

Interactive Modeling by Procedural High-Level Primitives

Lars Krecklau and Leif Kobbelt

RWTH Aachen University, Germany

Abstract

Procedural modeling is a promising approach to create complex and detailed 3D objects and scenes. Based on the concept of *split grammars*, e.g., construction rules can be defined textually in order to describe a hierarchical build-up of a scene. Unfortunately, creating or even just reading such grammars can become very challenging for non-programmers. Recent approaches have demonstrated ideas to interactively control basic split operations for boxes, however, designers need to have a deep understanding of how to express a certain object by just using box splitting. Moreover, the degrees of freedom of a certain model are typically very high and thus the adjustment of parameters remains more or less a trial-and-error process. In our paper, we therefore present novel concepts for the intuitive and interactive handling of complex procedural grammars allowing even amateurs and non-programmers to easily modify and combine existing procedural models that are not limited to the subdivision of boxes. In our grammar *3D manipulators* can be defined in order to spawn a visual representation of adjustable parameters directly in model space to reveal the influence of a parameter. Additionally, modules of the procedural grammar can be associated with a set of *camera views* which draw the user's attention to a specific subset of relevant parameters and manipulators. All these concepts are encapsulated into procedural *high-level primitives* that effectively support the efficient creation of complex procedural 3D scenes. Since our target group are mainly users without any experience in 3D modeling, we prove the usability of our system by letting some untrained students perform a modeling task from scratch.

Keywords: procedural modeling, interactive shape editing, shape grammars

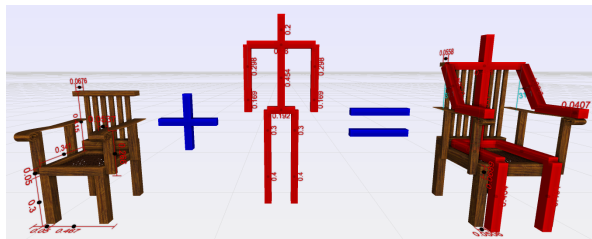


Figure 1: By creating complex manipulator objects like a stick figure and mapping it to another procedural object like a chair, we can achieve an abstract control of the chair in an ergonomic fashion. All important parameters can be directly adjusted in the 3D view for an intuitive and fast modeling process.

1. Introduction

High quality content creation gains increasing importance in computer graphics applications as hardware is getting more powerful in rendering highly realistic 3D scenes [1]. Using conventional modeling applications like *blender* or *Autodesk 3ds Max* is a time consuming task when massive and at the same time highly detailed scenes have to be produced. Moreover, large scale mod-

ifications for these models, like changing the number of floors in a facade, may become quite complicated. In contrast, procedural modeling describes a scene by a set of rules that recursively converts an input symbol (*non-terminal*) into a sequence of output symbols (*terminal or non-terminal*) [2, 3]. Changing parameters in such a grammar may trigger modifications on a coarse scale, like varying the number of windows on each floor of a building, as well as changing tiny features, such as setting the size of the window frames.

Since grammars describe a scene by a set of textual rules with a specific syntax, the procedural modeling workflow better compares to a programmer writing a script rather than an artist creating a 3D scene. Hence, interactive controls have to be included to make procedural modeling more intuitive and accessible to non-programmers. Unfortunately, many features of a procedural modeling language, like case switches and complex mathematical functions, cannot easily be mapped to interactive handles such that the full descriptive power of procedural models cannot be utilized anymore.

In order to find practically useful compromises between purely textual and purely interactive modeling approaches, we introduce two modes in our interactive procedural modeling framework. In the *professional mode* (*P-Mode*) the text-based authoring of procedural models is supported by interactive elements without restricting the descriptive power. The P-Mode is used to implement *high-level primitives* (*HL-Primitives*), i.e., encapsulated modules which take a well-defined set of parameters as input in order to create a specific class of geometric objects. In the *high-level mode* (*HL-Mode*), the user can change and combine these HL-Primitives in order to compose complex models. The HL-Mode is based on interactively adjusting geometric shape handles and the user is never exposed to the underlying textual grammar definition. However, the simplicity of the interface comes at the price of reduced flexibility since only those parameters provided by the HL-Primitives can be modified.

Both modes together combine the advantages of procedural and interactive modeling in the sense that experienced designers can develop a toolbox of HL-Primitives encapsulating expert knowledge of a certain *object domain* (e.g. architectural styles, manufacturable furniture). These HL-Primitives are then adjusted and combined in a fully interactive modeling session.

We introduce fundamental concepts for the creation of an interactive procedural modeling framework. The main challenges discussed in this paper are:

Parameter Manipulators — Any parameter of a procedural model can be mapped to a meaningful 3D manipulator that is interactively controllable in a 3D viewer. This includes manipulators for measure of length and angle measures, as they typically occur in most object domains, as well as of abstract manipulators such as a slider. By this visual interpretation of the parameters, the user gets a direct intuition of how a certain parameter will affect the procedural model thereby avoiding the need for testing random values by hand in a separate 2D interface.

Camera Views — A high number of parameter manipulators might result in a confusing visual representation. Parameter manipulators for small scale features as well as for large scale features are visible at the same time which makes an interactive control of any tiny parameter handles very hard. To overcome this drawback, a set of meaningful camera positions can be defined for a procedural model. Furthermore, a list of visible parameters can be attached to each camera to concentrate the view on only a few parameter manipulators.

High-Level Primitives — We present a novel modeling concept that makes the procedural modeling approach accessible for a wide range of users without programming experience. HL-Primitives that define parameter manipulators as well as camera views can be easily combined and locally modified without being exposed to the textual grammar or to atomic split operators at any time. This concept of simplicity brings procedural modeling to a whole new level, since non-programmers benefit from the interactive controls of the HL-Primitives which encapsulate a certain domain specific knowledge (like buildings, furniture, or plants).

Compound Primitives — HL-Primitives can be composed to create more complex objects. We demonstrate this mechanism by showing an extreme case, in which we create a complex manipulator object from the atomic parameter manipulators that is then associated with an existing HL-Primitive by defining parameter mappings. This is a powerful concept to produce intuitive control mechanisms such as the one depicted in Figure 1.

1.1. Related Work

The concept of rule based modeling was first pioneered by Lindenmayer and Prusinkiewicz who spend a lot of effort on L-Systems and their connection to the modeling of plants [3]. Based on a text replacement strategy, a LOGO-style turtle is controlled by visually interpreting all terminal symbols after a certain evaluation step. Textual grammars were extended by parameters, context-sensitivity and the stochastic application of rules in order to enhance the descriptive power of L-Systems [4]. The modeling of plants became even more realistic by using environmental information that influences the growth process thereby reducing the number of parameters that normally have to be tuned for a plausible plant growth [5, 6, 7]. Furthermore, interactive design metaphors were added to control any L-System parameters, i.e., spline curves and 2D function plots could be edited to give artists a more intuitive control over the structure of a plant [8].

The idea of L-Systems was also applied to the generation of street networks [9], but it turned out that the formalism of a growth process is not well suited for the creation of precise structures such as buildings. Instead, the decomposition of simple shapes has become a common concept for the modeling of architecture [10]. The basic idea of replacing shapes, rather than replacing text strings, was first pioneered by Stiny et al. [11]. Müller et. al. presented *CGA Shape* as a modeling grammar where a mass model (i.e. the coarse structure of a building) is created in a first step followed

by the application of splitting rules for the generation of small scale features [2]. While *CGA Shape* relies on the manipulation of boxes, Krecklau et al. extended this concept to multiple non-terminal classes within the unified grammar system G^2 [12]. Basically, each non-terminal class could be understood as a conventional modeling methodology like the manipulation of a box or of more complex entities like trilinear freeform deformation cages. Moreover, G^2 introduced the concept of abstract structure templates which is a technique to prevent rule explosion by combining existing grammars in just one line of code. We utilize their method in an interactive way to allow for an intuitive composition of a procedural 3D scene.

