*Volume 29 (2010), Number 2 pp. 1–12* COMPUTER GRAPHICS *forum*

# Generalized Use of Non-Terminal Symbols for Procedural Modeling

L. Krecklau, D. Pavic and L. Kobbelt

RWTH Aachen University, Germany

**Abstract**

*We present the new procedural modeling language $G^2$ (Generalized Grammar) which adapts various concepts from general purpose programming languages in order to provide high descriptive power with well-defined semantics and a simple syntax which is easily readable even by non-programmers. The term "Generalized" reflects two kinds of generalization. On the one hand we extend the scope of previous architectural modeling languages by allowing for multiple types of non-terminal objects with domain-specific operators and attributes. On the other hand the language accepts non-terminal symbols as parameters in modeling rules and thus enables the definition of abstract structure templates for flexible re-use within the grammar. By deriving $G^2$ from the well-established programming language* Python*, we can make sure that our modeling language has a well-defined semantics. For illustration, we apply $G^2$ to architectural as well as plant modeling in order to demonstrate its descriptive power with some complex examples.*

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Languages—I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—

## 1. Introduction

Procedural modeling has become a well established approach in applications where highly complex three-dimensional (3D) scenes with rich detail have to be generated [WMV*08]. In movie production and game industry, this approach is often used for the mostly automatic generation of realistically looking architecture, landscapes, or plants [Whi06, Zal04] (cf. Figure 1).

In contrast to conventional geometric representations like polygon meshes or voxel grids, procedural models describe a scene by a set of rules that recursively convert an input symbol (*non-terminal*) into a sequence of output symbols (*terminal or non-terminal*). There are several different production systems that can be applied for 3D content creation. For example, plant modeling often relies on string replacement. At a certain step the evaluation is stopped and the resulting string is visually interpreted. In contrast, architectural modeling is usually inspired by shape replacement in a coarse to fine fashion. Each step of the evaluation will bring more detail into the shape. As a consequence, the procedural description of a model usually looks like a snippet of source

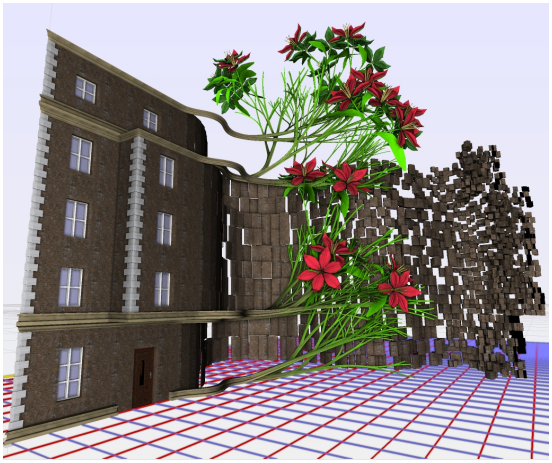


**Figure 1:** *We present $G^2$ which utilizes different modeling concepts in order to combine several object types like buildings or plants as seen in this artistic image.*

code written in some very basic programming language. The design of such a geometric programming language has to be done very carefully. If the language is too simple we lose the generality to design complex and general scenes. If it is too complicated, the code is no longer intuitively readable by non-programmers.

In this paper we focus on the idea of shape replacement as it is traditionally used in procedural modeling languages which are custom designed for the creation of buildings. Since rules for general shape replacement are hard to handle in practice, these languages in the domain of architectural models utilize translation, rotation, scaling and splitting operators working on simple boxes.

The novel procedural modeling language $G^2$ generalizes previous approaches by allowing for multiple *non-terminal classes* instead of limiting to simple boxes only. In order to increase the descriptive power while keeping the semantics clear and unambiguous as well as keeping the syntax human readable, we adapt various concepts from existing general purpose programming languages. In particular we extend existing procedural modeling systems by

**Abstract Structure Templates.** — We support the use of non-terminal symbols as user parameters to take the simplicity of the grammar one step futher by defining *abstract structure templates* which are encapsulated into *modules* in order to prevent variable propagation and to avoid name clashes.

**Non-Terminal Classes.** — *Non-terminal objects* can be of different types. Each type has special *operators* and *attributes* that are implicitly declared by the system. This freedom brings procedural modeling much closer to conventional modeling paradigms.

**Flags** — We introduce *flags* as a mechanism for local changes. Flags identify specific subtrees in the scenegraph without creating variable dependencies between the rules.

The name $G^2$ (Generalized Grammar) reflects two kinds of generalization. On the one hand the syntax of the grammar is generalized compared to previous approaches, i.e. it is easy extendable by new *non-terminal classes* thereby providing new modeling strategies. On the other hand user defined grammars become generalized by using *abstract structure templates* in order to make a scene description more compact and maintainable.

Furthermore, we derive our modeling language from the general-purpose language *Python* that allows for an easy implementation of our system.

### 1.1. Related Work

Procedural techniques have been used in several applications. Lindenmayer and Prusinkiewicz pioneered automatic plant generation [PL96] by using L-Systems. The resulting string can be visually interpreted as LOGO-style turtle graphics. This idea was then extended by several authors utilizing different formal language concepts like parameters, stochastic rule application, context-sensitivity [PHHM97] and environmental-sensitivity [PJM94] resulting in the plant modeling language *cpfg* [PHM00, PKMH00]. Positional information was later used in order to get better control over the growth process and its overall appearance [PMKL01]. L-Systems also fit perfectly well to the generation of street networks as long as the system provides self-sensitivity [PM01].

In contrast to a growth process that is achieved with L-Sytems, man-made structures like architecture are better designed in a coarse to fine fashion by utilizing splitting rules to decompose basic shapes into other shapes [WWSR03]. The concept of replacing shapes was pioneered by Stiny [Sti75, Sti80] who introduced the concept of shape grammars. Müller et. al. presented *CGA shape*, which utilizes scopes (i.e. bounding boxes containing any geometry) for the hierarchical build-up of the scene [MWH*06]. Mass models are the basis for the building structure. Occlusion queries and snapping lines lead to well arranged facade elements like windows or doors. However, the system is restricted to scopes acting as bounding boxes to load geometry which becomes a problem, e.g., if rounded objects have to be created. Our approach uses free-form deformation (FFD) [SP86] as an alternative *non-terminal object* to overcome this drawback.

Having footprints of buildings as a starting point, wall grammars can be used to create 2.5D facades [LG06]. Further research has also shown that splits [HBW06] can be used to automatically create the interior of buildings. Although several of the mentioned works use parameters to create dynamic content, they never take advantage of the procedural modeling in the sense of reusable structure patterns. In this paper, we introduce modules defining abstract structure templates by using non-terminal symbols as user parameters.

Combining different aspects, a visual framework was proposed by Ganster et al. using a model graph as representation for the scene generation [GK07]. In grammar based systems, previous work also showed a way of making local changes in the scene which is essential for artists creating individual features in their models [LWW08]. Tags at the operators and different locators are defined outside the grammar to spot a range of elements during the scene generation in order to let the rules behave differently on these located instances. In contrast to their work, we present the concept of flags to achieve local changes which are directly integrated in the grammar.

A number of other approaches address automatic scene creation like a stack-based programming language for generative geometry [GML], using noise for terrains [EMP*02] or taking a volumetric representation as a basis for modeling solids [CDM*02]. Other methods apply the idea of procedural modeling to the destruction of objects [MGDH04].

