Masarykova univerzita
Fakulta informatiky

Procedural modeling of buildings

Diploma thesis

Bc. Pavel Reichl

Brno, spring 2013

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Advisor: RNDr. Marek Vinkler

Acknowledgement

I would like to thank the advisor of my thesis RNDr. Marek Vinkler for his guidance and suggestions and I would also like to thank to Jaromír Punčochář for providing some of the models. The biggest thank belongs to my girlfriend Anna Kubalíková for her support and constant encouragement.

Abstract

This thesis describes theoretical basis of co-temporary techniques used for procedural modeling of buildings and thoroughly explains use of CGA shape grammars. An integral part of this thesis is a plug-in into a modeling tool Blender which demonstrates the usage and provides visual results of the CGA shape grammar technique.

Keywords

procedural generation, procedural modeling, CGA shape grammars, Blender, geometry synthesis

Contents

1 **Introduction** . . . 3
1.1 *Basic terminology and concepts* . . . 3
1.1.1 Procedural generation (PG) . . . 4
1.1.2 Procedural synthesis of geometry . . . 4
1.1.3 Procedural scene synthesis . . . 6
1.1.4 Notable techniques and concepts . . . 8
2 **Theory** . . . 13
2.1 *Grammar based approaches* . . . 13
2.1.1 L-systems . . . 13
2.1.2 Shape grammars . . . 15
2.1.3 Instant architecture . . . 17
2.1.4 Procedural modeling of buildings . . . 19
2.1.5 Further related works . . . 22
2.2 *Model synthesis methods* . . . 24
3 **Implementation** . . . 27
3.1 *Rules of CGA shape grammar* . . . 27
3.1.1 Rule implementation . . . 27
3.1.2 Split based rules . . . 28
3.1.3 Scope modification rules . . . 33
3.1.4 Other rules . . . 33
3.2 *Detailed models defined using CGA shape grammar* . . . 34
3.3 *Ad hoc procedurally defined models* . . . 36
3.4 *Artist made models* . . . 40
4 **Results** . . . 43
4.1 *Simple procedurally generated buildings* . . . 44
4.2 *Complex procedurally generated buildings* . . . 46
5 **Conclusion** . . . 51
Bibliography . . . 53
A **Installation** . . . 57
B **Control** . . . 59
C **Electronic attachment** . . . 61
D **Images** . . . 63

Chapter 1

Introduction

Since computer graphics has become common and sometimes even an essential element in film industry and key feature in booming video game industry a huge amount of work had to be done by 3D artists and animators.

Rising expenditures spent on growing number of artists were not the only problem that had to be dealt with. It became obvious that some scenes were not possible to be achieved by conventional tools directly controlled by artists but they had to be described in a more abstract way and the ins and outs had to be left for machine processing. For example incredible detailed jungle in movie Avatar could not be modeled without use of procedural generation [14].

One of the techniques which allows artists to work on higher level of abstraction and thus more efficiently is called a procedural generation. Techniques employing the procedural generation are used for creation of many asset types - among others storyline plotting, naming game objects (cities, characters, items), level design, synthesis of sounds and music but mainly creation of complete 3D objects including its geometry, texture and animation data.

In the first part of the thesis I would like to describe procedural generation methods used to synthesize geometry emphasizing those useful for building generation. Second part of the thesis presents implementation and results of developed plug-in Faid, which is based on one of the techniques described in first part - CGA shape grammars.

1.1 Basic terminology and concepts

This section is a short introduction into the world of procedural generation. In the next paragraphs some notable ideas, concepts and techniques are briefly described and accompanied with links to further sources of information.

1.1.1 Procedural generation (PG)

There is not an acknowledged definition and description details are domain dependent. Still one example of a definition is given at definition 1. The main feature for PG techniques is that the content must be generated algorithmically rather than manually. PG techniques are often used in real-time. This is useful because it can significantly reduce storage requirements since data are generated locally when user runs or installs application. However PG is sometimes used in process of development as it is often used as a part of content creation work-flow and its outputs are checked and further processed by artists.

Definition 1. *"Procedural techniques are code segments or algorithms that specify some characteristic of a computer-generated model or effect." [1]*

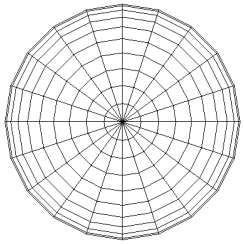
Procedural content The output data (textures, meshes, sound samples, game levels, story-plots, special visual effects etc.) of PG techniques are generally called *procedural content*.

Abstraction *Abstraction* is a commonly mentioned concept in description of PG techniques. This term has slightly flavoured meaning in context of PG. It is described as *"In a procedural approach, rather than explicitly specifying and storing all the complex details of a scene or sequence, we abstract them into a function or an algorithm (i.e., a procedure) and evaluate that procedure when and where needed."* [1].

