Procedural Modeling of Buildings

Pascal Müller* ETH Zürich
Peter Wonka† Arizona State University
Simon Haegler* ETH Zürich
Andreas Ulmer* ETH Zürich
Luc Van Gool* ETH Zürich / K.U. Leuven

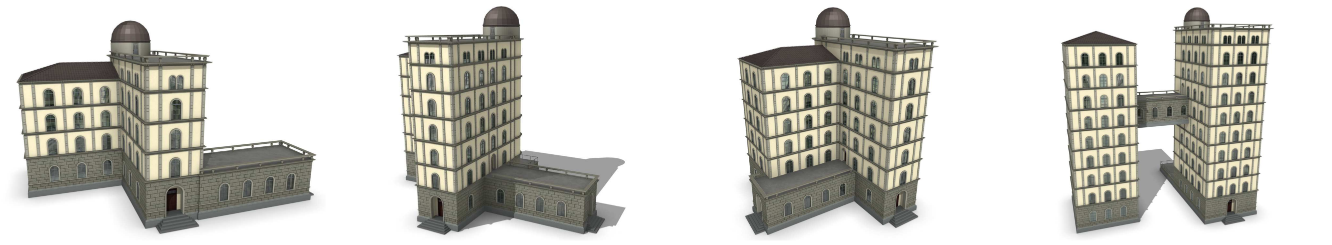


Figure 1: This figure shows the application of *CGA shape*, a novel shape grammar for the procedural modeling of computer graphics architecture. First, the grammar generates procedural variations of the building mass model using volumetric shapes and then proceeds to create façade detail consistent with the mass model. Context sensitive rules ensure that entities like windows or doors do not intersect with other walls, that doors give out on terraces or the street level, that terraces are bounded by railings, etc.

Abstract

CGA shape, a novel shape grammar for the procedural modeling of CG architecture, produces building shells with high visual quality and geometric detail. It produces extensive architectural models for computer games and movies, at low cost. Context sensitive shape rules allow the user to specify interactions between the entities of the hierarchical shape descriptions. Selected examples demonstrate solutions to previously unsolved modeling problems, especially to consistent mass modeling with volumetric shapes of arbitrary orientation. *CGA shape* is shown to efficiently generate massive urban models with unprecedented level of detail, with the virtual rebuilding of the archaeological site of Pompeii as a case in point.

CR Categories: F.4.2 [Mathematical Logic and Formal Languages]: Grammars and Other Rewriting Systems I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism I.6.3 [Simulation and Modeling]: Applications J.6 [Computer-Aided Engineering]: Computer-Aided Design (CAD)

Keywords: Procedural Modeling, Architecture, Chomsky Grammars, L-systems, Computer-Aided Design

1 Introduction

The creation of compelling models is a crucial task in the development of successful movies and computer games. However, modeling large three-dimensional environments, such as cities, is a very expensive process and can require several man years worth of labor. In this paper we will employ procedural modeling using shape grammars capable of efficiently creating large cities with high geometric detail and up to a billion polygons. It would be extremely time consuming to replicate these results with existing modeling software.

We use a shape grammar (called *CGA shape*) with production rules that iteratively evolve a design by creating more and more details. In the context of buildings, the production rules first create a crude volumetric model of a building, called the mass model, then continue to structure the façade and finally add details for windows, doors and ornaments. The main advantage of the method is that the creation of the hierarchical structure and the annotation of a model is specified in the modeling process. This semantic information is important for reusing design rules for procedural variations (see figure 1) and thereby creating a large variety of architecture populating a whole city.

The idea of modeling urban environments using shape grammars was recently explored by Parish and Müller [2001] and Wonka et al. [2003]: On the one hand, Parish and Müller showed how to generate *large* urban environments where each building consists of simple mass models and shaders for façade detail. On the other hand, Wonka et al. [2003] demonstrated how to generate *geometric details* on façades of individual buildings. Ideally, we would like to combine these two ideas to generate large and detailed urban environments. However, there is a significant challenge in the context of mass modeling that needs to be addressed and requires extensive changes to both models. (1) Parish and Müller could generate simple models by adding translated and rotated boxes and details were added with a shader. This strategy cannot generate sufficient geometric detail and there will be numerous unwanted intersections of architectural elements (see figure 2). (2) The split rules proposed by Wonka et al. are only sufficient for simple mass models. Complex mass models will require an excessive amount of splits. Further, the mass model cannot be easily changed because novel configurations will need additional production rules and objects of arbitrary orientation cannot be handled easily.

We present a grammar-based solution to generate detailed building shells stemming from complex mass models. Our approach is based on a new model for context sensitive shape rules that is suitable for computer graphics architecture. The major contributions of this paper are as follows:

- We are the first to introduce a procedural approach to model detailed buildings with consistent mass models. The buildings are not restricted to axis aligned shapes and include roof surfaces and rotated shapes. This also allows us to amplify

*e-mail: {pmueller|shaegler|ulmeran|vangool}@vision.ee.ethz.ch
†e-mail: peter.wonka@asu.edu

given mass models such as GIS databases consisting of extruded two-dimensional building footprints.

- We are the first to address application related details in the context of procedural modeling of buildings, such as the definition of the most important shape rules, the concise notation, and modeling examples detailing various modeling strategies. Our results will show massive urban models with unprecedented level of geometric detail.

1.1 Related Work

Procedural modeling can draw from a large spectrum of production systems such as Semi-Thue processes [Davis et al. 1994], Chomsky grammars [Sipser 1996], graph grammars [Ehrig et al. 1999], shape grammars [Stiny 1975], and attributed grammars [Knuth 1968]. However, the mere specification of a production system is only the basis. Several questions, such as the geometric interpretation, concise notation, control of the derivation, and the design of actual models, still have to be addressed.

For the geometric modeling of plants, Prusinkiewicz and Lindenmayer showed that wonderful results can be achieved by using L-systems to generate strings that are interpreted with a LOGO-style turtle [Prusinkiewicz and Lindenmayer 1991]. L-systems have been extended to query the turtle position [Prusinkiewicz et al. 1994], to incorporate general computer simulation [Měch and Prusinkiewicz 1996], self-sensitivity [Parish and Müller 2001], and user generated curves [Prusinkiewicz et al. 2001].

In architecture, shape grammars [Stiny 1975; Stiny 1980] were successfully used for the construction and analysis of architectural design [Downing and Flemming 1981; Duarte 2002; Flemming 1987; Koning and Eizenberg 1981; Stiny and Mitchell 1978]. The original formulation of the shape grammar operates directly on an arrangement of labeled lines and points. However, the derivation is intrinsically complex and usually done manually, or by computer, with a human deciding on the rules to apply. Shape grammars can be simplified to set grammars [Stiny 1982; Wonka et al. 2003] to make them more amenable to computer implementation. Cellular textures [Legakis et al. 2001] can be used to compute brick patterns and generative mesh modeling can generate complex manifold surfaces from simpler ones [Havemann 2005].

While the framework defined by the grammar is one essential part of procedural modeling, it is then necessary to abstract rules that create architectural configurations. While a larger library is necessary for this task, we would recommend starting with books that emphasize structure of architecture, such as a visual dictionary [Ching 1996], “The Logic of Architecture” by Mitchell [1990], “Space Syntax” [Hillier 1996], Design patterns [Alexander et al. 1977], and studies of symmetry [March and Steadman 1974; Shubnikov and Koptsik 1974; Weyl 1952].

1.2 Overview

The paper is structured as follows: We will first explain the basic shape grammar in section 2. In section 3 we will introduce extensions that allow to model complex shape configurations and shape interactions. Selected examples in section 4, 5, and 6 will show the application of the grammar to several modeling problems on a small scale. The results in section 7 will show the extension to larger urban environments. In section 8 we discuss our contribution and advantages and disadvantages of the approach; conclusions are given in section 9.