Regarding these different approaches, $G^2$ aims for the combination of several modeling strategies. As recently stated by Vanegas et al. [VAW*09] any Turing complete language like Python could be used to model an arbitrary scene, but a modeling language should create a well balanced relation between its expressiveness and its usability. Therefore, we make use of several non-terminal classes where each class encapsulates one modeling strategy. We will see, that this idea turns out to be both expressive due to its easy extensibility and simple because of the human readable syntax.

## 2. System Features

Context-free formal grammars typically consist of a set of non-terminal symbols $N$, a set of terminal symbols $T$ and a set of production rules $N \rightarrow (N \cup T)^*$. Additionally, there is always a specific non-terminal symbol $S \in N$ that acts as the start symbol. In a given sequence of symbols, a rule takes all non-terminal symbols that are equal to the left-hand side of the rule and replaces them by the new sequence of symbols that is given on the right-hand side of the rule.

Applying this idea to the modeling of complex scenes, some objects in the scene might have a non-terminal symbol attached to it. To state this more clearly, the non-terminal symbols are used to associate an object in the scene with a rule of the grammar. We refer to *non-terminal objects* when scene objects with an attached non-terminal symbol have to be addressed. Additionally, each non-terminal object is an instance of a specific *non-terminal class* which defines the object, e.g. non-terminal boxes are defined by their size.

Since non-terminal objects are supposed to be modified by the grammar, the rules contain a sequence of operators which can be interpreted as terminal and non-terminal symbols of a formal grammar. The only difference is, that operators encapsulate a certain modeling concept within its non-terminal class, e.g. non-terminal boxes can be resized or splitted along a local axis. Operators which only change the state of the *current non-terminal object* (e.g. resizing a box) can be interpreted as terminal symbols. Operators which create new non-terminal objects in the scene (e.g. splitting a box) can be seen as non-terminal symbols. Newly created non-terminal objects will take the current non-terminal object as their parent thereby creating an instance hierarchy leading to a scene-graph.

$G^2$ provides explicit user parameters that can be declared in each rule. In contrast to parameteric formal grammars [PHHM97], in our case, parameters can also contain non-terminal symbols. This extension as well as our definition of modules enables the creation of *abstact structure templates*. We already stated that each non-terminal class has implicitly declared attributes which describe the non-terminal object. Similarly, each operator in that class has implicitly declared attributes which contain information about its modeling concept. These implicit attributes can be easily accessed within our grammar.

### 2.1. Classes and their Attributes

In contrast to previous work, we introduce *non-terminal classes* which provide different modeling concepts. Therefore, the rules in Figure 2 have to be of a specific type since the operators of each rule can only be applied to a certain kind of non-terminal object. For example the class of the *non-terminal box* behaves similar to *CGA shape* by giving simple transformations or providing the repeat and split operators for a structured build-up of the scene. In contrast to that, FFDs provide operators to manipulate its control points. All the classes contain some implicitly declared attributes, which describe the non-terminal object. A box, for example, has the three attributes $size_x$, $size_y$ and $size_z$ whereas a FFD stores the 3D positions of all its control points $c_{xyz}$ with $x,y,z \in \{0,1\}$. These attributes can then be used for further calculations as seen in rule *A* of Figure 2.

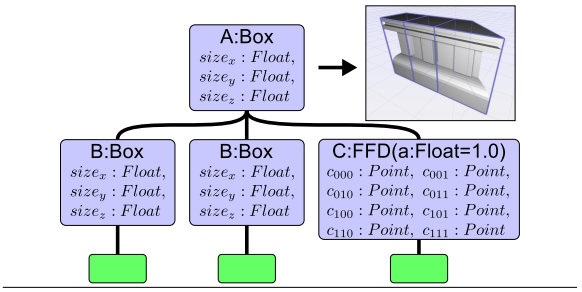
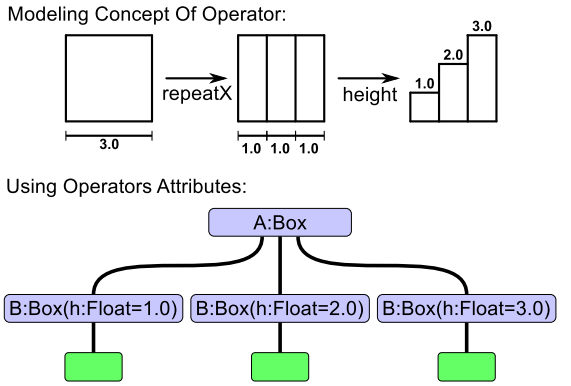



**Figure 2:** *Application of different rule types depending on the class of the non-terminal objects. Rule C provides an explicit parameter declaration, whereas rule A uses the implicit declared parameter $size_x$ of the non-terminal box which contains the current breadth. The upper right image illustrates the resulting geometry.*

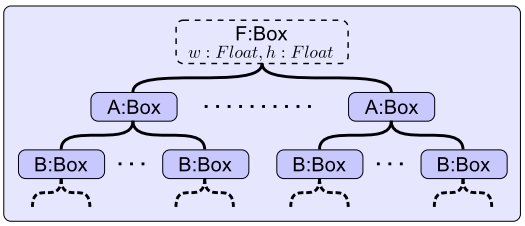
### 2.2. Operators and their Attributes

Operators are defined within each non-terminal class. Some of them create several new non-terminal objects, which can not be distinguished in some cases since they are associated with the same rule. Although this is the advantage of the procedural technique, because we apply only one rule to a bunch of new non-terminal objects, it is a drawback at the same time since repetitive structures are created. Therefore, those operators have to provide some kind of meta information through implicitly declared attributes while producing new non-terminal objects. Figure 3 outlines this concept by using an implicitly declared attribute *index* of the repeat operator.




```
A:Box-> repeatX(1,B(index));
B:Box(h:Float)-> height(h); renderBox();
```

**Figure 3:** *Using implicitly given information of the operator in order to get different results for each created non-terminal object.*



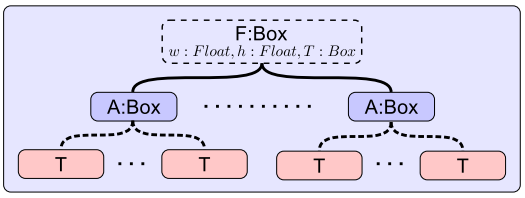

```
F:Box(w:Float,h:Float)-> repeatX(w,:A);
{
    A:Box-> repeatY(h,B);
    B:Box-> ··· Further subrules create window
}
```

**Figure 4:** *Modules are used to avoid parameter propagation and to encapsulate several rules into one unit. Please note, that in this example the rules A and B belong to the module F. We use the colon in ":A" in order to refer to a subrule and therefore, we do not need to explicitly pass the parameter h to subrule A since it is defined in the parent module.*

### 2.3. Dynamic Modules

As already mentioned, parameters are only declared in the scope of one rule and thus avoid dependencies to keep the grammar clear. The drawback of this convention is that parameters have to be propagated when they are given at an early stage of the evaluation process, but have to be used several rules later. In order to prevent parameter propagation we introduce *modules* (cf. Figure 4), which are rules encapsulating several subrules. For example, we want to create a dynamic facade by using the approximative tile width and height.