1.1.2 Procedural synthesis of geometry

This term is used for techniques generating new objects from scratch. This methods were originally developed to mimic natural shapes like plants but later evolved into technique capable to synthesize man-made objects like buildings, road networks or urban environments [4]. Algorithms for procedural synthesis of geometry are divided into two separated groups - *Data amplification algorithms* and *Lazy evaluation algorithms* [1].

Data amplification algorithms This class of algorithms is characteristic by transformation of relatively small amount of input data into possibly huge and intricate output of intermediate geometric representation - this is known as *amplification*. Work-flow of data amplification algorithms (shown in figure 1.2) is similar to a common production of 3D computer graphics assets which works in the following steps:



(a) Geometrical intermediate representation of a model procedurally described as `sphere(1, 20, 20)`. *Parameters are in order radius, number of slices and number of stacks.*

Length of string ``sphere(1,20,20)'' is 15 characters.

Size of procedural description is 15 bytes.

$$vertices \approx slices * stacks \Rightarrow 400$$

$$triangles \approx vertices * 4 \Rightarrow 1,600$$

$$size \approx vertices * 12 + triangles * 6 \Rightarrow 14,400$$

Size of polygonal geometrical description is over 14,400 bytes.

(b) Size comparison. *It is assumed that each vertex takes 12 bytes and every triangle is specified by 3 indices each taking 2 bytes summing up to 6 bytes per triangle.*

Figure 1.1: An example of different features of procedural and polygonal representation.

1. An art director or a game designer expresses his ideas to a 3D artist.
2. A 3D artist transforms the articulation given by a submitter into intermediate geometric representation (commonly in form of polygonal meshes).
3. The output of a modeler is then rendered in a form of a picture or of an animation sequence.

The difference stems from a fact that in PG techniques the role of a modeler is no longer a human role but it is replaced by an algorithm instead. This fact changes the form of articulation given by a user. It is no longer sufficient

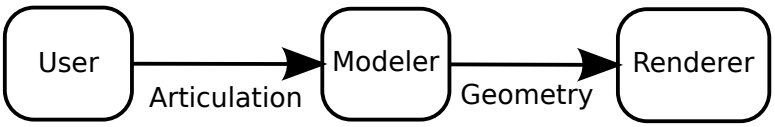


Figure 1.2: Work-flow of data amplification algorithms.

to supply input in a human capable form as a verbal description or some sketches. Precisely defined form of input data must be complied.

To emphasise the main feature of this class of algorithms repeat: the input given by a user to a modeler describing a procedural model can be as small as a few characters but the output of procedures performed by a modeler can be extremely large as it is illustrated in figure 1.1.

Lazy evaluation algorithms The main advantage of the Lazy evaluation algorithms is an avoidance to the data explosion characteristic for the Data amplification algorithms. This is accomplished for the price of more complex relation between a modeler and a renderer. A renderer takes a role of a client who claims geometry from a modeler (server). A client has to parametrize its request (customary by coordinates). A modeler does not have to generate a complete model in advance it just generates subparts asked by a client and by doing so it saves processing and memory sources. A diagram of this process is shown in figure 1.3.

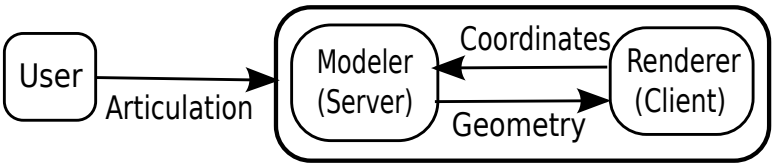


Figure 1.3: Diagram of lazy evaluation algorithms.

To demonstrate this principle in extent of the example 1.1 - a renderer would have to supply a position of an observer to a modeler which would then generate only the visible hemisphere as illustrated in figure 1.4.

1.1.3 Procedural scene synthesis

Concepts of procedural geometry synthesis (1.1.2) extend naturally to procedural scene synthesis [1]. While goal of procedural geometry synthesis is

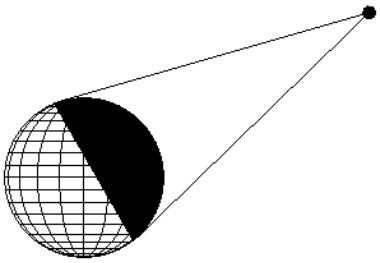
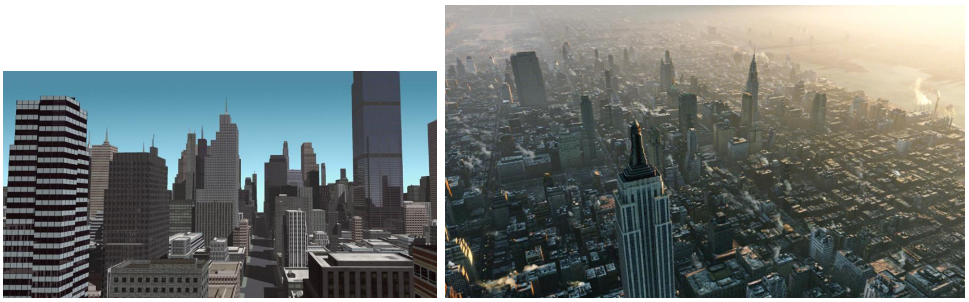


Figure 1.4: The little sphere on the right represents the position of an observer. The big sphere on the left is a model generated by a modeler. The solid part of the sphere is generated to meet demands of a renderer. The wire-framed part need not to be generated because it is not visible by the observer and thus resources can be saved.