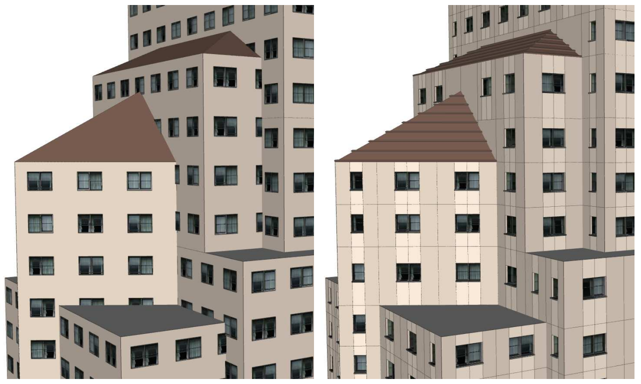


Figure 2: The motivation for our novel shape grammar. The modeled building consists of 14 volumetric primitives (cubes, roofs) placed by a stochastic shape grammar. Left: Existing methods of procedural architecture can either place shaders on the individual volumes or use split rules for procedural refinement. In both cases several unwanted intersections will cut windows (or other elements) in unnatural ways, as the volumes are not aware of each other. Right: Our approach allows the resolution of these conflicts. Additionally, we can place geometry on polygons of different orientation such as roof surfaces. This example was created using only 6 rules.

2 A Shape Grammar for CG Architecture

CGA Shape is an extension of set grammars, introduced by Wonka et al. [2003]. While the idea for the split rule was presented in previous work, the actual definition of the split including the repeat split and the scaling of rules are our contribution. Further we introduce a component split, the basis for modeling with one-, two-, and three-dimensional shapes. The notation of the grammar and general rules to add, scale, translate, and rotate shapes are inspired by L-systems [Prusinkiewicz and Lindenmayer 1991], but are extended for the modeling of architecture. While parallel grammars like L-systems are suited to capture *growth* over time, a sequential application of rules allows for the characterization of *structure* i.e. the spatial distribution of features and components [Prusinkiewicz et al. 2001]. Therefore, *CGA Shape* is a sequential grammar (similar to Chomsky grammars).

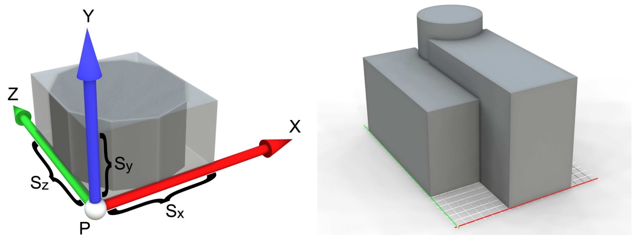


Figure 3: Left: The scope of a shape. The point P, together with the three axis X, Y, and Z and a size S define a box in space that contains the shape. Right: A simple building mass model composed of three shape primitives.

Shape: The grammar works with a configuration of shapes: a shape consists of a symbol (string), geometry (geometric attributes) and numeric attributes. Shapes are identified by their symbols which is either a terminal symbol $\in \Sigma$, or a non-terminal symbol $\in V$. The corresponding shapes are called terminal shapes and non-terminal shapes. The most important geometric attributes are the position P, three orthogonal vectors X, Y, and Z, describing a coordinate system, and a size vector S. These attributes define an oriented bounding box in space called *scope* (see figure 3).

Production process: A configuration is a finite set of basic shapes. The production process can start with an arbitrary configuration of shapes A, called the axiom, and proceeds as follows: (1) Select an active shape with symbol B in the set (2) choose a production rule with B on the left hand side to compute a successor for B, a new set of shapes $BNEW$ (3) mark the shape B as inactive and add the shapes $BNEW$ to the configuration and continue with step (1). When the configuration contains no more non-terminals, the production process terminates. Depending on the selection algorithm in step one, the derivation tree [Sipser 1996] can be explored either depth-first or breadth-first. However, both of these concepts do not allow enough control over the derivation. Therefore, we assign a priority to all rules according to the detail represented by the shape to obtain a (modified) breadth-first derivation: we simply select the shape with the rule of highest priority in step one. This strategy guarantees that the derivation proceeds from low detail to high detail in a controlled manner. Please note, that we do not delete shapes, but rather mark them as inactive, after they have been replaced. This enables us to query the shape hierarchy, instead of only the active configuration.

Notation: Production rules are defined in the following form:

id: predecessor : cond $\rightsquigarrow$ successor : prob

where *id* is a unique identifier for the rule, *predecessor* $\in V$ is a symbol identifying a shape that is to be replaced with *successor*, and *cond* is a guard (logical expression) that has to evaluate to true in order for the rule to be applied. The rule is selected with probability *prob*. For example, the rule

1: fac(h) : h > 9 $\rightsquigarrow$ floor(h/3) floor(h/3) floor(h/3)

replaces the shape fac with three shapes $floor$, if the parameter h is greater than 9. To specify the successor shapes we use different forms of rules explained in the remainder of this section.

Scope rules: Similar to L-systems we use general rules to modify shapes: $T(t_x,t_y,t_z)$ is a translation vector that is added to the scope position P, $Rx(angle)$, $Ry(angle)$, and $Rz(angle)$ rotate the respective axis of the coordinate system, and $S(s_x,s_y,s_z)$ sets the size of the scope. We use [and] to push and pop the current scope on a stack. Any non-terminal symbol $\in V$ in the rule will be created with the current scope. Similarly, the command $I(objId)$ adds an instance of a geometric primitive with identifier $objId$. Typical objects include a cube, a quad, and a cylinder, but any three-dimensional model can be used. The example below illustrates the design of the mass model depicted in figure 3 right:

1: A $\rightsquigarrow$ [T(0,0,6) S(8,10,18) I("cube")]
T(6,0,0) S(7,13,18) I("cube") T(0,0,16) S(8,15,8) I("cylinder")

Basic split rule: The basic split rule splits the current scope along one axis. For example, consider the rule to split the façade of figure 4 left into four floors and one ledge:

1: fac $\rightsquigarrow$ Subdiv("Y",3.5,0.3,3,3,3){ floor | ledge | floor | floor | floor }

The first parameter describes the split axis ("X", "Y", or "Z") and the remaining parameters describe the split sizes. Between the delimiter { and } a list of shapes is given, separated by |. We also use similar split rules to split along multiple axis ("XY", "XZ","YZ", or "XYZ"), nested splits, or nested combinations of splits and L-system rules.

Scaling of rules: From the previous example we can see the first challenge. The split is dimensioned to work well with a scope of size $y = 12.8$, but for other scopes the rule has to be scaled. From our experience not all architectural parts scale equally well, and it is important to have the possibility to distinguish between absolute values (values that do not scale) and relative values (values that do scale). Values are considered absolute by default and we will use the letter r to denote relative values, e.g.

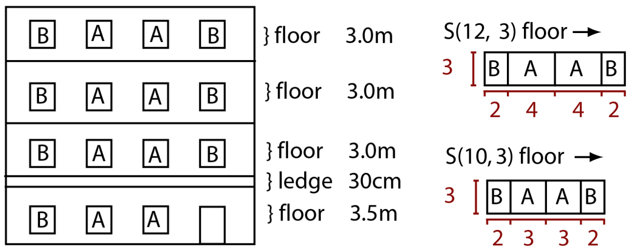


Figure 4: Left: A basic façade design. Right: A simple split that could be used for the top three floors.

1: floor $\rightsquigarrow$ Subdiv("X",2,1r,1r,2){ B | A | A | B }

where relative values r_i are substituted as $r_i * (Scope.sx - \sum abs_i)/\sum r_i$ ($Scope.sx$ represents the size of the x-length of the current scope). Figure 4 right illustrates the application of the rule above on two different sized floors (with x-length 12 and 10).

Repeat: To allow for larger scale changes in the split rules, we often want to tile a specified element. For example:

1: floor $\rightsquigarrow$ Repeat("X",2){ B }

The floor will be tiled into as many elements of type B along the x-axis of the scope as there is space. The number of repetitions is computed as $repetitions = \lceil Scope.sx/2 \rceil$ and we adjust the actual size of the element accordingly.

Component split: Up until this point all shapes (scopes) have been three-dimensional. The following command allows to split into shapes of lesser dimensions:

1: a $\rightsquigarrow$ Comp(type, param){ A | B | ... | Z }

Where *type* identifies the type of the component split with associated parameters *param* (if any). For example we write $Comp("faces")\{A\}$ to create a shape with symbol A for each face of the original three-dimensional shape. Similarly we use $Comp("edges")\{B\}$ and $Comp("vertices")\{C\}$ to split into edges and vertices respectively. To access only selected components we use commands such as $Comp("edge",3)\{A\}$ to create a shape A aligned with the third edge of the model or $Comp("sidefaces")\{B\}$ to access the side faces of a cube or polygonal cylinder. To encode shapes of lesser dimension we use scopes where one or multiple axis have zero size. To go back to higher dimensions we can simply use the size command S with a non-zero value in the corresponding dimension (e.g. to extrude a face shape along its normal and therefore transforming it into a volumetric shape).