We can simply create a module that takes the width and height as parameters. Within the module several rules are applied for the structure. Normally, we would have to propagate the given height and width until they are used. Within the module, this is not necessary anymore, since we can directly read-only access these parameters in any of the subrules.




```
F:Box(w:Float,h:Float,T:Box)-> repeatX(w,:A);
{
    A:Box-> repeatY(h,@T);
}
//Usage:
A:Box-> spawnBox(F(2.0,3.0,B));
B:Box-> ··· Further subrules create windows
```

**Figure 5:** *Non-terminal symbols are used to define abstract structure templates. The @ sign states, that the called rule is a non-terminal provided by a parameter and not a defined rule within the current module.*

| Set 1 | Set 2 | Set 3 | |
|---|---|---|---|
| S→A | S→A | S→C(A,B) | C(X,Y)→XC(X,Y)X |
| A→aAa | A→aAa | S→C(A,C) | C(X,Y)→D(Y) |
| A→B | A→B | A→a | D(Y) →YD(Y)Y |
| B→bBb | B→cBc | B→b | D(Y) →Y |
| B→b | B→c | C→c | |
| abbba | accca | abbba or accca | |

**Figure 6:** *Production rules with a set of non-terminal symbols {A,B,C,D,X,Y}, a set of terminal symbols {a,b,c} and the start symbol S. The last row shows a possible resulting string.*

### 2.4. Abstract Structure Templates

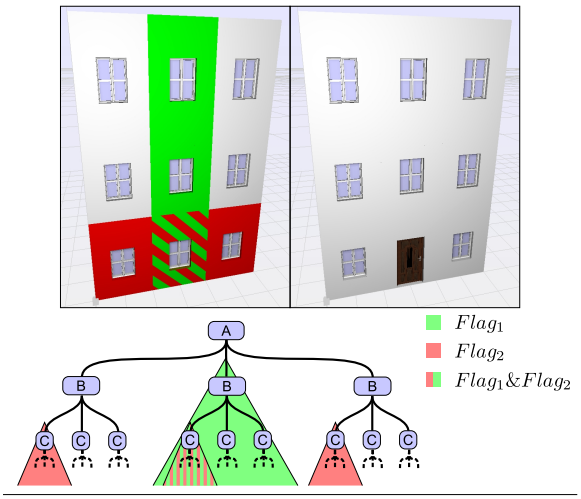
Modules are a good choice for enclosed, dynamic, procedurally generated objects in the scene, but the concept can be further extended by allowing non-terminal symbols as parameters. This creates *abstract structure templates* which can be reused at several places in the grammar. We use the term *abstract* since the evaluation of such templates does not yield a stand-alone object. The user has to pass non-terminal symbols for further evaluation. Regarding the last example (cf. Figure 4), we can create a module that just creates the grid tiling ignoring of what will be generated within the tiles. Figure 5 illustrates the use of non-terminal symbols as parameters.

By using abstract structure templates, rule explosion can be prevented in a more effective way. For illustration take a closer look at some sets of production rules in a string rewriting system (cf. Figure 6). The first two grammars of Figure 6 create similarly structured sequences of terminal symbols. To achieve this the whole grammar has to be duplicated and only the generated terminal symbols have to be exchanged in the grammar. Instead of this, we could create an abstract rule set such as grammar 3. By calling rule *C* with different non-terminal symbols we always achieve the same global structure but the resulting elements can still be further replaced. In this example, we only demonstrate how to

create the resulting strings of the first two grammars, but any non-terminal symbol could be used. This includes any other combination of already available non-terminal symbols (e.g. `S→C(B,C)`) and even other complex rule sets could be applied that generate new sequences of terminal symbols (cf. Figure 16).

### 2.5. Flags

The concept of *flags* makes local changes in the scene very easy. By convention undefined flags are associated with the value "false". A flag can be set to "true" whenever a new non-terminal object is created by a rule and it will be valid for the whole subtree that emerges from it, e.g. we do not need to pass the flag explicitly. Please note, that they differ from common explicitly declared local parameters, since a flag can either be defined at some certain non-terminal object within the scenegraph or not. In contrast to that, even a boolean variable would have three states, namely true, false or undefined. We prevent undefined variables or parameters by defining them explicitly within the rules. This is necessary, because if variables would also be available for a whole subtree there might occur dependencies between the rules. Some rule could want to read a variable that has never been created by the previously evaluated rules. Figure 7 shows the scope of the flags. Just like in *CGA shape*, rules can have conditions. Checking the defined flags within a condition allows for using other operations on a specific subtree to achieve local changes.




```
A:Box-> repeatX(2.0,B[If index is 1 set Flag1]);
B:Box-> repeatY(3.0,C[If index is 0 set Flag2]);
C:Box-> ··· If Flag1 and Flag2 are set, create a door.
          Otherwise create a window.
```

**Figure 7:** *Flags are used to apply local changes in the scenegraph. The green subtree has set $Flag_1$ whereas the red subtree has set $Flag_2$. The red and green subtree has both flags set, since $Flag_1$ was set by an early used operator and later another operator has set $Flag_2$.*

## 3. Derivation of a modeling language

In this chapter we derive our grammar from a general purpose language. We have chosen Python, since it already provides functional programming concepts which makes the example code of this section much easier. For later usage, we define $\mathcal{N}$ as the number of different non-terminal classes, $\mathcal{O}_i$ as the number of operators in non-terminal class $i$, where $i \in \{1,\ldots,\mathcal{N}\}$ and $\mathcal{R}$ as the number of rules that are defined in the grammar.

### 3.1. Basic Hierarchy

For completeness reasons we start the derivation by defining the base objects for building the scenegraph. Each object has one parent within the scenegraph and a transformation matrix that is relative to its parent (Figure 8).

```
class Object:
    def __init__(self):
        self.parent = None
        self.transformation = Matrix4x4f()
```

**Figure 8:** *Basic class for all objects in the scenegraph.*

For the derivation we will only take into account the non-terminal objects, since these instances have to be controlled by the grammar. Figure 9 makes clear that the non-terminals can be seen as group objects within the scenegraph, because rules working on them can create new child objects. Flags are recursively checked in the parent non-terminal objects returning true, if it is defined and false otherwise.

```
class NonTerminal(Object):
    def __init__(self):
        Object.__init__(self)
        self.children = []
        self.flags = []
    def addChild(self, child):
        child.parent = self
        self.children.append(child)
    def checkFlag(self, flag):
        if flag in self.flags:
            return True
        if self.parent:
            return self.parent.checkFlag(flag)
        return False
```

**Figure 9:** *Non-terminals build up the hierarchy.*

### 3.2. The System

The grammar makes use of implicitly declared attributes that are defined for classes and operators (cf. Sections 2.1, 2.2). From the systems point of view the attributes of the classes are local variables that are declared in $NTClass_i$. The attributes of an operator are represented by a dictionary within $operator_j$ in order to make them accessible in the grammar as we will see in a moment (cf. Figure 10).

```
class NTClass_i (NonTerminal):
    def init(self, ca_1, ···, ca_{n_i}):
        #implicit class attributes
        self.ca_x = ca_x  #∀x ∈ {1,···,n_i}
    def operator_j(self, arg_1, ···, arg_{m_{i,j}}):
        #implicit operator attributes (optional)
        oa = {'oa_1'=v_1, ···, 'oa_{l_{i,j}}'=v_{l_{i,j}}}
        #do something (predefined modeling concept)
```

**Figure 10:** *Non-terminal types that exist in the system ($j \in \{1,\ldots,\mathcal{O}_i\}$ and $i \in \{1,\ldots,\mathcal{N}\}$)*

As an example class we will introduce the *non-terminal boxes* (cf. Figure 11). The box contains three class attributes that define its size. A height operator is available that simply sets the size of the box along its local y-axis.

