Layered Shape Grammars for Procedural Modelling of Buildings

Diego Jesus António Coelho António Augusto Sousa

Abstract

The effort of creating virtual city environments is reduced by using procedural modelling techniques. However, these typically use split-based approaches wich can impose limitations on the final geometry, usually enforcing a grid-like structure and require complex geometry to be imported. Layered shape grammars can increase the variability of procedural buildings while the vectorial definition of shapes introduce the possibility of creating complex shapes that seamlessly blend into the model. We evaluate the contributions with a modelling example and a comparison with split-based procedural modelling techniques. Results show that layers allow more flexibility than split-based techniques in creating variations. Vectorially defined shapes are a step forward in shape grammar expressiveness.

1 Introduction

Urban scenery is required in a vast amount of applications, including urban planning, cinema and games. However, manually modeling these spaces is a labour intensive task that demands every element to be individually modelled.

Procedural modelling, however, is gaining ground as an alternative to the traditional modelling of urban scenes. These semi-automatic methods enable the generation of larger spaces with smaller amounts of effort, reducing production costs.

More specifically, procedural modelling of buildings is often achieved with shape grammars (Parish and Müller (2001), Wonka et al. (2003), Müller et al. (2006), Krecklau et al. (2010)) or with graph based approaches (Patow (2012), Silva et al. (2013), Silva et al. (2015)). In both cases, procedural methods encode the building generation in rules which are iteratively applied and replace one shape with a new set of shapes, incrementally adding detail to the models. The paradigm, however, is based on splitting operations that divide the geometry into parts that will be detailed separately.

However, split-based approaches impose too strict limitations on the geometry that can be procedurally generated. Namely, the created shapes are mostly rectangular and typically enforce a grid like structure. Some façades, however, don't possess such a trivial structure and, therefore, become quite difficult to encode with this paradigm.

Introducing layers in planar surfaces allows more complex layouts and more compact and natural representations of façades. (Zhang et al. (2013)). Also, layers are common concepts in many 2D or 3D content generation tools, with which most users are familiar with. Reordering layers or toggling the visibility of their contents can lead to different results and a non-linear creative workflow.

In split-based methods, complex geometry can often only be imported as externally modelled assets which are, sometimes, not easily integrated with the rest of the model. Allowing the vectorial definition of two-dimensional shapes within a model's planar surfaces has the capability of improving the expressiveness of procedural modelling methods.

We propose a system that extends shape grammars (e.g., CGA introduced by Müller et al. (2006)) with operations that allow the specification of both layers and the vectorial definition of 2D shapes. By doing so, the methodology is capable of generating non-trivial layouts, going beyond the regular grid structure, and seamlessly integrate more complex architectural elements such as arched doors and windows.

Overall, the contributions of this paper are:

- A set of procedural operations to specify layers and the vectorial definition of shapes.

- An algorithm to merge the layers, taking into account shape clipping and occlusion, according to their depth within layers, resulting in a new set of independent surfaces. We named this process planar shape normalisation.

- A procedural derivation mechanism which guarantees that all 2D shapes within layers have been created before adding volumetric detail.

2 Related Work

The seminal paper by Parish and Müller (2001) introduced L-Systems as a means to procedurally create road networks and buildings. However, due to building's spatial restrictions, Wonka et al. (2003) proposed the use of split and control grammars to procedurally create buildings. Eventually, due to existing limitations, Müller et al. (2006) introduce the CGA (Computer Generated Architecture) grammar which operates on geometry contained in scopes (oriented bounding boxes).

Krecklau et al. (2010) introduced the G^2 grammar (*Generalized Grammar*) which provides high descriptive power. The proposed system uses several types of non-terminals which can also be used as parameters in procedural rules.

The CGA grammar has been extended by Schwarz and Müller (2015) which present the CGA++ grammar, granting first-class citizenship to shapes, allowing access to information about created shapes during the grammar derivation process. Having such data allows to achieve more complex constructions.

Thaller et al. (2013) generalise the scope concept to convex polyhedra allowing more complex geometric operations, enhancing shape grammar expressiveness.

The approach presented by Leblanc et al. (2011) can generate complete buildings by using queries to select components (building parts) on which operations are performed and regions to create connections between components.

In these methods the procedural rules are defined using a text-based paradigm, which can be cumbersome for artists and content creators. Therefore, more interactive approaches have surfaced. Lipp et al. (2008), for example, introduce a visual frontend to define shape grammars interactively with direct visual editing. Kelly and Wonka (2011) allow the user to specify floor plans and extrusion profiles to generate complex architecture. Edelsbrunner et al. (2016) generate complete building shells with complex roof structures based on wall profiles. The models can be further detailed with shape grammars.

To overcome the fact that, in shape grammars, there is a disassociation between procedural rules, Patow (2012) presented a direct acyclic graph representation for shape grammars where edges connect distinct grammar rules. Another graph-based approach is the one presented by Silva et al. (2013), where users can encapsulate basic operations into semantically-rich and reusable components. More recently, Silva et al. (2015) presented a graph-based system that is able to integrate diverse types of content within the same visual language.

Zhang et al. (2013) presented a layered analysis of irregular façades which aims at retrieving a high-level understanding of façade structure by including depth layers which enables more compact and natural interpretations of building facades. The output is a hierarchical representation of a façade structure which can be used to generate variations. However, this method works only on the image domain and is intended for reconstruction operations.

Ruiz-Montiel et al. (2014) propose a Computer Assisted Conceptual Design system that includes layered shape grammar to aid users in the conceptualisation phase. While this system is targeted at generating plausible starting points to be further manually modelled by an artist, our methodology aims at generating final, detailed geometries for the outer building architecture. Moreover, the system presented by Ruiz-Montiel et al. (2014) does not provide an algorithm to merge the different layers.

More recently, Ilčík et al. (2015) proposed a layered based approach for the procedural design of façades. In their approach, layers are rectilinear grids of façade elements filled by two generator patterns. Despite the good results and the capability of generating intentional misalignments, the generated elements are based on rectangular shapes and complex geometries must be imported.

