Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS

Michal Rosa

Procedural modelling of buildings

Department of Software and Computer Science Education

Supervisor of the bachelor thesis: Mgr. Jan Beneš
Study programme: Computer Science
Specialization: Computer Science

Prague 2015

I would like to sincerely thank to my supervisor Mrg. Jan Beneš for all his advices, guidance, patience and time during development of this thesis.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature

Název práce: Procedurální modelování budov

Autor: Michal Rosa

Katedra: Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: Mgr. Jan Beneš, Katedra softwaru a výuky informatiky

Abstrakt: V této práci se zaměřujeme na implementaci aplikace pro procedurální modelování budov. Naši implementaci pak testujeme modelováním nekolika budov dle reálne předlohy. Kromě samotné implementace také podáváme stručný přehled již existujících prací. Součástí této práce je také programátorská a uživatelská dokumentace.

Klíčová slova: procedurální modelování architektura budova

Title: Procedural modelling of buildings

Author: Michal Rosa

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Jan Beneš, Department of Software and Computer Science Education

Abstract: In this thesis we focus on implementation of a standalone application for procedural modelling of buildings. We are then testing our implementation by modelling several buildings based on buildings in real life. Apart from our implementation, we are providing a short overview into already existing methods. This thesis also includes programming and user documentation.

Keywords: procedural modelling architecture building

Contents

1. Introduction

Computer graphics is extensively used branch of computer science. It is used for various types of problems, e.g. in medicine for processing magnetic resonance, X-ray, and nuclear scanner data, in education for various driving simulators or computer simulation programs, in military for shell trajectory visualization programs, navigation systems, army training programs, in architecture for building visualization and interior design programs or for remote sensing, such as LIDAR. All these applications requires to be swift, efficient an accurate. Our focus, which is generation of various buildings and structures, also requires these attributes.

Particularly in our work we are using computer graphics to model simple or more complex building structures. For example, when creating a film or a game we need to model many buildings or whole cities. Manual modelling on such a scale would be inefficient, time consuming and it would require a considerable amount of human resources, time, computing power or money. Procedural modelling is needed to solve these types of problems. Procedural modelling can be used in modelling individual buildings. These building models can be used as models in procedurally generated cities.

2. Previous work

This chapter contains brief explanation of rewriting systems and procedural modelling of buildings.

2.1 Rewriting systems

Both Lindenmayer systems (L-systems in short) and CGA Shape Grammar (CGA in short) are types of formal grammar [16, 9]. We will introduce both L-systems and CGA later. Firstly, we define a formal grammar: formal grammar is a structure, which describes a formal language. Formal grammar contains a finite set of rules P, finite set of terminal symbols V, finite set of non-terminal symbols $\sum$ and starting non-terminal $S \in \sum$ [17]. We call $x \in \sum$ a non-terminal symbol, if there exists a production rule $p \in P$, which can substitute for symbol x [17]. We call $a \in V$ a terminal symbol, if there is no production rule $p \in P$, which can substitute for symbol a [17].

```
 V = { a }
∑ = { x, b }
P = { S -> ab; S -> x; x -> ba; b -> xax }
```

Figure 2.1: An example of a formal grammar

Figure 2.1 shows us a simple formal grammar with terminal symbol a, non-terminals x and b, starting non-terminal S and a finite set of production rules. Firstly, these rules can substitute S either for ab or x. Secondly, they can substitute every x for ba and every b for xax.

Next, we introduce L-systems, which are an inspiration for CGA [9], which we will introduce later. L-systems were conceived as a mathematical theory of plant development [16]. The central concept of L-systems and CGA is based on rewriting process described above [16]. Rewriting process is a technique with which we can produce complex structures incrementally from simple structures by successively replacing parts of initial objects [16]. In L-systems, production rules are applied in parallel - see figure 2.2.

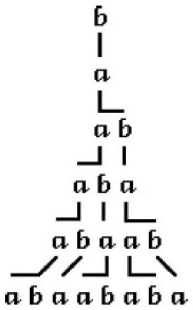

Figure 2.2: L-systems derivation tree. On this picture we can see parallel production rules application. Picture was taken from [16].

This is in contrast to Chomsky formal grammars, where production rules are applied sequentially [16]. There are various types of L-systems, e.g. DOL-systems (deterministic and context-free) or OL-systems (context-free) [16].

2.2 Procedural modelling of buildings

CGA Shape Grammar was originally inspired by L-systems [9]. CGA is an extension of set grammars [9], which were introduced by Muller et al. [9]. CGA is also based on formal grammars like L-systems, but unlike them, CGA substitutes non-terminals in production rules sequentially rather than parallel. Production rule is defined in following form [9]:

```
ID: predecessor:  cond -> successor:  prob
```

Figure 2.3: Production rule as defined by Muller et al. [9]. ID is a unique identifier for rule, predecessor $\in$ V is a symbol for shape to be replaced with successor $\in \sum$, cond means condition, it has to be true for rule to be evaluated and prob means probability of rule to be selected [9].

Major difference between CGA and formal grammars is in production rules structure, which in CGA needs to be non-circular. On the other hand, formal grammars allows us to circular production rules. An example of circularized CGA production rules in our implementation is listed in figure 2.4:

```
 Main --> split(v) { 10:Floor | 10:Floor }                    (1)
Floor --> split(u) { 10:Main | 10:Main }                      (2)
```

Figure 2.4: An example of circularized production rules

There are various types of production rules described by Muller et al. [9]. They can be divided in four categories: scope rules, split rules, repeat rule, component split. Scope rules are generally used to modify shapes (translations, rotations, scaling). Split function allows us to create more smaller scopes from already selected one, thus changing scope. Repeat rule is useful to handle larger scale changes in split rule. Component split allows splitting selected shape into shapes of lesser dimension [9]. Figure 2.4 introduces scope changing function called split.

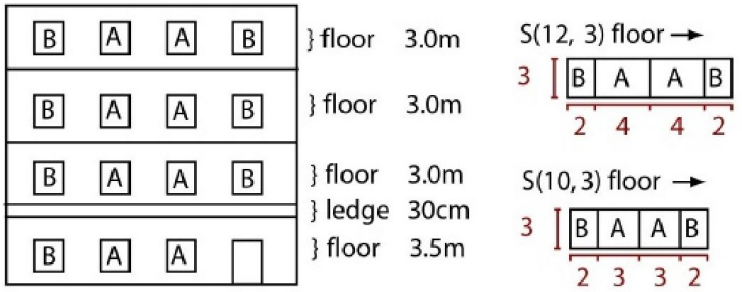

Figure 2.5: Possible result of multiple Split rule usage. Picture was taken from [9].

Set of production rules in figure 2.4 will target all $Main$ non-terminals, substituting them for two non-terminals $Floor$ with height 10 units (1). Then, it takes all non-terminals $Floor$ and substitutes them with two $Main$ non-terminals (2), thus creating infinite loop between $Main$ and $Floor$ production rules. Building generation with such loop will never finish.

Generating process begins with a special production rule, called start rule. This rule needs to be marked with special key word. Value of this key word can be different in various implementations. An example of start rule in our implementation can be seen in figure 2.6:

```
 @StartRule
Lot --> extrude(200, box) Main
```

Figure 2.6: An example of simple starting rule.

In our implementation, this start rule will create an invisible, pre-defined wall with default size referenced by non-terminal Lot. Then, it will substitute this Lot wall for a box with terminal $Main$, which will be 200 units tall and will have basic white color. To add more detail we can add more production rules with

non-terminal symbol *Main* on their left side. We can for example split whole building into a basement and some floors - possible example of such operation is listed in figure 2.7:

```
Main --> split(v) { 50:  Basement | 10:Floor[15] }
```

Figure 2.7: An example of a production rule.

This production rule will target a non-terminal called *Main*, and then it will perform split operation, which divides previous model and all its walls into 16 segments. Firstly, it will create a segment referenced as *Basement* with 50 units height. Then this production rule will take the rest and divides it into 10 *Floor* terminals with 10 units height each. Set of production rules together with basic model then creates more detailed model.

CGA works with a configuration of shapes. Shape consists of a symbol, geometry and numeric attributes [9]. Shapes are identified by either terminal $\in \sum$ or non-terminal symbol $\in$ V. For each terminal (non-terminal) symbol, there is a corresponding terminal (non-terminal) shape. The corresponding shapes are called terminal shapes or non-terminal shapes. Shapes also contains various geometric information in attributes. Most important geometric attributes are position P, orthogonal vectors X, Y, Z and size vector S. They define an oriented bounding box in space called scope. Scope is a box in space which contains the shape [9].

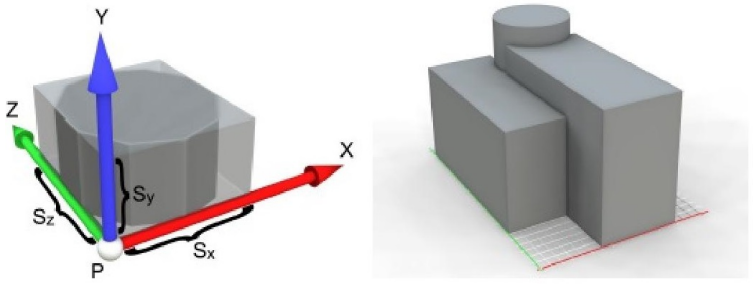

Figure 2.8: On left: example of boundig box or scope. On right: example of some basic shapes together in a simple building. Picture was taken from [9]

We will now describe a production process introduced by Muller et al. [9]. This production process is very similar to rewriting systems, which we introduced earlier. Production process requires a finite set of shapes. Process starts with an arbitrary configuration of shapes A, which are called the axiom [9]. The process alone proceeds in three steps: firstly, select an active shape with symbol B in the set; secondly, choose a production rule with B on the left part and compute a successor for B called BNEW; thirdly, mark shape B as inactive, add BNEW as set of shapes to configuration and continue with step one [9]. This algorithm is applied on every shape stored in set of shapes.

Next we would like to introduce a concrete implementation of procedural modelling of buildings made by Bc. Pavel Reichl [2]. We will start with rule implementation process. In this implementation, shape is implemented as a class and its public interface methods are called rules [2]. These return a reference to current object. There are also complex rules which are written using fluent interface technique [2].

Further on author of this thesis operates with rule selection probability, split based rules, repeat rule, component split, scope modification rules and other rules. Rule selection probability allows associating one rule identifier with multiple implementations. Each of them has a probability of being taken as an attribute. Split based rules allow dividing a scope into smaller scopes. Generally they respect divide-and-conquer principle [2]. Most important rule is Split. It has three parameters: axis, ratios and symbols. Ratios can be absolute or relative. Usage of relative ratios can be seen in figure 2.9.