Lipp et al. built an interactive framework to control the parameters of the production rules in a CGA shape grammar [13]. They describe the application of persistent local changes in order to break repetitive structures and to give artists more intuitive control over the resulting geometry. However, the interactive features in this framework are linked to atomic operations in a one-to-one fashion potentially providing an individual manipulator for each textual parameter in the shape grammar. Basically, our professional mode behaves quite similar to their work, except that we are not restricted to the subdivision of boxes due to the possibility of using several non-terminal classes. Our small user study shows, that this freedom comes naturally at the cost of simplicity, which is a necessary key factor to make procedural modeling accessible for unexperienced users. We effectively hide the grammatical structure and any parameter dependencies from the amateur user and non-programmer. The author of a procedural model can define a set of valid modifications and represent them by a set of manipulators like seen in one of the first constraint-based graphics systems [14].

Our high-level mode was also inspired by the user interface of Bokeloh et al. [15], who disassemble a given geometry into symmetric parts which can be reassembled to new objects. However, their method can be better compared to the idea of modeling by example [16] as they rely on a fixed set of static geometries which are not parameterized at all. Recently, Lau et al. presented an automatic approach to disassemble furniture into manufacturable parts [17] and therefore it could be understood as an inverse problem to our method for a specific object domain (i.e. furniture). Note, that the creation of manufacturable furniture has also been used in one of our examples as realistic modeling scenario, however, our method is definitely not limited to this object domain which can be seen in our other examples and in the accompanying video.

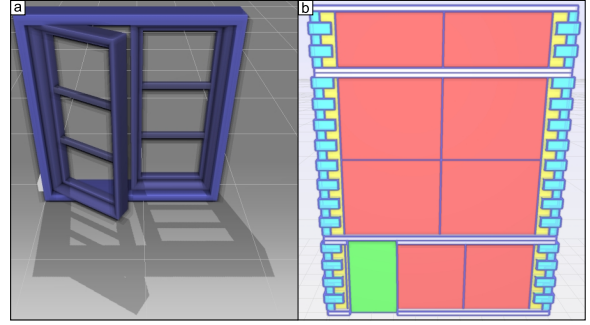


Figure 2: Procedural modules can be parameterized objects which are an encapsulated unit in the scene (a) or they can just describe an abstract structure where the details have to be generated by rules that are passed to the module (b).

2. Procedural Modeling

A formal grammar is a 4-tuple (N, T, P, S) , where N is a set of non-terminal symbols, T is a set of terminal symbols, P are production rules of the form $N \rightarrow (N \cup T)^*$ and $S \in N$ is the start symbol. Whenever there is a non-terminal symbol that has a matching left-hand side in one of the production rules, that rule can be applied and the symbol will be replaced by the right-hand side of the rule.

Our work is based on the procedural modeling language proposed by Krecklau et al. [12]. In their work, *non-terminal objects* are instances of a specific *non-terminal class* in a generated scenegraph which are associated with a certain rule of a grammar. In contrast to formal grammars, the rules contain a sequence of operators that are applied to their associated non-terminal object. On the one hand operators might change the current state of the non-terminal object like resizing a box (*box class*) or moving control points of a trilinear freeform deformation (*ffd class*) [18]. On the other hand operators can create new *non-terminal objects* or *terminal objects* in the scenegraph thereby creating an instance hierarchy. Most important for our system are several high level grammar concepts which were recently presented, i.e., *modules*, *abstract structure templates* and *locators*.

Modules — Often, several rules are needed to describe a certain procedural object. Those rules can be encapsulated into *modules* [12] to keep the grammar clear and unambiguous. A module can be used without understanding its underlying sub rules. The visual appearance of the resulting object can be influenced by setting meaningful parameters that are defined for a module. Figure 2.a shows an example of a window module.

Abstract Structure Templates — In principle, *abstract structure templates* are special modules which use non-terminal symbols as rule parameters [12]. This allows for the definition of rules that describe a coarse layout of a model such as an abstract facade style without specifying the generation of details (cf. Figure 2.b). The concept allows, e.g., to insert different versions of windows and doors in an abstract facade structure to get various facade styles with only one line of code.

Locators — Whenever an operator produces a sequence $NT = (N_1, \dots, N_n)$ of successive non-terminal objects $N_i \in NT$, a meaningful tag such as “floor” can be specified to clearly distinguish each of the resulting non-terminal objects and thereby also each corresponding subtree in the scenegraph hierarchy. Formally, a tag can be understood as a 2-tuple $TagName = (Index = i, Count = n)$ that is associated with every non-terminal object $N_i \in NT$. A *locator* [13] is a construct which is separated from the grammar and which specifies a certain set of non-terminal objects like “the second column in the third floor”. Thus, the user can apply persistent local modifications by changing the rule that is associated to the non-terminal objects that match a certain locator. In contrast to the detached locators, we express local changes directly in the grammar by accessing the tags within any conditional block of a rule. Formally, one can use the term *TagName* to check, if the tag is defined. In order to access the index or count of a tag one can use the terms *TagName.Index* or *TagName.Count*, respectively (cf. Figure 3).

3. Interactive Grammar Creation

The descriptive power of procedural modeling is not only based on the idea that a certain rule set is applied several times. If we could guarantee that exactly the same 3D object would be generated by a specific rule set, we could generate it once instead and just add an arbitrary number of references to that object into a scenegraph. Hence, the definition of public parameters and the use of conditions is a substantial feature for the creation of dynamic procedural objects. Each time we apply a certain rule set, the resulting 3D object can have a different shape based on the choice of parameter values or on geometric queries within any condition [2]. Unfortunately, the definition of public parameters and the use of arbitrary conditions are very abstract processes which can be hardly mapped to intuitive interactive interface metaphors in general. Therefore, we distinguish two points of view on our system. On the one hand, the professional mode (P-Mode) of the application is

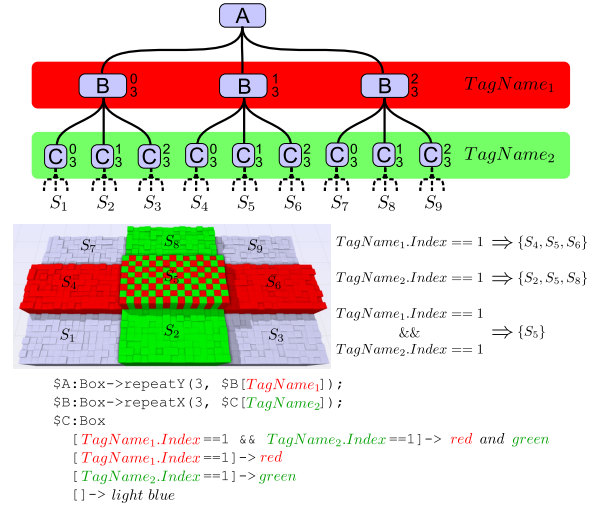


Figure 3: The bottom grammar generates the scenegraph at the top of the image. When an operator extends the scenegraph (e.g. repeatY of rule A), tags can be stored for the whole subtree (e.g. TagName₁). The indices (top) and element counts (bottom) for a specific tag are shown beside each node of the scenegraph whenever a tag has been defined in the grammar. The middle image shows the selection of different subtrees by using the conditions of rule C.

needed with the full descriptive power of the grammar in order to create arbitrary complex procedural models encapsulating a certain expert knowledge. On the other hand, the high-level mode (HL-Mode) of the system is essential to provide an easily operated program for a wide range of users that are then able to combine any of the existing modules.