3 Mass Modeling

The grammar explained in the previous section is powerful enough to specify complex shapes. The important remaining question is how to use them. We will first give an overview of how to generate mass models and then explain how to create façade and roof details. The proposed technique to solve the transition from mass modeling to façade and roof modeling is the key insight presented in this paper. We make use of two extensions that allow the specification of shape rules, the outcome of which depends on the spatial context.

3.1 Assembling Solids

Building mass models are most naturally constructed as a union of volumetric shapes [Le Corbusier 1985; Mitchell 1990]. The simplest construction uses a box as basic primitive. We can then generate simple mass models using scaling, translation or splits, such as the basic building blocks L, H, U and T described by Schmitt [1993] (see figure 5).

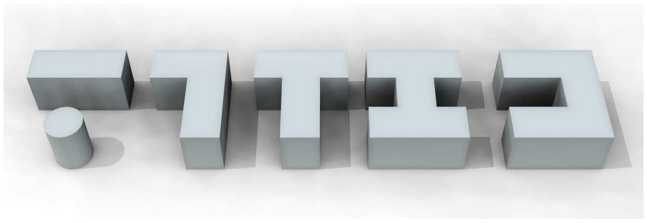


Figure 5: A basic shape vocabulary for mass modeling.

The next level of difficulty is to use arbitrary rotations of shapes and to include a cylinder in the shape vocabulary. A nice example using rotation are the Petronas Towers in Malaysia. The *CGA shape* reconstruction of one of the two identical towers is depicted in figure 6. The tower also shows the use of tapering prevalent in the construction of many skyscrapers. Such a construction can no longer be achieved by a split grammar alone.

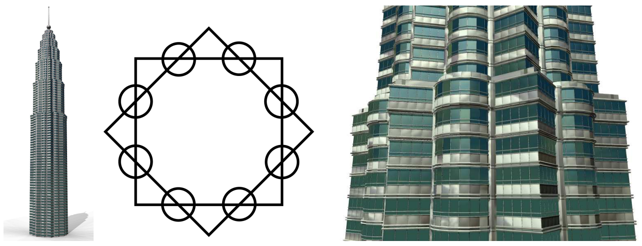


Figure 6: *CGA shape* reconstruction of a Petronas Tower. The mass model of the tower (left) and the footprint (middle) reveal the elementary assembling of cubes and cylinders. Right: The same façade rule has been applied onto the different types of solids.

Further, it becomes necessary to include basic roof shapes, such as the examples depicted in figure 7 and more general shapes, such as general L-shapes and extruded general quadrilaterals.

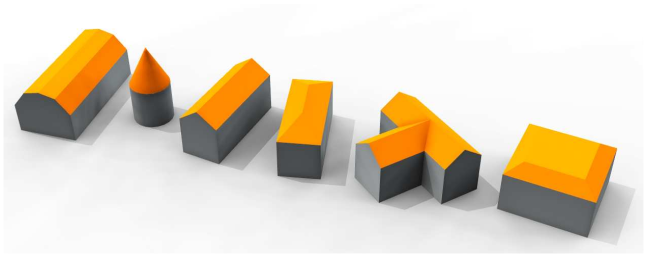


Figure 7: Selected roof types. From left to right: gambrel, cone, gabled, hipped, cross-gable, mansard.

Mass models can now be created in the following two fashions: (1) We are given a building lot as an axiom of the grammar. Then we are able to generate mass models, using scaling, translation, rotation, and split operations. Care has to be taken that the building mass does not protrude the parcel boundaries. (2) When importing data from a GIS database, or importing an existing architectural model, we are more restricted and can only make minor modifications to the mass model (if any). Currently we try to classify imported mass models as basic shapes already defined in our shape vocabulary. If this is not possible, we use a general extruded footprint together with a general roof obtained by a straight skeleton computation [Aichholzer et al. 1995; Eppstein and Erickson 1999] as shape primitives.

Problem of complex surfaces: This form of modeling is very intuitive and can create sufficiently complex building shapes. The question is how to proceed from here. One approach to solve this problem would be to compute the visible façade surfaces directly, but this would lead to the following complications: (1) the visible surfaces can be general polygons and it is not necessarily trivial to compute them. (see figure 8). (2) It is not clear how to write meaningful shape rules for general polygons. (3) There is no simple mechanism to assign non-terminal symbols for the façade grammar, because the surfaces are the output of an algorithm, rather than a production rule.

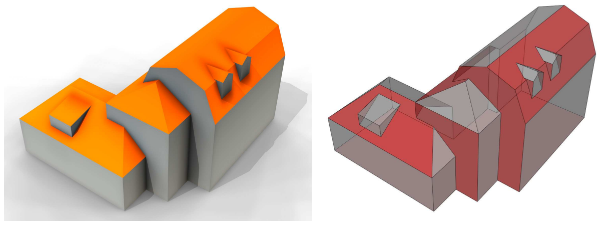


Figure 8: The union of simple volumetric shapes leads to complex polygons on the building shell: the resulting polygons can be concave, have many vertices, and multiple holes (marked red on the left).

Modeling strategy: Our solution to the problem retains the simplicity necessary for procedural modeling, while still working on many configurations of mass models. First, we use three-dimensional scopes to place three-dimensional shapes (volumes) to form a mass model. Then we generate two-dimensional scopes aligned with façade surfaces and roof surfaces by extracting the faces of the three-dimensional shapes with a component split. The resulting two-dimensional scopes will be correctly aligned and parameterized. Similarly we can extract edges by generating one-dimensional scopes. Please note that this modeling strategy works for arbitrarily aligned polygons such as roof surfaces as well as façade surfaces that are typically aligned with the up axis in the world coordinate system. The grammar can then proceed to refine the resulting quads and triangles (and in very limited cases general polygons). It is also important to note that after a shape (scope) is reduced to two-dimensions it is often replaced with a three-dimensional one by subsequent rule applications. Our solution to a consistent design are two mechanisms: (1) testing spatial overlap (occlusion) and (2) testing nearby important lines and planes in the shape configuration (snap lines). These two mechanisms are explained in the following.

3.2 Occlusion

An occlusion query tests for intersections between shapes. The simplest query can test if the current shape (the shape selected for derivation) is occluded by any other shape in the configuration. The result of this query can be either, no occlusion (”*none*”), partial occlusion (”*part*”), or full occlusion (”*full*”). For example, the rule below tests if the façade part *tile* is occluded before it is replaced by a door:

1: tile : Shape.occ(”all”) == ”none” $\rightsquigarrow$ door

There are several variations to query only a subset of the shape configuration: (1) We can make use of the fact that we store the derivation tree. The previous query would test for occlusion against all shapes, including the inactive ones that already have been replaced by a shape rule. To test only against active shapes we can use the keyword ”*active*” in the query. (2) We can restrict the queries to a subset of shapes with a specific label, e.g. $Shape.occ(”balcony”)$ tests only against shapes labeled balcony. (3) One of the most important subsets contains all shapes in the derivation tree except the current shape's predecessors. With this subset, we avoid the querying of parent shapes, which, in the case of a split, always occlude their successor shapes [Wonka et al. 2003]. A typical example is illustrated below (ε is the empty shape):

1: tile : Shape.occ(”noparent”) == ”none” $\rightsquigarrow$ window
2: tile : Shape.occ(”noparent”) == ”part” $\rightsquigarrow$ wall
3: tile : Shape.occ(”noparent”) == ”full” $\rightsquigarrow$ ε

The type of intersection computation can also be modified, to include distance, e.g. $Shape.occ(”noparent”,”distance”,4)$ tests if the current shape enlarged by 4 is occluded. In theory, the precise computation would require morphological shape operations, so that we resort to simple approximations in practice. Another modification is to test occlusion of sightlines, e.g. $Shape.visible(”street”)$ tests if the shortest sightlines to the street geometry are occluded.

