Recompose Grammars for Procedural Architecture — Supplemental Material

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

OVERVIEW

In this supplemental material, we give an overview of all new language elements introduced by Recomp (Sec. 1) and provide additional code (Sec. 2) that was used to generate the examples presented in the paper.

1 RECOMP REFERENCE

This section provides a list of all new keywords, operations, functions and selectors added in respect to CGA shape's most recent version [Esri 2023].

1.1 Inlined Derivation and Boolean 3D Operations

Inlined derivation is specified with the new keyword:

`inline` *symbol or subdivison operation*
`inline``(union|append)` *symbol or subdivison operation*
: Triggers the immediate derivation of the following symbol or subdivison operation. The resulting subtree is collapsed using the given strategy (default is recomposition) into one new shape that replaces the current.

Similarly, Boolean 3D operations can be used with separate keywords:

`intersect` *symbol or subdivison operation*
`subtract` *symbol or subdivison operation*
`union` *symbol or subdivison operation*
: Mirrors the functionality of `inline`, but instead of replacing the current shape with the new one, both shapes are input to the specified Boolean operation.

1.2 Geometry Component Tags

Tags are set and deleted with the following operations:

`tag``(`*name*`)`
: Adds *name* as a tag to all highest-dimensional components available. Any valid string is supported as *name*, except reserved wildcards characters (`?`, `*`), which are filtered out. The `.` (dot) character is interpreted as name separator.

`deleteTags``()`
`deleteTags``(`*query*`)`
: Deletes tags from all geometry components. The optional *query* parameter can be used to only delete matching tags.

Tags are queried with the following functions:

`getTags``()`
: Returns a list of all applied tags. The optional *query* parameter can be used to only get matching tags.

`hasTags``(`*query*`)`
: Returns `true` if any geometry component has a tag matching the *query*.

1.3 Component Selection

Tag-based selection is offered with the new selector functions:

`tagged``(`*query*`)`
: Returns `true` if the component has a tag matching the *query*. Edge and vertex components inherit the tags of adjacent higher-dimensional components for selection.

`tagged``(`*query*`, e|v)`
: Evaluates `tagged(`*query*`)` on the specified sub-components instead, and returns `true` if any sub-component is selected.

predicate expressions
: Static selectors (e.g., `front`) and functions can be logically combined. General functions are evaluated within the geometric context of the inspected component.

1.4 Shape subdivision

Operations can be locally applied and reintegrated with the rest using the new `select` operation:

`select``(f|e|v|fe|fv) {` *selector operator actions* `| ... }`
: Mirrors the functionality of `comp`, but with always inlined derivation. The subsequent recomposition procedure implicitely adds non-selected and missing higher-dimensional components to the result, recomposing them with the modified components.
Default components are `f` (faces), `e` (edges), `v` (vertices). The resulting scope of face components is always `x`-aligned to the first edge and `z`-aligned to the face normal. However, the direction of the `x`-axis of edge components depends on the underlying indexing, while only the `y`-axis of vertex components is generally predefined. For this reason, the `fe` (face edges) and `fv` (face vertices) options exist, generating a separate edge or vertex component per face with consistent scopes: the `x`-axis follows the direction of the face's boundary, while the `z`-axis points outwards.
The *operator* allows the selected components to be treated separately (`:`) or combined into a single shape (`=`).

2 EXAMPLES

This section provides additional code for the examples in the paper. The code snippets focus on selected modeling steps that use the new features of RECOMP. Attribute definitions and rules adding geometric details are omitted for brevity.

2.1 Slanted Tower Mass and Mountain Facade

This example is shown in Fig. 1b in the paper. The construction of the mass, which is similar to but slightly more complex than what is shown in Fig. 2 in the paper, creates a mass with slanted towers, raised bottom faces, and a subtracted box for a tree garden (Fig. 1a-c here). The mountain facade pattern is also created procedurally relying on recomposition (Fig. 1d-f here).