The system presented by Guerrero et al. (2015) is capable of learning shape placement in building façades. Given a set of examples, the system infers a probabilistic model of shape placement, with which variations can be generated with similar geometric relations. Although the system can generate complex, non-rectangular geometries and has extensions that allow volumetric modelling of buildings, there is no support for layers. Moreover, the user needs to provide the system with examples.

There is, to our knowledge, no system that employs the concept of layers in a shape grammar in the generation of detailed outer architecture of buildings while providing a way to vectorially define two-dimensional shapes that seamlessly integrate within the final geometry.

2.1 Shape Grammars

Shape grammars, firstly introduced by Stiny and Gips (1972), are production systems based on replacement rules. Starting from an axiom shape, the system iteratively applies rules, replacing one shape with a different subset of shapes.

In the CGA grammar Müller et al. (2006) shapes are identified by symbols and their geometry is contained within an oriented bounding box named scope. Shapes also contain other data such as material information. Rules have a symbol on their left side, called the predecessor, and a set of operations and successor shapes on their right side. Typically, rules are defined as follows:

```
Predecessor ⇝ Successor1 Successor2 ...
```

The rules are applied to shapes whose symbol matches the predecessor and replace them with the sucessor shapes. Shapes can be called non-terminal if its symbol matches a symbol in the left side of a rule, and terminal otherwise.

The grammar execution iteratively creates a shape tree, corresponding to a particular configuration of shapes, by appending successor shapes as children of predecessor shapes. The derivation continues until all leaves are terminal shapes.

Originally, Müller et al. (2006), defined several operations on shapes, of which we will describe the most relevant to our work in the remaining of this section.

Split: the split rule allows to split one shape into several subshapes along one of its axis, by defining the size of each subshape and the successor symbol.

Repeat split: the repeat split operation tiles the predecessor shape, along an axis, with a set of subshapes by also defining the size of the successor shapes. It is, then, possible to scale the split operation to shapes of arbitrary sizes.

Component split: this operation splits the predecessor shape into its components. Its parameters are the type of components (i.e., faces, edges or vertices) and which selected component to further derive (e.g., side or top faces).

Extrusion: the extrusion operation takes each face of the predecessor shape and performs an extrusion along its normal by a specified amount. This is useful to create mass models from planar shapes (e.g., a building outline).

Inert asset: this operation inserts a previously modelled object into the predecessor shape's scope. It is used to introduce more complex geometry into the final model.

3 Overview

Our system is based on shape grammars, similar to Müller et al. (2006) and Krecklau et al. (2010). We introduce support for the creation of layers and the vectorial definition of shapes within layers. This allows the creation of more complex shapes within façades, and more of variations with less modelling effort.

This is achieved by considering several types of procedural items: *volumetric shapes*, *planar shapes*, *layers* and *2D shapes* (Subsection 4.1). Planar shapes contain two dimensional scenes where 2D shapes can be organised in layers and ordered by depth. In this sytem, the procedural operations are overloaded, meaning that their execution and output depends on the type of the input procedural items (Subsection 4.2).

Unlike described by Müller et al. (2006), where the procedural rule execution scheduling is controled by user defined priorities, our system first executes all rules that add two dimensional detail to planar shapes, using the operations described in Subsection 4.3.

Once all 2D shapes have been created, the system merges all layers and 2D shapes in the planar shapes, taking into account occlusions between 2D shapes, creating a set of disjoint shapes, which are then converted into new planar shapes in three dimensional space. This process is called *planar shape normalisation* and is described in Section 5.

Operations that lead to volumetric shapes can, then, be executed, which add 3D detail to the façades. This process is repeated until there are only terminal shapes.

4 Layered Shape Grammars

This section describes the extensions to shape grammars that allow the specification of layers and the vectorial definition of shapes. Specifically, we describe the available procedural item types, how rules in our system are overloaded and the introduced operations.

4.1 Procedural Items

Procedural rules in this technique operate on Procedural Items (PI) which are identified by a symbol and, as in other procedural modelling systems (Silva et al. (2015), Krecklau et al. (2010)), our methodology supports several types of procedural items. The set τ of all available types contains the *VolumetricShape*, *PlanarShape*, *Shape2D* and *Layer*.

VolumetricShape: volumetric shapes represent shapes in three dimensional space whose bounding box has a volume superior to zero. In other words, these are 3D shapes that may or may not represent a closed volume.

Each volumetric shape has its own frame of coordinates and size, which is named scope, similarily to the one defined by Müller et al. (2006).

PlanarShapes: in contrast, planar shapes, are shapes that have a single face contained in an arbitrary plane. As with volumetric shapes, planar shapes also have an associated scope. However, its size along the z axis is zero and its xy-plane is coplanar with the shape.

There is a 2D scene associated with each planar shape, whose x and y axes are mapped to the scope's x and y axes.

This scene consists of a stack of layers, initially containing one layer with a 2D shape (the background shape), created by mapping the planar shape's 3D geometry into the 2D scene.

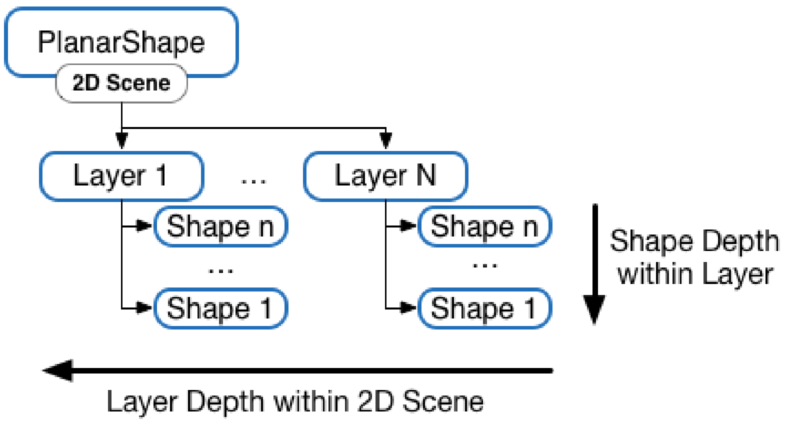


Figure 1: Relations between planar shapes, layers and 2D shapes.