The most interesting part of this code snippet is the argument *create_non_terminal_box* of the repeat operator. Python provides functional programming concepts, namely *lambda functions*. Since we use class derivation for the association between rules and non-terminal objects, *create_non_terminal_box* is a lambda function creating a rule that has to be derived from the class *NTBox*, because the repeat operator creates new non-terminal boxes and initializes them with the calculated size and position. Passing the operator attributes *oa* to this lambda function enables us to use them within the grammar.

```
class NTBox(NonTerminal):
    def init(self, x, y, z):
        self.size_x = x
        self.size_y = y
        self.size_z = z
    def height(self, v):
        self.size_y = v
    def repeatX(self, size, create_non_terminal_box):
        oa = {'index'=0,'steps'=round(self.size_x/size)}
        new_width = self.size_x / oa['steps']
        transformation = self.transformation.clone()
        for oa['index'] in range(oa['steps']):
            new_nt = create_non_terminal_symbol(oa)
            new_nt.init(new_width,self.size_y,self.size_z)
            self.add_child(tmp_nt)
            new_nt.setTransformation(transformation)
            transformation.translateXLeftSelf(new_width)
            enqueEvaluation(new_nt)
```

**Figure 11:** *An example of a non-terminal class. A non-terminal box is defined by its size along each local axis. Operators are defined as functions within the class. They are later called by the grammar in order to perform any modification on the non-terminal object itself (e.g. height) or to spawn new non-terminal objects in the scene (e.g. repeat).*

### 3.3. The Grammar

This section covers the background of the grammar, i.e. how the Python code has to be designed in order to fit our modeling purpose. Please note, that the Python code explained in this section does not need to be written by the user, because a parser will create it from the well-defined syntax which is introduced in Section 3.4. Rules of the grammar are defined by class derivation of a specific non-terminal class (cf. Figure 12). The explicitly declared parameters of a rule are stored locally in order to make them available for any subrule. A subrule can access such a parameter by just following the references to the parent module (*pm*).

```
class Rule_h(NTClass_i):
    def __init__(self, pm, p_1=v_1, ···, p_{k_h}=v_{k_h}):
        NTClass_i.__init__(self)
        self.pm = pm
        self.p_x = p_x  #∀x ∈ {1,···,k_h}
    def evaluate(self):
        #user defined sequence of operators
```

**Figure 12:** *Rules that are created by the user in the grammar ($h \in \{1,\ldots,\mathcal{R}\}$). Notice, that $p_i$ can contain any parameter values including lambda functions. The $v_i$ denote the default values which are assigned to the parameters.*

Most important part of the rules is the evaluation function. The user can apply any operator to the current non-terminal object by just calling a certain function of the system, i.e. an inherited function of the specified non-terminal class. Figure 13 shows an example of a simple grammar that just splits a non-terminal box along the x-axis. The example clearly demonstrates that lambda functions are an easy way to make implicit operator attributes accessible in the grammar (e.g. the current index of the repeat operator).

```
class Rule_S(NTBox):
    def evaluate(self):
        self.size(10.0, 1.0, 0.1)
        self.repeatX(1.0, lambda oa: Rule_A(oa['index']))
class Rule_A(NTBox):
    def __init__(self, index):
        NTBox.__init__(self)
        self.index = index
    def evaluate(self):
        self.trace("Index" + str(self.index))
```

**Figure 13:** *Example of a grammar that splits a box along the x-axis. The resulting boxes will have a width of* 1.0 *and the associated rule will print its index.*

Passing a non-terminal symbol as parameter can be achieved analogously, i.e. a lambda function of the following form is passed as parameter:

```
lambda *parameters: Rule_Target(parameters)
```

Note, that the parameters are propagated to our target rule in this case, because the abstract structure template could also pass some parameter values to the given rule.

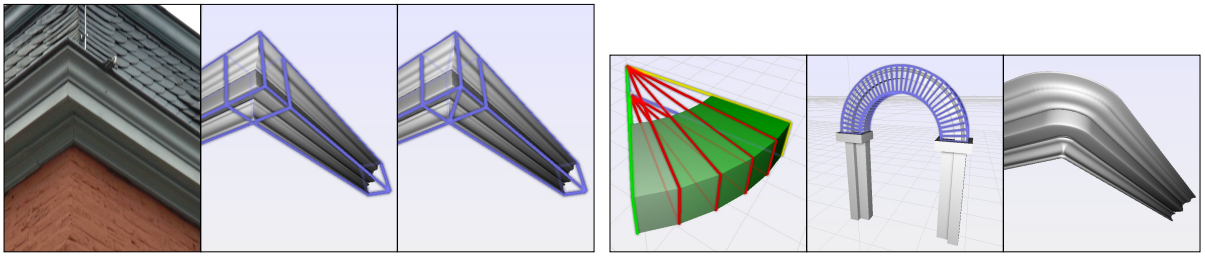
### 3.4. The Syntax

The syntax of our grammar is very close to *CGA shape* as seen in Figure 15. In contrast to their approach, our rules and parameters have to be of a specific type. Another difference is, that we define *rules*, which have a unique name within its *module*, where a module is just defined as a rule that has subrules. Please note, that parameters can be overloaded within the subrules. Each rule has a block of *conditions*, where each condition has a set of *operator sequences*.

```
$Rule:Type( parameter_1:Type, ···, parameter_n:Type)
    [ condition_1 ]
        < probability_1.1 > -> User defined sequence of operators
        ⋮
        < probability_1.p > -> User defined sequence of operators
    ⋮
    [condition_m]
        < probability_m.1 > -> User defined sequence of operators
        ⋮
        < probability_m.p > -> User defined sequence of operators
{
    Subrules
}
```

**Figure 15:** *Syntax of our grammar. Similar to CGA shape, we provide the use of parameters, conditions, and probabilities.*

The evaluation process will now be as follows. Whenever a new non-terminal object is created, we check the defined



```
$WallCornice:Box-> renderGeometry("Cornice.obj");
                   cornerFFD(45.0, $LeftCorner);
$LeftCorner:FFD-> renderGeometry("Cornice.obj");
```

```
$WallCornice:Box-> renderGeometry("Cornice.obj");
                   latheFFD(45.0, 2.0, $LeftCorner);
$LeftCorner:FFD-> renderGeometry("Cornice.obj");
```

**Figure 14:** *The left three images show a typical example of cornices going aroung an edge. In the second image, a new geometry has to be loaded in order to cover the sharp edge. In contrast, the third image shows that two FFDs solve the problem and that the geometry, used for the cornice along the wall, can be reused for the corner. The right three images show the creation of rounded edges by creating several FFDs. The given grammars correspond to the third and sixth image respectively. In both cases we cover an angle of 45 degrees. The whole angle of 90 degrees will be covered, since we apply this rule from both sides. When polygons are used as floor plans, the angles will be given as implicit operator attributes (cf. Figure 19).*

conditions of the associated rule sequentially. The operator sequences of the first condition that evaluates to true will be selected. Please note, that empty condition brackets always evaluate to true to serve as a default case. We will randomly choose one of the operator sequences belonging to the selected condition based on the given probabilities. Those are either defined in the range $[0.0, 1.0]$ or they are left empty. They are not allowed to sum up more than 1.0. All operator sequences of the same condition $x$ without probability will calculate their probability as $(1.0 - \sum_{i=1}^{p} probability_{x.i})/n$ where missing probabilities count as zero and $n$ denotes the number of missing probabilities.