We apply the `Mass` rule to a rectangular building footprint. First, context is setup by tagging the facades based on orientation, which is needed for the mountain facade pattern later on. Then, the top and bottom faces are split into sections (a) and locally transformed using inlined derivation (b).

```
Mass -->
    extrude(Base_Height + Building_Height)
    inline SetupContext
    select(f) { top: RoofPattern | bottom: BottomPattern }
    subtract TreeBoxVolume
    comp(f) { tagged("Facade")= MountainFacades
            | tagged("Roof")= ...
            | tagged("TreeBox")= ... }

SetupContext -->
    select(f) { front: tag("Facade.Front")
              | right: tag("Facade.Right")
              | back: tag("Facade.Back")
              | left: tag("Facade.Left") }

RoofPattern -->
    split(x) { ~Level_Width: Level(Switch)
             | { ~0.5*Level_Width: tag("Roof.Slanted.A")
               | ~Level_Width: Level(!Switch)
               | ~0.5*Level_Width: tag("Roof.Slanted.B")
               | ~Level_Width: Level(Switch) }* }

Level(switch) -->
    case switch: LowLevel
    else:        HighLevel

LowLevel -->
    t(0, 0, rint(rand(0,2))*Floor_Height)
    tag("Roof.Level.Low")

HighLevel -->
    t(0, 0, rand(4,8)*Floor_Height)
    r(0, rand(-30,30), 0)
    tag("Roof.Level.High")

BottomPattern --> ...
```

Next, a box is subtracted (c). Its position and size are parameterized for manual control.

```
TreeBoxVolume -->
    t(TB_tx, Base_Height+TB_ty, TB_tz)
    s(TB_sx, TB_sy, TB_sz)
    primitiveCube()
    tag("TreeBox")
```

The generation of the mountain pattern begins with splitting the facades into upper and lower sections and recomposing. Then, the shared edges between the upper and lower sections are recursively split, and the new vertices are randomly moved vertically (d). Finally, the vertices at the ends of adjacent facades are merged to create a continuous pattern around the building (e). The whole step is repeated for an additional level of detail (f).

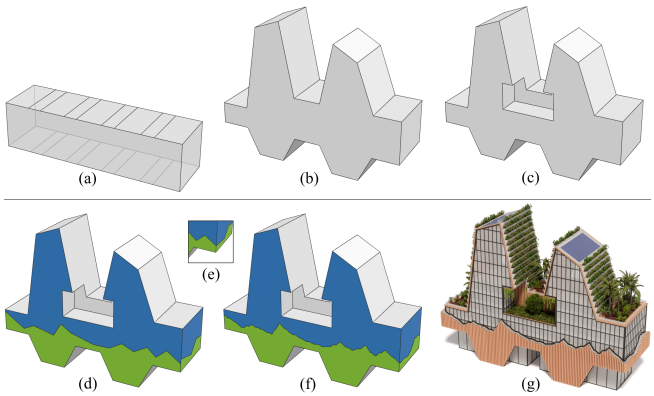

Fig. 1. Slanted tower building. (a-c) Basic mass model construction steps. (d-f) Modeling steps for the custom facade with a procedural mountain pattern wrapping around the building. (g) Final result.

```
Mountain_Res = 10
Mountain_Move_Percent = 0.9

MountainFacades -->
    select(f) { tagged("Facade")= SplitMountainFacade }
    inline MakeMountains(Mountain_Res, Mountain_Move_Percent)
    inline MakeMountains(0.25*Mountain_Res,1-Mountain_Move_Percent)
    comp(f) { tagged("Facade.Lower"): FacadeLower
            | tagged("Facade.Upper"): FacadeUpper }

SplitMountainFacade with (
    baseHeight := Base_Height + Mountain_Base_Height
) -->
    inline split(y) { baseHeight: tag("Facade.Lower")
                    | ~1:         tag("Facade.Upper") }
    select(e) { tagged("Facade.Lower") &&
                tagged("Facade.Upper"): tag("MountainEdge") }

MakeMountains(res, percent) -->
    select(e) { tagged("MountainEdge"): SplitRec(res, percent) }
    inline MergeMountainVertices

SplitRec(res, percent)-->
    case scope.sx < 1.5*res: MoveVertex(percent)
    else: split(x) { rand(0.5*res, res): MoveVertex(percent)
                   | ~1:                 SplitRec(res, percent) }

MoveVertex(percent) -->
    alignScopeToAxes(y)
    select(v) { 0: t(0, rand(0, percent*Max_Mountain_Height), 0) }

MergeMountainVertices -->
    inline MergeVertices("Front","Right")
    inline MergeVertices("Right","Back")
    inline MergeVertices("Back","Left")
    inline MergeVertices("Left","Front")

MergeVertices(a, b) -->
    select(v) { tagged("Facade."+a) && tagged("Facade."+b) &&
                tagged("Facade.Lower") && tagged("Facade.Upper")=
                    s(0,0,0) }
```

2.2 Building with Front Porch

This example shown in Fig. 1c in the paper demonstrates how adjacency information can be leveraged to build up context. More detailed construction steps are illustrated in Fig. 2 here. We start with the `Lot` rule on a square. The inlined `Layout` rule replaces it with a pentagon as building footprint and unions it with a scaled and translated copy representing the courtyard (a). Then we use tag removal and geometry cleanup to assign the intersection between both footprints to the courtyard, and tag the adjacency (b).