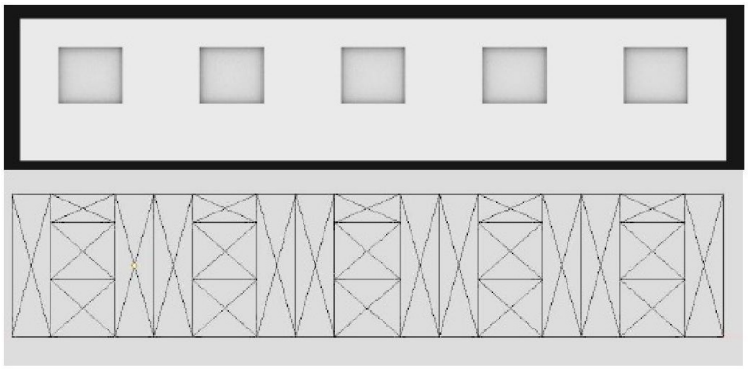

Figure 2.9: Multiple split rule with relative ratios. Picture was taken from [2].

Repeat rule allows us to describe patterns which can be repeated over a volume of arbitrary dimensions. Component split rule allows splitting selected scope into scopes of lesser dimensions. Result of component split rule combined with split rules can be seen in figure 2.10.

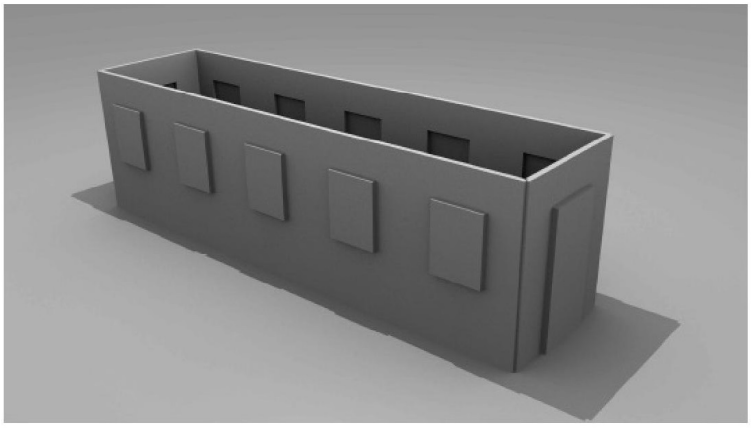

Figure 2.10: Combination of component split and split rule. Picture was taken from [2].

Scope modification rules contain various geometric operations, e.g. translation, scaling, positioning, etc. Other rules provide other functionality for scope manipulation. For example these rules are used to manipulate a stack in which we can store or load a current state of a scope, or they can instantiate a model specified by an input identifier [2]. In comparison to our implementation, implementation by Bc. Pavel Reichl can generate roofs, irregular shapes or round windows.

Another paper by Muller et al. [12] describes algorithms which can automatically derive 3D models of facades simply from facade images [12]. Proposed solutions consist of four parts called stages and together they form a pipeline. This pipeline can transform a single image into a full-scaled 3D facade model. A result of such process can be seen in figure 2.11.

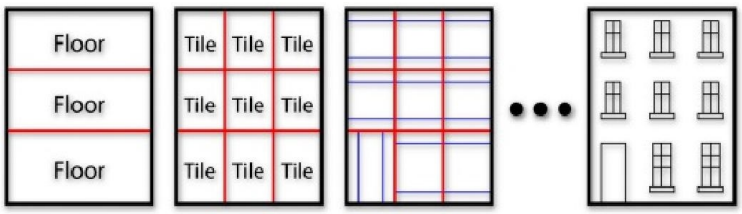

Figure 2.11: Possible result of a complete facade creation process. Picture was taken from [12].

The following sequence contains short description of each stage. In facade structure detection, input picture is divided into floors and tiles using mutual information, much like on figure 2.12.

Tile denotes an architectural element such as door or window together with a surrounding wall [12]. In second stage called tile refinement, we are us-

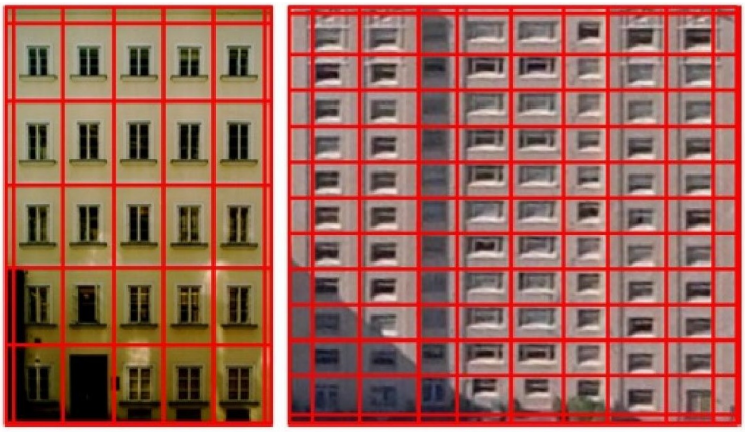

Figure 2.12: Illustration of first stage in facade creation process. Picture was taken from [12].

ing a sub-divided facade picture from stage one. Input also contains clustering information which describes similarity between tiles. This stage continues with sub-division of existing tiles into smaller rectangles using split grammars [12]. Next stage called element recognition we match smaller rectangles created in stage two with 3D objects, which are saved as a library of architectural elements [12]. Result of this stage is a 3D textured model, together with a semantic structure as a shape tree. In final stage called editing and shape grammar rule extraction, our shape tree from previous stage can be used for editing operations, e.g. extraction of shape grammar rule.

2.3 Procedural modelling of cities

Because of its relevance to our work and topic, we also discuss previous work in the field of procedural modelling of cities. The main system for generating cities is called CityEngine. CityEngine is capable of modelling a complete city using a small set of statistical and geographical input data [15]. Uline other city generating programs, CityEngine needs less input data information and no images of actual city [15]. CityEngine generation process consists of several different tools, which form a generation pipeline [15], showed in figure 2.13.

In the first step, all input data are inserted to the road-generation system. Second step defines areas between roads which are used as allotments for buildings, which will be generated in future steps. In third step buildings are generated as a string representation of Boolean operations on simple solid shapes. In last step an inner parser is used to interpret all results for visualization software, which should be able to process polygonal geometry and texture maps [15].

Virtual city is modeled from input data which are represented as 2D image maps [15]. They can be generated by drawing or scanning from statistical and geographical maps. Geographical maps contain elevation maps or

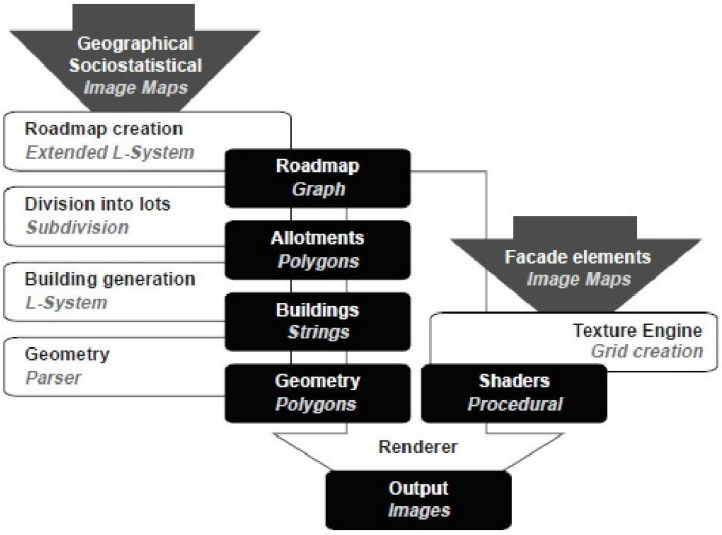

Figure 2.13: Illustration of entire pipeline in CityEngine generation process. Picture was taken from [15].

land/water/vegetation maps, statistical maps contain population density, zone maps (residential, commercial, industrial, mixed) or street patterns [15]. For the creation of virtual city two different L-systems are used: one for street generation, other for building generation [15]. An example of final product can be seen in 2.14.

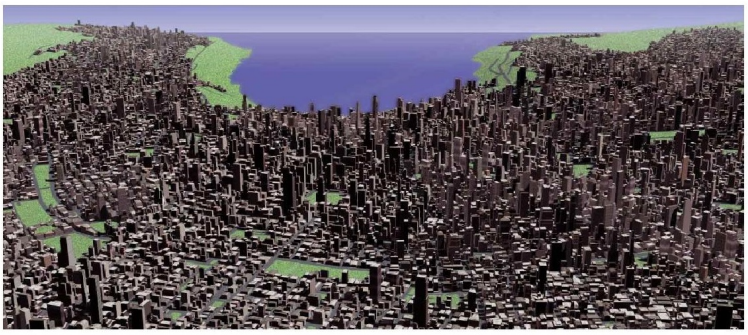

Figure 2.14: Practical example of a complete generated city by CityEngine system. Picture was taken from [15].

3. Our method

In this chapter we describe our method on procedural modeling of buildings. Our method is based on general procedural modeling system, introduced by Muller et al. [9]. Along with describing our method, this section will also introduce and document our modelling language.

Production rule in our method consists from left and right side. Left side of a production rule contains information about non-terminal to which it applies. We can also call this non-terminal a predecessor. Right side of a production rule contains operations such as scope changing functions, transformations or color/texture commands. Scope changing functions also contains information about possible successors of substitution performed by this production rule. Successor can be either non-terminal or terminal symbol. Also we define special production rule called starting rule - see figure 3.1, which is marked with special key word and needs to be first in set of production rules.

```
 @StartRule                                                    (1)
Lot --> extrude(200, box) Main                                 (2)
```

Figure 3.1: An example of a simple starting rule.

Starting rule at figure 3.1 will create an pre-defined pedestal with non-terminal *Lot*. This pedestal is invisible by default. This non-terminal is also starting non-terminal in this example. Line (1) indicates, that the next rule should be treated as a starting rule. Program will take line (2) and substitute *Lot* for a simple box with terminal $Main$ and with 200 units height, default size and place it so it's pivot is in the beginning of Cartesian coordinate system. Pivot is a point in the middle of bottom wall. This point is shared through all walls, meaning the entire building has exactly one pivot.

Our implementation allows user to use several functions. They can be categorized into three categories: scope changing functions, translations and color/texture functions. They will be described in more detail in the next section of this chapter.