Figure 1.5: A forest model consisting of four species of trees. Image is taken from [5].

to create a tree or a house, the goal of procedural scene synthesis is to create forests or cities. Great care must be taken to avoid an artificial appearance of generated systems. This is done by studying the area of concern and simulating the crucial relations. Simulation of forest is studied in [5]. Issues of city synthesis are described in [13]. The process of recreation of the 1930s New York for movie King Kong (2005) is the topic of [20]. Sample images are shown in figures 1.5, 1.6a and 1.6b.

Procedural level generation Procedural level generation is a subcategory of scene synthesis where generated scenes are maps for a video game fulfilling special restriction specific for a concrete game or rather a concrete game design. A level creation is traditionally job of level designers but in-



(a) Somewhere in a virtual Manhattan. The image is taken from [13].

(b) New York in early 1930s. Source King Kong (© Universal Studios 2005).

Figure 1.6: Two different instances of procedurally generated New York.

creasing number of games is being submitted with automatically generated levels. Some examples of these games are - *Diablo (1998), Diablo II (2000), Torchlight (2009), Spore (2008), Seven Kingdoms (1997)* [4]. Paper *Procedural level generation* [16] is one of a few works concerning actual techniques for generating a universally designed levels. A snap-shot of Diablo I is shown in figure 1.7.

1.1.4 Notable techniques and concepts

Following concepts are examples of techniques described in the previous text and some are used as stepping stones or sources of inspiration for techniques described in the following chapter 2.

Fractals Computer graphics and fractals are connected by a strong bound. Mathematicians were aware of fractals long before computer graphics was even established as a scientific discipline but there was no way of studying them because of their complexity. On the one hand a fractal structure is too complex to be grasped otherwise than to be visualized as an image which was not possible until computer graphics came on scene. On the other hand much of the complexity present at synthesized images delivered by computer graphics originates from the usage of fractals.

To describe fractals informally it is convenient to highlight those features which are potent to describe shapes and a phenomenon of nature manifesting some kind of a *self-similarity*. However it is not a universal tool usable for generation of nature elements such as living animals or plants. Precise definition of fractal is given at definition 2.

Figure 1.7: Snapshot of a video game Diablo. Levels in this game are procedurally generated on the fly. Every level is constructed from a set of pre-rendered tiles. Tile borders are easily distinguishable because of the step change of lighting. The image is taken from the developer homepage `http://blizzard.com`.

Definition 2. *Fractal is a geometrically complex object, the complexity of which arises through the repetition of a given form over a range of scales. [1]*

The usage of fractals can be considered as a source of an unlimited visual complexity that is generated from relatively short and simple source codes. This is sometimes called complexity-from-simplicity and it is an example of the *amplification* as described in the section 1.1.2.

Self-similarity divides fractals into separate groups [21]:

1. Exact self-similarity: the repeated pattern is the same at all scales; an example is shown in figure 1.8.

2. Statistical self-similarity: the pattern is repeated stochastically so statistical properties are preserved across scales; an example is shown in figure 1.9.

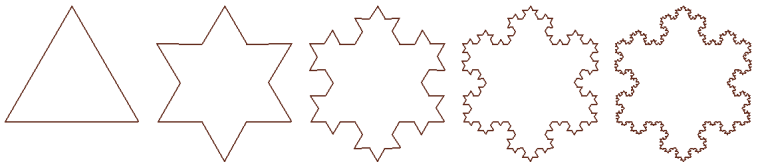


Figure 1.8: The first five iterations of exact self-similar fractal Koch snowflake (Definition is shown in figure 2.1).

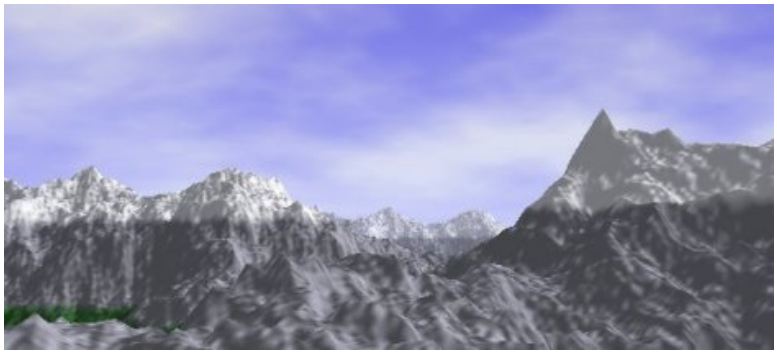


Figure 1.9: An example of a procedurally generated terrain using statistically self-similar fractals. Source `http://www.gameprogrammer.com/fractal.html`

L-systems Lindenmayer[1] system or for short called L-system is a rewriting system capable of an effective and elegant visualization of a subset of fractals. L-systems are very convenient and broadly used tool for procedural generation of plants [15]. L-systems successfully exploit the inherent common feature with vegetation - recursion. More details will be given in section 2.1.1.