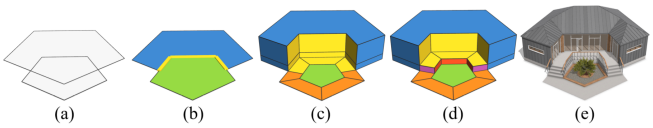

Fig. 2. Building with a front porch. (a-d) Visual guide for the construction steps. (e) Final result.

```
Lot -->
    inline Layout
    select(e) { tagged("bool.A") &&
                tagged("bool.B"): tag("Entrance") }
    comp(f) { tagged("bool.A"): House
            | tagged("bool.B"): Courtyard }

Layout -->
    primitiveDisk(5) // inserts a pentagon
    union CourtyardFootprint
    select(f) { tagged("bool.A") &&
                tagged("bool.B"): deleteTags("bool.A") }
    cleanupGeometry(edges)

CourtyardFootprint -->
    s(5, 0, 5) center(x) t(4, 0, 9)
```

The edge tag (yellow) previously applied to the shared edge is automatically propagated to the facade surfaces (c).

```
House -->
    extrude(3.5)
    split(y) { 0.8: Concrete | ~1: Floor }

Floor -->
    comp(f) { tagged("Entrance")= GlassDoors
            | side: Facade | top: Roof }
```

To model the courtyard area, we use the `offset` operation, which propagates the edge tags to the new offset border faces. This allows to determine the location of the stairs (purple) and railings (red) for the elevated front porch (d).

```
Courtyard -->
    select(e){ !tagged("Entrance"): tag("Path") }
    offset(-2)
    select(f) { inside: tag("TreeArea") }
    select(e) { tagged("Entrance") && tagged("Path"): tag("Stairs")
              | tagged("Entrance") &&
                tagged("TreeArea"): tag("Railings") }
    comp(f) { tagged("Entrance")= Porch
            | inside: TreeArea | all: Path }

Porch -->
    extrude(0.8)
    comp(f) { tagged("Stairs"): Stairs }
    comp(fe) { top && tagged("Railings")= Railings }
    Concrete
```

2.3 Warped Windows Facade and Wooden Slats

This example is presented in Sec. 5.1 in the paper. Derivation begins with the `WarpedWindowFacade` rule applied to a facade surface. We start by splitting out rectangular windows, and then we warp them by resizing selected window edges (using index-based selection). Although the facade is split into multiple pieces, we are able to recompose all the facade pieces into one shape in order to create a continuous facade of wooden slats.

```
WarpedWindowFacade -->
    inline FacadePattern
    select(f) { tagged("Window"): ResizeEdges }
    cleanupGeometry(edges)
    comp(f) { tagged("Window"): ... | all: WoodenSlats }
```

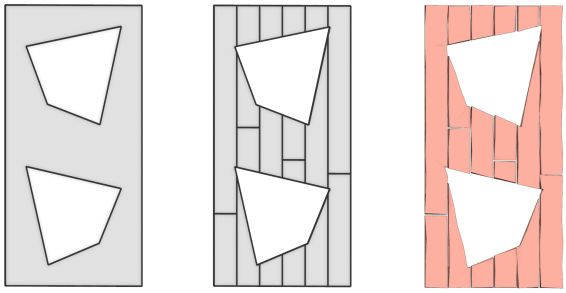

Fig. 3. Wooden slats construction.

```
FacadePattern --> split(y) { Floor_Height: Row }*
Row --> split(x) { Tile_Width: Tile }*
Tile -->
    split(x) { ~1: X.
             | Window_Width: split(y) { ~1: X.
                                      | Window_Height: tag("Window")
                                      | ~1: X. }
             | ~1: X. }

zeroOrTwo = 50%: 0 else: 2
ResizeEdges -->
    comp(fe) { zeroOrTwo:   s('rand(0.5, 0.8), 0, 0) center(x)
             | zeroOrTwo+1: s('rand(0.5, 0.8), 0, 0) center(x) }
```

