Collaborative Modeling with Symbolic Shape Grammars

Martin Ilčík[1], Michael Wimmer[2]
[1,2]TU Wien
[1,2]{ilcik|wimmer}@cg.tuwien.ac.at

Generative design based on symbolic grammars is oriented on individual artists. Team work is not supported since single scripts produced by various artists have to be linked and maintained manually with a lot of effort. The main motivation for a collaborative modeling framework was to reduce the script management required for large projects. We achieved even more by extending the design paradigm to a cloud environment where everyone is part of a huge virtual team. The main contribution of the presented work is a web-based modeling system with a specialized variant of a symbolic shape grammar.

Keywords: *collaboration, symbolic shape grammar, generative modeling*

INTRODUCTION

Procedural modeling is a popular approach in GIS, urban planning and for CGI film effects. In such applications detailed architectural content has to be generated on a very large scale, hence shape grammars with symbolic matching (Stiny 1982) are preferably used to simplify the modeling process. Procedural representation of objects allows to use a simple notation for complex symmetries and patterns. A whole class of objects can be instantiated from a single grammar. However, large scenes contain a variety of object classes, each exhibiting unique patterns. The number and complexity of involved grammars increases with each new class. Manual management and efficient re-usage of the designs soon becomes problematic.

Additional difficulties arise when the creative process is not limited to a single user. Our experience with the standard industry tool CityEngine by Esri [1] indicates that team work with a repository of grammars is really difficult. Production rules have to be manually imported and referenced across grammars. Constant manual synchronization between team members creates a lot of work overhead. Merging of updates requires a dedicated expert just for the repository management. Efficient collaborative tools are a big challenge for the field of generative design.

Contributions

We have created a collaborative design framework based on the theoretic work of Müller et al. (2006) which defined the successful CityEngine. Many aspects of the single-user oriented concept needed to be revised. The most innovative features of the proposed approach are:

1. A cloud repository of generative grammars.
2. A new language feature: procedural queries that automatically apply rules from the cloud.
3. No management overhead for team projects.
4. No special syntax, grammars are simple C# scripts.

Our experimental framework called Michelangelo [2] implements the presented features. All modeling is performed on the web, hence all grammars get immediately stored in Michelangelo's cloud. The main

theoretic novelty of the grammar definition is the concept of procedural queries which replace symbols as the main control element of the derivation process. Queries are the key to collaborative modeling. They automatically seek and link rules from the cloud and determine the order of their execution.

COLLABORATION FOR GRAMMARS

None of the existing grammar-based modeling tools was created with collaboration in mind. However, it is mainly the participation of artists with heterogeneous skills and ideas (Edmonds et al. 1994) that stimulates variety and drives the complexity of results to a new level. An optimal interface to a grammar with 5 rules looks different than the interface to 100 rules (Gips 1999). A team of artists will easily produce several such grammars. A cloud repository, once properly filled, will host several orders of magnitude higher amount of rules. With CityEngine, being the most popular tool implementing a symbolic shape grammar, we have analyzed the state-of-the-art pitfalls of collaboration in generative design.

CityEngine

CityEngine stores designs in a basic digital repository. The best practice for team work is to fragment new designs into short, well organized grammar snippets. Their connections need to be declared directly in the grammar files [3]. I.e., to link a symbol to a rule from a different grammar it requires manual import and mapping.

```
import B: "book.cga"
Book --> B.Book
```

In a collaborative environment, such static assignment has several drawbacks:

1. The whole script becomes invalid if someone renames the entry rule of the imported grammar. Figure 1 illustrates the case on a simple hierarchy of grammars. In the real world its complexity would be much higher. Renaming the *Book* rule to the more specific *HardcoverBook* requires to update a considerable number of grammars (in Figure 1 marked dark orange). Keeping permanent overview and precise synchronization is not feasible even for small projects.
2. If someone designs an alternative grammar for an item, other team members willing to use it must manually import the new file or rule into their grammars and include a conditional to decide when to use which source.

```
import B1: "book.cga"
Book1 --> B1.Book
import B2: "book2.cga"
Book2 --> B2.Book
#decide on Book1 or Book2
Book --> case { ... }
```