3.1. Professional Mode

The P-Mode enables all grammar features by providing a text editor. Implementing a rule set by writing text is primarily suitable for programmers, but since we are in the domain of procedural modeling, where operators are applied to geometrical objects in the scene, we can simply map all these built-in atomic operators to *3D manipulators* that are interactively controllable within the 3D viewer. In principle, this includes any modeling metaphors for the different non-terminal classes as presented by Krecklau et al. [12] such as scaling a box or moving the control points of an ffd. Since the operators might be applied to different non-terminal objects in the scene in parallel, the corresponding 3D manipulator might be visible several times (cf. Figure 4.b). An interaction with one of the 3D manipulators results in a change of the corresponding parameter in the textual grammar. A 3D manipulator becomes active, whenever the text cursor is above the corresponding operator in the grammar (cf. Figure 4). The fluid integration of a

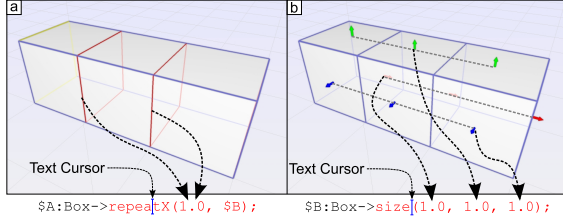


Figure 4: Operators are mapped to 3D manipulators which become active, whenever the text cursor is above a certain operator. In some cases a manipulator contains multiple interactive elements that are mapped to the same parameter of the corresponding operator (a). Some operators have several handles for different parameters (b). If operators are applied to several non-terminal objects, multiple manipulators become visible. Note, that all manipulators change the same parameter of the corresponding operator (b).

textual editor and 3D interaction metaphors as seen in conventional modeling applications, such as 3ds Max from Autodesk, does not only result in a faster scene assembling but also in a learning effect for users that are about to begin their work in the P-Mode.

For more complex grammars, the user can select any object in the 3D scene and trace back the whole *operator history*, i.e., the user can iterate over the sequence of all operators that were called during the generation of the selected object. Once a certain operator is found and highlighted, the textual grammar can be edited or the corresponding 3D manipulator can be used (cf. Figure 4). For example, if a user clicks on a window area within a facade structure (cf. Figure 5), the last used operator to generate the selected instance becomes active (i.e. 1). By using the mouse wheel any involved previous operator can be selected and edited (i.e. 2 – 6). This method speeds up the basic modeling process a lot, because manual text searches in the grammar are mostly avoided.

In the P-Mode, however, the author of a procedural model still has to care about the grammar itself, if parameters and conditions are involved. Due to the conditions, there might exist rules which are not executed at all and thus we are not able to select any object in the scene that made use of that rule. Let us assume, e.g., that rule *C* in Figure 5 has defined another first condition that identifies the fourth column on the bottom floor. In that case, we could not select the door area anymore, although the grammar still provides operators for this part of the facade. Hence it is not possible to get rid of the textual grammar or some other abstract view on the grammar in general.

The main goal of the P-Mode is to provide HL-Primitives that can easily be reused without understanding the insights (the expert knowledge) of the grammar.

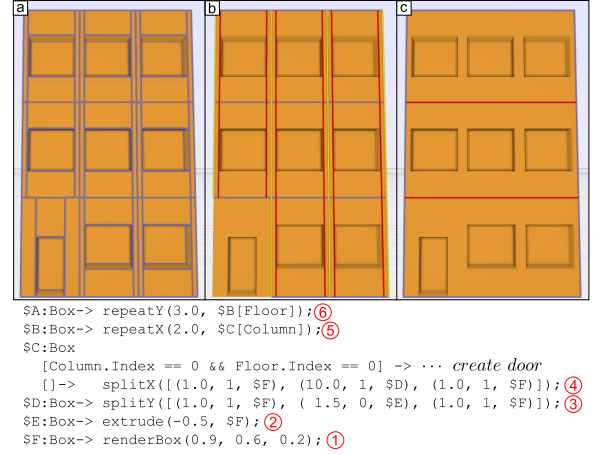


Figure 5: An operator history is created whenever the user selects an instance in the scene. All applied operators that were used to generate the selected object can be traced back in order to avoid tedious manual searches of operators in the textual grammar. In the example above the user clicked on one of the window areas, thereby generating the operator hierarchy (1-6). For simplicity, the images (a), (b) and (c) correspond to the operators 1, 4 and 6 respectively.

Therefore, meaningful *parameter manipulators* have to be visible when a certain procedural model is selected in the scene. In some situations, a module might become so complex that even the set of parameter manipulators is too confusing for any casual user. We provide the concept of *camera views* to overcome this problem by clustering some subsets of available parameter manipulators. When a procedural model is fully declared, its parameter manipulators are well-defined and several camera views have been set, the professional user has to provide some *prototypes* for its grammar as initial examples. This information will then be exported to a database which is later utilized in the HL-Mode.

3.1.1. Parameter Manipulators

A module typically defines parameters to influence the outcome of its evaluation. For non-trivial modules it might be not obvious what exactly happens if a parameter value is changed. In most of the cases, the parameters reflect either a measure of length or an angle measure. Hence the optimal choice is to display manipulators for meaningful length and angle parameters whenever a procedural model is selected. We also provide parameter manipulators for abstract parameters such as color values (cf. Figure 6). Within the grammar, the *parameter manipulators* are simply represented as operators of the following form:

```
manipulator(variable, ...);
```

The first parameter is a variable that is defined by a parent module *M*. Whenever an operator refers to *M*

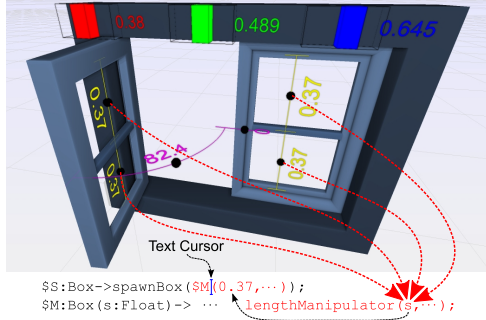


Figure 6: This image shows some parameter manipulators that are provided by our system like length measures (yellow) or angle measures (pink). Some parameters can only be controlled by abstract manipulators such as sliders to adjust the color values of the window. Whenever the text cursor is above a reference to a target rule, all parameter manipulators that correspond to the target rule become visible. An interaction with the manipulator changes the value of the referred parameter in the rule reference.

and the text cursor is above that reference, all involved parameter manipulators are displayed for an interactive editing process of all parameters that are defined by M . In contrast to the former defined 3D manipulators, any parameter manipulator changes the text (the numerical value) at the position where the module M is called and not at the position where the manipulator is defined (cf. Figure 6). This is the major key to the abstraction of interactive procedural models, because the formal definition of the module remains untouched.

3.1.2. Camera Views

A module might define a large set of parameters. This leads to usability problems, if length or angle measures are defined for small as well as for large scale features. Consequently, in the visualization we will observe either small parameter manipulators that are too tiny for any precise interaction or we have to zoom onto a small feature such that any large scale parameters are not visible at all. Another problematic case are overlapping parameter manipulators if they measure two features which are very close to each other.