Before main generation process, set of production rules is transformed into a grammar tree. Main generation process in our method contains five steps. Firstly, it creates a specific structure to contain all non-terminal or terminal symbols found in grammar tree. In the second step grammar tree is searched for existing scope changing functions: split, comp or extrude. If a function is found it is performed based on its production rule. Scope changing functions are performed differently depending of actual function. Scope changing functions are described in more detail in next section.

After we process all scope changing functions, we will search the grammar tree for possible transformations. If a transformation is found, it needs to be applied on the actual node and all its children. Our method allows using four types of transformations: translation, rotation, orbital rotation or scaling. Examples and usage can be found in next section.

In last two steps we are mapping all colors and textures provided by user in input set of production rules. Again we search grammar tree for possible color and texture production rules. If they are present in a particular node, we will apply them to that node and all of its children. At this point, model presented by set of production rules is fully generated.

Our implementation also allows us to use stochastic rules. They are also explained in more details in the next section.

3.1 Available functions

In this section, we will describe available functions in more detail. We can categorize our set of functions into three different categories: scope changing rules, transformation rules and color/texture rules.

3.1.1 Scope changing rules

This section contains information about scope changing rules. Scope changing rules rules manipulates already selected scope. Scope changing rules can produce more scopes from selected scope by means of split or extrusion or change selected scope dimension.

Extrude production rule

Extrude production rule is most basic scope changing rule. It allows us to alter dimension of selected scope by adding a third dimension onto two dimensional object. It also creates new scopes based on actual parameter presented by user in this type of production rule. Our method allows user to use two different types of extrusion: pool and box. Both of these types use normal vector to perform extrude.

Orientation of this newly created object depends on extrusion type. Box extrude production rule will extrude selected scope in direction of normal. Pool extrude production rule extrudes selected scope in direction opposed to normal vector of selected scope. Count of newly created scopes (faces) also depends of actual type of extrude production rule. Box extrude production rule will create six new objects, creating a box-like object, pool extrude production rule will create five new scopes, creating a (swimming-)pool like object.

```
 Main --> extrude(200, box) Base                    (1)
Main --> extrude(200, pool) Base                    (2)
```

Figure 3.2: An example of different extrude production rule types.

Figure 3.2 shows simple extrude functions. At line (1) we can see extrusion of a box. It will take every node with $Main$ non-terminal and substitute it for new $Base$ terminal by creating 6 new walls, thus changing scope from one wall to six. Newly created object will have height of 200 units. Example at line (2) takes every non-terminal symbol $Main$ and substitute it for new $Base$ terminal symbol, creating 5 new walls and rendering already selected scope - presented by non-terminal $Main$ - invisible. Newly created object will have depth of 200 units.

Component split function

Component split rule allows us to modify dimension of already selected scope by lowering it by one. We can for example change selected scope from 3D object to 2D object. There are various key words for precise selection and scope change in our method. We can categorize these key words into two different kinds: first for changing selected scope for a single scope of lesser dimension, second for changing selected scope for multiple scopes of lesser dimension. First key word set contains six key words: front, back, left, right, top, bottom. This set of key words maps entire object, which can be created by extrude function mentioned before. Second set of key words contains three key words: all, side and other. These key words actually works as a syntax sugar for user. During generation process, they are substituted for a set of key words from first key word set. Key word *all* is a substitution for all six key words from first set, key word *side* substitutes $left$, *right* and *back* key words. Key word *other* substitutes all key words from first set which were not taken in production rule before.

Please note that there is a major difference between *other* component split key word and *all*/*side* in our method. While *other* component split command is unoffensive, key words *all* and *side* are offensive, meaning they are dominant in selection process. An example of such behavior is presented in figure 3.3:

```
Main --> comp(f) { front :  Red | all :  Yellow }
```

Figure 3.3: An example source code which will trigger dominant behavior of key word *all*.

Please consider symbols *Red* and $Yellow$ in figure 3.3 as a non-terminal symbols, meaning somewhere in the set of production rules there is an actual rule which changes their color to red or yellow respectively. In example above, substitution of selected scope of front wall of an object with non-terminal symbol $Main$ to new non-terminal symbol *Red* was performed. Front wall of selected object should then appear as red in final model. However in final result all walls will have yellow color. This is because of dominant position of key word *all*, which will rewrite coloring for the front wall. Figure 3.4 shows this dominant behavior:

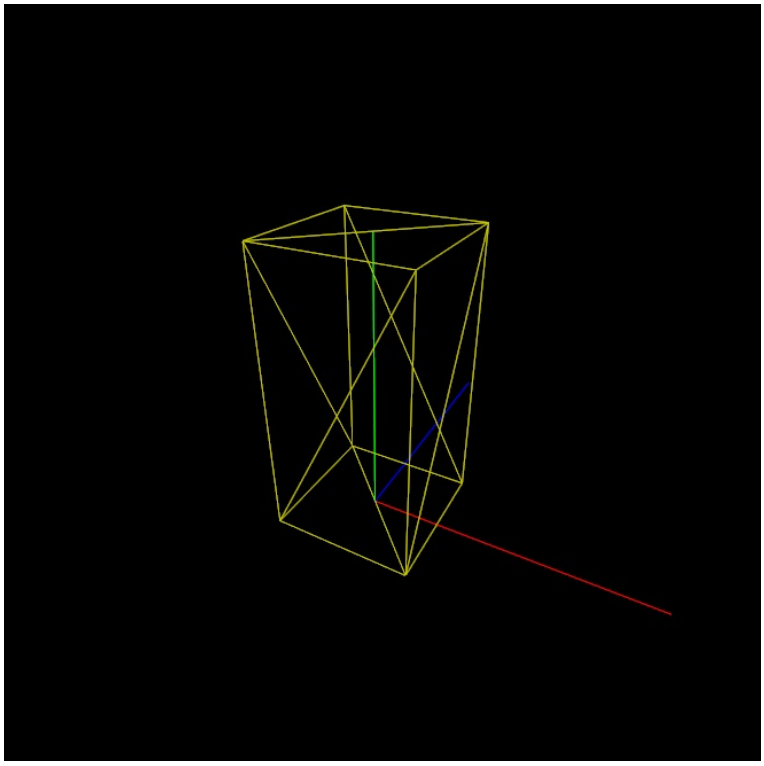

Figure 3.4: An example of dominant behaviour of *all* component split type. Please notice that all walls are yellow, even though we firstly colored front wall to red color.

```
Main --> comp(f) { front :  Front | other :  Other }
```

Figure 3.5: An example of combined component split function.

In figure 3.5 production rule will target every object with non-terminal $Main$ and substitute its front wall for a terminal symbol $Front$, while all other walls (left, right, top, bottom and back) will receive new terminal called $Other$. This example also shows usage of key word *other*. This command will target all wall types not selected previously in actual production rule. For example, if we use only *other* as key word, it will have the same effect as using *all*.

Split production rule

Split production rule allows most variability in our method. Split production rule can be used in different ways and two different directions. Split production rule creates new, smaller scopes from already selected scope by splitting them into smaller segments. These segments needs to have their respective size as an attribute. Sizes can be provided either in absolute or relative way. With absolute sizing of individual segments, two possible mistakes can occur. Sum of all absolute sizes can be lower than actual size of selected scope resulting in an empty space in selected scope. Sum of all absolute sizes can be also greater than actual size of selected scope. In this case, all segments above this size will be ignored. This can also result in an empty space in selected scope.

Split production rule contains one parameter in our method. Parameter of split production rule can have two different values. With this parameter, user provides us with information in which direction we need to perform split production rule. This parameter can be either u or v. If we imagine our selected object in front of us (meaning that camera position is parallel to the normal vector of selected object), parameter u will split object from left to right and parameter v from bottom to top.

```
Main --> split(v) { 40:Basement | 10:Floor }
```

Figure 3.6: An example of simple split production rule.

Figure 3.6 above illustrates simple split production rule, which will target all $Main$ non-terminals and split them into two new segments, thus changing scope from one wall to two walls. New segments will have terminal symbols $Basement$ and $Floor$, and they will be 40 units and 10 units tall respectively. Also, based on split rule parameter, our method will split selected scope from bottom to top.

```
Main --> split(v) { 40:Basement | ~1:Rest }
```

Figure 3.7: An example of split production rule with relative ratio.

Figure 3.7 shows simple split production rule with relative ratio. Production rule will aim all $Main$ objects, splitting them into two segments. First will have terminal symbol $Basement$ and height of 40 units, second will have terminal symbol $Rest$. Exact height of segment with terminal symbol $Rest$ is unknown and will be calculated during generation process as the size of the selected scope $Main$ minus the size of segments with already known sizes.

```
Main --> split(v) { ~2:Basement | ~1:Floor | ~1:Floor | ~1:Floor
}
```

Figure 3.8: An example of split production rule with multiple relative ratios.

Production rule from figure 3.8 above will substitute all *Main* non-terminal symbols for a set of new four segments. First segment will have terminal symbol *Basement*, others will have terminal symbols *Floor*. By parameter provided in this production rule, we already know that we will split this scope from bottom to top. Relative ratios works in the following way. Firstly, depending on parameter by split, relevant size of actually selected scope is computed. Then, whole split rule is searched for absolute ratios. All absolute ratios will be summed and subtracted from already known size of our selected object. Result is the size of a rest segment, which will be divided between segments with relative ratios proportionally to the given ratios. In next step, split production rule is searched for repeaters. An example of a production rule with repeater is listed in figure 3.9:

```
Main --> split(v) { 10:Basement | ~1:Floor[5] }
```

Figure 3.9: An example of split production rule with repeater.

Production rule from figure 3.9 above will target all non-terminal symbols *Main*, splitting them into minor segments. Firstly, it will create segment *Basement* with 10 units of height, then it will create 5 segments with terminal *Floor*, all with relative ratios one. Please note that repeater is just a syntactic sugar for user. Program will parse this production rule into a production rule `Main --> split(v) { 10:Basement | ~1:Floor | ~1:Floor | ~1:Floor | ~1:Floor | ~1:Floor }` . In the next step, whole production rule is searched for relative ratios, summing them. Finally, absolute ratios are calculated on previously found information. This calculation is handled by specific formula:

```
Ar = ((Ws / Rp) * Rr) / R
```

Figure 3.10: A formula for absolute ratios computation based on relative ratios.