3. The control is fully passed to the linked grammar, which acts like a black-box. It is not possible to override its rules or imports. We consider this limitation as the main obstacle for emergence of alternative designs within a collaborative environment. It is not possible to alter materials or model parts created by imported grammars without changing the original script. If it comes from a different author, it is necessary to read and understand it first. Rewriting someone else's grammar file is a bad practice which is mostly avoided by creating an edited copy leading back to problem 2.

During intense artistic work the administration of a complex hierarchy of grammars can easily overwhelm the authors. To support efficient team work it is necessary to simplify the control over the grammars.

Imperative vs. declarative design

Starting with the first shape grammars (Stiny 1982) and L-Systems (Prusinkiewicz and Lindenmayer 1990), through GML (Havemann 2005) and CGA-Shape (Müller et al. 2006), up to the most recent PCGR (Silva et al. 2015) and CGA++ (Schwarz and Müller 2015), all theoretic concepts are very close to

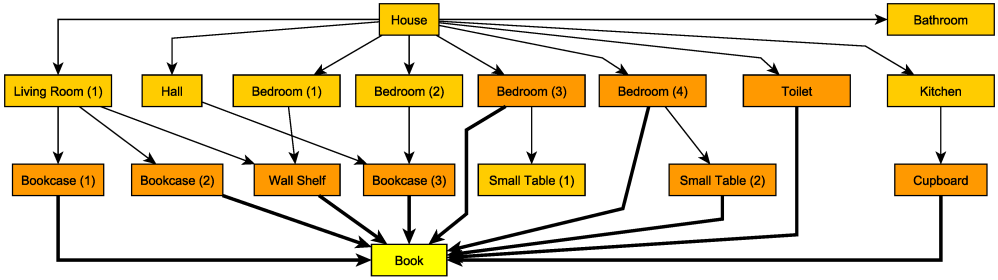


Figure 1
Example of a hierarchy of grammars. Renaming the book rule (yellow) or adding a new one would require to manually introduce changes in all grammars linking the book grammar (orange).

common imperative programming languages. They demand that the designer specifies a sequence of modeling steps which creates the desired object. Fixed linking of production rules is the main obstacle for collaborative modeling. Artists without a computer science background are not skilled to assemble and link complex scripts, however they can easily describe their goals. Generative design tools seem to overlook this fact, but for example in logic programming goal-driven computation is quite popular. It utilizes the declarative programming paradigm (Kowalski 1979). Applied to generative design it means that instead of specifying how, the designer should mainly describe what to generate. The imperative "*then continue with the modeling operation X*" attached to each modeling operation gets replaced by the declaration "*then try to transform the output shape to S*".

Replacing a pointer to a particular rule *X* by a shape description *S* of the desired result is a subtle syntactic difference with major impact on the collaborative modeling process. Production rules are still listed in grammars, but there are no explicit connections between them. Such freedom allows to combine grammars automatically and arbitrarily in the background without exposing the complex management tasks to the artist. At the same time, the specification of shape descriptions introduces a query mechanism to the grammar. Using a list of simple goals, e.g. *chair*, *wooden*, the author can specify what a non-terminal shape should become. If the repository contains a design that fulfills the given goals it will be embedded automatically.

DECLARATIVE SHAPE GRAMMAR

The proposed declarative shape grammar is based on a symbolic set grammar over an algebra of shapes (Wonka et al. 2003). String symbols provide only limited meaning to shapes. Therefore, in the declarative grammar they are replaced by a more complex structure for shape description with semantics consisting of:

- *G* a set of string goals.
- *T* a set of tags T∩G={}. Tags are goals which were fulfilled.
- τ a Boolean flag marking terminal shapes.
- *A* a set of named attributes (numeric or string).