Space colonization Different approach for PG of plants is based on a fact that real plants compete for sunlight. The modeling of a plant can be then viewed as a simplified simulation of growth from its root to the treetop. This method [17] has two advantages in comparison to L-systems. First, as space get colonized it is not possible that two branches (whether from the same tree or two distinct trees is not important) intersect each other. The second

1. Aristid Lindenmayer (1925 - 1989) was a Hungarian biologist who developed a formal language for description of a cell behaviour which is called after him - Lindenmayer Systems.

advantage stems from the fact that the same space or rather the same state of space colonization can be shared to generate multiple plants and thus simulate their mutual influence.

Design grammar Design grammar is a type of a formal grammar used to generate languages which are interpreted as some kind of a design plan. Examples of design grammars are shape grammars [18] used for generating designs of painting and sculptures and L-systems [15] for design of plants or building designs [11].

Chapter 2

Theory

In this section different techniques of procedural generation are described emphasizing methods suitable for building structures generation. The key section (2.1.4) of this chapter is *Procedural Modeling of Buildings* which establishes theoretical bases of implemented plug-in.

2.1 Grammar based approaches

2.1.1 L-systems

L-systems, which were briefly introduced in section 1.1.4, are parallel rewriting systems. Specifically they are a variant of a formal grammar. Main influence of L-systems on procedural techniques which generate buildings [11, 13, 23] was the demonstration of marvelous success based on the usage of design grammars. Though L-systems are not directly applicable for this goal as they are more suitable for imitation of natural growth and suffer from a problem of self-intersection. However the notion of grammar and some of the rules (scale, translate, rotate) were adopted by techniques for PG of buildings [11]. Definition of the simplest class of the L-systems - *DOL-systems*, which are deterministic and context-free, is given in the definition 3.

Definition 3. *DOL-system [15] is an ordered triple* $G = (V, \omega, P)$ *where*

- V *is an alphabet of the system,*
- $\omega \in V^+$ *is an* axiom*,*
- $P \subset V \times V^*$ *is a finite set of production rules.*

Comparison to Chomsky grammars

- The essential difference between L-systems and Chomsky grammars is the method in which the rules are applied and hence strings derived. Chomsky grammars are *sequel* which means that grammar production are applied sequentially whereas *parallel* grammar production

rules are applied in parallel and at the same time on all symbols of a designated string.[1]

- Axiom, which is used as a starting seed of a derivation, is a word unlike Chomsky grammars where the axiom is just a symbol from a set of non-terminals.
- In the definition of DOL-systems there are not any terminals like in Chomsky grammars - instead, all symbols are treated like being non-terminals. For those symbols where termination property is desired a workaround is presented in a form of the identity rule. Definition of the identity rule is given in definition 4 but it simply can be described as: *"symbols are rewritten to itself"*.

Definition 4. *For symbols which do not form left-hand side of any rule* identity rules *are assumed:*
$\forall a, b \in V : \neg(\exists\, p \in P : a \rightarrow b) \Rightarrow a \rightarrow a \in identityRules$
which are considered as a part of P *during derivation process.*

Geometry interpretation Geometry interpretation of L-systems is based on *turtle graphics (TG)* [22], which is a method for describing *vector graphics*. Turtle graphics dates back to 1960s and is connected with Logo programming language. An essential idea of TG is a concept of a *turtle* which can be considered as some kind of a pen which is controlled by simple commands such as *move forward* or *turn left*. More practical definition states: a *state* of the turtle is defined as an ordered triplet (x, y, α) where (x, y) is a tuple describing a turtle's position and α is a heading which is a direction where the turtle is facing [15]. To provide a complete specification of TG systems some parameters such as d for an actual distance of command *move forward* and δ for turning must be determined.

A turtle graphics interpretation for common symbols which are used in a simple example of L-system 2.1 is given as:

- 'F' - moves the turtle forward in its heading for distance d.
  $(x, y, \alpha) \rightarrow (x', y', \alpha)$ where $x' = x + dcos(\alpha)$ and $y' = y + dsin(\alpha)$
  Line is drawn between points $(x, y), (x', y')$.
- '+' - turns the turtle left.
  $(x, y, \alpha) \rightarrow (x, y, \alpha + \delta)$

1. At *The algorithmic beauty of plants*, which is a bible of L-systems, following explanation is provided: *"This difference reflects the biological motivation of L-systems. Productions are intended to capture cell divisions in multicellular organisms, where many divisions may occur at the same time."* [15]

- '-' - turns the turtle right.
  $(x, y, \alpha) \rightarrow (x, y, \alpha - \delta)$

$G = (V, \omega, P)$
$V = \{+, -, F\}$
$\omega = F + +F + +F$
$P = \{(F \rightarrow F + F - -F + F)\}$
$\delta = 60°$

Figure 2.1: L-system G which defines Koch snowflake at figure 1.8.