Layers: layers are stacked by their depth within a 2D scene, where the bottom-most layer represents the deepest layer in a 2D scene. Each layer contains a list of 2D shapes sorted by depth. Figure 1 illustrates the relations between planar shapes, layers and 2D shapes.

Layers are considered procedural items since they are created and manipulated by procedural rules.

Shape2D: Finally, a 2D shape is a bidimensional shape contained in a layer of a 2D scene. All shapes are clipped inside another shape (except for the background shape), which must be of a higher depth, and are possibly occluded by shapes of lower depth. The clipping and occlusion process is described in Section 5.

These procedural items have two boolean attributes that can be set independently. These are *placeholder* and the *fixed geometry* flags.

All 2D shapes are created as placeholder shapes, which means that they are derived normally but they will not be transformed into planar shapes during the planar shape normalisation (Section 5). Before the normalisation process, 2D shapes are automatically marked as non-placeholders if their derivation requires a planar shape.

For example, if the next operation to be applied to a 2D shape is an extrusion, then the shape is marked as non-placeholder. The shape is, consequently, converted into a planar shape which can be correctly extruded.

The *fixed geometry* flag is used to control how the occlusion process affects each shape. During occlusion calculations (Subsection 5.1) if a planar shape with fixed geometry is occluded, then its geometry is discarded. If the flag is not set, the final geometry is calculated by subtracting the occluding geometry from its original geometry.

4.2 Overloaded Procedural Rules

The rules in our system are similar to those in regular shape grammars in the sense that they define the replacement of a predecessor shape with a set of successor shapes through the sequential execution of a list of shape operations, $RuleOps$.

However, because our system supports several types of procedural items, the operations are overloaded. This means that the same operation can have a different execution and can output procedural items of different types, depending on the type of input, while maintaining the operation semantics. Furthermore, it can restrict the application of an operation to some types.

The *split* operation, for example, can be applied to volumetric and planar shapes, outputting volumetric or planar shapes respectively. The *extrude* operation always outputs volumetric shapes, although it accepts volumetric and planar shapes. The *create layers* operation (Subsection 4.3.1) only accepts planar shapes and outputs layers.

More formally, an operation O is a function $O : t_{in} \in \tau \rightarrow t_{out} \in \tau$, that acts on procedural items of type t_{in} and outputs zero or more procedural items of type t_{out}. Each operation has an associated type mapping $\omega_O : \tau \mapsto \tau$ that describes the type transformations the operation performs. The output type of an operation for input procedural item with type t is given by $\omega_O(t)$. If t is not in the domain of ω_O, then $\omega_O(t) = \varnothing$. Likewise, $\omega_O(\varnothing) = \varnothing$.

The output type of a procedural rule R given an input procedural item of type t is, then, given by the operator $out\_type(R, t)$ defined by the following algorithm:

```
function $out\_type(rule, item\_type)$:
  current_type $\leftarrow$ item_type
  for(op in rule.RuleOps)
    current_type = out_type(op, current_type)
  return current_type
```

Before applying a rule to a procedural item, the system runs the above algorithm and checks its result. A return value different from $\varnothing$ indicates that the rule can be executed with the given procedural item, since each operation is able to manipulate the results of the operation executed before. Otherwise, an error is raised and the execution is halted.

4.3 Procedural Operations

In addition to the operators introduced in Müller et al. (2006) and briefly described in 2.1 we have defined a new set of operations that act on procedural items. In the following we describe their operation and define their syntax, loosely following the syntax in Müller et al. (2006).

4.3.1 Layer Creation Operation

The layer creation operation allows to create layers in planar shapes in decreasing order of its depth. Successor rules can add further detail to each layer separately.

For example, the following example shows the creation of two layers: one that will contain the façade's floors description and another defining the entrance.

```
Fac $\rightsquigarrow$ layers("floors","door"){Floors | Door}
```

The n^{th} parameter defines the name of the n^{th} layer to be created in the planar shape. For each layer, a rule name must be given between the { and } delimiters, separated by a | character. Although it seems redundant, having a layer name and a rule name adds a level of semantics to this operation. The user can, for example, swap the rule for the *door* layer with another one, while the semantics is still retained. Furthermore, it allows to externally turn on or off the generation of the layer via its composite name: *Fac.door*, increasing the variation of results without modifying the rulebase.

4.3.2 Segment Operation

The segment operation is used to partition a planar shape, 2D shape or layer into 2D shapes along one of its axis (x or y), placing the resulting shapes into one of the 2D scene's layers, thus defining its depth within the scene as $d = depth_{pred} - 1$ where $depth_{pred}$ is the predecessor procedural item's depth. If the predecessor is a 2D shape, then $depth_{pred}$ is that shape's depth, and the resulting shapes are placed within the same layer. Whenever the predecessor is a planar shape, $depth_{pred}$ is the 2D scene's background shape's depth and the shapes are inserted in the deepest layer. If, however, the predecessor is a layer, then the shapes are included in that layer and $depth_{pred}$ takes the value of the minimum depth of the 2D shapes in the layer underneath or, if the predecessor is the bottommost layer, the background shape's depth.

This operation is also responsible for creating a clipping tree of a 2D scene's shapes, where nodes represent the 2D shapes and edges represent a clipping relation such that the geometry of children shapes must be contained inside the parent shape. The clipping tree root node is the 2D scene background shape. When the predecessor procedural item is a planar shape or a layer, then the shapes output by the segment operation become children of the background shape. On the other hand, if the predecessor is a 2D shape, then the shapes are clipped by such shape.

By default, this operation creates only placeholder shapes, meaning that they will not be converted into planar shapes unless their derivation leads to a procedural item of a type other than *Shape2D*. This keeps the background geometry intact for as long as possible allowing the correct and seamless integration of complex geometry into planar shapes and 2D shape occlusion during the planar shape normalisation step (Section 5).

As with the operations described in Section 2.1, there is also a repeat segment operation. We indicate this by appending an asterisk character to the operation body (after the last } character). Moreover, segment sizes can also be defined as relative to the predecessor size (by prepending the value with a ' character) or as a float size (by using a ~before the value) proportionally distributing the remaining size by all float segments. This is also present in the CGA grammar.