Rules must be adapted to the semantics as well. Instead of a symbol, a rule in the declarative grammar contains a set of predicates over the shape semantics on its left side. The next section provides an example. There are predicates available for goals, tags and attribute names (with *s* being semantics of the evaluated shape):

- *IfGoal*(s, $g_0 \ldots g_n$) = $s.G \supseteq \{g_0 \ldots g_n\}$
- *IfTag*(s, $t_0 \ldots t_n$) = $s.T \supseteq \{t_0 \ldots t_n\}$
- *IfHas*(s, $a_0 \ldots a_n$) = $s.A_{keys} \supseteq \{a_0 \ldots a_n\}$

All of the predicate functions also have negated counterparts with the prefix *IfNot*. For predicates based on attribute values there is a general predicate *If* which directly evaluates the given function *f* de-

fined over the semantics *s*.

- *If(s, f) = f(s)*

Derivation step

In imperative approaches the shape symbol directly specifies the next rule to be applied.

```
Symbol1 --> commands Symbol2
```

In the declarative grammar the set of tags defines the current state while the set of goals specifies the final state to be reached. Hence, the equivalence of symbols for rules matching has to be replaced by subset relations over the shape semantics.

```
Predicates(Semantics1) --> commands
 ↪ Semantics2
```

Using the predicates described above one defines conditions for the rule to be applicable to a shape. A special command *Fulfills* checks for existence of goals (using *IfGoal*) and converts them to tags.

A typical condition is that a rule can be applied to a shape only if the set of its goals contains all goals fulfilled by the rule. Further requirements can be given for and shape attributes. E.g., a rule that initiates table decoration in a certain style may require the shape to be tagged *restaurant*, *table* with goals *romantic*, *dinner* and its size should be less than 0.8 meters in each dimension:

IfIs(s, "restaurant", "table") ∧
IfGoal(s, "romantic", "dinner") ∧
Fulfills(s, "setting") ∧
If (s.Size.X < 0.8 ∧ s.Size.Z < 0.8)

IMPLEMENTATION AND USAGE EXAMPLE

The system is implemented as a cloud service called Michelangelo [2]. For the modeling basis we use a variant of the CGA-Shape by (Müller et al. 2006). Grammars are written as C# scripts providing a familiar environment and powerful features at once. Let us demonstrate the syntax using a very simple example of a chair. The main difference to standard symbolic grammars is that instead of a symbol the user sets goals (a set of strings) for the axiom and shapes. They represent semantics of the desired modeling result.

```
new Axiom("chair", "wooden").Size
 ↪ (0.4,0.8,0.4);
```

Rules modify the model to match the given goals and introduce more specific goals in a top-down manner. The following rule divides the shape vertically into three parts using relative sizing with respect to the shape extent on the *Y* axis. It also adds a new goal to each one of them. A randomized numeric attribute is assigned to the *support* part using the *Rnd(min, max)* function.

```
new Rules.Split(Axis.Y,
  Relative(0.4).Goal("support").
   ↪ Attribute("Leg.Size", Rnd
   ↪ (0.03,0.05)),
  Relative(0.05).Goal("seat"),
  Relative(0.55).Goal("lean"))
.Fulfills("chair");
```

Contrary to the standard symbolic matching, we introduce a goal-driven declarative approach. Rules are sought and applied to each shape so that its goals get fulfilled. The previous example rule fulfills the goal *chair*, thus it can be applied to the axiom. Fulfilled goals become tags, hence all shapes produced by the rule will be tagged *chair*. Tags and numeric attributes accumulate in the shapes over the whole derivation process.

Most of the rules fulfill at least one goal, but it is not required. Matching of rules to shapes is further controlled by predicates: *IfGoal* checks a goal without fulfilling it, *IfIs* checks a tag (i.e. a fulfilled goal), *IfHas* checks a named attribute. More complex predicates are explained in the documentation of Michelangelo.