2.1.2 Shape grammars

Shape grammars were developed as a base instrument for purely visual computation. First they were used as tool aiding with design of paintings and sculptures [18] later, their ability to formalize creation of new design was studied in architecture and was a source of influence for later works such as *Instant architecture* [23].

Shape grammars are build upon a concept of a shape which is a 2D or 3D geometrical object, rather than symbol (which is basic primitive of symbolic grammars such as Chomsky grammars). To describe relations and operations used for shape grammars spatial aspects must be considered e.g. similarity, translation, scale, rotation rather than symbolic aspects [3]. Production rules of shape grammars may be applied serially (in the same manner like rules of Chomsky grammars) or in parallel (like rules of L-systems).

Automation of a rule selection and application in derivation process of shape grammars is difficult because of frequent emergence of new shapes [23]. This problem was addressed in [19] by an introduction of a novel grammar which treats shapes as symbolic objects. A definition of a shape grammar [18] is given as:

A shape grammar is a 4-tuple: (V_T, V_M, R, I) where:

- V_T is a finite set of shapes. Elements of this set are called *terminals*.
- V_M is a finite set of shapes such $V_T^* \cap V_M = \emptyset$. Elements of this set are called *non-terminals or markers*.
- R is a finite set of ordered pairs (u, v) where u is a shape consisting

of an element of V_T^* combined with an element of V_M. v is a shape consisting of one of the following:

- the element of V_T^* contained in u
- the element of V_T^* contained in u combined with an element of V_M
- the element of V_T^* contained in u combined with an addition element of V_T^* and an element of V_M

Elements of this set are called *shape rules*.

- I is an *initial* shape consisting of elements of V_T^* and V_M.

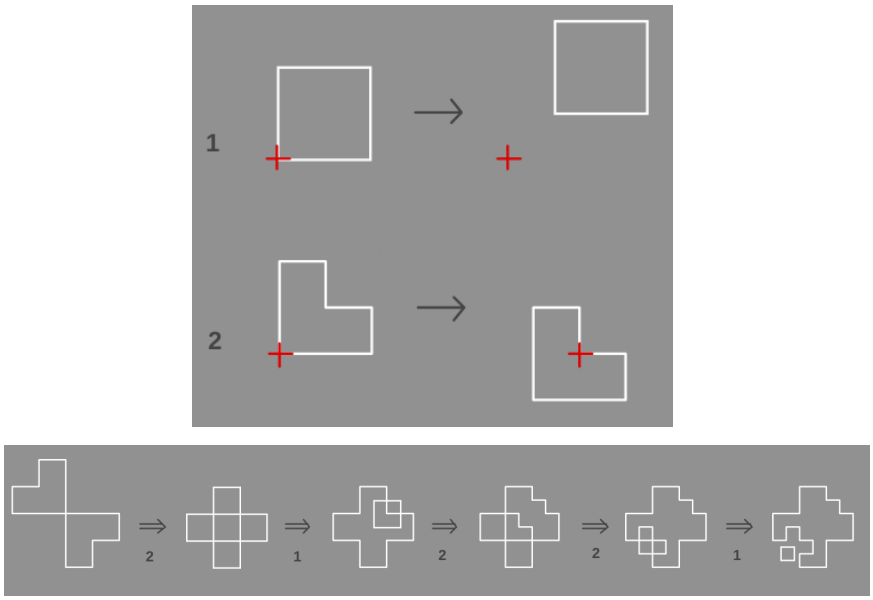


Figure 2.2: Top: Example of two shape grammar rules. Bottom: Derivation process using rules above. Pictures are taken from [10].

In the context of Chomsky grammars expression V_T^* would have semantics of the Kleene star whereas in the context of shape grammars a different meaning is supplied. For shape grammars V_T^* is defined as a set which elements are formed as a finite arrangement of one or more elements of V_T where each of them can be used multiple times with any scale and orientation. Elements of V_T^* appearing in some pair of R or in I are called *terminal shape elements*. Element of V_M are called *non-terminal shape elements*.

Rule application A process of shape generation, which is illustrated in figure 2.2, is seeded by the shape I and then it is followed by a recursive call of shape rules from R. The rule application consists of the following steps:

- Find a part of the shape that is similar to the left-hand side of any rule.
- Find geometrical transformations (e.g. scale, translation, rotation) to make the left-hand side of the rule and the part of the shape in question to completely match.
- Apply the transformations from the previous step to the right-hand side of the rule and then substitute it for the matched part of the image.