We overcome these conflicts by creating a set of meaningful *camera views* for a procedural model. Those are then stored in special variables that are defined in a parent module. Therefore, we extend the formal definition of rules in the following way:

$$\$Rule:Type(p_1, \dots, p_n)(c_1, \dots, c_m)$$

The declaration of the camera views c_i is optional. Setting a c_i variable does not differ from setting any other variable. They only need a separate definition, because they reflect a concept that is utilized by the user interface and not by the procedural generation of the 3D

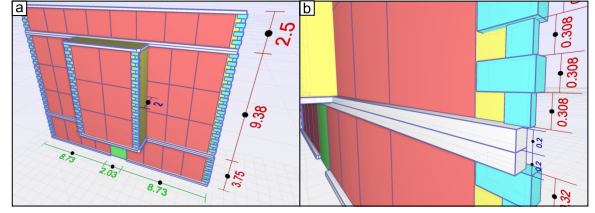


Figure 7: If a procedural model is very complex, it might define a high number of parameters. Camera views help the user to concentrate on a small subset of parameter manipulators which cover either large scale features (a) or small scale features (b). Furthermore, possible overlaps of parameter manipulators are avoided by this method.

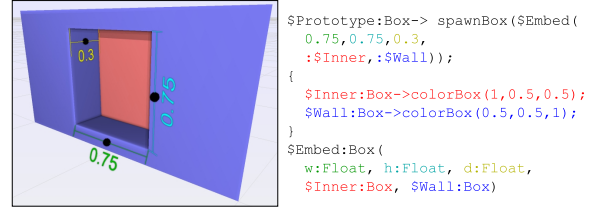


Figure 8: A prototype for the module *Embed* is created which has five parameters two of which are non-terminal symbols that refer to some subrules. When a HL-Primitive is imported in the HL-Mode, one of the prototypes is included into the grammar under a certain name.

scene. The following method makes the definition of a camera view quite simple by taking the position of the camera target and calculating the camera position by adding a specified vector to the position of the target:

$$c_i = \text{CameraView}(\text{Target}, \text{Vector}, \text{ParamList});$$

The last parameter specifies a list of all variables for which the 3D parameter manipulators have to become visible if the camera view is selected. This resolves the problem of overlapping manipulators. Close-up and distant camera views can now be defined for small and large scale features, respectively (cf. Figure 7).

3.1.3. Prototypes

All the procedural models that are designed by a professional user have to be made available for other users. Different *prototypes* (i.e. example rules) have to be provided for a certain module in order to clarify how the parameters can be initially set. If an abstract structure template has been designed, the prototypes have to include some target rules for further evaluation. Typically, non-terminal objects that are associated with the same rule are colored in the same way to visualize their *shared identity* (cf. Figure 8). Finally, a HL-Primitive will be saved in a *database* consisting of the module (the actual rules), the prototypes (template rules for the instantiation), a textual description, some keywords, and a screenshot for each prototype.

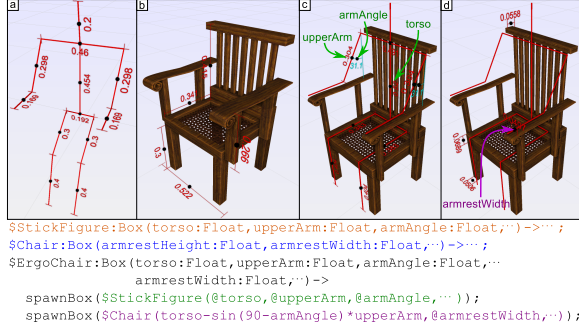


Figure 9: (a,orange) Stick figure manipulator object composed of several parameter manipulators. (b,blue) Procedural chair providing several parameters. After the parameter mapping, some of the chair parameters are controlled by the stick figure (c,green) while others are still handled by the chair (d,purple). Note, that (a) and (b) only show one camera view, so that not all parameter manipulators are visible.

3.1.4. Compound Primitives

Modules can be composed of other modules to share as many rules as possible. Since a module is a textual description of a geometric object it can be understood analogously to the concept of classes in object-oriented programming languages. An external module X can simply be imported by using the following command:

```
#import("X");
```

The above command requires the existence of a file X that contains the respective definition of the module X :

```
$X:Type( $p_1^x, \dots, p_n^x$ )( $c_1^x, \dots, c_m^x$ )->...;
```

For the interactive control of X , some of the parameters p_i^x are mapped to 3D manipulators. If X is imported by another module Y , the author of Y can decide which of these manipulators should still be available for interactive control and which are statically defined by Y . For example, if X defines 3 parameters (with corresponding 3D manipulators) and Y defines 2 parameters, we can use the $@$ sign to pass on a parameter that should still be interactively adjustable by Y ($p_1^x \leftarrow @p_1^y$), specify any mathematical expression for a pre-defined parameter dependency ($p_2^x \leftarrow p_1^y * p_2^y$), or simply set a constant value ($p_3^x \leftarrow 1.0$):

```
#import("X"); //Defines $X:Box( $p_1^x, p_2^x, p_3^x$ );
$Y:Box( $p_1^y, p_2^y$ )->spawnBox($X(@ $p_1^y, p_1^y * p_2^y, 1.0$ ));
```

The parameter p_2^y can be mapped to a new 3D manipulator in Y that also implies new semantics to control X , e.g., an angle in Y can be mapped to a certain length in X . Note, that the definition of camera views for Y is handled in a similar same way to the parameters, i.e., by passing a camera view from Y to X , the author of Y can decide which of the camera views should be reused.

As an extreme case, one could create a complex *manipulator object*, like a stick figure, and attach it to some other module, like a chair, by just defining the parameter dependencies between these objects resulting in an ergonomic chair that is mainly controlled by the attached stick figure (cf. Figure 9). If we want to control the parameters of the chair C (cf. Figure 9.b) by the stick figure F (cf. Figure 9.a), we just create a new module for an ergonomic chair E (cf. Figure 9.c-9.d) that makes use of the modules C and F in the following way. Whenever a parameter of C is dependent on F , a mapping function has to be evolved, e.g.:

```
armrestHeight $_C$ :=torso $_F$ -sin(90-armAngle $_F$ )*upperArm $_F$ 
```

The ergonomic chair E provides a subset of parameter manipulators by passing on some of the parameters to the stick figure F or the chair C such as the height of the torso, the angle of the arm, the length of the upper arm or the width of the armrest. Furthermore, the author of E can decide to define new camera views or to reuse any existing one from C or F by passing on the corresponding camera view variables.

Note, that a compound primitive does not need to be necessarily that complex. As a light weight example, the embed and window modules of Figure 11 could be composed to exchange the continuous sizing policy for a window element of a certain brand by a set of pre-defined sizes that are manufacturable.

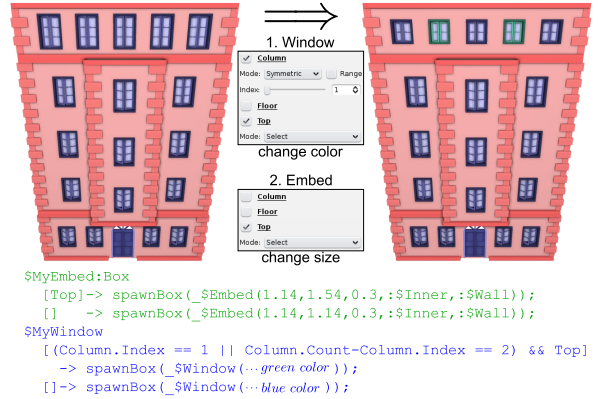


Figure 10: Local modifications are handled by a 2D user interface. When an instance in the scene has been selected, all available tags that are valid for that instance are displayed to the user. Whenever interactive modifications are applied to a HL-Primitive, the user can choose, if the changes should only be applied to a subset of instances by picking a certain selection pattern for each available tag (e.g. symmetric, i-th from left/right, every i-th element). In the upper example, the sizes of all windows in the topmost floor are changed as well as the colors of the second window from the left and from the right (symmetric). The grammar of Figure 12 is automatically changed as seen in the bottom grammar snippet.