In formula listed in figure 3.10, Ar means absolute ratio, Ws means wall size, which is previously computed wall size of selected scope subtracted by absolute ratios, Rp means relative pieces, which is sum of all relative ratios presented in production rule, Rr means relative ratio, which is a single relative ratio in actual part of production rule and R means repeat, which indicates how many time we will repeat this particular part of production rule. After this calculation, all repeaters and relative ratios are computed. Resulting production rule will contain only absolute ratios and no repeaters. We will list and example of a more advanced production rule below, explaining all operations in one place:

```
Main --> split(v) { 40:Basement | ~3:BigFloor[2] | ~1.5:Floor[5]
| ~1:UnderRoofFloor | 10:  TopRecess }
```

Figure 3.11: An example of advanced split prodution rule.

Production rule in figure 3.11 will target all non-terminal symbols *Main*. Based on parameter we will split all *Main* objects from bottom to top. We will

now transform split production rule in this form into a clean form, so production rule will contain no repeaters and relative ratios. For this example, let us assume that *Main* has its respective size of 400 units. Firstly, we will search for all absolute ratios presented in this production rule and sum them. We have two absolute ratios in our production rule, one referenced by terminal *Basement* with 40 units and one referenced by terminal *TopRecess* with 10 units. That makes a total of 50 units of absolute ratios. Size of the rest segment (size of the object subtracted by sum of absolute ratios) will be 350 units. Secondly, we will transform repeaters into clean form. Resulting production rule will look like an example in figure 3.12:

```
Main --> split(v) { 40:Basement | ~3:BigFloor | ~3:BigFloor |
~1.5:Floor | ~1.5:Floor | ~1.5:Floor | ~1.5:Floor | ~1.5:Floor |
~1:UnderRoofFloor | 10:  TopRecess }
```

Figure 3.12: Advanced split production rule without repeaters.

We will now transform all relative ratios into absolute ratios. Firstly, we will sum all relative ratios. There are eight relative ratios in this production rule, which will give us a sum of 14.5 relative pieces. This number indicates on how many pieces we will distribute our 350 units. One relative piece will have size of approximately 24,14 units. For example segment with terminal symbol *TopRecess* will have exactly this size. One segment of *Floor* will have approximately 36.21 units and one segment of *BigFloor* will have a size of approximately 72.42 units.

3.1.2 Transformation rules

In this section we will introduce commands that allows further manipulation with selected scope. This manipulation will be geometrical rather than scope manipulating. There are four transformation commands already implemented in our method: translation, rotation, orbital rotation and scaling. Transformations are part of production rules. There are listed in right half, before actual scope changing function. All transformations have three parameters. With these parameters we can manipulate different axis. First parameter manipulates along X axis, second along Y axis and finally, third manipulates along Z axis.

```
Main --> t(200,0,0) extrude(400, box) Mass
```

Figure 3.13: An example showing an extrude scope changing function with transformation in single production rule.

Production rule from figure 3.13 will target all non-terminal symbols *Main* and substitute them into six new objects with terminal symbols *Mass* via extrusion, creating a box with height of 400 units. Then a transformation is applied, in this case translation. This particular translation will move our newly created object in 200 units along X axis, translating it from its original position

on the beginning of Cartesian coordinate system.

With translation command, we can change position of entire selected scope. With rotation command, we can rotate selected scope around its pivot. Orbital rotation allows us to rotate selected scope around the beginning of Cartesian coordinate system. Finally, scaling allows us to change size of selected scope.

```
 translation - t(200,0,200)                                    (1)
rotation - r(0,45,45)                                          (2)
orbital rotation - ro(15,0,30)                                 (3)
scaling - s(2,1,2)                                             (4)
```

Figure 3.14: All transformation functions.

Above examples in figure 3.14 shows transformation functions in their basic form. At line (1) translation is showed. This command will change position of selected scope - translating scope it 200 unit in X and 200 units in Z axis. Second line (2) contains example of rotation command - rotating scope around buildings pivot by 45 degrees in Y and 45 degrees in Z axis, while next line (3) shows different type of rotation, called orbital rotation. This type of transformation will rotate selected scope by beginning of Cartesian coordinate system by 15 degrees in X and 30 degrees in Z axis. Lastly, example at line (4) shows scaling transformation. This particular example allows us to expand actual scope size two times in X and two times in Z axis.

3.1.3 Coloring, texture rule, stochastic functionality

In this section, we introduce two commands, together with stochastic functionality that our implementation provides. One command allows us to change coloring of selected scope, other will apply texture provided by user on selected scope.

Colors and textures

Color and texture rule allows us to change visual appearance of final model. While color can be applied as a part of any production rule, texture can be only applied on terminal symbols. We assume that coloring a scope can be used for testing purposes only - thus providing an opportunity to color every scope provided by user. Every non-object referenced by a non-terminal symbol will be rendered invisible in the generation process, so applying texture on object referenced by non-terminal symbols is meaningless.

Coloring command consists of three parameters with respect to RGB system. They will apply as red, green and blue ingredient respectively. They values can be from 0 to 255. Texture command consists of one parameter, which is an absolute path to selected texture. In the parsing stage of generation process, all texture paths will be checked and searched for textures. If there is no texture

at provided path, an error will be displayed, terminating generation process.

```
 Main --> color(55,55,55)                                    (1)
Main --> texture(C:/Textures/texture.jpg)                    (2)
```

Figure 3.15: Color and texture command examples.

In figure 3.15 on line (1) listed above will perform coloring on selected scope with terminal symbol *Main*. All vertices in selected object will have color attribute set to (55,55,55). On line (2) of the same example we can see texture command. A texture *texture.jpg* from provided file will be applied at the end of generation process to all objects with terminal symbol *Main*.

```
 Lot --> extrude(200, pool) color(0,0,200) Main
Main --> comp(f) { front:  color(200,0,0) Front | side:
color(0,200,0) Side | other:  color(0,0,200) Other }
Main --> split(u) { 10:  color(200,0,0) TileBlue | ~1:
color(0,200,0) RelativeTile | 10:  Tile(0,0,200) TileRed }
```

Figure 3.16: Examples showing different combinations of color command and other functions.

Stochastic functionality

By stochastic functionality we will understand a rule with certain selection probability. This probability is relative in comparison to other production rules with same non-terminal symbol on their left side. All stochastic rules are mapped to their respective node in grammar tree based on non-terminal symbol on their left side. Basically, all production rules are stochastic in our method - if no relative probability is presented by user, production rule will appear with probability one. If there is only one production rule for a non-terminal symbol, this production rule will appear in 100% occasions.

```
 Main --> split(u) { ~1:Tile | ~5:Tile }
Main(1) --> split(u) { ~1:Tile | ~1:Tile }
Main(5) --> split(u) { ~1:Tile | ~1:Tile }
```

Figure 3.17: Example showing equivalent stochastic production rules.

An example in figure 3.17 shows us different stochastic rules. Please consider them apart of each other, meaning they are not presented in same production rules set. First production rule has no relative probability. In the generation process, this production rule will obtain a relative probability of one. Same principle goes for second example, although here is probability explicitly presented by user as a number between parenthesis on the left side of production rule. Third example contains probability of five, but since it is the only production rule for non-terminal *Main*, generator handles it in the same way as previous

two examples, meaning it will show in all occasions.

Trough generation process, when generator finds a node in grammar tree which contains more production rules, it will sum all their relative probabilities, creating a set of intervals. Then it will pick a random number and find out where in these intervals this picked number lies. Based on that knowledge a production rule for next substitution will be picked from relevant subset of production rules.

```
 Main(3) --> color(200,0,0)
Main(4) --> color(0,200,0)
Main(3) --> color(0,0,200)
```

Figure 3.18: Multiple coloring stochastic rules.

Example in figure 3.18 shows practical usage of stochastic rules to set different color to selected scope with non-terminal symbol $Main$. In this example, sum of relative probabilities is 10. We will then pick up random number and based on this number, we will add color to our selected scope. In this case, our scope $Main$ will have red color in 30%, green color in 40% and blue color also in 30% occasions. Stochastic rules can be also used combined with texture command to provide a set of textures, which can be applied to certain terminal symbol. This will make texture variability and each result will have different design then previous one.

4. Programing documentation

In this chapter we focus on more detailed technical explanation of our implementation. This chapter can be also viewed as a documentation for programmers.

Our program is implemented in C# language using Visual Studio 2012. Our program also uses graphical engine called Irrlicht Lime, which is an C# wrapper for Irrlicht graphical engine.

Our method consists of a main generator that transforms source code into 3D model. Generator requires specialized structure called grammar tree as an input structure. Grammar tree is specialized structure which contains information about set of production rules stored in nodes. Node is a class with various attributes. Node stores information about its identifier and also information about an identifier of its parent. Both these identifiers are stored as integers and together with a list of children identifiers they produce tree structure. Node also contains information about its terminal or non-terminal symbol stored as attribute name and information about production rule, which will be later parsed and applied on this node.

Parsing process contains these major steps: production rules (see chapter 2 for definition) check, parsing into rule storage and generating grammar tree. Firstly, we need to check production rules for possible syntax or parse errors. We assemble all production rules into simple storage, which we search trough. Secondly, previously checked production rules are assembled into rule storage, which is fed to a grammar tree generator in third step. Grammar tree generating contains several sub-steps. Firstly, whole set of production rules is parsed into plain nodes with only their terminal/non-terminal symbols as names and their respective production rules as substitutions. At this point, search for inner split production rules is also in place.

Inner split detection works inside curly braces. For each parsed split production rule, it searches trough whole production rule character by character. If inner split detector finds left curly brace "{", it will increment a level counter and possibly turns on inner split assembling. If inner split detector finds right curly brace "}", it will decrease level counter. If at the same time this level counter equals zero, inner split assembling is turned off and resulting inner split is saved as an anonymous node into grammar tree. Anonymous nodes are actually Node classes with random non-terminals as name attribute. To create this random non-terminal, random number is generated with an underscore as first character. Possible example of a randomly generated non-terminal could look like _98145. Our program also handles possible collisions of anonymous node names simply by generating new anonymous node name.

As a second step of parsing process, whole newly created tree is searched for all production rules, parsing them into scope changing functions, translation, color/texture functions or children. Lastly, we search trough entire structure

again to make connection between individual nodes based on substitution order from input set of production rules, thus creating whole grammar tree.

All previously parsed production rules are stored in their respective node classes as Lists of different structures. In the parsing process we take whole information between set parenthesis, parsing it by the pipe character | into structures called giblets. Individual giblets store all information about possible size, orientation or type of required operation. We use three different types of giblets, depending on which scope changing rule we are currently parsing.