To generate wooden slats across the entire facade, we apply the `WoodenSlats` rule to the facade face after window holes are removed (Fig. 3 here). The procedural slats have small deformations, giving each slat individual character. This is done by adding vertices, randomly perturbing them, and recomposing:

```
Cut_Percent = 0.6
Slat_Spacing = 0.01
Slat_Deviation = 0.7

WoodenSlats -->
   split(x) { ~Slat_Width: CutSlat }*

CutSlat -->
   case p(Cut_Percent):
      split(y) { 'rand(0.1,0.9): DeformSlat | ~1: DeformSlat }
   else:
      DeformSlat

DeformSlat with( a:=Slat_Deviation*Slat_Spacing ) -->
   offset(-Slat_Spacing, inside)
   select(e) { all: split(x) { ~0.3: X. }* }
   extrude(Slat_Depth)
   select(fv) { tagged("extrude.top"):
        t(rand(-a,a),rand(-a,a),rand(-a,a)) }
   color(Facade_Color)
```

2.4 Arch Subroutine

This example presented in Sec. 5.1 in the paper illustrates how rules can act as geometric subroutines on individual edges. The

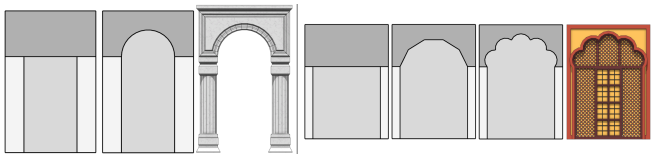

Fig. 4. Demonstration of the arch rule: (left) round arch, (right) multifoil arch.

MakeArch rule expects a single edge as input and curves it, offering two parameters: the number of vertices to insert and the angle span of the curved section. The CurveSharedEdge rule is a wrapper for MakeArch ensuring a correct scope for the local transformations. It takes two face tags as input — bottom and top — and curves each shared edge outwards from the bottom face.

```
CurveSharedEdge(bottom, top, n, angleSpan) -->
    select(e) { tagged(top) && tagged(bottom): tag("EdgeMarker") }
    select(f) { tagged(bottom):
        select(fe) { tagged("EdgeMarker"):
            MakeArch(n+1, angleSpan) } }
    deleteTags("EdgeMarker")

MakeArch(n, angleSpan) with (
   w := scope.sx
   r := w / 2
)-->
    split(x) { { (w/n): MoveVertex(r, angleSpan) }*
             | (w/n): X. }

MoveVertex(r, angleSpan) with (
    frac  := (split.index+1)/split.total - 0.5
    angle := frac * angleSpan
    delta := (180 - angleSpan) / 2
    dx := 2*r*frac - r*sin(angle)/cos(delta)
    dz := r*cos(angle) - r*sin(delta)
) -->
    select(fv) { 1: t(dx, 0, dz) }
```

We use the CurveSharedEdge rule to make different arches (Fig. 4 here). The RoundArch and MultifoilArch rules are applied to a face, which is first split into a basic tagged layout. The tags identify the shared edge that should be curved.

```
RoundArch -->
    inline BasicPattern(0.4)
    inline CurveSharedEdge("Bottom", "Top", 16, 180)
    comp(f) { tagged("Bottom"): NIL
            | tagged("Top"): ArchTop
            | all: Pillar }

BasicPattern(side) -->
    split(y) { ~1: split(x) { side: X.
                            | ~1:  tag("Bottom")
                            | side: X. }
             | scope.sx/2: tag("Top") }
```

To create the recursive pattern on a multifoil arch, we apply the arch rule twice:

```
MultifoilArch -->
    inline BasicPattern(0.18)
    inline CurveSharedEdge("Bottom", "Top", 4, 160)
    inline CurveSharedEdge("Bottom", "Top", 10, 140)
    comp(f) { tagged("Bottom"): Window
            | all= OuterFrame }
```

2.5 Single Door

This modeling step (illustrated in Fig. 5 here) is part of the low-poly house example presented in Sec. 5.2 in the paper. The single door is achieved because all eligible facade surfaces (a) are within a single shape. We use the comp function to get the width of all front-facing facades on floor level (b) and the sortIndices function to determine the index of the widest one (c), and place a door there (d).

```
Facades with (
    widths := comp(f) { tagged("Lvl.1") && front: scope.sx
                      | all: 0 }
    doorFacadeIndex := sortIndices(widths)[0]
)-->
    comp(f) { doorFacadeIndex: FacadeWithDoor }
            | all = NormalFacades }
```

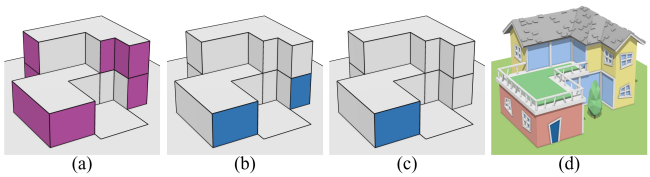

Fig. 5. Determining the single door facade.

REFERENCES

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