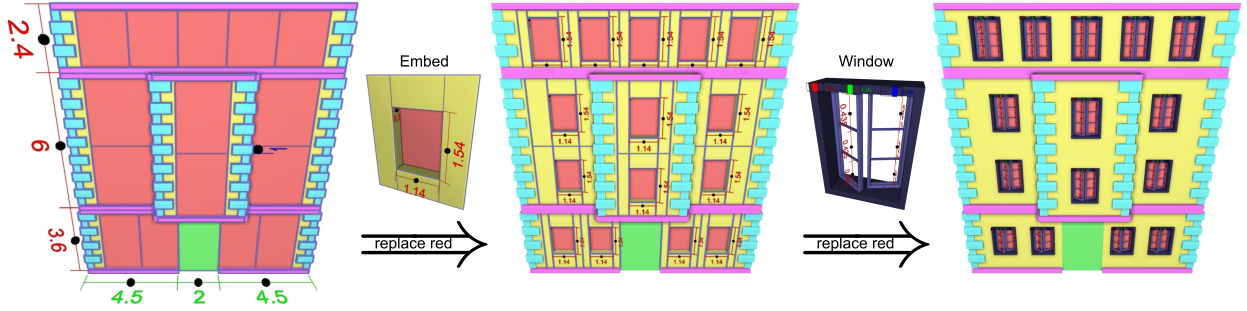


Figure 11: Illustration of the workflow in the HL-Mode. While professional designers provide a toolbox of complex procedural models equipped with interactive parameter manipulators, a casual user can easily combine and modify these HL-Primitives in an intuitive way. In the example above, the high-level user selects a certain facade style at the beginning. Each manipulator in the scene directly visualizes the effect of a certain parameter. Then, all red non-terminal objects are replaced by a HL-Primitive to bring more details into the facade. Finally, a window is chosen for the remaining red non-terminal objects.

3.2. High-Level Mode

In contrast to the P-Mode, the high-level mode (HL-Mode) has to provide a very simple interface to reach a lot of users that are not familiar with procedural modeling. This mode is primarily suitable for beginners or artists, because the workflow is optimized for easily combining and manipulating existing procedural models without regarding the textual grammar at any time. It is a purely interactive process, which consists of the following three interaction concepts:

Replacement — Starting from an empty scene, the user has to import some HL-Primitive from the database. The entries of the database provide a textual description and a sequence of keywords in order to search for a specific class of elements, e.g. “facade styles” or “windows”. Screenshots of the prototypes are displayed for each entry to give the user an impression of the procedural object. Importing a HL-Primitive will put the import statement and a renamed copy of one prototype into the grammar (cf. Figure 12). Whenever new non-terminal objects are created by a HL-Primitive (i.e. it is an abstract structure template), they can be further replaced by importing other HL-Primitives. All non-terminal objects that share their color (shared identity) belong to the same *non-terminal group*, i.e. all non-terminal objects in a non-terminal group are associated with the same rule. The non-terminal object itself might define some *filter tags* to pre-select a small number of items from the database in order to guide the modeling process so that the user can only apply sensible replacements. For example, the database of the second replacement in Figure 11 will be reduced to only display window items. This results in an iterative replacement process that does not require to understand which atomic operations were used (cf. Figure 11).

```
#import("Facade", "Embed", "Window");
$$Box-> size(11, 12, 0); spawnBox($MyFacade);
$MyFacade:Box->spawnBox(_$Facade(2,3,...,$WindowTile,$Wall,...));③
{
  $WindowTile:Box-> spawnBox($MyEmbed);
  $Wall:Box-> colorBox(1.0,1.0,0.5);
}
$MyEmbed:Box->spawnBox(_$Embed(1.14,1.54,0.3,$Inner,$Wall));②
{
  $Inner:Box-> spawnBox($MyWindow);
  $Wall:Box-> colorBox(1.0,1.0,0.5);
  $MyWindow-> spawnBox(_$Window(...)); ①
}
```

Figure 12: This grammar corresponds to Figure 11. First, the facade module is imported (red). Then, we use the embed module to generate details in each of the tiles (green). Finally, the inner part of the embed module is replaced by a certain window. Analogously to the operator hierarchy, any applied modules can be traced back in the HL-Mode (1-3). This makes subsequent changes of any parameters quite easy for the user. For simplicity, we left away some parameters and subrules.

Parameter Adjustment — Similar to the P-Mode, the user can click on any object (terminal or non-terminal) and trace back all HL-Primitives that are involved to generate the selected object by simply scrolling the mouse wheel. Whenever a HL-Primitives is activated, the user can toggle between the different camera views and interactively control any visible parameter manipulators (cf. Figure 7).

Local Modifications — Some of the abstract structure templates generate non-terminal objects that are associated with the same rule. In order to apply a modification to a subset of these non-terminal objects, we utilize the tags which are reflected by a 2D user interface. Any existing tags for a selected instance in the scene can be easily combined and different selection patterns can be chosen for each tag (cf. Figure 10). The user can either change any parameter values for the specified HL-Primitive or load a completely different one. The example of Figure 10 shows some local modifications that are applied to the procedural object of Figure 12.

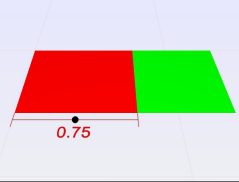
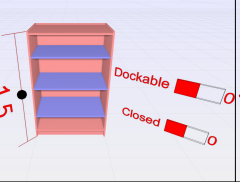
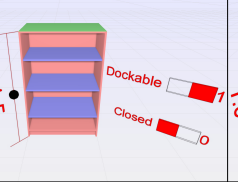
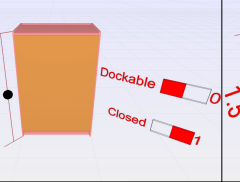
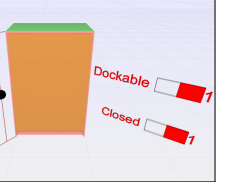
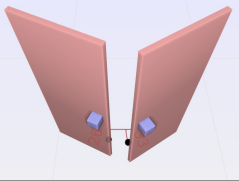
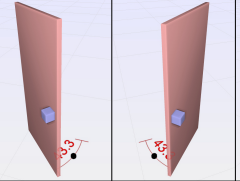
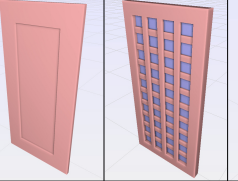
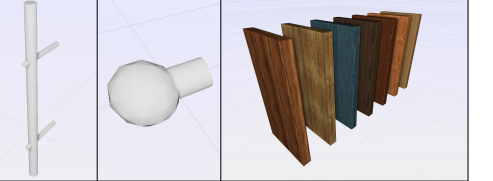
ShelfBasePlan	ShelfContainer			
				
Description: The plan is the basic construct to place shelf containers. It can be extended to the right. Parameters: width of the shelf column NT: ShelfContainer, ShelfBasePlan	Description: A shelf container creates several non-terminal objects and defines the basic shape of the shelf. This high-level primitive has a several options to change the overall appearance. Beside a parameter to change the height of the container, the user may also specify if another shelf can be placed on top of it or if the shelf is closed so that the user can add some doors. The inner and outer planks can be replaced by a wooden material. Parameters: height of the shelf container, dockable (allow to place another shelf container on top), closed (allow to put some doors in front of the shelf) NT: MaterialWood, MaterialWood, ShelfContainer, ShelfDoorLayout			
ShelfDoorLayout	ShelfDoorDesign	ShelfHandle	MaterialWood	
				