Generator operates in five stages. Generator is a standalone class and contains all information needed in generation process. Generator uses previously generated grammar tree and a structure called Dictionary to store all abstract walls. Generator updates Dictionary several times during generation process, performing different operation in every search. In first stage, whole grammar tree is breadth-first searched and Dictionary structure is assembled. All terminals or non-terminals found in grammar tree will serve as keys to Dictionary. We can easily access all abstract walls based on terminal or non-terminal symbol during generation process.

A wall is actually an abstract class called Primitive. Two classes are inherited from Primitive class - a Rectangle and a Triangle class. They differs in vertices count. We use only Rectangle class at this point in our program, Triangle class is prepared for further development. Abstract Primitive class contains information about vertices and indices stored in static array. Primitive class also contains information about normal vector, which is stored as a unit vector, a pivot, which is stored as a vector, an information about its visibility as a boolean and list of all texture path that needs to be searched and applied in the end of generation process. The primitive class can be used to extend the implementation in the future by using it as a base class for a new primitive type, e.g. a cylinder or a NURBS plate.

Second stage of generation process searches grammar tree via breadth-first search for scope changing functions, applying them based on which scope changing function is presented in production rule. We update previously created Dictionary with new walls to create an abstract model of desired building. Third, fourth and fifth stage performs breadth-first search on whole grammar tree in order to find and perform all transformations, color and texture commands. We separated these stages to ensure generation stability. If a transformation, color or texture command is present in searched node class, it needs to be applied on actually selected Node class and all its children. We accomplish this by using depth-first search on all children of actually selected node.

From Dictionary, final model will be assembled as an end result after end of main generation process. During assembly process, whole Dictionary is searched and all abstract walls stored in Dictionary are transformed into triangle mesh. This triangle mesh is then fed to graphical engine, which is Irrlich Lime in our program.

Main class of our program is called Core. This class performs all operations during generation process and it also serves as an interface between GUI and main generator. Core class firstly check input set of production rules for possible syntax or parsing errors. If input file is successfully validated, Core will perform parsing operation on input file, creating grammar tree. Then Core performs main generation operation, thus creating whole triangle mesh. As a next step, Core initialize whole graphical engine and starts infinite rendering loop. Core class also contains method to export created triangle mesh.

5. User documentation

This chapter describes user functionality of our program. For more detailed technical description please see previous chapter.

After starting our program we will see a screen much like in figure 5.1. We can see various control buttons on this screen together with an empty text box. This text box which is suited on the left upper corner is necessary for input set of production rules. In this text box we can either write our own set of production rules, or load already existing file.

We can load already existing set of production rules by clicking on the *Loadfile* button listed below our input box. By clicking on this button a load screen will appear in which we can choose existing file. Please note that only plain text files can be loaded. We also recommend to load only relevant files that contain set of production rules based on our syntax.

After creation of our own set of production rules either by loading a file or by writing them we can save our new set of production rules. There are two different save buttons. One is located on the left side of our screen below *Load file* button. This button called *Save* will save our actually opened file in case of editing a file that already existed. If we are writing our set of production rules without loading, save as screen will trigger instead of just saving the file. This screen is exactly the same as the one showed by clicking on *Save as new* button, which is located in the middle of our program screen. After successful save is performed, an message will appear, informing that save process was completed successfully.

Let us assume that our set of production rules is now prepared and saved. We can proceed to generation of our model. This can be done by clicking on *Generate* button between *Save* and *Save as new* buttons. After clicking on this button our inner generation process starts. We can monitor our generation progress via progress bar located below *Generate* button. After whole generation process completes our result model will be displayed on the right half of our program.

We can now use our mouse to control our Maya style camera. By clicking and holding **left mouse button** we can do **tumble**, by clicking and holding **center button** we can **dolly** and finally, by clicking and holding **right mouse button** we can **track**. Since there is no reason to generate our model twice in a row, *Generate* button will render itself unusable until changes in text box are made. We can make changes in our production rule set by clicking into text box. Please note that clicking into text box will render output screen unusable and additional generation will be necessary.

After our generation process is completed we can **export** our created model into **Wavefront .obj file** so we can use it in other programs that can

read this type of files. In our case we were using MeshLab program. By clicking *Export Model* button located above *Save as new* button a specialized save window will appear. We need to choose file where we want to export our model. After making this choice, program will check if our chosen directory is empty. In case it is not, warning message will be displayed before export. This is necessary since exporting also copies all provided textures in our set of production rules, thus our destination folder needs to be empty. Exporting mechanism then copies all provided textures and creates two files, one with extension .obj and one with extension .mtl. Please notice that generating is not necessary to export our set of production rules. If we want only to export our model without displaying it, we can load relevant production rules file and click $ExportModel$. After successful export a message will be displayed. Please note that incorrect input files can not be exported. Trying exporting such files will result in a message that our file can not be exported.

We can change visual mode by checking *Enable wireframe* which is located above *Generate* button. After checking this checkbox a wireframe model will be generated instead of full model, consisting from individual triangles. This mode can be used in early stage developing to see detailed results of some production rules. Please note that checking this ckechbox will render output screen unusable and re-generation will be required. We can perform this by clicking *Generate* button again.

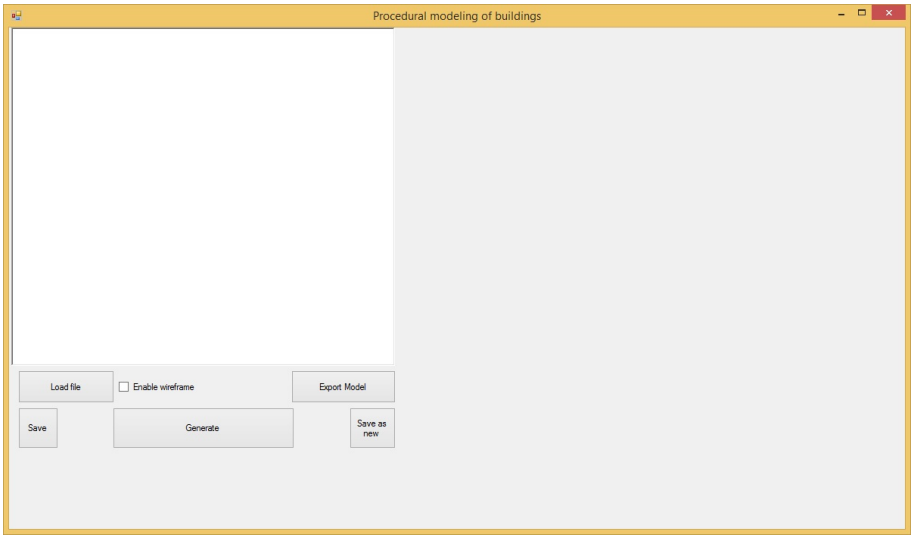

Figure 5.1: Empty window of our program, showing all controls.

6. Results

In this chapter, we will present results and various models, which can be generated with our implementation as a functionality demonstration of our procedural modeling of buildings implementation. Firstly, we will present some basic results to show different types of production rules together with color command. Secondly, we will introduce more complex solutions via three buildings introducing texture command. First will be a dormitory at Prague 5 called "17. listopadu", second will be a prefab building in the "Na Petřinách" street in Prague, and finally third example will be a dormitory at Prague 6 called "Na Větrníku".

6.1 Basic results

At this section, we will present some basic outputs of our implementation. This section will consists mostly of plain boxes with little features as it is mean to show different production rules or commands rather than complex buildings. These types of examples can be seen in next section. We will also provide necessary code snippets. In order to have cleaner view, we will sometimes use wireframe of our model instead of full model.

First example section will contain various extrude production rules. We will combine them with color command and transformations to have a better output.

```
 @StartRule
Lot --> extrude(200, box) color(200,0,0) Main
```

Figure 6.1: Source code for basic extrude production rule.

Figure 6.1 shows us basic extrusion. Set of production rules contains only one production rule, which is also starting rule. In this example our program substitutes starting non-terminal *Lot* for a 3D model of box, adding red coloring in the process.

```
 @StartRule
Lot --> t(150,0,0) extrude(200, box) color(200,0,0) Main
extrude(100, pool) color(0,0,200) Main
```

Figure 6.2: Source code for multiple extrude functions of same terminal symbol.

Figure 6.2 shows us multiple extrusions, one of type box, other of type pool. It also moves first object by 150 units to the right and adds red or blue coloring respectively. Interesting about this example is usage of same terminal symbol $Main$ for two different objects. Creating a production rule with $Main$ as a non-terminal on its left side will select both of objects into scope, allowing

further manipulation like component split or classic split function.

```
 @StartRule
Lot --> t(150,0,0) extrude(200, box) Object1
extrude(100, box) Object2
t(-150,0,0) extrude(300, box) Object3
Object1 --> comp(f) { front:  Front }
Object2 --> comp(f) { front:  Front }
Object3 --> comp(f) { front:  Front }
Front --> color(200,0,0)
```

Figure 6.3: Multiple object with simple component split rule.

Figure 6.3 demonstrates simple component split production rule. Please note that component split targets three different objects and assigns single terminal symbol to their front walls.

```
 @StartRule
Lot --> t(150,0,0) extrude(200, box) Object1
extrude(100, box) Object2
t(-150,0,0) extrude(300, box) Object3
Object1 --> comp(f) { front:  Front }
Object2 --> comp(f) { front:  Front }
Object3 --> comp(f) { front:  Front }
Front --> split(v) { 40 :  Basement | ~1:Floor[10] }
Floor --> color(0,150,0)
Basement --> color(150,0,0)
```

Figure 6.4: Multiple objects with simple component split rule.

Figure 6.4 demonstrates simple split production rule with both absolute and relative ratios. With component split, we mapped all three object into one non-terminal symbol. We can access all three of them by writing a production rule with this non-terminal on the left side. This split rule creates a red colored *Basement* segment of height 40, and then splits the rest of selected front wall into 10 segments, each referenced by non-terminal *Floor*. Please notice that *Floor* segments on each building have different sizes. This behavior is due to relative ratio in split production rule.