Consider the following listing that further details the previous example.

```
Floors ⇝ segment("Y", 3,2,2)
  { ϵ | Floor2 | ϵ }
Floor2 ⇝ segment("X", 3,2,3)
  { ϵ | Balcony | ϵ }
Balcony ⇝ extrude(0.1)
```

This results in a façade with a single square extrusion centered in the middle floor. Note that the segment operation syntax is similar to the split operation described before

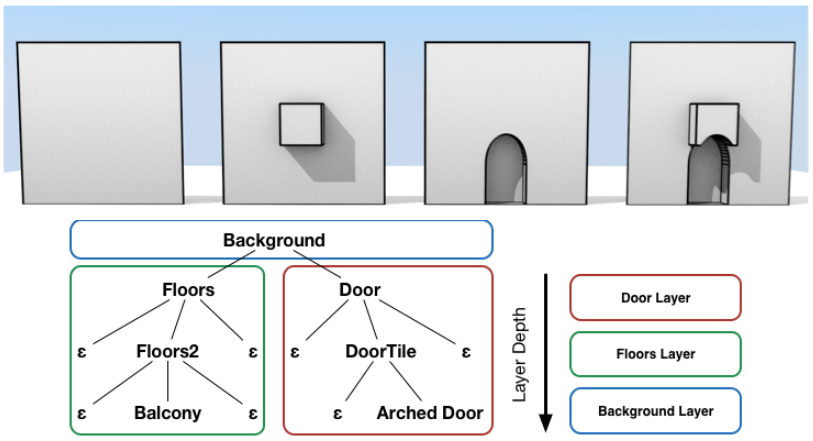


Figure 2: Various stages of a layered shape grammar execution demonstrating layers and vectorial shapes (top) and the final clipping tree (bottom).

and, as in Müller et al. (2006), ϵ is the empty shape. No practical consequences result from including empty shapes since only non-empty, non-placeholder 2D shapes have influence on the final geometry (see Section 5 for more details). In this scenario, only the *Balcony* shape results in modifications to the final model since its derivation leads to a volumetric shape via the extrude operation and the empty shapes are used as padding to correctly place the balcony. The results are illustrated in the center-left image in Fig. 2.

4.3.3 Vectorial Shape Operation

The vectorial shape operation is used to create complex 2D shapes within other shapes, taking as input the SVG specification of a vectorial closed path, allowing to specify complex architectural elements such as arched windows or doors. The vectorial definition is, afterwards, sampled down to a polygon P using an user specified sampling factor, allowing to control the level of detail. The sampled points $p_i(x, y) \in P$, where $x, y \in [0, 1]$ are used to create a polygonal 2D shape by interpolating them inside the predecessor shape's bounding rectangle, thus the vectorial shape aligns with any predecessor shape's bounding box.

The following example illustrates the creation of an arched door in the previous examples.

```
Door ⇝ segment("X", 3,2,3)
  { ϵ | DoorTile | ϵ }
DoorTile ⇝ segment("Y", 4,~1)
  { ArchedDoor | ϵ }
ArchedDoor ⇝
  vectorial("M0,0 L0,0.5 C0,1 1,1 1,0.5 L1,0Z")
  extrude(-0.1)
```

The first argument to the vectorial operation is the SVG path specification and a second, optional, argument would be the sampling factor indicating how many points are to be sampled per path segment (in our implementation and examples, this factor defaults to 30).

As depicted in the center-right image in Fig. 2, the window is no longer generated since, by default all 2D shapes are created having fixed geometry meaning that in case of being occluded by a shape with a smaller depth, they are discarded. Because the door is generated in a less deep layer, it takes precedence.

4.3.4 Set Property Operation

Similar to other procedural modelling frameworks, our technique provides an operation to set properties on procedural items. This is specially useful, in our case, to define 2D shapes with non fixed geometry.

The following example sets the *fixed_geometry* property to false in the *Balcony* rule, causing the balcony's geometry to be modified adapting to the arched door (see right image in Figure 2).

```
Balcony ⇝ set_property
  ("fixed_geometry", false)
  extrude(0.1)
```

5 Planar Shape Normalisation

The normalisation of a planar shape is a process that has the goal of converting the 2D scene associated with each planar shape to a set of planar shapes. This way the procedural derivation can continue to add volumetric detail.

In this system, the normalisation process is triggered by processing a special non-terminal procedural item, whose type is *NormalisationToken*, during the derivation process. This procedural item is automatically created as soon as a 2D shape is created within a planar shape and the system guarantees that there is at most one present at a time, which is discarded after the normalisation process. All 2D shapes are created before normalising the planar shapes (see Section 6).

The process consists of two stages for each planar shape. The first is to merge all layers in the 2D scene, calculating occlusions between shapes. This process can, usually, be found in 2D graphics software. The second stage is to convert these 2D shapes to planar shapes in three dimensional space.

More formally, the normalisation process is a function $normalise : PlanarShape \rightarrow \langle Bg_{shape}, Front_{shapes} \rangle$ which takes a PlanarShape p, along with its 2D scene and shapes, and transforms it into a a disjoint set of shapes such that $p \equiv Bg_{shape} \cup Front_{shapes}$. The Bg_{shape} represents the converted background shape in the 2D scene while $Front_{shapes}$ is a set containing the remaining planar shapes.

After the normalisation process, the background shape is marked as a terminal shape, while the remaining shapes are considered non-terminals.

5.1 Layers Merge

The first step in the process of merging all layers is to sort all 2D shapes by increasing order of depth, taking into account the shape's depth within a layer and the relative position of such layer. This results in a list L where the last element is the background shape.

All non-terminal 2D shapes are set as non-placeholder shapes if their following derivation results in procedural items other than *shape2D*. This guarantees that only shapes that will be derived further are converted into planar shapes.

The clipping tree is traversed and the 2D shape, s, of each node is clipped inside its parent shape p, such that its resulting shape s' is computed as $s' = s \cap p$.