A local *goal* (note the lowercase) can be fulfilled only by the same grammar to avoid interference with other goals in the cloud. The compiler takes care of setting the corresponding *Fulfills* to be local as well. Shapes marked *Void()* become terminals with no geometry.

```
new Rules.Split(Axis.X,
```

```
        Absolute(@Double("Leg.Size")).
         ↪ goal("split-legs"),
        Relative(1).Void(),
        Absolute(@Double("Leg.Size")).
         ↪ goal("split-legs"))
.Fulfills("support");

new Rules.Split(Axis.Z,
        Absolute(@Double("Leg.Size")).
         ↪ Goal("leg"),
        Relative(1).Void(),
        Absolute(@Double("Leg.Size")).
         ↪ Goal("leg")
.Fulfills("split-legs");
```

The previous pair of rules produces the legs of the chair. To assure random, yet consistent sizing for all of them, the previously defined attribute *Leg.Size* is accessed using the special *@Double* function. It implicitly adds a corresponding *IfHas* predicate to the rule. Attribute getters can be composed to complex expressions. To finish the basic chair model we add a simple rule for the lean construction.

```
new Rules.Size(Axis.Z, 0.04)
.Fulfills("lean");
```

At this point the grammar already produces a simple chair (see figure 2 left) with variable leg thickness. Note that using *Void()* is the only way of explicit terminal assignment. Any other termination directive would contradict the basic idea of connecting grammars in the cloud. The derivation process shall continue until there are no more rules applicable. A predefined limit of derivation steps assures termination.

Cloud queries

The cloud-based derivation process is the most revolutionary feature of the system. It allows efficient collaborative modeling without any overhead. It might seem that the chair grammar contained modeling operations only. In fact, it already exposed procedural queries. The concept of goals *is the query mechanism* integrated within the grammar. There was no rule for material assignment in the grammar, but the chair in figure 2 is brown. As the wooden goal of the axiom was not fulfilled by any local rule, the cloud was queried instead and returned a rule assigning brown color to wooden objects. Our design was refined automatically without any intervention necessary. This trivial example already shows the solution to the first two pitfalls listed in section "Collaboration for Grammars".

Let us demonstrate the queries on a more explicit example. The following grammar adds an alternative rule for the chair support. It only defines three rules and relies that any other rules necessary for a wooden chair will be fetched from the cloud. That is equivalent to overriding a part of an existing design without any direct knowledge about it. Hence, the third pitfall of CityEngine is solved as well. Figure 2 (right) shows the resulting chair.

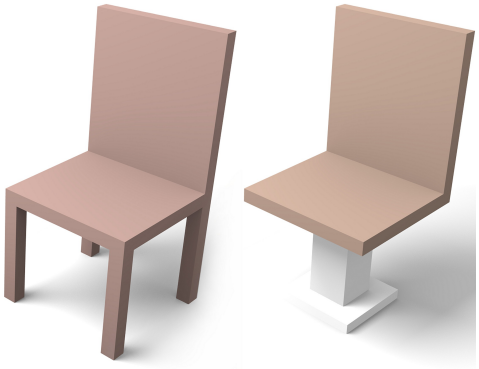


Figure 2
Brown material for the chair is automatically linked from the cloud. Support of the basic chair model (left) can be easily overriden (right).

```
new Axiom("chair", "wooden").Size
 ↪ (0.4,0.8,0.4);

new Rules.Split(Axis.Y,
  Absoulute(0.4).goal("stand"),
  Relative(1).goal("leg"))
.Goal("metalic")
.FulFills("support");

new Rules.Scale(Vec3(0.5,1.0,0.5),
 ↪ Pivot.Middle)
.Fulfills("leg");

new Rules.Scale(Vec3(0.25,1.0,0.25),
 ↪ Pivot.Middle)
.Fulfills("stand");
```

Now there are at least two rules stored in the cloud which fulfill the support goal. In such case, one of them is selected in a non-deterministic way. Stochastic selection is possible as well. Primarily, rules from the current grammar get selected. If none fits, the cloud of grammars is queried for rules matching the current shape. The derivation process in the background seamlessly jumps between the local and the cloud rules all the time.