3.3 Snapping

Occlusion makes it possible to avoid placing façade elements such as windows and doors on the intersection of volumetric shapes constituting the mass model. While this gives some improvement, we can further improve the layout of the façade structure, if we alter existing shape rules to snap to a dominant face or line (snap shapes) in the shape configuration. In the simplest form, all faces of the volumetric shapes of the mass model are stored as global construction planes. If we are selecting a planar (two-dimensional) scope on the side of a façade as the active rule, the scope can be intersected by the global construction planes defining snap lines. The snap lines work for a repeat split and a subdivision split as follows: (1) For the repeat rule the snap lines divide the scope into different parts and the repeat rule is invoked for each part separately. (2) For a subdivision rule the snap line just alters the closest split and leaves everything else unmodified. See figure 9 for a two dimensional example of a snap line changing the outcome of a repeat-split and a subdivide-split. Figure 10 shows an example model created with the help of snap lines. The notation for snap lines is illustrated below. The keyword *Snap* inserts a snap shape (in case of rule 2 with label ”*entrancesnap*”) and we modify the splitting axis (e.g. ”XS” instead of ”X”) to snap to existing snap shapes.

1: floors $\rightsquigarrow$ Repeat(”Y”,$floor\_height$){ floor Snap(”XZ”) }
2: entrance $\rightsquigarrow$ Snap(”Y”,”entrancesnap”) door
3: floor $\rightsquigarrow$ Repeat(”XS”,$tile\_width$){ tile }

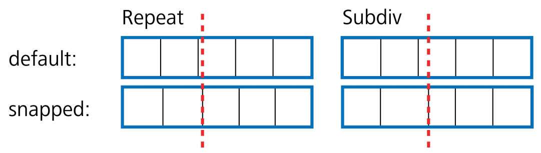


Figure 9: This figure illustrates the effect of snap lines (red). Left: A scope is split with a repeat split not reacting to the snap line (top). The snapped version of the repeat split first splits the scope with the snap line and then invokes the repeat for each half separately, thereby changing all shape sizes. Right: The subdivide split only alters the splitting line closest to the snap line. Only the two shapes adjacent to the snap line are changed.

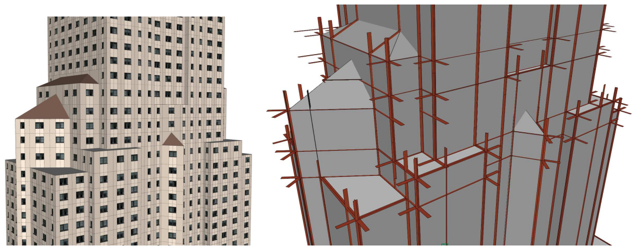


Figure 10: Left: A building generated with snap lines. The thin lines on the building show the scopes of the final shapes illustrating the structure of the grammar. Note how the floor levels are automatically aligned over all solids, e.g. a higher floor was forced below the tapering (common skyscraper feature). Right: The snap lines used during the construction.

3.4 Implementation

To store the shapes and snap lines for spatial queries, we use an octree [Berg et al. 2000] as acceleration data structure. The main reason is simplicity of implementation, especially due to frequent runtime modifications of the shape configuration. We also experimented with a discretized data structure (modified octree, see figure 11). Another acceleration strategy is to replace occlusion queries of shapes with occlusion queries of scopes. For example, $Scope.occ$ can be used to test the current scope for occlusion. To compute geometric intersections (e.g. for the determination of snap lines), we use a splitting algorithm based on [Mäntylä 1986].

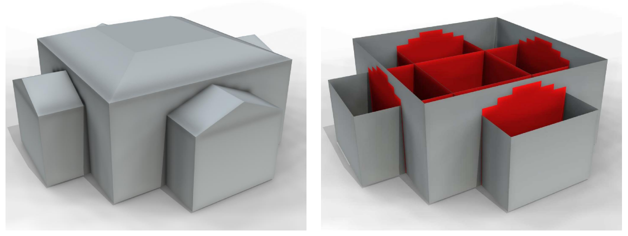


Figure 11: Left: A volumetric model. Right: The approximate occlusion data-structure intersected with the the occluded façade surfaces is shown in red.

4 A Simple Building Model

This section details an introductory example of modeling with our shape grammar. The example grammar will generate a simple generic building including its roof from an arbitrary building footprint (see figure 12). This building footprint is the axiom of the grammar. We aim to illustrate the following concepts:

Readability: The rules are human readable and can therefore be reused and understood by other users. We use parameters that are set in cursive font, e.g. $building\_height$ and $building\_angle$ and we use shape symbols from a consistent architectural vocabulary [Mitchell 1990].

Occlusion: The grammar can be applied to several footprints of a city. Even this simple example works on concave footprints and avoids placing windows and doors where the building intersects neighboring buildings (see figure 13 left). Rule 2 breaks down the mass model into its side faces. Rule 3 identifies the street facing building side to place an entrance. Rule 7 uses the occlusion query to avoid placing windows at the intersection of mass models.

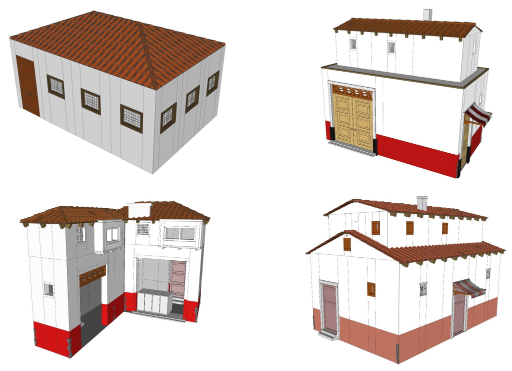


Figure 12: The top left shows a building generated with the rules described in section 4. The other three models were generated by extending the grammar.

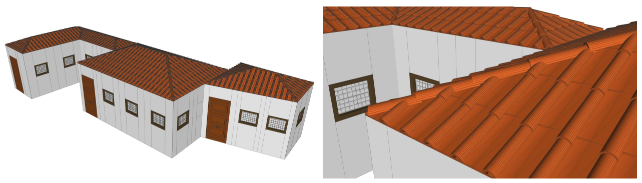


Figure 13: Left: The simple grammar is applied to three different footprints. Please note how the occlusion query avoids placing windows at the intersection of two neighboring buildings. Right: The roof is modeled by bricks on the roof planes and the roof edges.

Roof construction: The placement of bricks on the roof construction is illustrated in figure 13 right. The grammar works as follows: Rule 12 splits the roof with a component split and generates scopes aligned with the edges and the faces. Then the edges are covered with *roundbrick* and the faces with *flatbrick*. Please note how the grammar generates overlapping bricks, i.e. the geometry is not just a simple displacement map.

```
PRIORITY 1:
1:   footprint ⇝ S(1r,building_height,1r) facades
        T(0,building_height,0) Roof(”hipped”,roof_angle){ roof }
PRIORITY 2:
2:   facades ⇝ Comp(”sidefaces”){ facade }
3:   facade : Shape.visible(”street”)
        ⇝ Subdiv(”X”,1r,door_width*1.5){ tiles | entrance } : 0.5
        ⇝ Subdiv(”X”,door_width*1.5,1r){ entrance | tiles } : 0.5
4:   facade ⇝ tiles
5:   tiles ⇝ Repeat(”X”,window_spacing){ tile }
6:   tile ⇝ Subdiv(”X”,1r,window_width,1r){ wall |
        Subdiv(”Y”,2r,window_height,1r){ wall | window | wall } | wall }
7:   window : Scope.occ(”noparent”) != ”none” ⇝ wall
8:   window ⇝ S(1r,1r,window_depth) I(”win.obj”)
9:   entrance ⇝ Subdiv(”X”,1r,door_width,1r){ wall |
        Subdiv(”Y”,door_height,1r){ door | wall } | wall }
10:  door ⇝ S(1r,1r,door_depth) I(”door.obj”)
11:  wall ⇝ I(”wall.obj”)
PRIORITY 3:
12:  roof ⇝ Comp(”sidefaces”){ covering }
        Comp(”sideedges”){ roofedge } Comp(”topedges”){ roofedge }
13:  covering ⇝
        Repeat(”XY”,flatbrick_width,brick_length){ flatbrick }
        Subdiv(”X”,flatbrick_width,1r){ ε |
          Repeat(”X”,flatbrick_width){ roofedge } }
14:  roofedge ⇝
        Subdiv(”Y”,overlap,brick_length-2*overlap,1r){ ε |
          roundbrick | Repeat(”Y”,brick_length-overlap){ roundbrick } }
15:  flatbrick ⇝ S(1r,1r,flatbrick_height) T(0,0,-flatbrick_height)
        Rx(-3) I(”flatbrick.obj”)
16:  roundbrick ⇝ S(roundbrick_w,Scope.sy+overlap,roundbrick_h)
        T(-roundbrick_w/2,-overlap,-roundbrick_h)
        Rx(-3) I(”roundbrick.obj”)
```