The final 2D geometry of each shape is, then, calculated by iterating the list of shapes, performing occlusion calculations with a mask shape M, while considering the *placeholder* and *fixed geometry* flags. The final shape is calculated as $s'' = \mathit{occlude}(s', M)$, where *occlude* is a function described in the following listing. The mask shape is constructed by accumulating the occlusion results.

```
function occlude(shape, $M$):
  if(shape.placeholder):
    return $\varnothing$
  else if($shape \cap M = \varnothing$):
    return shape.geometry
  else if(shape.fixed_geometry):
    return $\varnothing$
  else:
    return $shape.geometry \setminus M$
```

Placeholder shapes do not have any impact on the final model and, so, its geometry is discarded. If the shape does not intersect the mask then there is no occlusion and the shape's geometry is not modified. If there is an occlusion the geometry is discarded if the shape is marked as fixed geometry. On the other hand, the geometry of shapes not marked as fixed geometry is calculated by subtracting the mask from its original geometry.

The clipping and occlusion process can, eventually, lead to shapes being split into two or more shapes. In these situations, the resulting shapes are considered separate shapes, and will lead to different planar shapes with the same symbol.

5.2 Conversion to Planar Shapes

The conversion of a 2D scene associated with a planar shape P into a set of new planar shapes, is achieved by noting that the scene is actually embedded in the plane defined by the x and y axis of P's scope.

There is, thus, a trivial mapping $\rho_P : \mathbb{R}^2 \mapsto \mathbb{R}^3$ that transforms points in a 2D scene in a planar shape P into points in the 3D scene.

For each 2D shape in a planar shape P, the system converts its 2D polygon, with ρ_P, into a 3D polygon, which is then used to create a new planar shape. Its scope is defined by translating P's scope to the point nearest to its origin and its size is adjusted to the new polygon. Finally, the 2D shape's symbol is copied to the new planar shape.

The planar shape can now be further detailed through the normal derivation process.

5.3 Deferred Procedural Items

To achieve a greater control of when the normalisation process takes place, it is possible to circumvent the normal rule execution scheduling (Section 6) and defer the derivation of a procedural item until a normalisation occurs. These items are, thus, called deferred procedural items.

This can be encoded in the grammar by prepending the @ character before a successor symbol. After the normalisation process occurs, all deferred items are unmarked as such and their derivation can continue as normal.

Deferring a procedural item is particularly useful when segment operation sizes must consider the final size of a planar shape, instead of its size before normalisation. The following grammar generates the example in Fig. 3a. Omitting the @ character before the *Windows* successor (second line), results in the model shown in Fig. 3b.

```
Start -> layers("windows", "frames")
  { @Windows |  Frames }
Frames -> segment(x,1,~1,1)
  { extrude(0.1) | ϵ | extrude(0.1) }
Windows -> segment(x,0.5,2,0.5)
  { ϵ | RepeatV | ϵ }*
RepeatV -> segment(y,~1,2,~1)
  { ϵ | extrude(-0.1) | ϵ }
```

6 Derivation Process

The derivation process starts with an arbitrary configuration of procedural items. These are, necessarily, planar or volumetric shapes, since they have a three dimensional representation.

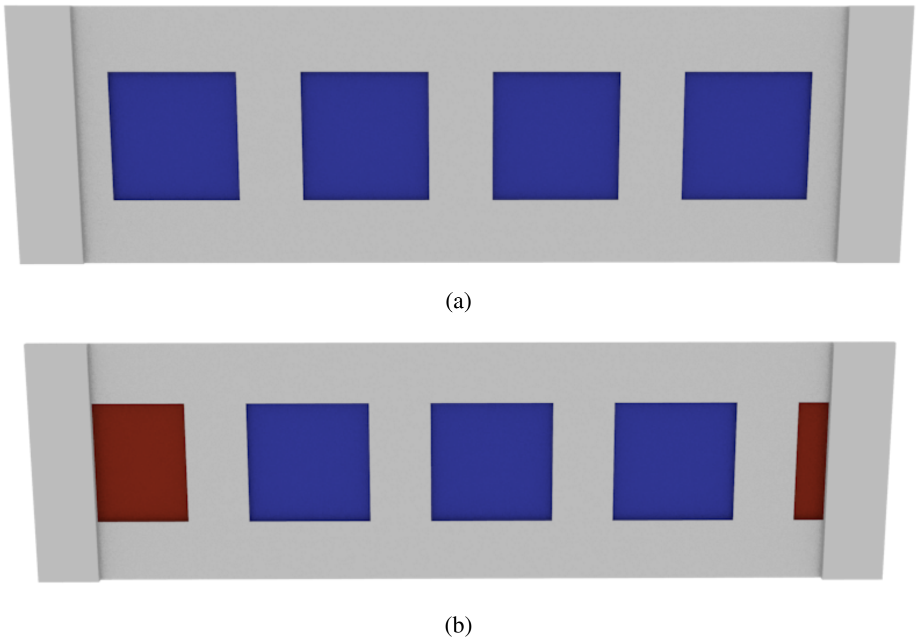
The procedural items are, then, inserted into a priority queue *NT* of non terminals, which sorts the items decreasingly using the *priority* function, where $future\_type(item)$ indicates the type of the resulting items after *item* has been derived.

$$priority(item) = \begin{cases} 3 & \text{if } future\_type(item) = Shape2D \\ 2 & \text{if } item.type = NormalisationToken \\ 1 & \text{if } item.deferred \\ 0 & \text{if } otherwise \end{cases}$$

Therefore, procedural items whose $future\_type$ equals *Shape2D* have a higher priority, followed by the *normalisation token*, the deferred shapes and finally the remaining procedural items. Therefore, the system creates all 2D shapes possible, ensuring that 2D scenes are fully detailed and the normalisation process has all available clipping and occlusion data to generate correct results. The generation of geometry evolves naturally from less detailed to more detailed models.

The derivation algorithm is detailed in the following listing:

```
function derivation(NT):
  while(NT not empty)
```


(a)

(b)

Figure 3: Deferring the Windows rule results in the correct window placement (a), when compared to (b).