Description: Three door layouts are available, which can be placed in front of the shelf if it has been defined as a closed one. The door layout can be either left or right handed or both. The user can also specify the opening angle(s). Parameters: angle(s) to open the door(s) NT: ShelfDoorDesign, ShelfHandle	Description: Two different door designs are available to raise the complexity of the model. Parameters: none NT: MaterialWood, MaterialWood	Description: Two different handles are available that can be attached to the shelf doors. Parameters: none NT: none	Description: A collection of wooden materials is provided by this module. Parameters: type of the wood NT: none	

Figure 13: This image depicts the available HL-Primitives in our shelf assembly scenario (cf. Figure 15 for an assembled shelf). Each HL-Primitive has a short description for its typical usage and a list of adjustable parameters. Furthermore, the *filter tags* of the newly generated non-terminal objects are listed (NT). Their color corresponds to their shared identity which can be seen directly in the 3D model. As an example, the orange non-terminal object of the *ShelfContainer* should be replaced by one of the available *ShelfDoorLayouts* and therefore, the database will only show these three options when the orange non-terminal object is selected.

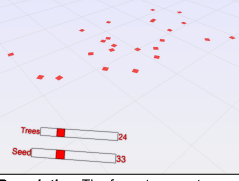
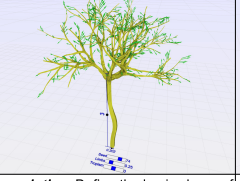
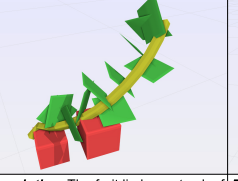
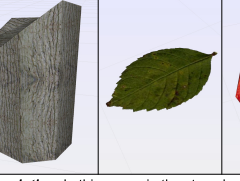
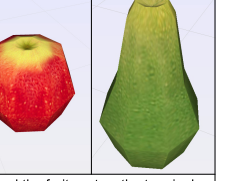
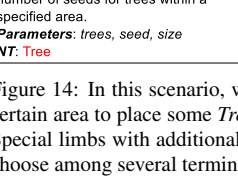
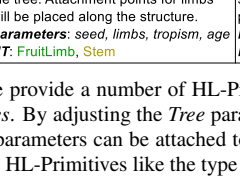
Forrest	Tree	FruitLimb	Stem	Leaf	Fruit	
						
Description: The forrest generates a number of seeds for trees within a specified area. Parameters: trees, seed, size NT: Tree	Description: Define the basic shape of the tree. Attachment points for limbs will be placed along the structure. Parameters: seed, limbs, tropism, age NT: FruitLimb, Stem	Description: The fruit limb creates leaf seeds and places fruits with a specified probability. Parameters: lenght, fruitProbability NT: Fruit, Leaf, Stem	Description: In this scenario the stem, leaf and the fruits act as the terminal symbols which are placed during the modeling process. Note, that the stem is defined in a free-form deformation cage and will be deformed accordingly. Parameters: none NT: none			

Figure 14: In this scenario, we provide a number of HL-Primitives for the creation of fruit trees. The *Forrest* automatically generates seeds in a certain area to place some *Trees*. By adjusting the *Tree* parameters, the user can define the limb density or influence the growth process (tropism). Special limbs with additional parameters can be attached to the tree structure, e.g. *FruitLimbs* for the creation of fruit trees. Finally, the user can choose among several terminal HL-Primitives like the type of wood, leaves and fruits that define the overall appearance of the tree.



Figure 15: This example shelf has been assembled by the HL-Primitives depicted in Figure 13. Note, that any parameters of the object are still adjustable, so that we are able, e.g., to open the doors, change the type of wood or adjust any of the container dimensions.

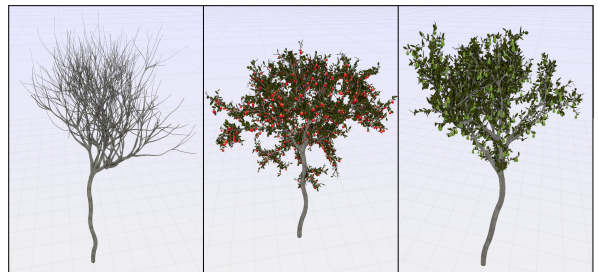


Figure 16: These example trees have been generated with the HL-Primitives depicted in Figure 14. Adjusting the parameters and replacing non-terminal objects in different ways rapidly results in a collection of unique tree models.

3.2.1. Examples

In this chapter we demonstrate the flexibility of our system by showing several examples of different object domains, namely buildings, furniture, and plants. Please also watch our accompanying video containing all presented examples.

Buildings — Most explanatory figures throughout this paper demonstrate the use of HL-Primitives for the creation of buildings (cf. Figure 11, 10). In this case, the user first chooses the basic style of the facade that defines the coarse structure. Afterwards a database of windows, doors, ornaments, and cornices allows for the creation of a wide variety of different buildings [12]. Unfortunately, the geometric complexity for a single building can already become very high such that one can only display and edit a small number of buildings at interactive frame rates. Consequently, further research needs to be done to automatically derive a continuous level-of-detail (LoD) of the HL-Primitives to enable cooperative modeling sessions of whole cities.

Furniture — Section 3.1.4 already illustrated an example of an ergonomic chair that was composed of two existing modules. In this case, we are able to interactively adjust the skeletal features with respect to a target person and the parameterized chair will automatically adapt these changes.

In an alternative scenario for the creation of manufacturable furniture, we assume to have a database of HL-Primitives containing shelf parts of a certain brand (cf. Figure 13). For simplicity, the company is specialized in the assembly of wooden shelves like the one of Figure 15. Due to the filter tags, the modeling process becomes much simpler since the number of replacement options is reduced significantly. From the database of Figure 13, e.g., the user will only see the two *ShelfDoorDesign* entries when one of the red non-terminal objects of the *ShelfDoorLayout* is selected. As another example, the inner and outer planks of the *ShelfContainer* have the filter tag *MaterialWood* assigned. Therefore, all planks have to get a wooden material, however, the type of wood which is chosen for the inner and outer planks is allowed to differ because they are defined in different non-terminal groups shown in blue and red, respectively (cf. Figure 15). This modeling scenario has also been used for our small user study which can be found in the appendix of this paper.

Plants — For this scenario, we provide a small database of HL-Primitives that contain organic parts for the composition of tree models (cf. Figure 14). For simplicity,

we only provide a small set of HL-Primitive for the creation of fruit trees like the ones in Figure 16. Basically, the user starts with a HL-Primitive (*Forrest*) that randomly places a number of seed points within a rectangular area. At any time, the random seed can be altered to change the final outcome of the forrest. Afterwards, the HL-Primitive *Tree* can easily generate a high number of different branching structures. One useful parameter for this HL-Primitive is the tropism, i.e., the limbs of the tree will favor to grow towards the sky (positive value) or towards the ground (negative value). Beside these parameters, the basic tree structure will place further seed points to place limbs with a special behavior. In our case, we provide a special *FruitLimb* that places leafs and fruits during the growth process. The number of fruits that are attached to the limb can be controlled by a special parameter. Finally, our database contains a small number of terminal HL-Primitives to define the type of wood of the limbs and to specify the leafs and fruits that are attached to the tree.

4. Discussion

Usability — Although the P-Mode provides interactive handles for basic operations like transformations and splits, we cannot get rid of the P-Mode with direct access to the grammar (cf. Section 3). Especially, if arbitrary conditions or complex mathematical functions are involved, the procedural modeling process is more like scripting instead of designing. However, with a slightly higher workload, an expert can define parameter manipulators such that the procedural model becomes an easy operated HL-Primitive for a large community.