5 A Model for Office Buildings

The following example shows firstly a small rule set to generate various mass models using a stochastic grammar. The axiom of this grammar is a building lot, a two-dimensional shape. The rules work as follows: First, the lot is extruded by *building_height* with a size command to yield a three-dimensional shape. This shape is then split into two smaller shapes. One volumetric shape (*facades*) that is the largest solid in the mass model and one shape that will later be broken down into two side wings. This split is performed by Rule 2. The rule also generates a gap between the side wings, thereby creating a U-shape. Rule 3 shows the use of stochastic rules to generate a variety of mass configurations. Please note that we use a combination of random numbers and stochastic rule selection to create a variety of side wing shapes with different heights and widths. For example, there is a fifty percent chance that the side wing is as high as the largest solid. Rule 4 is the transition to façade modeling. An example model using these four rules is depicted in figure 14.

```
PRIORITY 1:
1:   lot ⇝ S(1r,building_height,1r)
        Subdiv(”Z”,Scope.sz*rand(0.3,0.5),1r){ facades | sidewings }
2:   sidewings ⇝
        Subdiv(”X”,Scope.sx*rand(0.2,0.6),1r){ sidewing | ε }
        Subdiv(”X”,1r,Scope.sx*rand(0.2,0.6)){ ε | sidewing }
3:   sidewing
        ⇝ S(1r,1r,Scope.sz*rand(0.4,1.0)) facades : 0.5
        ⇝ S(1r,Scope.sy*rand(0.2,0.9),Scope.sz*rand(0.4,1.0))
           facades : 0.3
        ⇝ ε : 0.2
4:   facades ⇝ Comp(”sidefaces”){ facade }
```

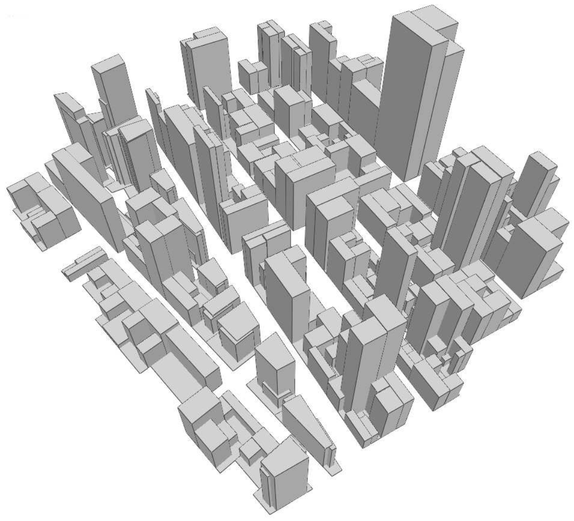


Figure 14: Stochastic variations of building mass models generated with only four rules (starting with the building lot as axiom).

The idea of the second part of this rule set is to first derive the dominant shape in the mass volume and force dominant planes of the construction (floors) on the other shapes. The rules also demonstrate the use of labeled and unlabeled snap lines. The first rule generates the front façade and rule 7 will be used for the remaining faces of the building. The second production subdivides the ground floor into several parts, including a door and a labeled snap line. This snap line is inserted as an inactive shape in the shape configuration. Rule 8 splits into individual floors ($floor$) and adds snap planes parallel to the ground floor. Please note that this rule snaps to existing snap planes as well as creating new ones. The use of labeled snap lines is illustrated by rule 9 and 15. Rule 9 places a labeled snap line in the up direction, so that the fire escape is aligned with the façade structure. Details, such as windows, doors, entrance, and walls are build in similar way to the example in section 4. An example model is depicted in figure 15.

```
PRIORITY 2:
5:  facade : Shape.visible(”Street”) == 0 ⇝
      Subdiv(”Y”,groundfloor_height,1r,topfloor_height)
        { groundfloor | floors | topfloors } firescape
6:  groundfloor ⇝ Subdiv(”X”,1r,entrance_width,1r){ groundtiles |
      entrance SnapLines(”Y”,”entrancesnap”) | groundtiles }
PRIORITY 3:
7:  facade ⇝ floors
8:  floors ⇝ Repeat(”YS”,floor_height){ floor Snap(”XZ”) }
9:  floor ⇝ Repeat(”XS”,tile_width){ tile Snap(”Y”,”tilesnap”) }
...
15: wall : Shape.visible(”Street”) ⇝ I(”frontwall.obj”)
PRIORITY 4:
16: fireescape ⇝ Subdiv(”XS”,1r,2*tile_width,7r,”tilesnap”)
      { epsilon | escapestairs | ε }
17: escapestairs ⇝ S(1r,1r,fireescape_depth)
      T(0,0,-fireescape_depth) Subdiv(”YS”,groundfloor_height,1r)
        { ε | Repeat(”YS”,floor_height){ I(”fireescape.obj”) } }
```

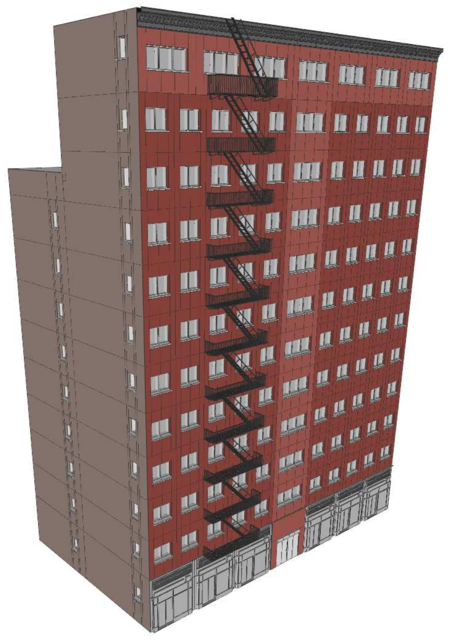


Figure 15: A procedurally generated building modeled with snap lines. Note the alignment of important lines and planes in the construction.

6 A Model for Single Family Homes

The shape grammar and shape queries can also be used to generate and place other components in an urban environment. The following example shows the nice interplay, between one, two and three-dimensional modeling (see figure 16). The grammar in this example uses the following strategy: (1) split of the property edges with a component split and place shrubs near the fence, (2) split the property to model the front yard, back yard and the main building, (3) generate a sidewalk and place trees (generated with Greenwork's Xfrog) in regular intervals next to the street, and (4) generate a driveway connected to the garage door and a pathway connected to the entrance door.

The nice aspect of our modeling system is that the initial stages of a grammar can work with overlapping, but well formed shapes and then only later resolve conflicts with occlusion queries. For example we resolve intersections between sidewalk and driveway and place trees and shrubs in sufficient distance from the house and other vegetation.