The number of results returned from the cloud can be immense. The more common an object is, the more alternative designs are expected to be found. Moreover, the cloud is a dynamic repository with new designs coming in all the time. The amount and variability of results produced by each grammar will increase over time as the cloud grows. Therefore, it is advised to keep open goals for all shapes so that grammars written in the future can automatically add details to all related designs.

Cloud-based modeling

The cloud can also impose negative effects on the design process. The more alternatives the less predictable the results. Wrong semantic annotations in a single grammar can easily propagate to a large class of related designs and produce unacceptable results. Many interesting aspects emerging from the work with a cloud need to be addressed in addition to the basic concept of a declarative grammar.

Ambiguous rule semantics. Rule predicates should be precise, but at the same time simple. Designing a good matching function for each rule is not easy. Keeping predicates too general may have negative impact on the whole cloud. In the previous example the chair terminated with a *leg* goal opened for each of its legs. If someone later writes a grammar for the human body with the following rule it will ruin many chairs by attachment of human legs.

```
new Rules.Split(Axis.Y, /*foot,ancle,
 ↪ etc.*/).Fulfills"(leg");
```

It is very important to include predicates for the context in which the rule should be applied. There are two ways of limiting the rule context. It can be done manually by adding an *IfIs* predicate to each rule, or it can be set for a block of rules with a *using* block.

```
using (Predicates.IfIs("chair")) {
/*All rules in this block will inherit
 ↪  the IfIs*/}
```

Both chair grammar examples from the previous section should be wrapped by such a *Predicates* block, except for the first rule which fulfills the *chair* goal.

Synchronization of queries. Procedural queries are executed independently and in random order. If there is a number of shapes with identical goals a different sequence of production rules will be applied to each of them. But often their appearance should be synchronized, e.g, for windows on a façade. Synchronization of shapes is possible using the *Synchronize* object.

```
var windows = new Synchronize();
```

Shapes to be synchronized get attached to the *Synchronize* instance by calling the *Sync* function. Synchronization then applies for their children as well. For all synchronized shapes, as long as their goals allow, the same sequence of production rules gets applied. Synchronization of rule parameters is left for future work.

```
new Rules.Repeat(Axis.X,
    Absolute(1.5)
    .Goal("Window")
    .Sync(windows))
.Fulfills("Floor")
.IfIs("Facade");
```

Quality control. Michelangelo is still a research prototype. The system is used by a closed group of trusted users. Despite of several protection measures a single malicious rule could destroy almost every produced model. In the current development stage it can be avoided by utilizing restrictions on linking rules from the cloud. The default mode *all* can be reduced to employ only rules by *friends*, *mine* or from *this* grammar.

For the future we plan to utilize gamification in the style of Stack Overflow (Jin 2015). Users will collect achievements and reputation points. Based on

the reliability of the grammar's author new rules will become accessible in the cloud only after approval by respected users. At the same time, objective reviewing of grammars will be encouraged.

COMPLEX RESULTS

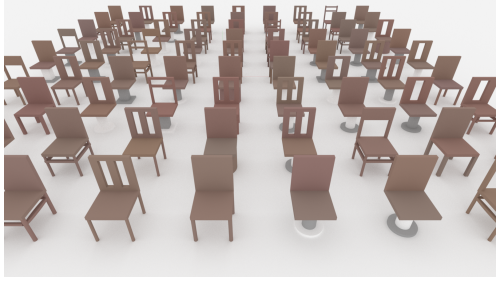


Figure 3
Almost 100 chairs combined randomly from only 4 source designs.