The formal definition to reference a rule for further evaluation is done by the following expression:

$$Prefix\texttt{\$Rule}(exp_1,\ldots,exp_n)\,[cond_1 : flag_1,\ldots,cond_m : flag_m]$$

The prefix can be either ":" to call a subrule, "_" to call a root rule which belongs to no module, "@" to use one of the non-terminal symbols which are given in an explicit parameter or "../···../" to access a subrule of any parent module. Since the applied rule might have defined some parameters, we can pass that many values by using arbitrary expressions. We have seen in Sections 2.1 and 2.2, that both classes and operators have attributes, which are implicitly given in the grammar. In order to avoid name clashes we use a different prefix for accessing those attributes. For example, if we want to access the current breadth of a box, like in Figure 2, we can use $Class.size_x$ whereas if we want to access the current index during the non-terminal object creation of the repeat operator, as seen in Figure 3, we have to use $Operator.index$. Flags are also implicitly given in a subtree of the scenegraph, so that we have to use an additional prefix in this case. For example in Figure 7 we could just use $Flag.Flag_1 \&\& Flag.Flag_2$ as a condition in order to perform the demonstrated local change of placing a door in the respective tile.

## 4. Use Case: Architectural Modeling

In this section, we introduce some non-terminal classes which are suitable for the generation of architectural geometry. As already mentioned, modules can act as structure templates by taking advantage of non-terminal symbols as parameters or they just build a unit to generate a dynamic object in the scene. Therefore, the modeling strategy will be as follows:

1. Create different details by defining dynamic modules (i.e. windows, doors, decorations, etc.)
2. Create abstract structure templates for coarse layouts (i.e. modern, old, etc.)
3. Combine structures and details
4. Use flags for local changes of structures or details

Before we start the modeling of a facade, we introduce several operators. By following the *CGA shape* convention, we are able to use the component, repeat, split and transformation operators in a similar manner. In order to make cornices go around corners of buildings, non-terminal FFDs are created. Taking advantage of this deformation we are able to reuse the same geometry that was already placed on the wall of the facades (cf. Figure 14). There are currently two methods supported by our system to handle geometry in the case of a building corner. On the one hand, we can define a FFD to create sharp corners. This is done by taking the base xy-rectangle of a non-terminal box and extending one edge along the local z-axis. The result will be a prism shaped FFD that covers half of the given angle. On the other hand, our system provides an operator that creates several FFDs in order to approximate a round shape. It also takes the xy-rectangle of a non-terminal box and uses the local y-axis for the rotation. The operator needs the total angle and an approximative angle for each step. The calculation of the real angle for each segment is done analogously to the repeat operator. Figure 14 illustrates the operators for round and sharp building corners.

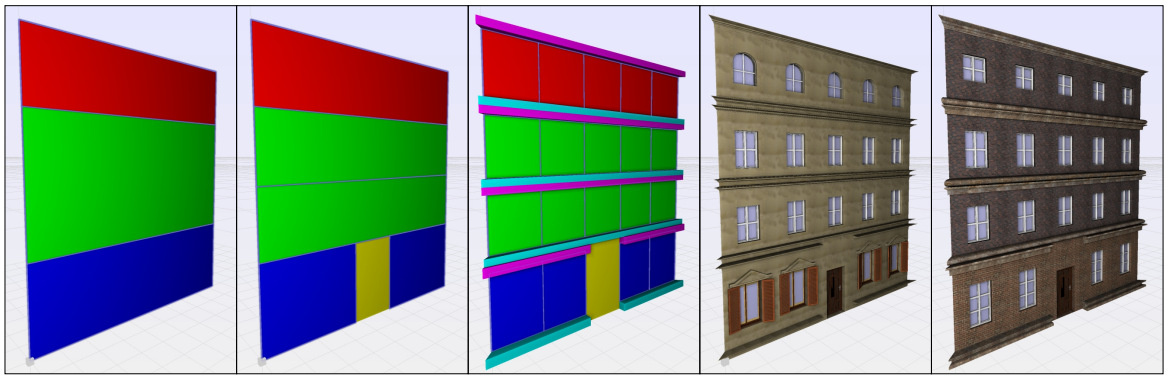



**Figure 16:** *Stepwise creation of the abstract structure template presented in Figure 17. First, we create different parts of the facade (a). Then, we split the middle part into the floors as well as the bottom floor into three parts in order to get a segment for a door (b). Finally, we create the cornices where the height of each cornice is related to the floor height by some given cornice ratio (c). We also create tiles as window areas in each floor. Images (d) and (e) show the usage of different dynamic modules based on our abstract structure template.*

```
$AbstractFacade:Box(th:Float, mh:Float, bh:Float,
                    cr:Float, la:Float, ra:Float, d:Bool,
                    $CorniceCorner:FFD, $CorniceWall:Box,
                    $WindowTile:Box, $DoorTile:Box)
    Split the box into the coarse features, i.e. bottom, middle and top floors. The
    split operator can be understood analogously to CGA shape. Therefore, each
    tuple uses a 0 to declare an absolute part and 1 to specify a relative part.
    ->splitY([(bh,  0, :$BottomFloor[BF]),
              (1.0, 1, :$MiddleFloors[MF]),
              (th,  0, :$Floor(la, ra)[TF])]);
{
    If we use a door, split the bottom floor. The left and right floor parts will not
    have a corner at the door side, so we use an angle of 0.0 degrees in this case.
    $BottomFloor:Box
        [d]->splitX([(1.0, 1, $Floor(la, 0.0)),
                     (1.8, 0, @$DoorTile),
                     (1.0, 1, $Floor(0.0, ra))]);
        [] ->spawnBox($Floor(la, ra));
    The middle floors should all have approximately the same height of mh meters.
    $MiddleFloors:Box->repeatY(mh, $Floor(la, ra));
    We overload la and ra internally in order to skip the corners beside the door tile.
    $Floor:Box(la:Float, ra:Float)
        ->splitY([(Class.size_y*cr/2.0, 0, :$Cornice[BC]),
                  (1.0,                 1, :$Tiling),
                  (Class.size_y*cr/2.0, 0, :$Cornice[TC])]);
    {
        The cornice will have the same depth as height. Further proceed with the
        left (0) and right (1) faces of the box.
        $Cornice:Box
            ->depth(Class.size_y);
              compFace([(0, $CreateCorner(la)),
                        (1, $CreateCorner(-ra))]);
              spawnBox(@$CorniceWall);
        Create the FFD corners with the given angle. If the angle is negative, we
        create the right corner.
        $CreateCorner:Box(a:Float)
            [a > 0]->cornerFFD(a, @$CorniceCorner);
            []      ->tX(Class.size_x); rY(180);
                      cornerFFD(a, @$CorniceCorner);
        All window tiles should all have approximately the same width of 1.8 meters.
        $Tiling:Box->repeatX(1.8, @$WindowTile);
    }
}
```

**Figure 17:** *Practical use of an abstract structure template. The first three parameters define the height of the top (th), middle (mh) and bottom (bh) floors. This is followed by a cornice ratio (br), the left (la) and right (ra) angle for the edges of the cornices and a boolean, if we want to use a door (d). The last four parameters are non-terminal symbols for further evaluation. Note, that this abstract structure template also sets some flags to distinguish the tiles for the top (TF), middle (MF) and bottom (BF) floors as well as the top (TC) and bottom (BC) cornices. The flag color referes to the Figure 16(c).*

```
$Facade01:Box(la:Float, ra:Float, d:Bool)
  ->spawnBox(_$AbstractFacade(2.5, 2.8, 3.2,
                              0.2, la, ra, d,
                              :$CorniceCorner,:$CorniceWall,
                              :$WindowTile,:$DoorTile));
{
    $CorniceCorner:FFD
        [Flag.TC]-> renderColor(1, 0, 1);
        [Flag.BC]-> renderColor(0, 1, 1);
    $CorniceWall:Box
        [Flag.TC]-> renderColor(1, 0, 1);
        [Flag.BC]-> renderColor(0, 1, 1);
    $WindowTile:Box
        [Flag.TF]-> renderColor(1, 0, 0);
        [Flag.MF]-> renderColor(0, 1, 0);
        [Flag.BF]-> renderColor(0, 0, 1);
    $DoorTile:Box-> renderColor(1, 1, 0);
}
```