With the implemented concepts of the HL-Mode, the whole modeling process can be rather compared to choosing and combining your favorite elements from a catalogue and does not behave like a traditional modeling software. This is our main goal in order to make procedural modeling accessible to everyone. In our small user study (given in the appendix of this paper), the test subjects had to solve several tasks like assembling the shelf of Figure 15 from scratch given the database of Figure 13. In summary, all participants were very satisfied with their result (all outcomes were barely distinguishable from the requested shelf) and — maybe even more important — they had a lot of fun using the application which was partially caused by the direct integration of the parameter manipulators in the 3D viewer (cf. Appendix). Please also watch our accompanying video in which we demonstrate the usability of our system.

Size	Tri.	Inst.	Eval.	Upl.	Ren.
20×10	65972	4351	9	20	1
20×20	142676	10047	24	46	2
20×40	258932	18841	46	78	3
20×80	528596	39027	113	118	4

Table 1: This table shows some performance statistics of our application related to the facade of Figure 11 right. We measured the number of triangles and instances in the scenegraph as well as the timings for the grammar evaluation, the upload of the generated geometry to the graphics card and the rendering of the triangles in milliseconds.

Implementation — Our application is written in C++ utilizing meta programming concepts. Although all fixed types of the grammar are compiled to native data structures, the system remains flexible in terms of new non-terminal classes or built-in operators for the P-Mode. To enhance the visual appearance of the real time viewer, we use screen space ambient occlusion for the generated objects and glow effects for the manipulators. We run our application on a Intel Core i7 at 2.67GHz with 6 GB ram and a GeForce GTX 285 with 1 GB ram. Since our application is not parallelized, we only utilize one of the cores. We have taken some performance statistics (cf. Table 1) of the facade scene of Figure 11 right. It is extremely important to notice, that the number of instances is the crucial factor for measuring the evaluation performance and not the number of triangles, because the number of triangles can already be very large for a single loaded terminal shape.

Limitations — Camera views have turned out to be a necessary concept if a lot of parameters are defined by a module. Unfortunately, defining appropriate camera views can become challenging, because of the dynamic behavior of a procedural model. Even for static geometry, a good viewpoint estimation is not trivial and often relies on a set of heuristics to determine meaningful visibility criteria [19, 20]. In future work, semantics could be attached to object parts and integrated into these algorithms in order to derive better visibility criteria.

Furthermore, the set of camera views can change on different parameter values due to certain conditions (cf. Section 3). The presented prototypes are a convenient solution to this problem, because the author of a grammar can already define some meaningful start configurations. A minor problematic situation occurs if small parameter manipulators are defined. In that case, the user does not get a direct feedback how an edited parameter changes the global shape of the object. This problem could be solved by providing two viewers, i.e. one for editing with respect to the current camera view and one user defined camera location to see any global changes.

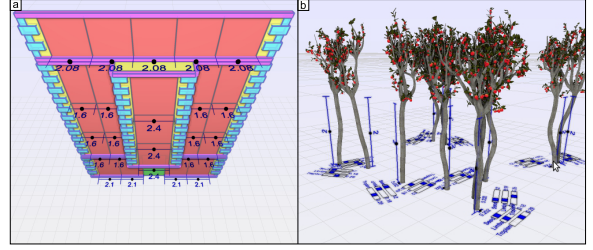


Figure 17: In a repetitive pattern, a *single* parameter manipulator definition can result in *multiple* interactive 3D handles. (a) In this example, one parameter is used to define the approximate width of each window tile, however, in the visualization, several parameter manipulators are displayed labeled with the actual width instead of the parameter value itself. (b) For irregular patterns, showing several instances of one parameter manipulator definition might become confusing.

If a single parameter manipulator definition is evaluated multiple times (e.g. due to the repeat operator), several 3D manipulator objects are generated. In this case, our definition of camera views will not be able to reduce the number of visible handles since they all map to the same parameter value. Unfortunately, there is no general answer to solve this problem. In some situations, especially for regularly repeated elements (cf. Figure 17.a), it is reasonable to display all manipulators as indication for actual lengths and angles that occur in the model. On the contrary, for irregular structures, like randomly placed trees (cf. Figure 17.b), multiple visualizations of the same parameter manipulator might become confusing. For this configuration, only the manipulator instances of the selected item could be displayed. Other entities that are dependent on the manipulated parameter could be highlighted with a colored silhouette to reveal the global influence of that parameter.

5. Future Work

We want to provide several other non-terminal classes in our application, such as NURBS, to bring the P-Mode of our application one step closer to conventional modeling software like Maya or 3ds Max from Autodesk. For this purpose, we simply need to map the basic operators of a new class to common 3D user interfaces.

Once our system provides a higher number of classes and operators, we make the application available to a large community. Professional users create their own HL-Primitives which are then stored in a globally accessible database. If the HL-Primitives are approved by moderators, the database will provide a well-defined set of HL-Primitives which are then downloaded by any other user for compositing. When several people work within the same procedural environment, a huge and likewise detailed scene could be generated.

Although our application can already handle single complex procedural models, it is not possible to generate a large scenes with millions of objects with interactive frame rates. In future work, we will develop and evaluate different methods on how those scenarios can still be rendered and edited in real time such as automatic level of detail creation for procedural models. This is one of the most essential points for the former explained community based system.

6. Conclusion

This paper has covered fundamental concepts for the interactive creation, modification and composition of complex grammars for procedural models. A *professional mode* provides all features of the procedural system by allowing direct access to the underlying textual grammar. Geometrically meaningful parameters that describe a certain procedural model are mapped to interactive manipulators. Camera views are defined to reduce the number of parallel visible parameter manipulators. Finally, the author provides prototypes for his grammar to make it available as high-level primitive for a wide range of people. The fully interactive *high-level mode* serves for the composition of existing procedural models in an easy way. With a small community of professional users and a large community of high-level users, procedural modeling becomes the key solution to the creation of huge and likewise detailed scenes.