Figure 3 shows a variety of chair models with parts combined from four incomplete grammars. The image was generated by a grammar which contained only axioms for wooden chairs, but no rules were given. All modeling decisions were left to the cloud. Variance implied by the high number of possible combinations is the first important use-case of the presented method.

Instancing of complex model hierarchies is the most powerful use-case of the presented approach. Procedural models, if defined well, are able to fit automatically into different space volumes. Figure 4 shows an office building with 2500 m^2 on three floors populated purely by procedurally generated content. Shape grammars were in particular helpful with furniture layout and design due to extensive instancing of similar objects. The most frequent items are 240 desks and office chairs. A part of them is visible on the floor plan of the top floor on Figure 5. Detailed variations of an office are presented in Figure 6 and Figure 7. It is mainly the arrangement of furniture that changes in the examples.

Evaluation of the whole building axiom was performed in less than two minutes. The generation process took 47,284 derivation steps controlled by 329 rules which were combined from 27 grammars. The interiors include items designed independently in a time span of more than half a year. Nevertheless, no management was required for the maintenance or merging of the sources. The cloud automatically derived the models using a combination of all the stored grammars.

Limitations

During several informal sessions we collected feedback for Michelangelo. Designers appreciate the web-based platform for providing easy scripting and good overview out-of-the-box. On the other hand, rendering performance of large detailed scenes in a browser can not match stand-alone applications.

Our most ambitious experiment was the presented office building. For such architectural tasks the main drawback is the lack of collision detection and missing support for floor planing. Floor plans created for the office building would not comply with a different one. A different boundary polygon and different placement of vertical structures like columns, pipes and elevators would be the main issues. A dedicated rule would be necessary for a universal procedural approach to floor planing.

The current scope of Michelangelo are furniture items. Our main goal is to improve the available deformation rules to allow a wider range of forms. The current set is limited to simple taper, twist and shear rules.

FUTURE WORK

We hope that more out-of-house artists will try out Michelangelo. As the cloud will grow there will be new challenges for keeping the system consistent and focused on the goals given in the axioms. The topic of cross-reviewing grammars is part of a larger research package planned for the next months. We see a large potential in turning the cloud intelligent. By observing user behavior and collecting feedback on the produced models the cloud should be able to learn and provide a highly customized experience:

- Learn new concepts and select rules better. If there are many rules without an explicit pred-

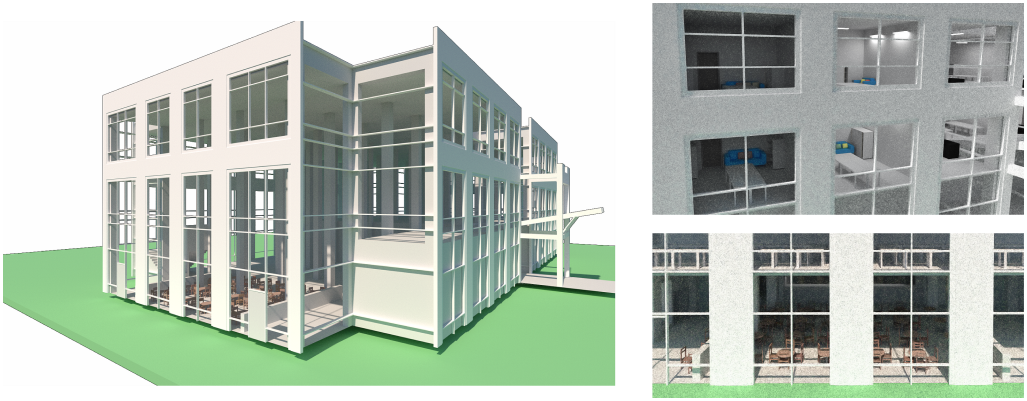


Figure 4
An office building with its full interior generated from the cloud.

