modeling of Buildings. *ACM Trans. Graph.* 25 (07 2006), 614–623. https://doi.org/10.1145/1141911.1141931

Giacomo Nazzaro, Enrico Puppo, and Fabio Pellacini. 2021. geoTangle: Interactive Design of Geodesic Tangle Patterns on Surfaces. *ACM Trans. Graph.* 41, 2, Article 12 (nov 2021), 17 pages. https://doi.org/10.1145/3487909

Christian Santoni and Fabio Pellacini. 2016. gTangle: a grammar for the procedural generation of tangle patterns. *ACM Trans. Graph.* 35, 6, Article 182 (dec 2016), 11 pages. https://doi.org/10.1145/2980179.2982417

Michael Schwarz and Pascal Müller. 2015. Advanced Procedural Modeling of Architecture. *ACM Trans. Graph.* 34, 4, Article 107 (jul 2015), 12 pages. https://doi.org/10.1145/2766956

Michael Schwarz and Peter Wonka. 2014. Procedural Design of Exterior Lighting for Buildings with Complex Constraints. *ACM Trans. Graph.* 33, 5, Article 166 (sep 2014), 16 pages. https://doi.org/10.1145/2629573

Markus Steinberger, Michael Kenzel, Bernhard Kainz, Joerg Mueller, Peter Wonka, and Dieter Schmalstieg. 2014. Parallel Generation of Architecture on the GPU. *Computer Graphics Forum* 33 (04 2014), 73–82. https://doi.org/10.1111/cgf.12312

George Stiny. 1980. Introduction to Shape and Shape Grammars. *Environment and Planning B: Planning and Design* 7, 3 (1980), 343–351. https://doi.org/10.1068/b070343

George Stiny. 1982. Spatial Relations and Grammars. *Environment and Planning B: Planning and Design* 9, 1 (1982), 113–114. https://doi.org/10.1068/b090113

George Stiny and James Gips. 1971. Shape Grammars and the Generative Specification of Painting and Sculpture, In Information Processing. *IFIP Congress* 71, 1460–1465.

Wolfgang Thaller, Ulrich Krispel, René Zmugg, Sven Havemann, and Dieter W. Fellner. 2013. Shape grammars on convex polyhedra. *Computers & Graphics* 37, 6 (2013), 707–717. https://doi.org/10.1016/j.cag.2013.05.012 Shape Modeling International (SMI) Conference 2013.

Andrew R. Willis, Prashant Ganesh, Kyle Volle, Jincheng Zhang, and Kevin Brink. 2021. Volumetric procedural models for shape representation. *Graphics and Visual Computing* 4 (2021), 200018. https://doi.org/10.1016/j.gvc.2021.200018

Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. 2003. Instant Architecture. *ACM Trans. Graph.* 22, 3 (jul 2003), 669–677. https://doi.org/10.1145/882262.882324

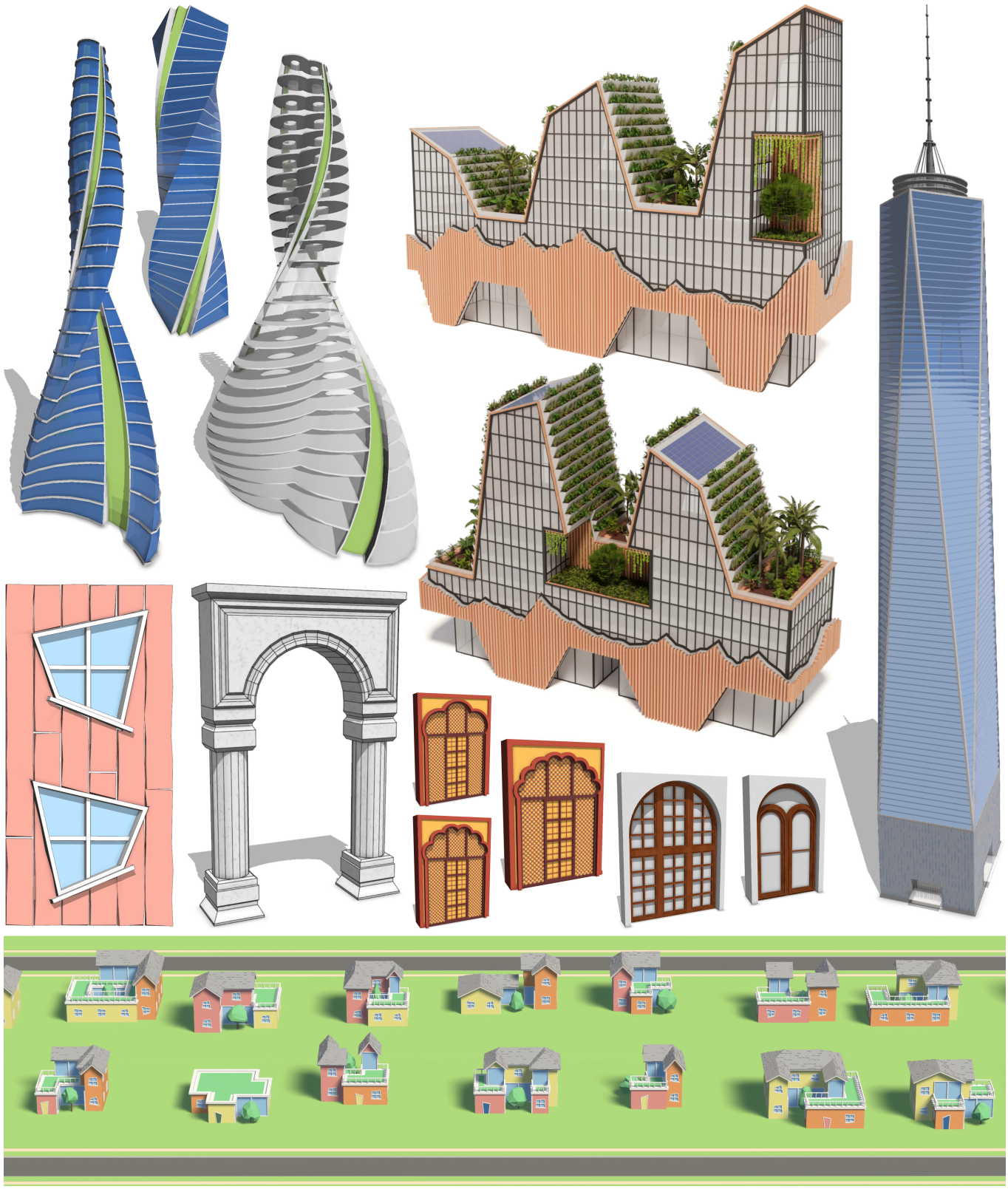

Fig. 11. More variations of the examples.