**Figure 18:** *Simple application of the abstract structure template of Figure 17 in order to create Figure 16(c).*

```
Load a city map and associate a mesh rule with it.
$S:Box->size(100.0, 100.0, 0.0);
       spawnMesh($CityPlan, "CityPlan.obj");
Decompose the mesh into its polygons.
$CityPlan:Mesh->compPoly($FloorPlan);
Extrude the polygons by placing boxes at the lines. The width equals the length
of the line, the height is taken from the extrude operator (10.0) and the depth is
degenerated to 0. In order to let the facade cornices fit perfectly well along the
building edges, we use the outer angles of the polygon. Note, that the gap at the
corners is 180 degrees less than the outer angle itself. A door will be placed within
the first side of the extruded polygon (Operator.index == 0).
$FloorPlan:Polygon->extrude(10.0, $Facade01(Operator.la / 2 - 90,
                                            Operator.ra / 2 - 90,
                                            Operator.index == 0));
```

**Figure 19:** *Using the facade of Figure 17 on several floor plans. Please note, that the extrude operator provides the outer left (Operator.la) and right (Operator.lr) angles of the current line as well as the current index (Operator.index) as implicit attributes while creating the different sides as non-terminal boxes.*

Once the user has created several dynamic modules (cf. Figure 20 for some examples), he is able to reuse them in a given abstract structure template. Therefore, we take a closer look at an example creating an abstract structure template of a simple facade in Figure 17. We use comments to explain the code snippet as well as a sequence of images in Figure 16 to illustrate different stages of the evaluation.

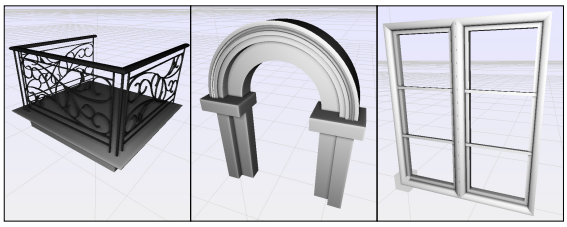


**Figure 20:** *Several dynamic modules, which will change their appearance on different input values.*

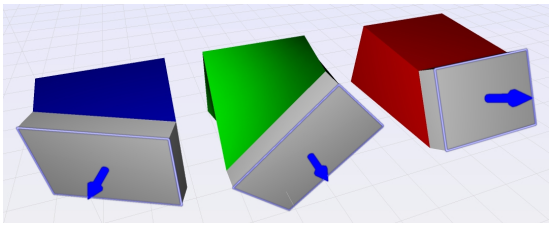
After the creation of abstract structure templates and other dynamic modules, we are able to combine them. In Figure 18 we just draw different colored boxes and FFDs in order to visualize different parts that are created by the *AbstractFacade* rule. Instead, we could use any dynamic module to create the details. This could, for example, result in the the last two images of Figure 16. Please note, that previous approaches would need to copy the *AbstractFacade* rule for each combination of different windows, doors or cornices. In contrast, our abstract structure templates allow for keeping the rules untouched and reusing them several times.

In order to generate a building or even an entire city, we are able to load *non-terminal meshes* which can be seen as the floor plans. A component operator will disassemble the mesh and spawn a *non-terminal polygon* for each face. At this point we can easily extrude each polygon by spawning non-terminal boxes for the sides which are zero scaled along the z-axis (cf. Figure 19). In general, this method can be understood analogously to the component split of *CGA shape*, except that we distinguish different geometric objects by the non-terminal classes. The advantage of our system is here, that special operators could be defined for those geometry classes, e.g. a triangulation operator for *non-terminal polygons* or a subdivision operator for *non-terminal triangles*. These examples clarify, that the distinction of non-terminal objects by different classes is an substantial feature if the system is supposed to be easily extendable in order to make it fit to other modeling domains.

## 5. Use Case: Plant Modeling

In contrast to architecture, plants are generated by simulating a growth process. Our approach takes the idea of generalized cylinders as they are used by Prusinkiewicz et al. [PL96] and transfers this concept to a controlled creation of FFDs. Therefore, our basis is also a LOGO-style turtle that can roll, yaw, pitch and move forward.

With generalized cylinders the user can insert a new control point for the creation of a Bezier curve at the current position and orientation of the turle. Since we are using FFDs for the deformation of geometry, we adapt this idea in the following way. At each position we can specify a given width and height which states the size of a cross-sectional rectangle that is orthogonal to the forward vector. At any point we can specify, that the current rectangle will be used



```
$S:Box->size(3,2,0); spawnFFDTurtle($Yaw);
$Yaw:FFDTurtle->forward(2); yaw(-25); setFront();
                renderColor(0,0,1); spawnFFDTurtle($Forward);
$Forward:FFDTurtle->forward(0.5); setFront(); renderColor(1,1,1);
```

**Figure 21:** *Turtle FFDs: We first set the size of the turle to a width of 3 and a height of 2 using this rectangle as the back of the current FFD. Then we go forward by a value of 2 and yaw (blue), roll (green), pitch (red) the turtle respectively without changing its size. Using this as the front rectangle yields the illustrated FFD shapes. The grammar below only demonstrates the yaw transformation, since roll and pitch would be created analogously.*

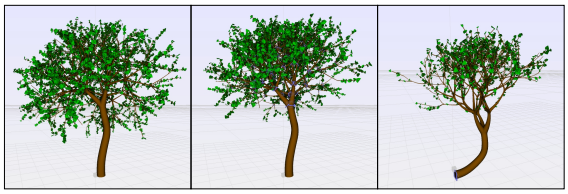


**Figure 22:** *Recursive application of rules leads to tree structures. Please note, that even the branch transitions look very smooth due to the Turtle FFD principle. In the middle and in the right image we use a vector pointing upwards as parameter in order to simulate a growth into the direction of the sunlight.*

as front or back of the FFD as seen in Figure 21. The growth process can then be simulated by using random values and recursive application of rules like illustrated in Figure 22. A simple upward directed vector as parameter can be utilized to guide the growing direction. Loading geometry that holds an extruded contour of a certain part of a plant, we can create similar flowers as generated by *L-studio/cpfg* as long as they do not need to simulate the propagation of internal information as proposed by Prusinkiewicz et al. [PHL*09].

Using abstract structure templates in plant modeling allows for an easy application of existing grammars by just combining them (cf Figure 24, Figure 23). Experienced users with knowledge about plant modeling can provide several rules for tree structures, fruits, or leafs whereas beginners just use and combine them to get a variety of trees. Previous approaches would have to copy the base tree structures for each possible combination of dynamic modules. The grammar would become very long and thereby confusing. Furthermore, changes in the base structure would have to be done in each copy. Our approach of using non-terminal symbols as parameters serves as a elegant solution to this problem holding the syntax clear and unambiguous.