Figure 6.5 shows an extensive usage of inner split. Production rule with non-terminal *Floor* on its left side is actually taken from one of advanced results, which we will show in next section. In figure 6.11 we can see that using relative ratios creates fitting *windows* on every object, depending on its actual size. We can also see one of the multi-targeting component split key word. This key word targets all walls on selected object and references them to terminal symbol *Floor*, thus spliting all of them with next production rule. Please notice blue rectangles on top or on right side of middle building.

```
 @StartRule
Lot --> t(150,0,0) extrude(200, box) Object1
extrude(100, box) Object2
t(-150,0,0) extrude(300, box) Object3
Object1 --> comp(f) { front:  Front }
Object2 --> comp(f) { all:  Front }
Object3 --> comp(f) { front:  Front }
Front --> split(v) { 40 :  BrickTile | ~1:Floor[10] }
BrickTile --> color(150,0,0)
Floor --> split(u) { ~1,3:  { ~1:  { ~1:BrickTile | ~6:  { ~2:
BasementWindowBase | ~1 :  BrickTile | ~2:  BasementWindowBase
} | ~1:  BrickTile } |  1:  { ~1:BrickTile | ~6:  { ~2:
BasementWindowBase | ~1:  BrickTile | ~2 :  BasementWindowBase
} | ~1:  BrickTile } } |  1:BrickTile }
BasementWindowBase --> split(v) { ~1:  BrickTile | ~4:
BasementWindow | ~1:  BrickTile }
BasementWindow --> color(0,0,200)
```

Figure 6.5: Multiple objects with advanced inner split production rule.

Last figure in this section, figure 6.6 contains multiple transformation usage. Again, we are extruding pre-defined starting non-terminal *Lot* into three different objects referenced by different non-terminal symbols. Then, different transformations are applied to each object. On complete *Object*1 we apply new scaling vector, which will increase its size four times along X axis and also four times along Z axis. Then we will position *Object*1 by 600 units into right direction. In figure 6.12 box in bottom left corner is *Object*1. Next to it is *Object*2. This object has only one transformation called rotation. This classic rotation rotates selected object around its pivot. In this case, by 40 degrees around X axis, by 50 degrees around Y axis and finally by 60 degrees around Z axis. In the top part of figure 6.12 we can see *Object*3. Our set of production rules indicates that we are applying a total of four transformations. They are executed sequentially from left to right, beginning with new translation vector, which moves selected object *Object*3 by 350 units along X axis in negative direction. With second transformation we will change size vector for selected object, doubling it along X axis and Z axis. Thirdly we added orbital rotation. This means that our selected object will be rotated by the beginning of Cartesian coordinate system. In this case, it will rotate by 25 degrees in direction of X axis, by 150 degrees in direction of Y axis and finally by 47 degrees in direction of Z axis. Last transformation is classic rotation, rotating our selected object by 40 degrees along X axis.

```
 @StartRule
Lot --> s(4,1,4) t(600,0,0) extrude(150, box) Object1
r(40,50,60) extrude(100, box) Object2
t(-350,0,0) s(2,1,2) ro(25,150,47) r(40,0,0) extrude(300, box)
Object3
Object1 --> comp(f) { top:  Front }
Object2 --> comp(f) { all:  Front }
Object3 --> comp(f) { front:  Front }
Front --> split(v) { 40 :  BrickTile |  1:Floor[10] }
BrickTile --> color(150,0,0)
Floor --> split(v) { ~1:  BrickTile | ~4:  BasementWindow | ~1:
BrickTile }
BasementWindow --> color(0,0,200)
```

Figure 6.6: Multiple objects with various transformations.

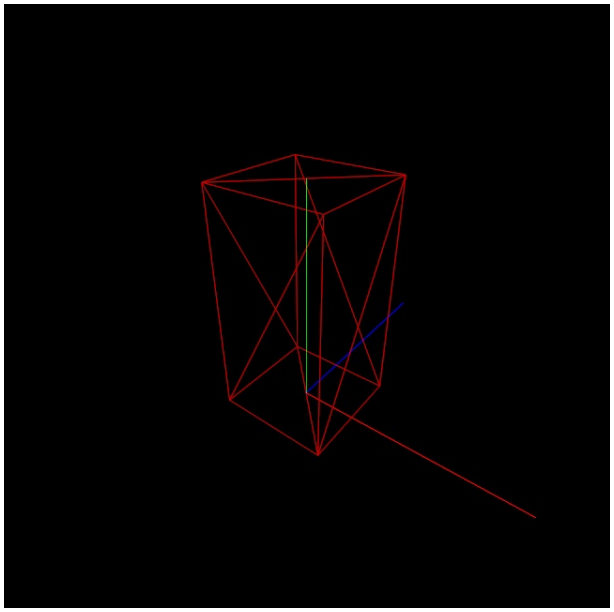

Figure 6.7: Simple extrusion with red coloring.

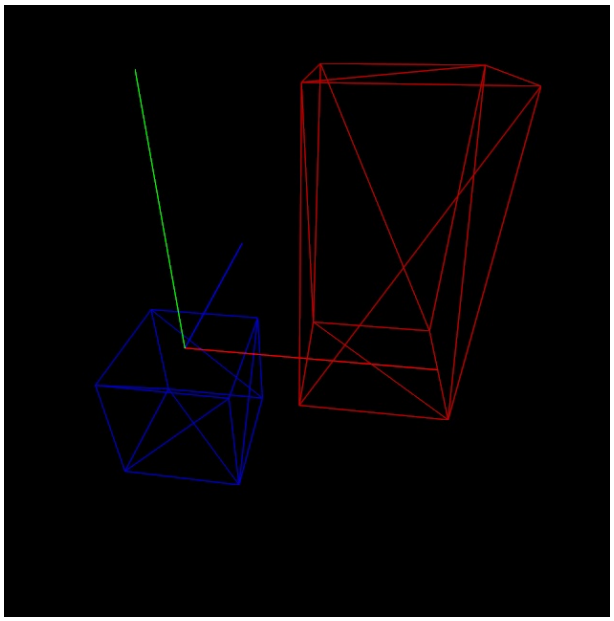

Figure 6.8: Multiple extrude functions of different type with same terminal symbol.

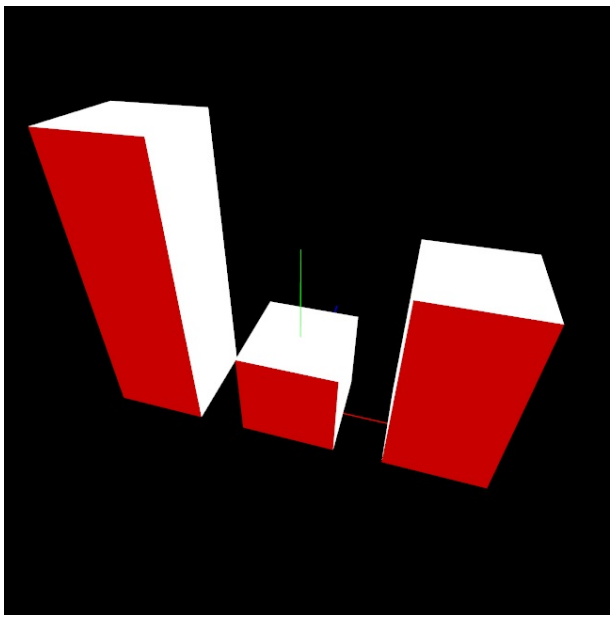

Figure 6.9: Multiple objects with front component split.

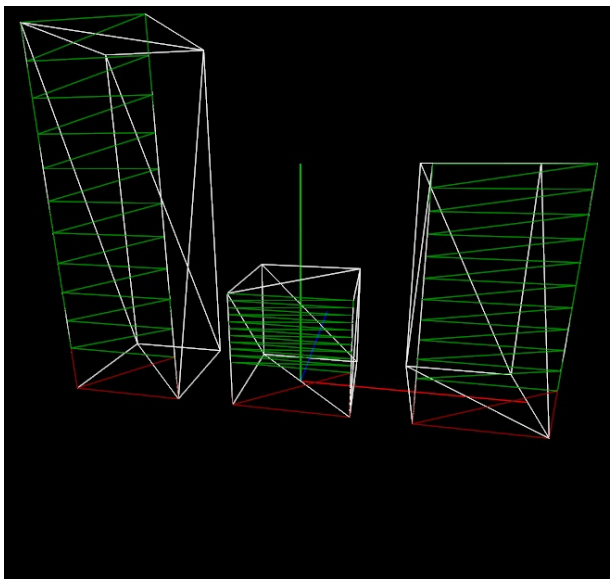

Figure 6.10: Multiple objects with front component split and split rule with combined ratios.

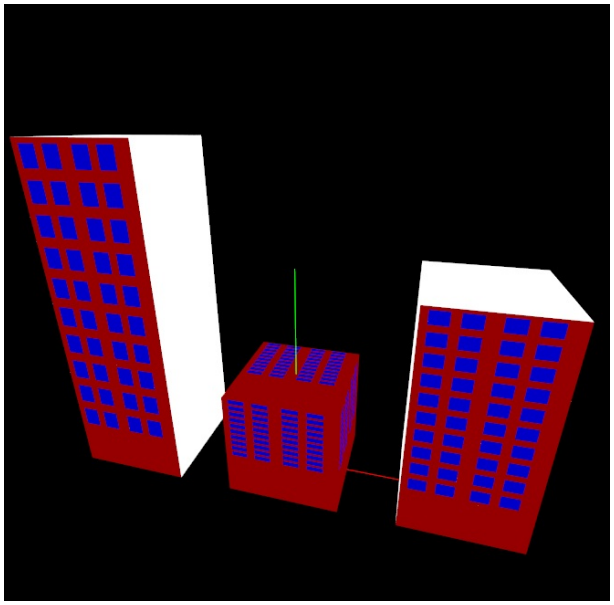

Figure 6.11: Multiple inner split production rule with different component split command. While in left and right model we target only front wall with component split, in middle model we target all walls.

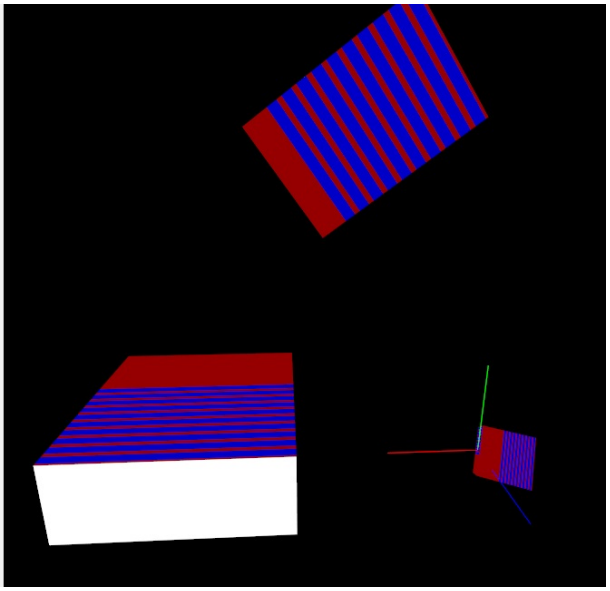

Figure 6.12: Multiple transformation example.