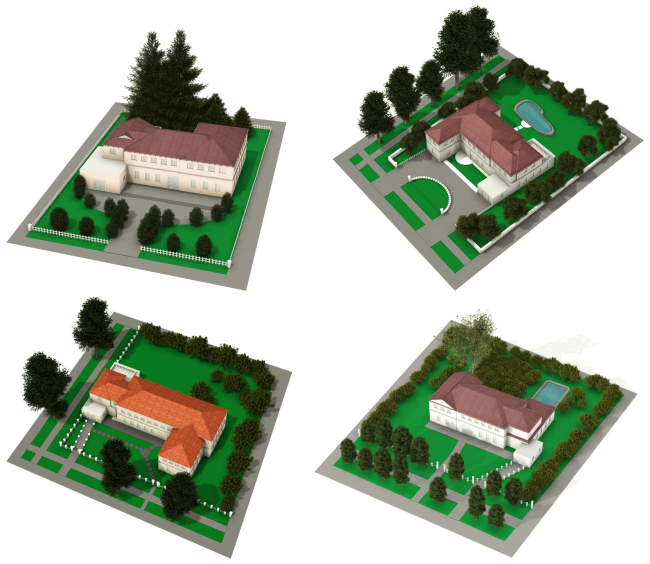


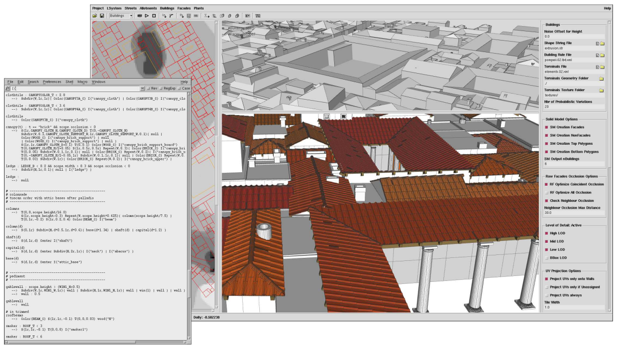
Figure 16: Different buildings in a suburban environment. *CGA shape* can also be employed for the procedural generation of the building environment e.g. walkways or vegetation.

7 Results

7.1 User Interface and Workflow

The C++ implementation of *CGA shape* is integrated in the *CityEngine* framework [Parish and Müller 2001]. Figure 17 shows a screenshot of the *CityEngine* user interface. We can import most forms of GIS data, including rasterized maps and Google Earth's KML format for building mass models. Similar to other modeling applications we rely on many different views of the model guiding an iterative design process. The most frequently used views are: (1) An overview mode to show building footprints, streets, and parcel boundaries. (2) A three-dimensional view of the partial derivation of the grammar e.g. up until to the mass models. (3) Preview modes of the final geometry of selected subsections. (4) Several tools to visualize the shape configurations, the scope of shapes, trim surfaces, snap lines and the topology of shape derivation trees for visual debugging. (5) A rule editor for the shape grammar.

To develop new rules, our implementation provides several interactive methods to restrict the derivation of the shape grammar to parts of the model (this includes one single building and parts of a building). Once the user is happy with the results he can generate a larger and detailed model and write it to the disk. The actual computation of a model with 50 thousand polygons (like in figure 1 left) takes about one second. In addition, half a second is needed for writing such a model to the hard disk. To setup the lighting and camera animations we use Maya with simple building mass models. Rendering a larger city requires a scalable rendering solution to work with billions of polygons. Therefore, we use Pixar's RenderMan with its memory-saving instancing support (with delayed read archives) and reliable level-of-detail (LoD) functionality to create the renderings. We modeled the different detail levels manually e.g. by interchanging high-resolution terminals with low-resolution ones or by adjusting rules which produce high polygon counts.



Project LSystem Streets Allotments Buildings Facades Plants
Buildings
File Edit Search Preferences Shell Macro Windows Help
Rev RegExp Case
Dolly: -0.502238

```
clothtile : CANOPYCOLOR_T < 2.8
   --> Subdiv(V,1r,1r){ Color(CANOPY3A_C) I("canopy_cloth") | Color(CANOPY3B_C) I("canopy_clo

clothtile : CANOPYCOLOR_T < 3.6
   --> Subdiv(V,1r,1r){ Color(CANOPY4A_C) I("canopy_cloth") | Color(CANOPY4B_C) I("canopy_clo

clothtile
   --> Color(CANOPY3B_C) I("canopy_cloth")

canopy(t) : t == "brick" && scope.occlusion < 0
   --> S(1r,CANOPY_CLOTH_H,CANOPY_CLOTH_D) T(0,-CANOPY_CLOTH_H)
       Subdiv(V,0.1,CANOPY_CLOTH_SUPPORT_W,1r,CANOPY_CLOTH_SUPPORT_W,0.1){ null |
       Color(WOOD_C) I("canopy_brick_support") | null
       | Color(WOOD_C) I("canopy_brick_support") | null }
       S(1r,1r,CANOPY_CLOTH_D+0.1) T(0,0.1) Color(WOOD_C) I("canopy_brick_support_board")
       T(0,CANOPY_CLOTH_H/2+0.05) S(1r,0.5r,0.5r) Repeat(V,0.3){ Color(BRICK_C) I("canopy_bri
       T(0,0.05) Subdiv(V,0.1,1r,0.1){ null | Color(BRICK_C) Repeat(V,0.2){ I("canopy_brick_u
       T(0,-CANOPY_CLOTH_H/2-0.05,1r) Subdiv(V,0.1,1r,0.1){ null | Color(BRICK_C) Repeat(V,0.
       T(0,0.03) Subdiv(V,1r){ Color(BRICK_C) Repeat(V,0.2){ I("canopy_brick_upper") }

ledge : LEDGE_B < 0.2 && scope.width > 0.3 && scope.occlusion < 0
   --> Subdiv(H,1r,0.1){ null | I("ledge") }

ledge :
   --> null


# ----------------------------------------------
# colonnade
# tuscan order with attic bases after palladio
# ----------------------------------------------

columns
   --> T(0,0,scope.height/16.0)
       S(1r,scope.height-0.3) Repeat(V,scope.height*0.625){ column(scope.height/7.5) }
       T(0,1r,-0.2) S(1r,0.2,0.4) Color(BEAM_C) I("beam")

column(d)
   --> S(0,1r) Subdiv(H,d*0.5,1r,d*0.6){ base(d*1.34) | shaft(d) | capital(d*1.2) }

shaft(d)
   --> S(d,1r,d) Center I("shaft")

capital(d)
   --> S(d,1r,d) Center Subdiv(H,2r,1r){ I("neck") | I("abacus") }

base(d)
   --> S(d,1r,d) Center I("attic_base")


# ----------------------------------------------
# pediment
# ----------------------------------------------

gablewall : scope.height > (WIN1_H+0.5)
   --> Subdiv(V,1r,WIN1_W,1r){ wall | Subdiv(H,1r,WIN1_H,1r){ wall | win(1) | wall } | wall }
   --> wall : 0.5

gablewall
   --> wall

# is trimmed
roofbeams
   --> Color(BEAM_C) S(1r,1r,-0.1) T(0,0,0.03) wood("H")

smoker : ROOF_T < 3
   --> S(1r,1r,-0.1) T(0,0,0) I("smoker1")

smoker : ROOF_T < 6
```

Figure 17: Screenshot of the *CityEngine*, the *CGA shape* modeling environment. In the left panel of the main window is a GIS-like viewer to display the city layout and on the right an OpenGL preview to show selected parts of the generated geometry. The window in the front contains the rule editor.

7.2 Examples

First, we modeled Pompeii, an ancient Roman town destroyed in 79 AD, in collaboration with archeologists who gave us ground plans and figures of selected building types. We used this information to abstract 190 design rules to model the complete city including the streets and placement of trees. The basic modeling concept was introduced in section 4. The resulting city has about 1.4 billion polygons in high LoD, 31 million polygons in middle LoD, and 170 thousand polygons in low LoD. Figure 18 shows views of different elevations over the city and a view inside the street. The exterior lighting is simulated with ambient occlusion.

As a usability test we invited a professional modeler to develop an example model. On the first day we explained the user interface, the workflow, and several example rule sets to demonstrate the rule syntax and modeling strategies. Afterwards, with minor help support from our side and the rules of section 5 as starting point, he independently created the small city model depicted in figure 19. As a conclusion, modeling with *CGA shape* proved to be easy and efficient. Even the concept of snapping (which is suited for high-rise buildings) was understood and successfully applied.

The third model is inspired by aerial images of Beverly Hills, but the complete model is generated procedurally. We used about 150 rules, including rules for parcel subdivision, urban vegetation, swimming pools, sidewalks and streets. The basic modeling concept was introduced in section 6. The complete model consists of about one thousand buildings for a total of about 700 million polygons in full LoD (without the trees, which are only transformed instances). See figure 20 for renderings of the model.