Acknowledgement

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

References

- [1] B. Watson, P. Müller, O. Veryovka, A. Fuller, P. Wonka, C. Sexton, Procedural urban modeling in practice, *IEEE Computer Graphics and Applications* 28 (3) (2008) 18–26.
- [2] P. Müller, P. Wonka, S. Haegler, A. Ulmer, L. V. Gool, Procedural modeling of buildings, in: *SIGGRAPH*, ACM, NY, USA, 2006, pp. 614–623.
- [3] P. Prusinkiewicz, A. Lindenmayer, *The algorithmic beauty of plants*, Springer-Verlag New York, Inc., NY, USA, 1996.
- [4] P. Prusinkiewicz, M. Hammel, J. Hanan, R. Měch, Visual models of plant development (1997) 535–597.
- [5] P. Prusinkiewicz, M. James, R. Měch, Synthetic topiary, in: *SIGGRAPH*, ACM, NY, USA, 1994, pp. 351–358.
- [6] R. Měch, P. Prusinkiewicz, Visual models of plants interacting with their environment, in: *SIGGRAPH*, ACM, NY, USA, 1996, pp. 397–410.
- [7] W. Palubicki, K. Horel, S. Longay, A. Runions, B. Lane, R. Měch, P. Prusinkiewicz, Self-organizing tree models for image synthesis, in: *SIGGRAPH*, ACM, NY, USA, 2009, pp. 58:1–58:10.
- [8] P. Prusinkiewicz, L. Mündermann, R. Karwowski, B. Lane, The use of positional information in the modeling of plants, in: *SIGGRAPH*, ACM, NY, USA, 2001, pp. 289–300.
- [9] Y. I. H. Parish, P. Müller, Procedural modeling of cities, in: *SIGGRAPH*, ACM Press, NY, USA, 2001, pp. 301–308.
- [10] P. Wonka, M. Wimmer, F. Sillion, W. Ribarsky, Instant architecture, in: *SIGGRAPH*, ACM, NY, USA, 2003, pp. 669–677.
- [11] G. Stiny, Introduction to shape and shape grammars, *Environment and Planning B* 7 (1980) 343–361.
- [12] L. Krecklau, D. Pavic, L. Kobbelt, Generalized use of non-terminal symbols for procedural modeling, *Computer Graphics Forum*.
- [13] M. Lipp, P. Wonka, M. Wimmer, Interactive visual editing of grammars for procedural architecture, in: *SIGGRAPH*, ACM, NY, USA, 2008, pp. 1–10.
- [14] G. Nelson, Juno, a constraint-based graphics system, in: *SIGGRAPH*, ACM, NY, USA, 1985, pp. 235–243.
- [15] M. Bokeloh, M. Wand, H.-P. Seidel, A connection between partial symmetry and inverse procedural modeling, in: *SIGGRAPH*, ACM, NY, USA, 2010, pp. 104:1–104:10.
- [16] T. Funkhouser, M. Kazhdan, P. Shilane, P. Min, W. Kiefer, A. Tal, S. Rusinkiewicz, D. Dobkin, Modeling by example, in: *SIGGRAPH*, ACM, NY, USA, 2004, pp. 652–663.
- [17] M. Lau, A. Ohgawara, J. Mitani, T. Igarashi, Converting 3d furniture models to fabricatable parts and connectors, in: *SIGGRAPH*, ACM, NY, USA, 2011, pp. 85:1–85:6.
- [18] T. W. Sederberg, S. R. Parry, Free-form deformation of solid geometric models, *SIGGRAPH* 20 (4) (1986) 151–160.
- [19] H. Fu, D. Cohen-Or, G. Dror, A. Sheffer, Upright orientation of man-made objects, in: *SIGGRAPH*, ACM, NY, USA, 2008, pp. 42:1–42:7.
- [20] P.-P. Vázquez, M. Feixas, M. Sbert, W. Heidrich, Viewpoint selection using viewpoint entropy, in: *Proceedings of the Vision Modeling and Visualization Conference 2001, VMV '01*, Aka GmbH, 2001, pp. 273–280.
- [21] S. G. Hart, L. E. Staveland, Development of nasa-tlx (task load index): Results of empirical and theoretical research, in: *Human Mental Workload*, Vol. 52 of *Advances in Psychology*, North-Holland, 1988, pp. 139 – 183.

Appendix

This appendix contains a user study to compare the concepts of the professional mode (P-Mode) and the high-level mode (HL-Mode). We show, that an intuitive and simplistic user interface is the key factor to make procedural modeling available to a wide range of casual users with no 3D modeling experience. Most important for the motivation of the participants is to avoid an emotional link of the modeling process with some kind of stressful work, but instead it should feel like playing a game which is emotionally linked with fun.

Introduction — We performed the study with 10 mainly undergraduate students. With respect to the modes, our user study is split in two phases. In the first phase, we explained the principal concepts of procedural modeling to the participants and we created a small shelf layout from scratch by using very basic split operators to demonstrate the P-Modeling mode in action (similar to the *ShelfContainer* of Figure 13).

Note, that the interaction metaphors in this mode behave quite similar to the ones presented by Lipp et al. [13] such that we also compare our HL-Mode against their method in a certain manner. After the introduction, we handed over a sheet explaining all important operators (i.e. transformations, repeat, split) to the participant, who had to repeat the modeling process of the shelf layout within a limited time period of roughly 10 minutes. In the second phase, we shortly introduced the interaction methods of the HL-Mode and handed an image of a shelf over to the participant, who had to compose the illustrated shelf within 10 minutes (cf. Figure 15). We did not discuss the available HL-Primitives in detail, because in a real world scenario the user would also be faced with unknown HL-Primitives of different modeling domains (see Figure 13 for a subset of the provided HL-Primitives). Notice, that the two tasks are quite similar in the number of interactions that have to be performed.

The NASA-TLX — The main goal of our user study is to compare the workload between the P-Mode and the HL-Mode based on the NASA Task Load Index (NASA-TLX) [21]. The results of our study can be seen in Figure 18. The diagram clearly shows, that the cognitive workload for the HL-Mode is much below the cognitive workload of the P-Mode. Note, that the number of participants is already sufficient for a reliable statement on the workload which can be seen from the calculated confidence intervals, that do not overlap. All participants were able to model exactly the shelf from the image in the HL-Mode within the given time (average 7:44) whereas only three participants produced a more or less pleasing result in the P-Mode within the 10 minutes (average 11:27). During the test, the participants did not notice that they were already exceeding the given time interval and they stayed motivated to produce the requested structure, however, most of the test subjects had problems to apply the split operations in a strict coarse-to-fine manner which led to suboptimal workarounds in rules that were applied later on. Furthermore, only one participant used the repeat operator properly, whereas all others created the inner planks by a fixed number of splits resulting in a redundant usage of certain parameters.

The Questionnaire — Beside the NASA-TLX, all participants had to give a rating (from 1 (little/rarely) to 5 (much/often)) on the following 8 questions (Q1–Q8). The result of this survey is presented as a box plot in Figure 18. Additionally, they were asked to give some comments on the different questions.

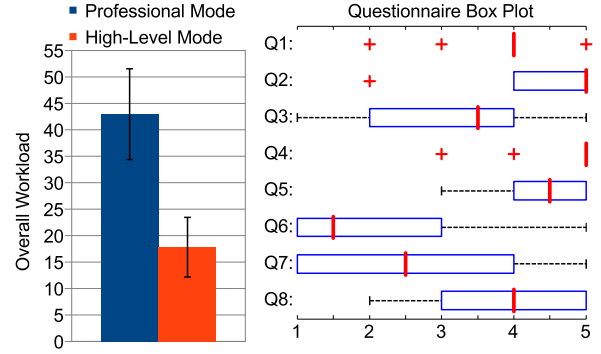


Figure 18: The left diagram compares the cognitive workloads of the HL-Mode (17.81) and the P-Mode (42.95) based on the NASA-TLX. We illustrate 95% confidence intervals for the bar charts to prove the high reliability of our result. The right diagram shows a box plot of the answers related to the questionnaire (red plus \equiv outlier, red line \equiv median).

1. How much do you like the professional mode?
2. How much do you like the high-level mode?
3. How much do you think you would have to learn to model the shelf from Figure 15 with the professional mode?
4. Are you satisfied with your results from the second task?
5. How much do you like the integration of the manipulators directly in the 3D modeling interface?
6. Your knowledge about procedural modeling techniques?
7. Your experience with 3D modeling software?
8. Your experience with scripting languages?

First of all, we see from the box plot, that both modes were liked in general (Q1, Q2). A common sense among the participants why they liked a certain mode was the flexibility in the P-Mode and the simplicity as well as the interaction techniques in the HL-Mode. For question Q3 we can see that the answers are distributed over the full range which most probably correlates with the experience of the test subjects, which is also scattered along the whole range (Q6-Q8) with a tendency to know less about procedural modeling techniques and more about scripting languages. The results of Q4 and Q5 are one possible answer to the question, why both modes were favored so much. Especially the direct integration of the manipulators in the 3D scene was explicitly mentioned several times to be a convenient method to adjust parameters since one can fully focus on the 3D viewer.