6.2 Advanced results

In this section we will present some advanced results. These results represent real buildings. We will compare these results with actual photographs made for this purpose. The source codes of these examples can be found in the attachement.

6.2.1 ”17. listopadu” dormitory

In this part, we will focus on our first advanced model, based on dormitory called ”17. listopadu”. This dormitory can be located in Prague 7. It consists of two buildings called block A and block B. Block A has 20 floors, block B has 16 floors. We are also using low detailed and low resolution textures for this model to show difference between final visual design compared to other two examples, which are using high quality and high resolution textures.

At figure 6.13 we can see a model of taller block A in front view. Please note various window textures, which shows multiple use of stochastic texture commands. On this picture we can also see entrance, which was modeled by special combination of both extrusion types available.

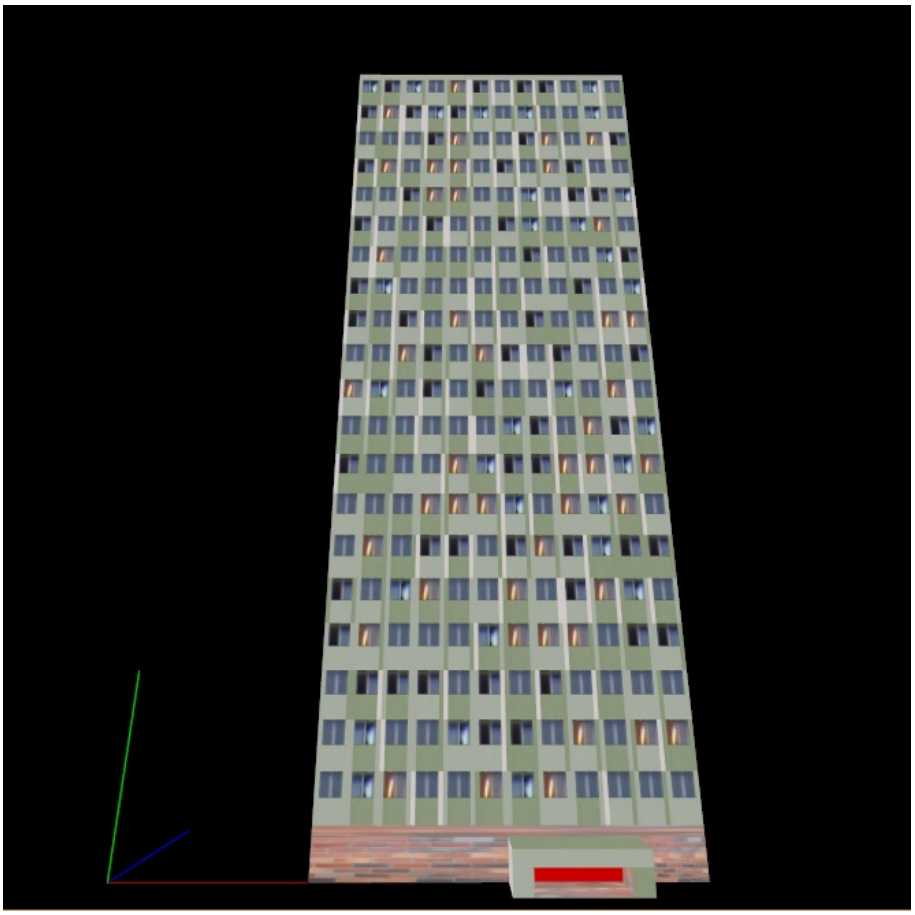

Figure 6.13: ”17. listopadu”, block A, front view. Entrace was modeled using both type of extrusion available.

As a next example, we will compare two pictures from very similar angles. Firstly, figure 6.14 shows our model of "17. listopadu" from a side view, showing both blocks. Taller block closer is block A. In figure 6.15 we can see a photography of real dormitory, also from a side view. Please notice difference in panel ordering in our model compared to real picture. This difference is not a bug and it can be easily removed by further increasing detail in our model. Side parts in our model are referenced by single non-terminal, which also reference terminal symbol 'Tile'. This terminal symbol is also referenced by all other walls, and it contains three different panel textures, which are stochastic. In order to repair this side view to look it exactly as in figure 6.15, we will have to treat each side separately, defining new set of *Tile* (WhiteTile, GreyTile and DarkGreyTile) on which we can map their respective textures. By further ordering of these new *Tile* terminals we could achieve exact look as in 6.15.

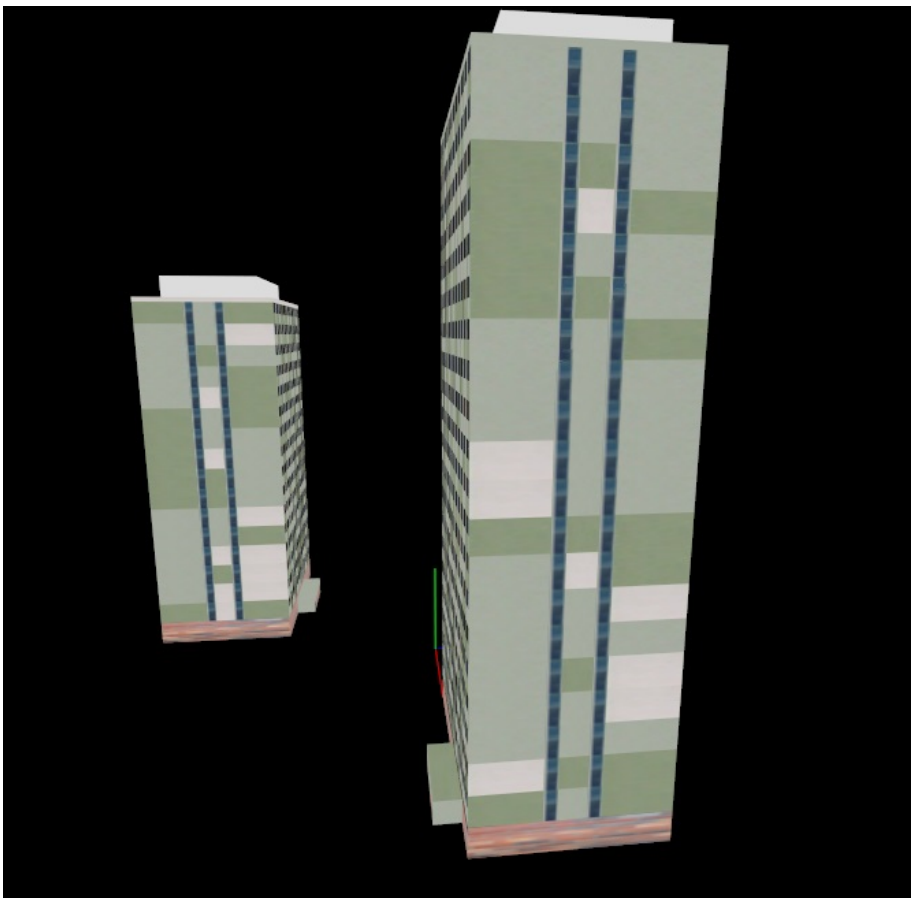

Figure 6.14: "17. listopadu", side view on both blocks.

We will now compare detailed view on our model with actual photography. Since front and back wall are identical on both pictures, we can compare them. Our model can be seen in figure 6.16, displaying both blocks, closer is block A. This picture was taken from behind of block A. In figure 6.17 we can see block A from front corner view several floors above ground. Please notice both panel and window texture variations in our model.

Figure 6.15: "17. listopadu", photography of both blocks from side view.

At our last example of "17. listopadu" dormitory we will compare another picture of our model with real photography. At figure 6.18 we can see our model loaded in MeshLab program. To achieve this, we exported our generated model first. We compare this model to figure 6.19 which was taken from opposite direction than our model. From this comparison, we can see that we achieved quite good positioning and rotation. We can also see texture diversity on our model, which looks similar to real picture. We would like to compare height on both pictures, but since our real photography was taken from ground in a big angle, we can not compare height adequately.

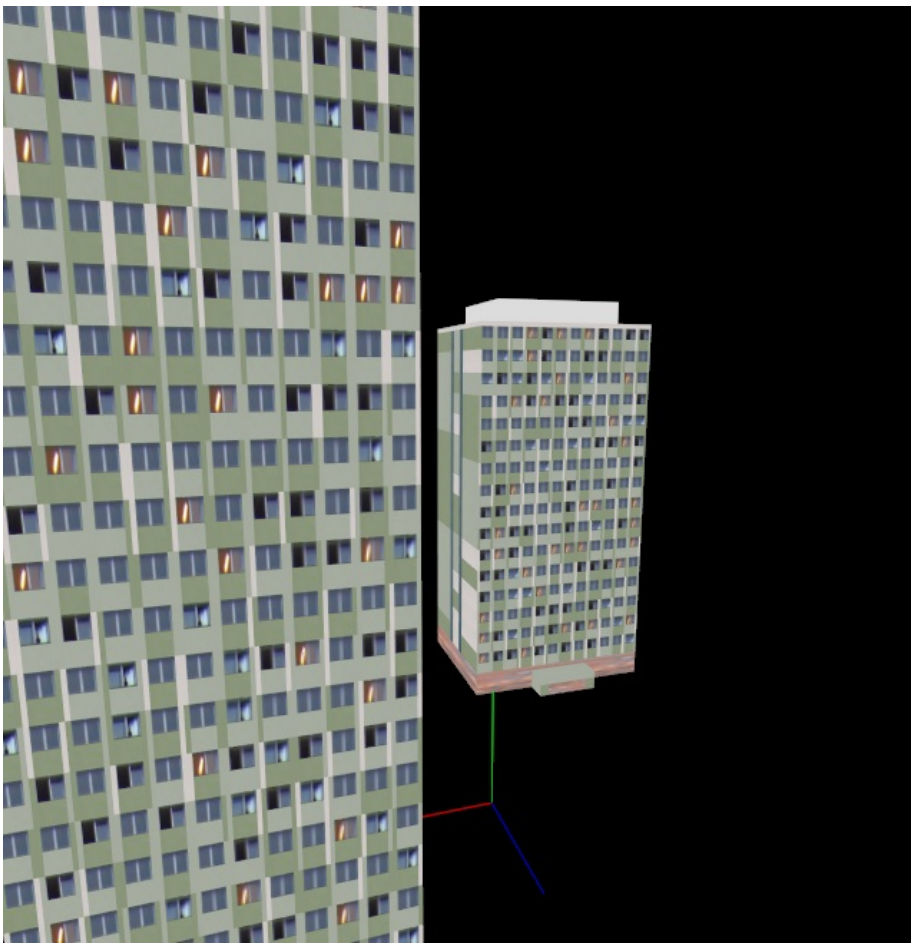

Figure 6.16: ”17. listopadu”, block A from behind.