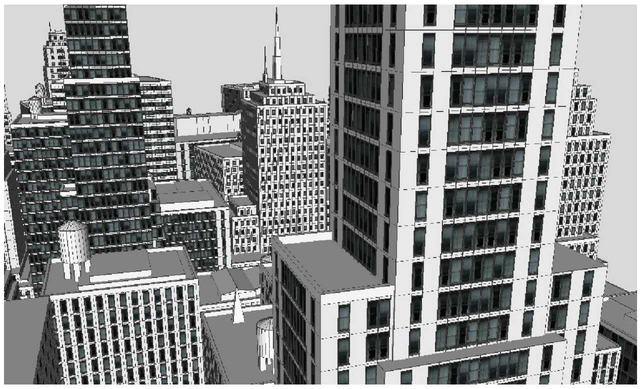


Figure 19: This figure shows a modern city model which was created from scratch in two days only.

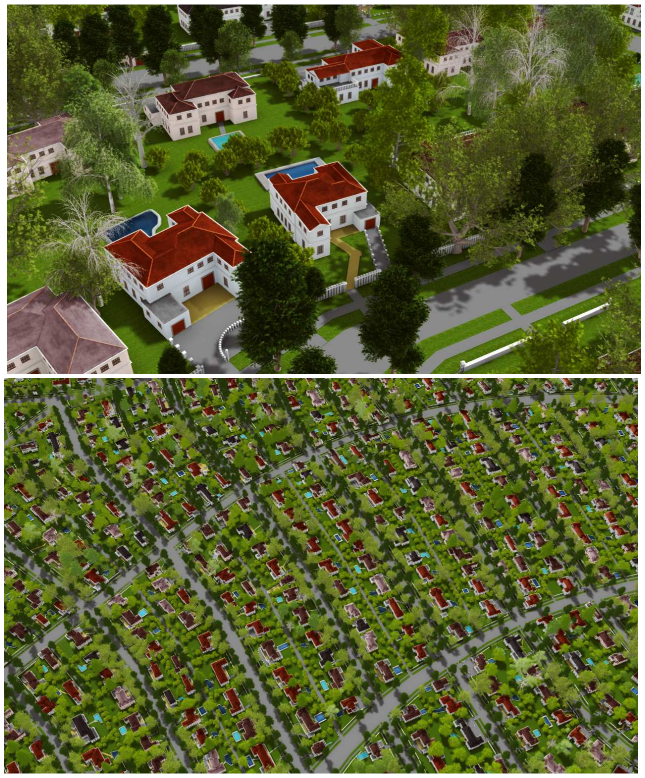


Figure 20: Our wealthy suburbia model was inspired by Beverly Hills. *CGA shape* was also used to distribute the tree models.

8 Discussion

In this section we want to compare to previous work, identify contributions and open problems that are of interest for future research.

Comparison to mesh modeling tools: A comparison to existing modeling tools can only be done informally. It is possible to use scripting in commercial modeling software to accelerate the modeling process. However, we believe that these scripts would most likely only replicate parts of our shape grammar. We were in discussion with three software companies who are interested in our procedural city modeling tools because current costs of content creation are a major challenge in the industry. As an example from the movie industry we can mention that the urban models for *Superman Returns* took 15 man years to complete. Still, our mod-

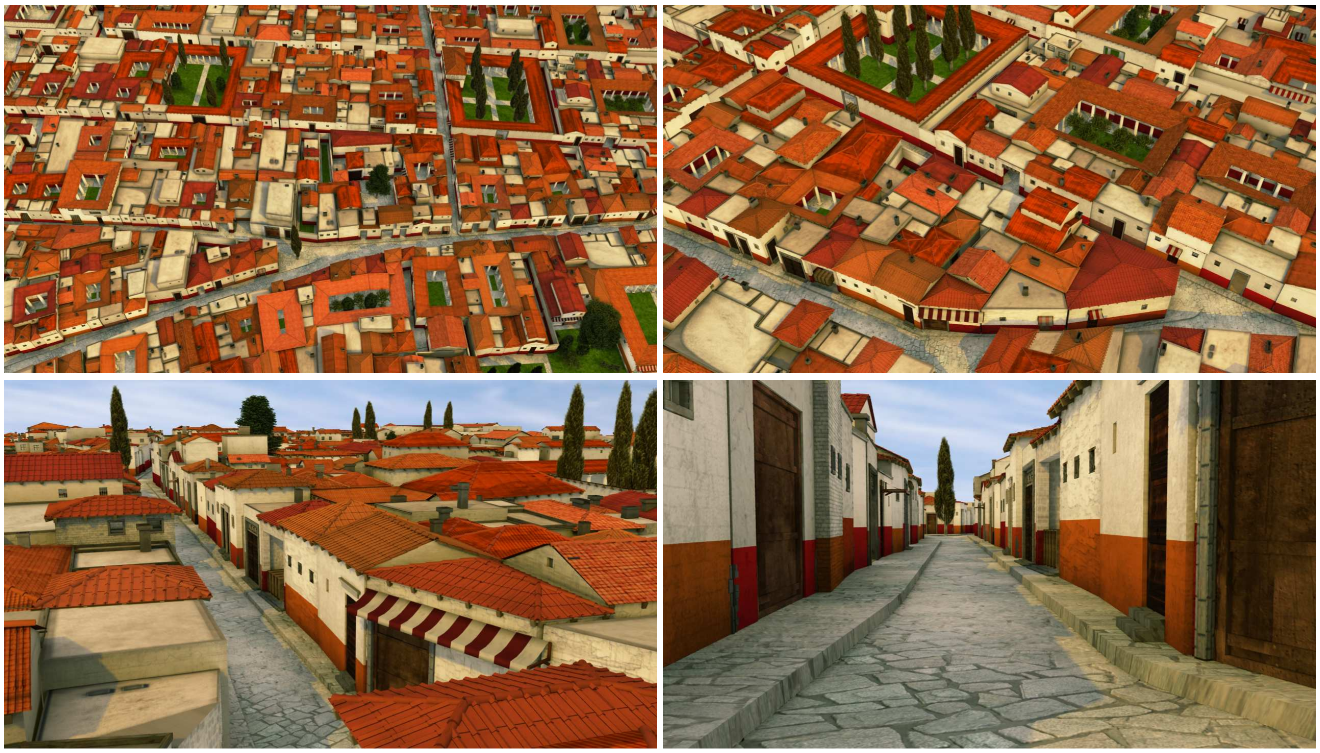


Figure 18: Various views of the procedural Pompeii model. Based on real building footprints, the city was generated with 190 manually written *CGA shape* rules. Hence, the whole model is a rule-based composition of 36 terminal objects (plus 4 tree types and the environment).

els with up to a billion polygons are significantly more detailed than the current modeling practice in the entertainment industry. However, the shape grammar introduced in this paper does not aim to replace existing 3d modeling software, but relies on a tight integration in a complete modeling environment. While the global structure, the positioning of individual shapes, level-of-detail control, and the data handling of large models is a strength of our framework, the generation of smaller complex geometric details is sometimes inefficient. We used Maya to generate geometry, such as roof bricks, the capitals, and window grills. We belief that *CGA shape* is a significant step forward that reduces modeling times by orders of magnitude.

Efficiency and robustness: We found that designing with our grammar is robust and efficient in most cases since no complex and error-prone geometric computations have to be executed (like boolean operation algorithms, which are very difficult to implement reliably). We can formulate meaningful rules for simpler shapes that together create complex polygonal surfaces on the building shell. We believe that we found a very good tradeoff between visual quality and speed. A global optimization might be able to produce better results, but it is much more difficult to model and modeling times could be prohibitively high. In contrast, the derivation of the shape grammar is reasonably fast so that massive one billion polygon models can be generated in less than one day. A general disadvantage of a procedural approach is that it sometimes generates configurations of shapes that are not plausible. This is especially the case when starting from arbitrary building footprints given by a GIS dataset. In this context, we believe that it would be a promising avenue of future research to employ shape grammars for shape understanding.

Usability: The learning curve to use *CGA shape* is similar to that of other scripting languages. We make a conscious effort to write the rules in human readable form, as demonstrated in the paper. Since it is possible to reuse rules and share rules with other users, even inexperienced users will be able to quickly model satisfactory results by importing and modifying available rule sets. However, carelessly written rules can be very contrived and will be only understood by the original author and may produce unwanted side effects. We expect that modeling with *CGA shape* is most naturally understood by people with computer science background, but many professional modelers are familiar with scripting and will be able to use shape grammars.