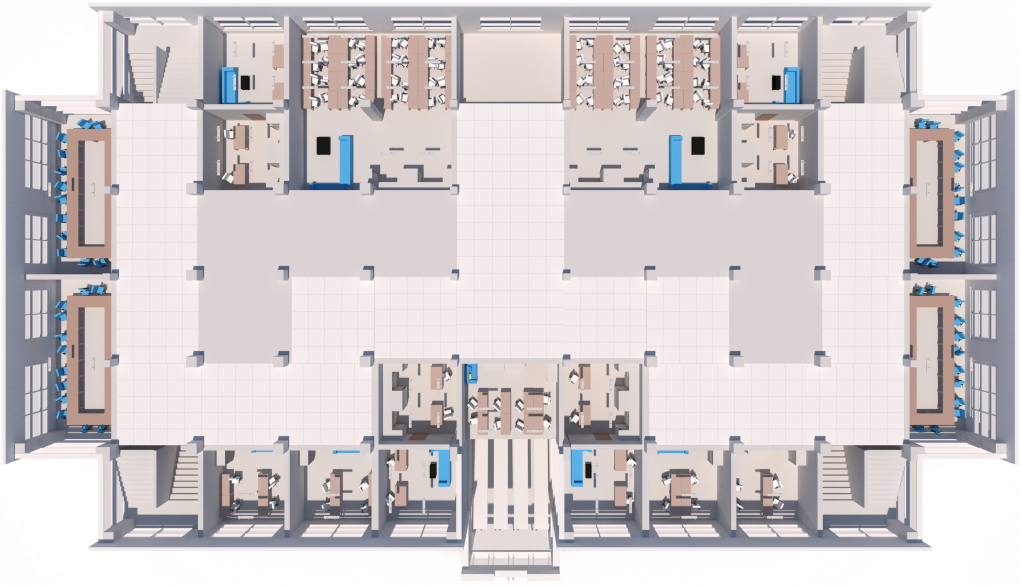


Figure 5
Interiors of offices in the top floor.

Figure 6
Variations of a medium office.

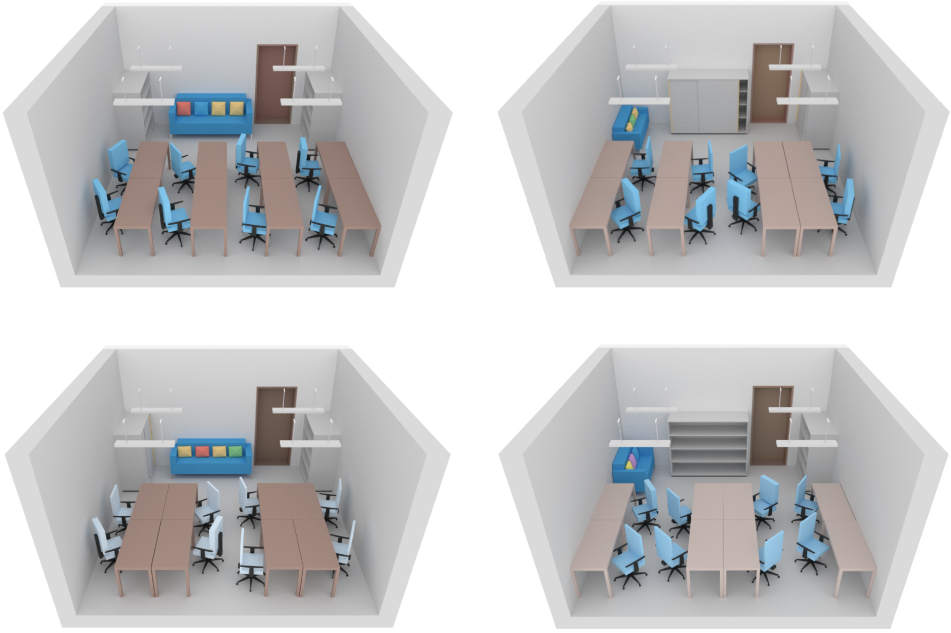
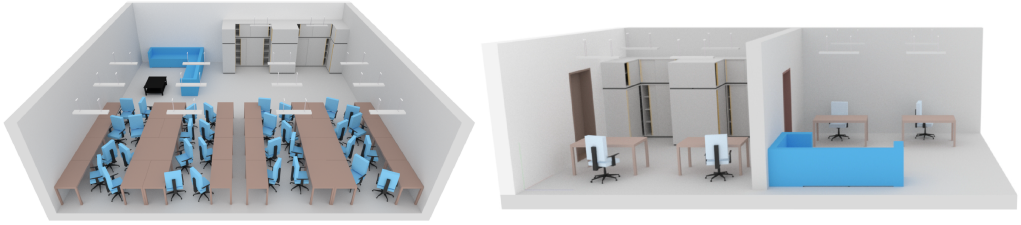