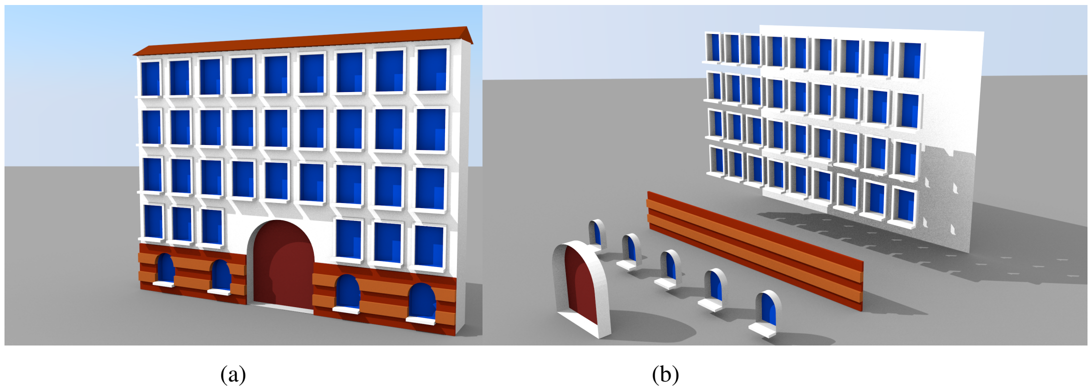

(a) (b)

Figure 4: The example façade (Subsection 7.1), on the left, and its decomposition into layers, on the right.

```
 3      S ← NT.pop()
 4      rule ← fetch_rule(S.symbol)
 5      if(out_type(rule ,S.type) = ∅)
 6         error_and_exit()
 7      newItems ← rule.execute(S)
 8      for(item in newItems)
 9         if(non_terminal(item))
10            NT.push(item)
11            if(should_generate_token(item))
12               NT.push(new normalisationToken)
```

The third line retrieves and removes the next non-terminal procedural item S from the priority queue NT and, afterwards the procedural rule that matches the item's symbol is fetched (line 5). Lines 6 through 8 check if the rule can be executed with the given item type, in which case, the rule is executed resulting in a new list of items (line 9). This list is iterated and the algorithm checks if each item is a terminal or not, by querying the rulebase for a matching symbol on the left side of a rule. If an item is non-terminal, it is inserted into the NT for further processing (line 14). A normalisation token is created as soon as a procedural item whose type is *Shape2D* is generated and there is no normalisation token present in NT (lines 15 through 17).

Including the normalisation token as a non terminal procedural item, guarantees that the planar shape normalisation process occurs in the right moment, while keeping the derivation process simple and similar to other approaches (e.g., Müller et al. (2006)).

7 Results and Discussion

In this section we provide a set of examples modelled with our system and a comparison to split-based methods.

7.1 Simple Façade Example

This example shows a simple façade containing several layers and shapes created with a vectorial definition. Note that the model is completely generated with procedural methods and does not include any pre-modelled asset, representing an improvement in shape grammar expressiveness.

The *Facade* rule creates two layers. As implicit, the *Floors* layer will contain the ground and top floors definition, while the *Entrance* layer will contain the arched door. The separation into two layers, allows the door to span several floors while the normalisation process guarantees that there are no geometry conflicts with the windows. Also, the door placement is completely independent from the floors and windows, enabling more variations by simply adjusting a few parameters.

The *GroundFloor* rule also creates two layers: one for the horizontal stripes and one for the arched windows. Note the seamless integration of the arched shapes with the stripes, resulting in more complex shapes.

```
Facade ⇝ layers("Floors","Entrance")
  { FloorsSplit | Entrance2 };
```

```
FloorsSplit ⇝ segment(y,2,~1)
  { GroundFloor | TopFloors };
TopFloors ⇝ segment(y,2)
  { Floor }*;
Floor ⇝ segment(x,1.5)
  { TileSegment }*;
TileSegment ⇝ segment(x,'0.1,~1,'0.1)
  { ϵ | TileV | ϵ };
TileV ⇝ segment(y,'0.1,~1,'0.1)
  { ϵ | WindowTile | ϵ };
GroundFloor ⇝ layers("stripes","windows")
  { Stripes | GroundWindows };
GroundWindows ⇝ segment(x,0.9,1,0.9)
  { ϵ | GroundWindowTile | ϵ }*;
GroundWindowTile ⇝ segment(y,0.35,0.1,~1,0.1)
  { ϵ | extrude(0.35) | ArchedWindow| ϵ };
ArchedWindow ⇝
  vectorial("M0,0 L0,0.5 C0,1 1,1 1,0.5 L1,0Z")
  extrude(-0.3);
Entrance ⇝ segment(x,~1,3,~1)
  { ϵ | EntranceTile | ϵ };
EntranceTile ⇝ segment(y,3.5,~1)
  { ArchedDoor | ϵ };
ArchedDoor ⇝
  vectorial("M0,0 L0,0.5 C0,1 1,1 1,0.5 L1,0Z")
  extrude(-0.5);
Stripes ⇝ segment(y,'0.2,'0.2')
  { extrude(0.1) | extrude(0.2) }*;
```

Some of the rules were ommited for brevity. The resulting façade can be seen in Fig 4a and its decomposition into layers can be seen in Fig 4b. Note that, although the vectorial definition for the door and the windows is the same, the generated shapes are of different size since the polygon is interpolated inside the predecessor's procedural item bounding rectangle.

7.2 Comparison to Split Based Methods

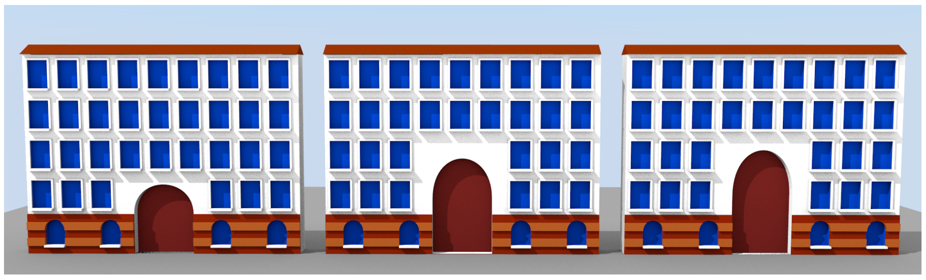
To generate the previous example using split based methods only, several modifications were introduced and are present in the following listing.

```
Facade ⇝ split(y,4,~1)
  { GroundFloor | TopFloors }
TopFloors ⇝ split(y, 2)
  { Floor }*
Floor ⇝ split(x,1.5)
  { 1.5: Tile }*
Tile ⇝ split(x,'0.1,~1,'0.1)
  { ϵ | TileV | ϵ }
TileV ⇝ split(y,'0.1,~1,'0.1)
  { ϵ | WindowTile | ϵ }
```