Architecture and computer graphics: Rules in architectural literature are very powerful, but typically abstract and under specified, so that they can only be applied by humans. The major contribution of our shape grammar is to adapt architectural concepts and derive a set of specific shape rules that can be implemented and are powerful enough to generate detailed high quality models. While we believe that the application of *CGA shape* in computer graphics is very successful, we acknowledge that the application to architectural design is not yet explored and might require significant changes.

Comparison to L-systems: Our work is inspired by pioneering work in plant modeling [Prusinkiewicz and Lindenmayer 1991; Měch and Prusinkiewicz 1996] and the beautiful images that were created. Similarities include: (1) the notation of the rules, (2) the idea of the scope is an evolution of the L-system turtle, and (3) the basic idea for the necessity of context sensitive rules. However, the details and modeling challenges are fundamentally different. A major distinction of our grammar is that we emphasize the concept of shape, and rules replacing shapes with shapes, rather than building on string replacement. We also use a large set of shape rules not existing in L-systems. Furthermore, the rules governing a biological system do not directly relate to the modeling of buildings. We found that a direct application of L-Systems to architecture overemphasizes the idea of growth, a concept that is often counterproductive for the procedural modeling of buildings.

Comparison to Instant Architecture: We built on the idea of the split rule [Wonka et al. 2003] as an important ingredient for *CGA shape*. As split grammars maintain a strict hierarchy, modeling is fairly simple, but also limited. However, after introducing rules for combinations of shapes and more general volumetric shapes such as roofs, the strict hierarchy of the split-grammar can no longer be enforced. We can confirm, that the idea of split rules is a very suitable primitive to generate façade details, but we did not find it suitable for many forms of mass modeling. We made use of the control grammar to generate procedural variations together with simple stochastic rules. We believe that our model of context sensitive shape rules, together with the interplay of one, two, and three dimensional modeling are an elegant and efficient solution to a challenging problem. Besides this major conceptual contribution, we are also the first to address application related details, such as the definition of the most important shape rules, the concise notation, and modeling examples detailing various modeling strategies.

Real-time rendering: Although we are currently collaborating to build a real-time rendering solution this requires additional post-processing algorithms not yet developed. One main challenge for this future work is to develop levels-of-detail techniques for massive city models. As we currently do not optimize for consistent topology, existing algorithms would fail.

9 Conclusion

This paper introduces *CGA shape*, a novel shape grammar for the procedural modeling of building shells to obtain large scale city models. The paper is the first to address the aspect of volumetric mass modeling of buildings including the design of roofs. These two elements form the major contributions of this paper. Furthermore we introduced several extensions to the split grammar to obtain a complete modeling system. We believe that our work is a powerful adaption of Stiny's seminal shape grammar idea for computer graphics and we demonstrate the creation of massive city models that have significantly more geometric detail than any existing urban model created in industry or academia.

Acknowledgments

The authors thank Robbie Müller for testing the ease of use, Tijl Vereenooghe for archaeological consulting, and the anonymous reviewers for their constructive comments on improving this paper. This project is supported in part by EC IST Network of Excellence EPOCH, EC IST Project CyberWalk, and NGA grant HM1582-05-1-2004.

References

AICHHOLZER, O., AURENHAMMER, F., ALBERTS, D., AND GAERTNER, B. 1995. A novel type of skeleton for polygons. *Journal of Universal Computer Science 12*, 12, 752–761.

ALEXANDER, C., ISHIKAWA, S., AND SILVERSTEIN, M. 1977. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York.

BERG, M. D., KREVELD, M. V., OVERMARS, M., AND SCHWARZKOPF, O. 2000. *Computational Geometry*. Springer-Verlag.

CHING, F. D. K. 1996. *A Visual Dictionary of Architecture*. Wiley.

DAVIS, M., SIGAL, R., WEYUKER, E. J., AND DAVIS, M. D. 1994. *Computability, Complexity, and Languages : Fundamentals of Theoretical Computer Science*. Academic Press.

DOWNING, F., AND FLEMMING, U. 1981. The bungalows of buffalo. *Environment and Planning B 8*, 269–293.

DUARTE, J. 2002. *Malagueira Grammar – towards a tool for customizing Alvaro Siza's mass houses at Malagueira*. PhD thesis, MIT School of Architecture and Planning.

EHRIG, H., ENGELS, G., KREOWSKI, H.-J., AND ROZENBERG, G. 1999. *Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools*. World Scientific Publishing Company.

EPPSTEIN, D., AND ERICKSON, J. 1999. Raising roofs, crashing cycles, and playing pool: applications of a data structure for finding pairwise interactions. In *Proceedings of the 14th Annual Symposium on Computational Geometry*, ACM Press, 58–67.

FLEMMING, U. 1987. More than the sum of its parts: the grammar of queen anne houses. *Environment and Planning B 14*, 323–350.

HAVEMANN, S. 2005. *Generative Mesh Modeling*. PhD thesis, TU Braunschweig.

HILLIER, B. 1996. *Space Is The Machine: A Configurational Theory Of Architecture*. Cambridge University Press.

KNUTH, D. 1968. Semantics of context-free languages. *Mathematical Systems Theory 2*, 2, 127–145.

KONING, H., AND EIZENBERG, J. 1981. The language of the prairie: Frank lloyd wrights prairie houses. *Environment and Planning B 8*, 295–323.

LE CORBUSIER. 1985. *Towards a New Architecture*. Dover Publications.

LEGAKIS, J., DORSEY, J., AND GORTLER, S. J. 2001. Feature-based cellular texturing for architectural models. In *Proceedings of ACM SIGGRAPH 2001*, ACM Press, E. Fiume, Ed., 309–316.

MÄNTYLÄ, M. 1986. Boolean operations of 2-manifolds through vertex neighborhood classification. *ACM Transactions on Graphics 5*, 1, 1–29.

MARCH, L., AND STEADMAN, P. 1974. *The Geometry of Environment*. MIT Press.

MĚCH, R., AND PRUSINKIEWICZ, P. 1996. Visual models of plants interacting with their environment. In *Proceedings of ACM SIGGRAPH 96*, ACM Press, H. Rushmeier, Ed., 397–410.

MITCHELL, W. J. 1990. *The Logic of Architecture: Design, Computation, and Cognition*. MIT Press.

PARISH, Y. I. H., AND MÜLLER, P. 2001. Procedural modeling of cities. In *Proceedings of ACM SIGGRAPH 2001*, ACM Press, E. Fiume, Ed., 301–308.

PRUSINKIEWICZ, P., AND LINDENMAYER, A. 1991. *The Algorithmic Beauty of Plants*. Springer Verlag.

PRUSINKIEWICZ, P., JAMES, M., AND MĚCH, R. 1994. Synthetic topiary. In *Proceedings of ACM SIGGRAPH 94*, ACM Press, A. Glassner, Ed., 351–358.

PRUSINKIEWICZ, P., MÜNDERMANN, P., KARWOWSKI, R., AND LANE, B. 2001. The use of positional information in the modeling of plants. In *Proceedings of ACM SIGGRAPH 2001*, ACM Press, E. Fiume, Ed., 289–300.

SCHMITT, G. 1993. *Architectura et machina*. Vieweg & Sohn.

SHUBNIKOV, A. V., AND KOPTSIK, V. A. 1974. *Symmetry in Science and Art*. Plenum Press, New York.

SIPSER, M. 1996. *Introduction to the Theory of Computation*. Course Technology, Boston.

STINY, G., AND MITCHELL, W. J. 1978. The palladian grammar. *Environment and Planning B 5*, 5–18.

STINY, G. 1975. *Pictorial and Formal Aspects of Shape and Shape Grammars*. Birkhauser Verlag, Basel.

STINY, G. 1980. Introduction to shape and shape grammars. *Environment and Planning B 7*, 343–361.

STINY, G. 1982. Spatial relations and grammars. *Environment and Planning B 9*, 313–314.

WEYL, H. 1952. *Symmetry*. Princeton University Press.

WONKA, P., WIMMER, M., SILLION, F., AND RIBARSKY, W. 2003. Instant architecture. *ACM Transactions on Graphics 22*, 3, 669–677.