Figure 7
More office layouts.



icate on the current shape semantics it is hard to pick the best match. The cloud can use the collected data to learn new predicates and improve the decision process even in contexts not handled by the original grammar.

- Adapt stylistic details based on learning preferences of the user. By learning semantic patterns preferred by the user it should be possible to predict which cloud rules he or she will like more.

While the concept of an intelligent cloud has clear priority, many more ideas for the development of the system are waiting for implementation. The variety of geometric operations should be increased to allow more attractive and complex forms. A visual interface hiding the programming part would be for sure appreciated by non-experts for simplicity and by experts for efficiency. Interactive visual analysis features for the produced models are becoming clearly necessary as the complexity of the produced models grows. Last but not least, the complexity of the generated models is limited by memory and bandwidth of the display devices. Simplifications by level-of-detail techniques and textures baking will be necessary to reduce the data load.

ACKNOWLEDGMENTS

This research was financed by the Austrian Science Fund project Nr. FWF P24600-N23.

REFERENCES

Edmonds, E A, Candy, L, Jones, R and Soufi, B 1994, 'Support for collaborative design: agents and emergence', *Communications of the ACM*, 37(7), pp. 41-47

Gips, J 1999 'Computer implementation of shape grammars', *NSF/MIT Workshop on Shape Computation*, pp. 56-66

Havemann, S 2005, *Generative Mesh Modeling*, Ph.D. Thesis, TU Braunschweig

Jin, Y, Yang, X, Gaikovina Kula, R, Choi, E, Inoue, K and Iida, H 2015 'Quick Trigger on Stack Overflow: A Study of Gamification-Influenced Member Tendencies', *IEEE/ACM 12th Working Conference on Mining Software Repositories*, Florence, pp. 434 - 437

Kowalski, R 1979, 'Algorithm = Logic + Control', *Commun. ACM*, 22(7), pp. 424-436

Leblanc, L, Houle, J and Poulin, P 2011 'Component-based Modeling of Complete Buildings', *Graphics Interface 2011*, pp. 87-94

Müller, P, Wonka, P, Haegler, S, Ulmer, A and van Gool, L 2006, 'Procedural Modeling of Buildings', *ACM Trans. Graph.*, 25(3), pp. 614-623

Prusinkiewicz, P and Lindenmayer, A 1990, *The algorithmic beauty of plants*, Springer-Verlag, New York

Schwarz, M and Müller, P 2015, 'Advanced Procedural Modeling of Architecture', *ACM Trans. Graph.*, 34(4), pp. 107:1-107:12

Silva, P, Eisemann, E, Bidarra, R and Coelho, A 2015, 'Procedural content graphs for urban modeling', *International Journal of Computer Games Technology*, 2015, p. N/A

Stiny, G 1982, 'Spatial relations and grammars', *Environment and Planning B: Planning and Design*, 9(1), pp. 113-114

Wonka, P, Wimmer, M, Sillion, F and Ribarsky, W 2003, 'Instant Architecture', *ACM Trans. Graph.*, 22(3), pp. 669-677

[1] http://www.esri.com/software/cityengine

[2] http://michelangelo.graphics

[3] http://cehelp.esri.com