2.1.3 Instant architecture

Instant architecture [23] is a seminal paper formulating a new direction in the development of procedural modeling techniques for the building generation. 3D models are obtained using a novel grammar called *split grammar* which is a new type of parametric symbolic grammar based on the concept of the shape. Images demonstrating this technique are shown in figure 2.3.

Main contributions of this work are:

- An introduction of the split grammars which power stems from restrictions on a type of allowed rules. While grammar is powerful enough for the modeling of buildings, it is still granting an automatic and controllable process of derivation.
- A parameter matching system is defined to provide a way of specifying high-level design goals and harness randomness common for PG techniques.
- Conformity with a distribution of design ideas towards architectural patterns rather then randomness is assured via simple context free grammar called *control grammar*.

Split grammar is a special type of symbol grammar operating on shapes. Main purpose of shape grammars is to produce 3D layouts of modeled buildings in a form of attributed shapes. These shapes are used a during post-processing phase to define geometry and material used for visualization of shapes. The objects manipulated by the grammar are called basic shapes and forms basis building blocks as cuboids, polygonal cylinders and prisms.

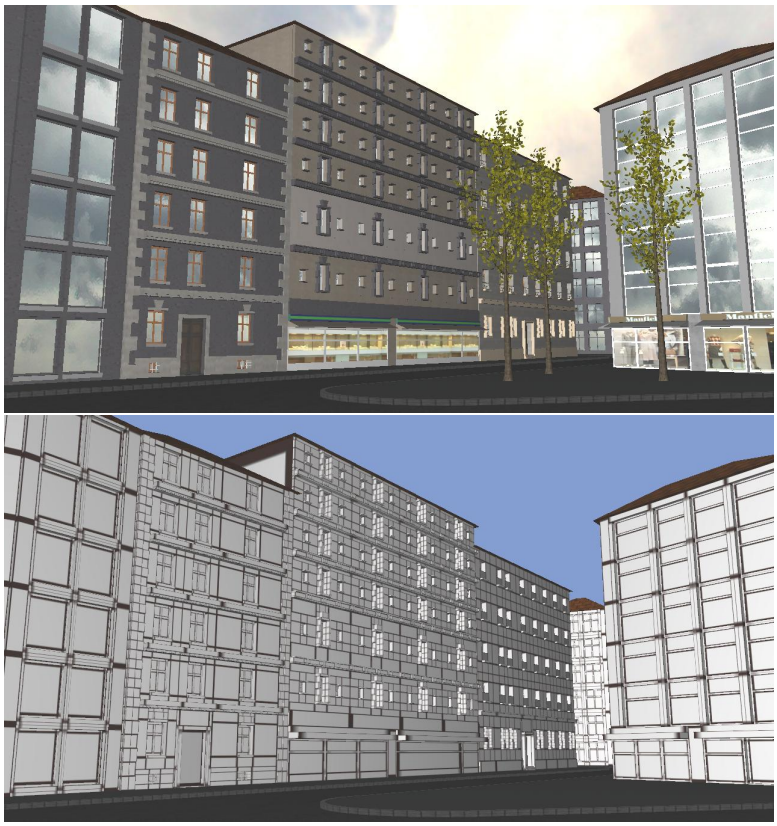


Figure 2.3: Top: Example image of buildings generated using split grammars. Bottom: Render of the same scene but in this case terminal shapes are modeled as boxes instead of 3D models. Pictures are taken from [23].

Common weakness of the design grammars viable for an automatic derivation (most notably, L-systems) is intersection of a generated structure with itself. This artefact may be tolerable for a vegetation generation, but it is strictly unwanted in modeling of architectural structures. The split grammar requires shapes created in a rule to be always a part of their parent shape's scope and by doing so deals with self-intersection problem.

User working with split grammars has the advantage of a relatively high control over the resulting models. The algorithm is configurable in these way (sorted from the most to least powerful):

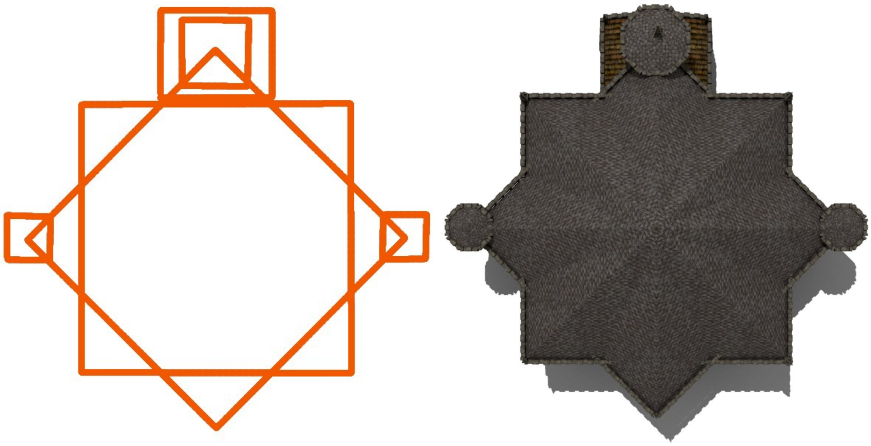
1. modification of split and control grammar rules,
2. updating attribute values for a starting shape of split grammar,

3. modification of attribute values attached to the rules of split and control grammars.