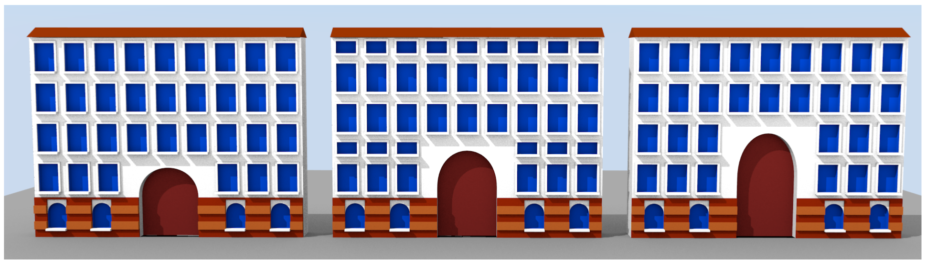
```
GroundFloor ⇝ split(x,4.5,4.5,4.5)
  { MFSide | EntranceTile | MFSide }
MFSide ⇝ split(y,2,2)
  { GroundFloorSide | TopFloors }
GroundFloorSide ⇝ split(x,~1,~1,~1)
  { Stripes | GroundWindowTile | Stripes }*
GroundWindowTile ⇝ segment(y,0.35,~1,0.1)
  { Stripe1 | WindowAsset | Stripe1 }
EntranceTile ⇝ split(x,~1,3,~1)
  { EStripes |  DoorAsset | EStripes }
EStripes ⇝ split(y,2,~1)
  { Stripes | ϵ }
```

Split operations now have to, simultaneously, account for the entrance, floors and horizontal stripes. Layers, on the other hand, treat these elements separately, which provides a greater amount of variability with fewer modifications to the rulebase, while still providing correct results.

For example in our system, modifying the entrance height in the façade can be achieved by tuning the segment operation parameters in the *EntranceTile* rule in the previous example. In the split based method, this height is, already, implicitly defined in the the *Facade* rule. Modifying the split parameters leads to unwanted repercussions in other parts of the façade. This is illustrated in Fig. 5, where the images show an increasing entrance height and, while our methodology (Fig. 5a) keeps the windows with a constant size, the split-based method deforms two of the façade floors (Fig. 5b).



(a)



(b)

Figure 5: Changing the entrance size using the layered approach, 5a, leads to better results than with the split based approach, 5b.

Another consequence of the layered approach, is that shape placement is not constrained by split lines and, as such, intentional misalignments of shapes are easily achieved. In Fig. 6 we see the same façade furthed detailed with a two-storey balcony in an aditional layer which is not aligned with the windows regular grid. This example also shows that complex modifications can be introduced with new layers. On the other hand, with traditional shape grammars several procedural rules would have to be rewritten.

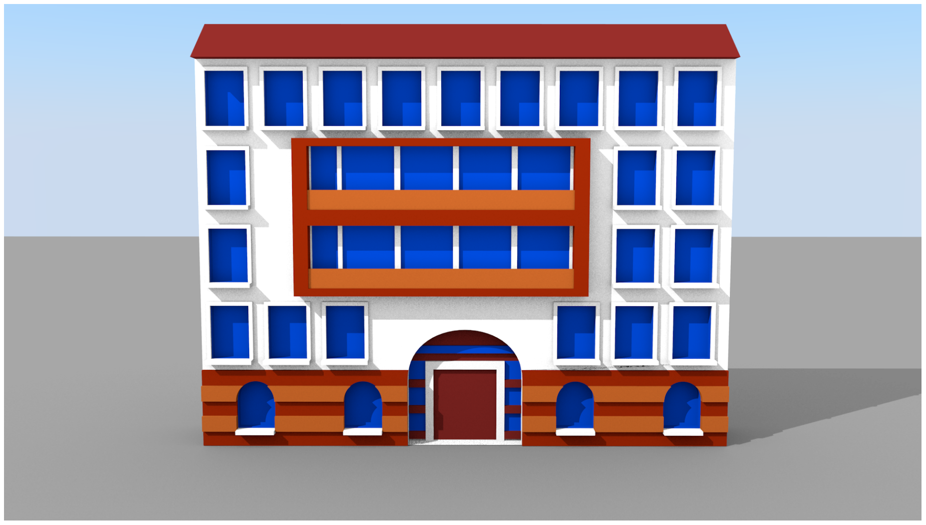


Figure 6: Intentionally misaligned balconies in the example façade.

Moreover, in split-based methods, the arched elements have to be imported as pre-modelled assets and, as seen in the ground floor windows (Fig. 7), can be difficult to integrate. By using a vectorial definition, the planar shape normalisation process allows the seamless composition of these windows and the horizontal stripes.

8 Limitations and Future Work

Although layered shape grammars allow a greater flexibility in placing geometry within planar shapes, our system currently does not offer any explicit way to constrain alignments between elements in different layers. The current solution is to parameterise the rules to encode these alignments. However, the normalisation process provides a synchronization point during the procedural derivation where 2D shapes can be modified (e.g., translated or resized). This would allow the system to perform alignments with procedural items already created.

The manual input of a SVG path to vectorially define shapes is an error prone task. However, this can be alleviated in a number of ways. One is to externally define and import these paths, resembling an asset import operation. A more elaborate solution is to provide the designer with tools to sketch the shapes over the models in an interactive procedural modelling application.