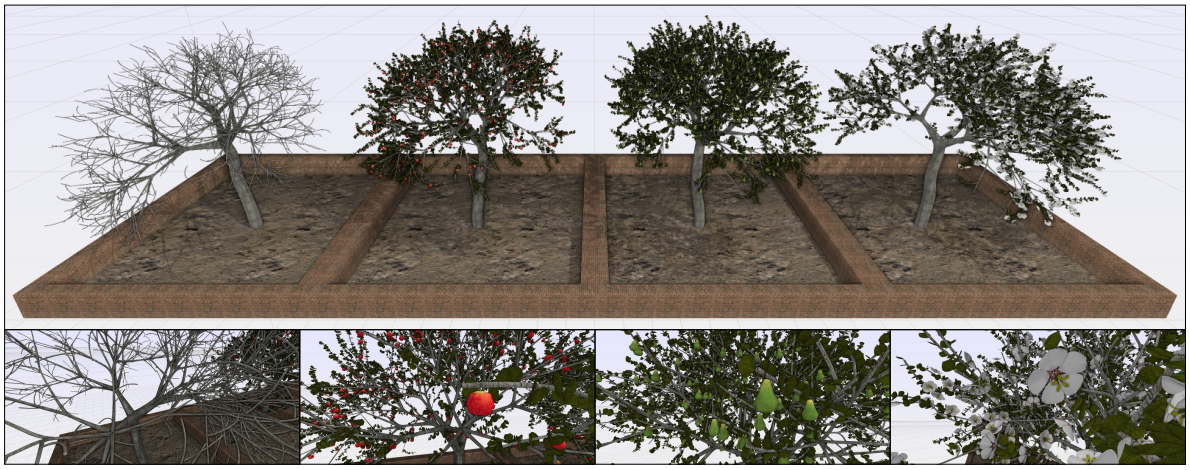



**Figure 23:** *These images demonstrate the usage of abstract structure templates in plant modeling. (a) A small rule set defines a basic tree structure. (b-d) Dynamic modules serve for the creation of leafs, apples, pears and petals which are passed as parameters to the basic tree. This enables various combinations by just changing a few non-terminal symbol parameters in the grammar (cf. Figure 24).*

```
Experienced users create abstract structure templates of trees as well as dynamic
models of fuits and leafs.
$BaseTree:FFDTurtle($Leaf:FFDTurtle, $Fruit:FFDTurtle)-> ···
$Leaf :FFDTurtle-> ···
$Petal:FFDTurtle-> ···
$Apple:FFDTurtle-> ···
$Pear :FFDTurtle-> ···
Beginners benefit of abstract structure templates and dynamic modules, since they
only need to combine different grammars without understanding them in detail.
$AppleTree:FFDTurtle-> spawnFFDTurtle($BaseTree($Leaf, $Apple));
$PearTree :FFDTurtle-> spawnFFDTurtle($BaseTree($Leaf, $Pear));
$PetalTree:FFDTurtle-> spawnFFDTurtle($BaseTree($Leaf, $Petal));
```

**Figure 24:** *Creation of various trees by combining abstract structure templates and dynamic modules in plant modeling.*

| Fig. | Eval. | Rendering | Triangles | Inst. |
|---|---|---|---|---|
| 25 left | 27 ms | 15 ms | 276226 | 11035 |
| 1 | 267 ms | 98 ms | 1346280 | 58653 |
| 26 | 1272 ms | 520 ms | 6914247 | 281607 |
| 23 top | 2443 ms | 1005 ms | 3114704 | 556629 |

**Table 1:** *The evaluation time is related to the generation of the whole scene. Additionally, we list the rendering time, the number of triangles, and the number of instances.*

## 6. Implementation

In order to demonstrate the integration into professional 3D animation software, we provide a little script package in Python related to this paper [Gen10]. We chose Houdini from Side Effects Software [Hou09], which suits perfectly well on our procedural modeling approach.

For runtime measurements, we use the C++ version of our system which is much faster. This leads to very fast scene assembling, since all the changes result in a direct visual feedback. Some statistics of the grammars that are shown throughout the paper are listed in Table 1. We run our application on a Intel Core i7 with 2.67GHz with 6 GB ram and a GeForce GTX 285 with 1 GB ram. Since our application is not parallelized so far, we only utilize one of the cores. Additionally, the rendering is currently not optimized, i.e. we issue as many draw calls as instances in the scenegraph. This bottleneck could be solved by uploading the geometry to the graphics card before rendering it.

## 7. Discussion

**Expressiveness** — A general disadvantage of procedural modeling is, that it is usually limited to one specific modeling domain. Since our language provides several non-terminal classes, we are not restricted to a certain modeling concept. The advantage of this method is not only that different use cases are covered, but that they can be combined as seen in Image 1. This opens a whole range of new possibilities for a dynamic scene creation. We currently only use 6 non-terminal classes (i.e. Box, FFD, FFDTurtle, Mesh, Polygon and Triangle), but the system could easily be further extended by new classes such as spline curves or NURBS surfaces. We also applied our grammar to the reconstruction of different object types as seen in Figure 25.

**Local Changes** — During the creation of an abstract structure template, flags are a simple way to partition the different created non-terminal objects. If the designer of the template also provides an example with a color scheme like seen in Figure 16(c), the flags can be utilized easily.

**Real-time Rendering** — All images in this paper are rendered with our real-time application. Since we currently have no level-of-detail techniques implemented, the framerate may drop in cases of massive scenes (cf. Figure 26).

**Figure 25:** *The left two examples show the reconstruction of a real building that we created with our grammar. Please note, that the round hallways are also generated by the system using FFDs and that the details are dynamic objects that build a unit within the grammar. Therefore, the windows on each floor are based on the same module, but called with different parameters. The right Figure shows that our FFD approach is well-suited for the creation of plants. In this case we modeled a lily analogue to the algorithms presented by Prusinkiewicz et al. [PKMH00].*

**Usability** — The learning curve of our grammer is similar to that of scripting languages. The human readable syntax supports the designer to understand grammars of other persons very fast. Especially, dynamic modules and abstract structure templates serve as a good interface for shared work, because they are closed units. Other designers do not need to analyze the whole grammar for variables, which have to be set when using a specific rule, since we do not have parameter dependencies between the rules. *Writing* complex grammars can still take a while, even for experienced users. For example creating the base structure and all modules (windows, doors, cornices), used throughout the images of Figure 16, took approximately 1 hour. In contrast, *using* non-terminal symbols as parameters brings the combinatorial possibilities to a whole new level. Even beginners are now enabled to combine nontrivial structures and dynamic modules within seconds thereby creating complex scenes.

**Limitations** — Currently, our system does not provide any geometric queries. In general, this could be achieved nearly analogously to *CGA shape*, i.e. using the conditions to query any generated geometry. In order to get the queries more performant, *CGA shape* can be restricted to only checking intersections between bounding boxes. From this point of view, queries become a little bit more complex in our system, since a non-terminal class could represent any geometrically meaningful object. For example, intersections with any free-form deformed object could be checked by first using only the convex hull of the FFDs.

Currently, we are limited to trilinear FFDs, since our C++ version of the system uses the vertex shader of the graphics card to apply the deformation in real-time, although the scenes might consist of several thousand FFDs. We also experimented with FFDs of higher degree, but we found that trilinear FFDs are sufficient for our use cases; especially, if low resolution polygon geometry is loaded, any high resolution FFD would not contribute to the quality of the resulting geometry unless the geometry itself is subdivided. Instead, using the low resolution polygon geometry in several trilinear FFDs, which share a common face, serves for a similar effect when rounded objects have to be generated.

## 8. Future Work

In future work, we would like to extend $G^2$ by other non-terminal classes which address different modeling scenarios reaching from detail creation like furniture to the generation of massive scenes which include nature and man-made artifacts. Furthermore, attaching semantic information to generated objects could help to create even more abstract grammars, e.g. an ivy grammar only needs to know that it can climb along a stone surface, no matter if it is part of a facade or if it is a stand-alone wall. Another extension could be to interchange signals between the created non-terminal objects, e.g. a plant hits a wall during its growth process; this could trigger a signal event within the wall in order to modify it at a certain position.