The process of a creating a complete rule set is a complex task requiring skilled designer working together with an architect. It is assumed that other users of the system will modify mostly only attributes of shapes and rules.

2.1.4 Procedural modeling of buildings

This section presents an approach based on CGA shape grammars, which are novel shape grammar developed for procedural modeling of computer graphics architecture [11]. This is the key technique implemented in the accompanied plug-in. In the following paragraphs theoretical bases of this method are described. The aspects regarding implementation details of this data amplification algorithm are topic of the next chapter 3.



(a) Mass model (b) Solid model

Figure 2.4: Top view on bounding volumes defining the mass model of a complex building is shown in figure (a). The final generated solid model is shown in figure (b) and at higher resolution in figure D.6.

Parametrization For all PG techniques a common resemblance characterizing relation between the size of input data and the possibility to control the output of an algorithm exists. Providing little input data is sufficient for some methods like Model synthesis methods [7, 8, 9] to provide some interesting results which may be sufficient and thus little user interaction is needed. However, the downside of this approach is the impossibility of

influencing the results which may appear to be too random or may look too different from what the user intended. CGA shape grammars differ considerably from methods described previously in a huge size of required input data to provide good results.

Main drawback of this method is a burden of specifying a grammar which may be cumbersome for non programmer users. The typical end users of content generating tools are 3D artists and level designers who may lack the favorable skills (scripting, programming basics) as a domain of their knowledge is more artistic. On the other hand, the mayor advantage is the high level of control over the production process enabling tunning even of the fine details of generated models by utilization of CGA shape grammar's rules.

Input data for this technique can be divided into two separate categories. The first category is represented by a set of rules describing the grammar used for derivation of output. This input category is persistent in the sense that it can be reused several times for generating different scenes of the same class of architectural elements. The other input category, which is called *mass modelling*, generally describes attributes of a scene to be generated. It is composed of bounding volumes which carry positional and bounding information as illustrated in figure 2.4. Placing of bounding volumes is not a subject of any restrictions as special cases such as intersection of bounding volumes is handled using *conditional statements* described in next paragraph.

Conditional statements The rules itself provide a powerful tool for guidance of visual form of a generated building which is augmented by a possibility to incorporate a conditional statements. The conditional statements specification is a part of rule definition statements. Process of a rule selection profits greatly from the possibility to use sophisticated restrictions on rules. This feature enables the user to specify a global condition on a whole scene and by doing so to project a surrounding neighborhood to a building appearance. Example of conditional statements are local tests for proximity of some other objects, an occlusion or some local structural constraints such as the maximal number of times the rule can be used in the derivation of a given building. The impact of the occlusion test on the generated building is shown in figure 2.5.

Shape A shape is the most important concept of CGA shape grammars. It binds together bounded space with a desired type of structure to generate in this space. Rules of CGA shapes grammar are functions which are applied

Figure 2.5: On the left image the occlusion test is not performed. On the right image the occlusion test is enabled and thus no strange window intersections are present.

to shapes and as a result return modified shape or newly generated shapes. A shape comprises of:

- A grammar symbol which is a string that identifies a shape. Symbol can be terminal or non-terminal which directly applies to shapes and divides them to two distinct groups - terminal shapes or non-terminal shapes.
- geometrical properties which are described as oriented bounding box - a scope.

Scope Scope is a term describing space properties of a shape. It is defined as a bounding volume consisting of:

- (local) position
- (local) orientation
- scale
- parent scope if any

For implementation purposes it seems practical to create a tree-like graph structure of scopes. If this is the case then a position and a orientation stored

in a scope are considered to be local. Global values of these attributes must be computed recursively via a parent scope.

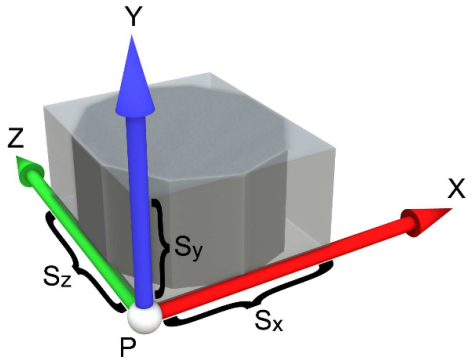


Figure 2.6: The scope of a shape. Point P defines position, axis X, Y, Z provide orientation, Sx, Sy, Sz are scalars representing scale. The image is taken from [11].