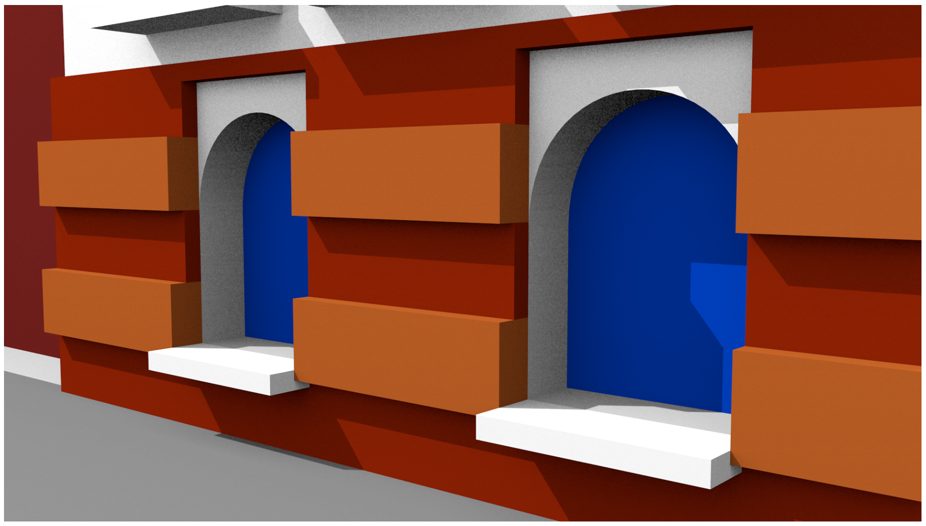


Figure 7: Detail of the ground windows in the split-based approach.

As with any shape grammar approach to the procedural modelling of buildings, our methodology suffers from the same drawbacks. For instance, these are text-based which is impractical for artists (Patow (2012)) and have limitations in its readability and manageability (Silva et al. (2015)). The grammar could be translated to a graph-based approach.

9 Conclusions

In the procedural modelling of buildings, most current methodologies rely on a split-based paradigm, imposing too strict limitations in the generation of geometry.

This paper presented an extension to shape grammars that allows the specification of layers within planar surfaces of buildings. This permits more complex layouts that go beyond regular grid structures. Moreover, this concept can, easily, provide a non-linear design workflow and produce more variations with less effort.

In most procedural modelling methods, complex geometry must be imported as a pre-modelled assets. We take a step further in procedural expressiveness by allowing the vectorial definition of 2D shapes within layers.

Results show that non-trivial variations of procedural models using our approach can be achieved with fewer modifications to the rule base. The integration of curved shapes (e.g., arched windows and doors) with other architectural elements is also achievable with our method.

Acknowledgements

Funded by the Portuguese government by FCT throuhg the PhD scholarship SFRH/BD/92019/2012 and Project “NORTE-01-0145-FEDER-000020”, funded by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement and through the European Regional Development Fund (ERDF).

References

Yoav I H Parish and Pascal Müller. Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques - SIGGRAPH ’01*, SIGGRAPH ’01, pages 301–308, New York, 2001. ACM Press.

Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. Instant architecture. *ACM Trans. on Graphics*, 22(3):669–677, 2003.

Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. *ACM Transactions on Graphics*, 25(3):614–623, jul 2006.

Lars Krecklau, Darko Pavic, and Leif Kobbelt. Generalized Use of Non-Terminal Symbols for Procedural Modeling. *Computer Graphics Forum*, 29(8):2291–2303, 2010.

Gustavo Patow. User-Friendly Graph Editing for Procedural Modeling of Buildings. *IEEE Computer Graphics and Applications*, 32(2):66–75, mar 2012.

PB Silva, Pascal Müller, Rafael Bidarra, and A Coelho. Node-Based Shape Grammar Representation and Editing. In *Proceedings of PCG 2013 - Workshop on Procedural Content Generation for Games, co-located with the Eigth International Conference on the Foundations of Digital Games*, 2013.

Pedro Brandão Silva, Elmar Eisemann, Rafael Bidarra, and António Coelho. Procedural Content Graphs for Urban Modeling. *International Journal of Computer Games Technology*, 2015:1–15, 2015.

Hao Zhang, Kai Xu, Wei Jiang, Jinjie Lin, Daniel Cohen-Or, and Baoquan Chen. Layered analysis of irregular facades via symmetry maximization. *ACM Trans. Graph.*, 32(4):1, 2013.

Michael Schwarz and Pascal Müller. Advanced procedural modeling of architecture. *ACM Transactions on Graphics*, 34(4):107:1–107:12, 2015. ISSN 07300301. doi: 10.1145/2766956.

Wolfgang Thaller, Ulrich Krispel, René Zmugg, Sven Havemann, and Dieter W. Fellner. Shape grammars on convex polyhedra. *Computers and Graphics*, 37(6):707–717, 2013.

Luc Leblanc, Jocelyn Houle, and Pierre Poulin. Component-based Modeling of Complete Buildings. In *Proceedings of Graphics Interface 2011*, pages 87–94, 2011.

Markus Lipp, Peter Wonka, and Michael Wimmer. Interactive visual editing of grammars for procedural architecture. *ACM Transactions on Graphics*, 27(3):102:1—-102:10, 2008.

Tom Kelly and Peter Wonka. Interactive architectural modeling with procedural extrusions. *ACM Transactions on Graphics*, 30(2):14:1—-14:15, apr 2011.

Johannes Edelsbrunner, Ulrich Krispel, Sven Havemann, Alexei Sourin, and Dieter W. Fellner. Constructive Roofs from Solid Building Primitives. *Trans. on Computer Science XXVI*, 1:17–40, 2016.

Manuela Ruiz-Montiel, María Victoria Belmonte, Javier Boned, Lawrence Mandow, Eva Millán, Ana Reyes Badillo, and José Luis Pérez-De-La-Cruz. Layered shape grammars. *CAD Computer Aided Design*, 56:104–119, 2014.

Martin Ilčík, Przemyslaw Musialski, Thomas Auzinger, and Michael Wimmer. Layer-Based Procedural Design of Façades. *Computer Graphics Forum*, 34(2):205–216, may 2015.

Paul Guerrero, Stefan Jeschke, Michael Wimmer, and Peter Wonka. Learning Shape Placements by Example. *ACM Transactions on Graphics*, 34(4):108:1—-108:13, aug 2015.

George Stiny and James Gips. Shape grammars and the generative specification of painting and sculpture. In *Information Processing 71 Proceedings of the IFIP Congress 1971. Volume 2*, volume 71, pages 1460 – 5, Amsterdam, Netherlands, 1972.