## 9. Conclusions

This paper presents $G^2$, a novel procedural modeling language combining the generative power of shape grammars with common modeling concepts. We introduce the idea of having several non-terminal classes in order to encapsulate different modeling strategies like working with boxes or free-form deformations. Abstract structure templates are presented allowing for well-defined non-terminal object arrangement of more complex patterns. Local changes can also be applied by adding flags during the scene creation. We combine these concepts in a grammar with a human readable syntax, which makes the modeling process very intuitive.

## Acknowledgement

This work was supported in part by NRW State within the B-IT Research School.

## References

[CDM*02] CUTLER B., DORSEY J., MCMILLAN L., MÜLLER M., JAGNOW R.: A procedural approach to authoring solid models. *ACM Trans. Graph. 21*, 3 (2002), 302–311.

[EMP*02] EBERT D. S., MUSGRAVE K. F., PEACHEY D., PERLIN K., WORLEY S.: *Texturing & Modeling: A Procedural Approach, Third Edition (The Morgan Kaufmann Series in Computer Graphics)*. Morgan Kaufmann, December 2002.

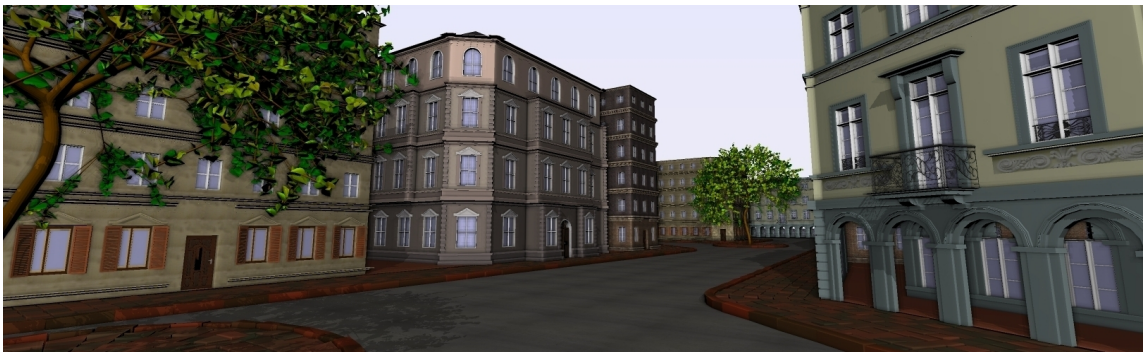


**Figure 26:** *This image shows a massive scene of a street with various houses. Rectangular structures, such as buildings, are generated by recursive application of decomposition rules based on non-terminal boxes whereas organic structures, e.g. trees, are better simulated as growth process based on FFD turtles.*


[Gen10] Generalized grammar. http://www.graphics.rwth-aachen.de/software/generalizedgrammar/, April, 2010.

[GK07] GANSTER B., KLEIN R.: An integrated framework for procedural modeling. In *Spring Conference on Computer Graphics 2007 (SCCG 2007)* (Apr. 2007), Sbert M., (Ed.), Comenius University, Bratislava, pp. 150–157.

[GML] *Generative Mesh Modeling / Havemann, Sven*. PhD thesis.

[HBW06] HAHN E., BOSE P., WHITEHEAD A.: Persistent real-time building interior generation. In *Sandbox '06: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames* (New York, NY, USA, 2006), ACM, pp. 179–186.

[Hou09] Side effects houdini. http://www.sidefx.com/, October, 2009.

[LG06] LARIVE M., GAILDRAT V.: Wall grammar for building generation. In *GRAPHITE '06: Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia* (New York, NY, USA, 2006), ACM, pp. 429–437.

[LWW08] LIPP M., WONKA P., WIMMER M.: Interactive visual editing of grammars for procedural architecture. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers* (New York, NY, USA, 2008), ACM, pp. 1–10.

[MGDH04] MARTINET A., GALIN E., DESBENOIT B., HAKKOUCHE S.: Procedural modeling of cracks and fractures. In *SMI '04: Proceedings of the Shape Modeling International 2004* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 346–349.

[MWH*06] MÜLLER P., WONKA P., HAEGLER S., ULMER A., GOOL L. V.: Procedural modeling of buildings. *ACM Trans. Graph. 25*, 3 (2006), 614–623.

[PHHM97] PRUSINKIEWICZ P., HAMMEL M., HANAN J., MĚCH R.: Visual models of plant development. In *Handbook of formal languages, vol. 3: beyond words* (New York, NY, USA, 1997), Springer-Verlag New York, Inc., pp. 535–597.

[PHL*09] PALUBICKI W., HOREL K., LONGAY S., RUNIONS A., LANE B., MĚCH R., PRUSINKIEWICZ P.: Self-organizing tree models for image synthesis. *ACM Trans. Graph. 28*, 3 (2009), 1–10.

[PHM00] PRUSINKIEWICZ P., HANAN J., MĚCH R.: An l-system-based plant modeling language. In *Applications of Graph Transformations with Industrial Relevance*, Lecture Notes in Computer Science. 2000, pp. 258–261.

[PJM94] PRUSINKIEWICZ P., JAMES M., MĚCH R.: Synthetic topiary. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1994), ACM, pp. 351–358.

[PKMH00] PRUSINKIEWICZ P., KARWOWSKI R., MECH R., HANAN J.: L-studio/cpfg: A software system for modeling plants. In *AGTIVE '99: Proceedings of the International Workshop on Applications of Graph Transformations with Industrial Relevance* (London, UK, 2000), Springer-Verlag, pp. 457–464.

[PL96] PRUSINKIEWICZ P., LINDENMAYER A.: *The algorithmic beauty of plants*. Springer-Verlag New York, Inc., New York, NY, USA, 1996.

[PM01] PARISH Y. I. H., MÜLLER P.: Procedural modeling of cities. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2001), ACM Press, pp. 301–308.

[PMKL01] PRUSINKIEWICZ P., MÜNDERMANN L., KARWOWSKI R., LANE B.: The use of positional information in the modeling of plants. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2001), ACM Press, pp. 289–300.

[SP86] SEDERBERG T. W., PARRY S. R.: Free-form deformation of solid geometric models. *SIGGRAPH Comput. Graph. 20*, 4 (1986), 151–160.

[Sti75] STINY G.: *Pictorial and Formal Aspects of Shape and Shape Grammars*. Birkhauser Verlag, Basel, 1975.

[Sti80] STINY G.: Introduction to shape and shape grammars. *Environment and Planning B 7* (1980), 343–361.

[VAW*09] VANEGAS C. A., ALIAGA D. G., WONKA P., MÜLLER P., WADDELL P., WATSON B.: Modeling the appearance and behavior of urban spaces. In *Eurographics 2009 - State of the Art Reports* (2009), pp. 1–16.

[Whi06] WHITE C.: King kong: the building of 1933 new york city. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches* (New York, NY, USA, 2006), ACM, p. 96.

[WMV*08] WATSON B., MÜLLER P., VERYOVKA O., FULLER A., WONKA P., SEXTON C.: Procedural urban modeling in practice. *IEEE Computer Graphics and Applications 28*, 3 (2008), 18–26.

[WWSR03] WONKA P., WIMMER M., SILLION F., RIBARSKY W.: Instant architecture. *ACM Trans. Graph. 22*, 3 (2003), 669–677.

[Zal04] ZALZALA J.: Procedural building destruction for "the day after tomorrow". In *SIGGRAPH '04: ACM SIGGRAPH 2004 Sketches* (New York, NY, USA, 2004), ACM, p. 144.