Figure 6.17: ”17. listopadu”, photography of block A, front corner view.

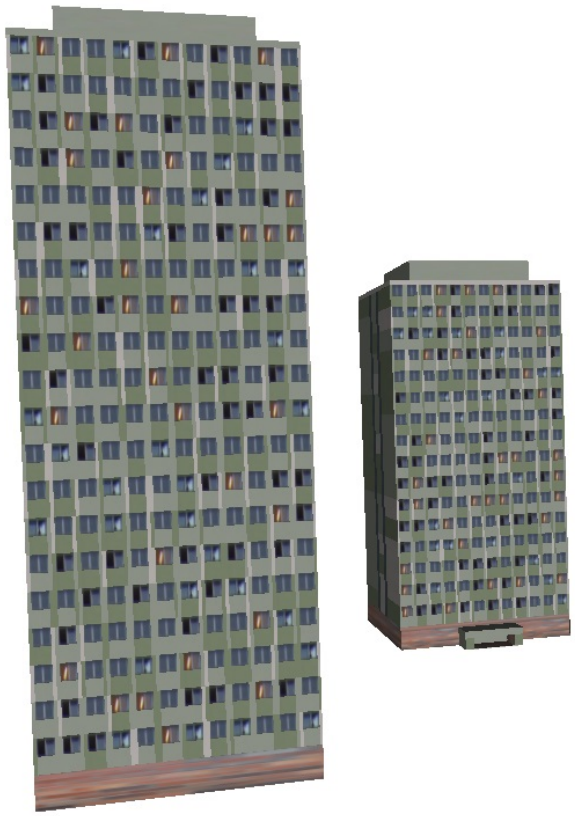

Figure 6.18: ”17. listopadu”, loaded in MeshLab, back view on both blocks, closer is block B.

Figure 6.19: ”17. listopadu”, photography of block A taken from behind.

6.2.2 Prefab building in Prague 6

In this part, we will present fully textured result. This model is also based on a real building in Prague 6. On figure 6.20, taken on this model we can see whole model from front. We can see basic structure of our model, together with various textures. We are using high resolution textures in this example to enhance visual detail on our model. On this picture we can also see a recess just under roof. This particular shape was accomplished by using extrusion into small segments referenced by non-terminal symbol *TopRecess*.

Figure 6.20: Prefab building, front view.

In next figure 6.21 we can see detailed view on balconies at our prefab model. This particular shape was created by mixing both extrusion types implemented in our program together with split rules. Firstly, we extruded already splited tile into a plain box. Secondly, by using component split, we selected front wall into scope. Thirdly, by multiple split production rules we separated center of our previously selected scope. Lastly, we used second type of extrusion to pull the wall back in the building, resulting in balcony. We can compare this result with real balconies in figure 6.22.

In our last example, we will focus on entrance. Entrance of our model can be seen in 6.23, and we can compare it to entrance of real building at 6.24. We can already see major differences. For example, top recess in our model is

Figure 6.21: Prefab building, balcony detail.

smaller than recess in real entrance. Also our model lacks side ornaments. However, our model is a good overall approximation of the original building.

Figure 6.22: Prefab building, corner view detail.

Figure 6.23: Entrance detail in our prefab building model.

Figure 6.24: An entrance of real preafab building.

6.2.3 "Na Větrníku" dormitory

At our last advanced example of our program, we will display our model of dormitory "Větrník". This model is also fully textured and combines various functionality. This model is our most complicated example. Dormitory "Větrník" is a complex of three buildings - entrance hall and two block, block one and block two. However our model consists of nine different objects. We will explain this further next part.

Figure 6.25: Entrance hall from behind on our model.

Our next pictures focuses on detailed back view of middle section. In figure 6.27 we can also see rear part of block 2. We will compare this picture with real photography in figure 6.28.

Figure 6.26: Entrance hall from behind.

Figure 6.27: Detailed view on middle section of block 2 on our model, back view.

Figure 6.28: Detailed view on middle section of block 2, back view.

Figure 6.29 shows detailed view on block 2, as it can be seen from a parking lot in reality somewhere around main entrance. We can compare this result with 6.30. We can see, that our model shows almost identical structure of various elements such as window shapes, window textures or distance between individual elements. Please note, that figure 6.29 shows block 2 from main entrance and figure 6.30 shows rear part of block one from behind, thus having different order of major elements.

Figure 6.29: Detailed view on block 2 from space before main entrance.

Last three pictures shows our model in MeshLab. In figure 6.31 we can see whole complex from back side. Please notice non-trivial shape of both blocks. This shape was accomplished by combining various objects together with rotations, translations and scaling. Both blocks contains three parts. Two are similar and creates rear and front parts. There are both created with rectangular pedestal. Middle part is also created with rectangular pedestal, but it is also perfectly positioned between front and rear part, thus creating an illusion of one single shape. Together with entrance hall and two boiler rooms located on back side of each middle part, there are total of nine different object presented on this model. On next two pictures we can see our model from two different angles. In figure 6.32 we can see entire model from front side and in distance, while in figure 6.33 we focus on back side of our model. This picture was also taken from distance to show entire model.

Figure 6.30: Detailed view of block 1, back view.

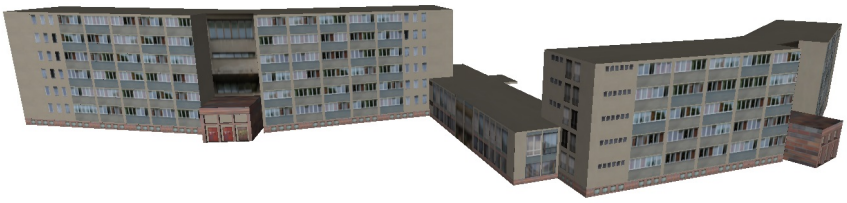

Figure 6.31: Side view of whole buildings system from behind.

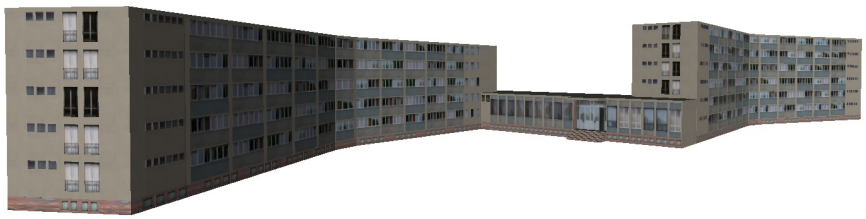

Figure 6.32: Whole dormitory Větrník from front view.

Figure 6.33: Whole dormitory Větrník from back view.

7. Conclusion

Our focus in this thesis was to implement a functional program for procedural modelling of buildings. We described method which we are implementing together with functions available to user. We also briefly described theory behind procedural modelling of buildings at the beginning of this thesis together with some other implementations. Then we tested our implementation by modelling buildings based on real buildings.

Further development of our program could include additional functions to either manipulate scope or change it, include simple roof generator or allow us to generate non-rectangle pedestals for our buildings. Also our existing functionality can be improved, e.g. using relative paths in texture command.

Bibliography

[1] Bradley, Benjamin. *Towards the Procedural Generation of Urban Building Interiors*. Master thesis, University of Hull, 2005.

[2] Reichl, Pavel. *Procedural modeling of buildings*. Diploma thesis, Masarykova univerzita, 2013.

[3] Finkenzeller Dieter, Bender Jan, and Schmitt Alfred. *Feature-based decomposition of façades. Proceedings of Virtual Concept*, 2005.

[4] Kelly Tom, and Wonka Peter. *Interactive Architectural Modeling with Procedural Extrusions. ACM Trans. Graph., vol. 30, no. 2*, 2011.

[5] Lipp Markus, Wonka Peter, and Wimmer Michael. *Interactive Visual Editing of Grammars for Procedural Architecture. ACM Trans. Graph., vol. 27, no. 3*, pp. 102-112, 2008.

[6] Lopes Ricardo, Tutenel Tim, Smelik Rubem M., Jan de Kraker Klaas, and Jan de Bidarra Rafael. *A constrained growth method for procedural floor plan generation. GAMEON'10*, 2010.

[7] Merrell Paul, and Manocha Dinesh. *Model Synthesis: A General Procedural Modeling Algorithm. IEEE Trans. Graph., vol. 17, no. 6*, pp. 715-728, 2011.

[8] Merrell Paul, Schkufza Eric, and Koltun Vladlen. *Computer-generated residential building layouts. ACM Trans. Grpah., vol. 29, no. 6*, 2010.

[9] Muller Pascal, Wonka Peter, Haegler Simon, Ulmer Andreas, and van Gool Luc. *Procedural modeling of buildings. ACM Trans. Graph., vol. 25, no. 3*, pp. 614–623. 2006.

[10] Muller Pascal, Zeng Gang, Wonka Peter, and van Gool Luc. *Computer-generated residential building layouts. CM Trans. Graph., vol 26, no. 3*, 2007.

[11] Ebert David S. *Texturing and Modeling: A Procedural Approach.. Morgan Kaufmann Series in Computer Graphics and Geometric Modeling*. Elsevier Science, 2003.

[12] Muller Pascal, Zeng Gang, Wonka Peter, and van Gool Luc. *Image-based Procedural Modeling of Facades. ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 2007.

[13] Becker S., Fritsch D., Philipp D., Baier P., and Dibak C. *Combined grammar for the modeling of building interiors. ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences, Volume II-4/W1*, 2013.

[14] Wonka Peter, Wimmer Michael, Sillion Francois, and Ribarsky William. *Instant architecture. In ACM SIGGRAPH 2003 Papers, SIGGRAPH*. New York, NY, USA, 2003.

[15] Parish Yoav I H, and Muller Pascal. *Procedural modeling of cities. In Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. Press, 2001.

[16] Prusinkiewicz Przemyslaw, and Lindenmayer Aristid. *The algorithmic beauty of plants. Springer-Verlag New York, Inc.* New York, NY, USA, 1996.

[17] Chomsky Noam. *Syntactic Structures. The Hague: Mouton.* 1957.

Attachments

With this thesis we provide CD with program with source codes, examples listed in results and unit test source codes. Program is located in directory *bachelor program*. Program can be opened in Windows 8 64-bit version.