CityEngine Esri CityEngine[2] is a commercial multi-platform 3D modeling software developed for the purpose of generating 3D urban environments. According to its homepage it is used by professionals in urban planning, architecture, visualization, game development, entertainment, GIS, archeology and cultural heritage. It is a vivid example of a successful application based on PG techniques explained in this section and in previous section - *Instant Architecture* 2.1.3.[11, 12, 13].

2.1.5 Further related works

Interactive Visual Editing of Grammars for Procedural Architecture The creation of rule sets is usually done by direct editing of text files. This approach is somewhat unfriendly to non-programming users and even for programmers it is an error prone and cumbersome task. The main goal of this research paper [6] is to develop ideas enabling visual and real-time editing of rule sets and thus bind together advantages of shape grammars and traditional modeling techniques. To provide useful solution several problems had to be addressed:

2. Homepage of this project is at `http://www.esri.com/software/cityengine`.

- *Local modifications* - to adopt generated models accordingly to artist's ideas without need of changing the underlying grammar a new way of storing modification done by artists and a relation with a process of grammar derivation had to be designed.

- *Selection problem* - as an underlying PG technique is based on *split grammars* a hierarchical tree of a derivation is always produced and a selection of nodes on the same level is rather simple to be achieved. As this is often useful, a more sophisticated tool for a selection is needed - *a semantic selection*. The semantic selection is a process of selecting objects based on its properties such as a number of a floor or a column of windows on a facade.

- *Persistence* - the problem of keeping local modifications even if whole objects are regenerated or applied grammar rules are changed.

Figure 2.7: On the left: a simple image used as input. On the right: A render of a generated 3D model. Pictures are taken from [12].

Image-based procedural modeling of facades Image-based procedural modeling of facades [12] is a method based on a combination of a procedural modeling pipeline of split grammars with an image analysis resulting in a meaningful hierarchical facade subdivision. The main addressed problem is a generation of a plausible 3D model along with semantic information from a single input image. As a solution a pipeline transforming an input image into textured 3D model including analyzed structure is presented. A pipeline consists of four stages.

Facade structure detection - It is an algorithm which automatically sub-

divides a facade image into flours and tiles.[3]

Tile refinement - It is a stage in which detected tiles are subdivided into smaller regions. The method recursively selects the best splitting line of a current region. The structure subdivision is a common concept in procedural modeling [11, 23] and yields a hierarchy of elements.

Element recognition - A process of matching elements obtained in previous step with a vast 3D object library (demonstrated version contained 150 elements).

Editing and shape grammar rule extraction - In this phase a facade interpretation is encoded as a shape tree. Due to the lack of depth information editing operation for setting the depth of facade elements are required to improve the visual perception.

This method can be used for an urban reconstruction or a facade reconstruction if source images are in an adequate quality.[4] An automatic derivation of shape grammar rules from facade images is also providing compliant results. The image 2.7 shows an example of the method output.

2.2 Model synthesis methods

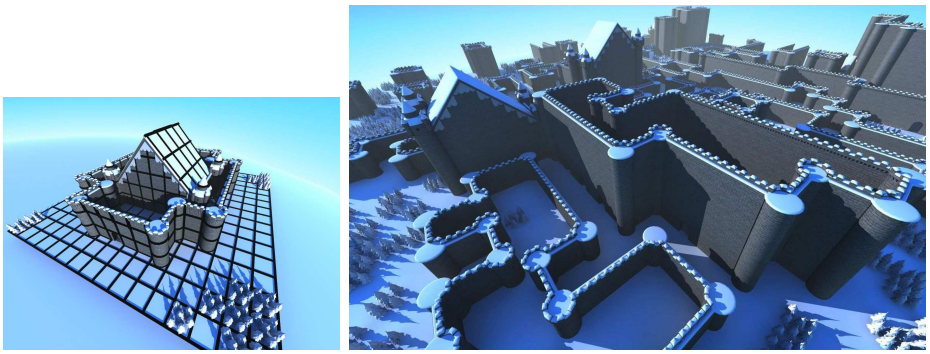
The technique called *example-based model synthesis* [7] takes a user defined model as an input and generates a larger output model that resembles a smaller input model. Model synthesis principle of operation is the same as a texture synthesis - both take a little input data and try to generate a bigger output data by repeating and expanding a contained pattern. Example is shown in figure 2.8. A significant advantage of model synthesis methods over L-systems and CGA shape grammars, which are capable of generating only a limited class of objects - plants or buildings, is universality of objects which can be generated with no need to update the method itself. The only source of a change is an input model made by an artist.

The biggest disadvantage of a model synthesis are restrictions put on the input model which must be satisfied. The model must be decomposable into a few unique building blocks so called model pieces. These can be rearranged together on a 3D grid to yield a new model which respects designers intentions. To satisfy this goal, which is called the *Consistency problem*, a set of rules must be created to assure that pieces fit together correctly and seamlessly. This is demonstrated in figure 2.9.

---

3. A tile is a common concept of procedural modeling denoting an architectural element such as a window or a door including a surrounding wall.

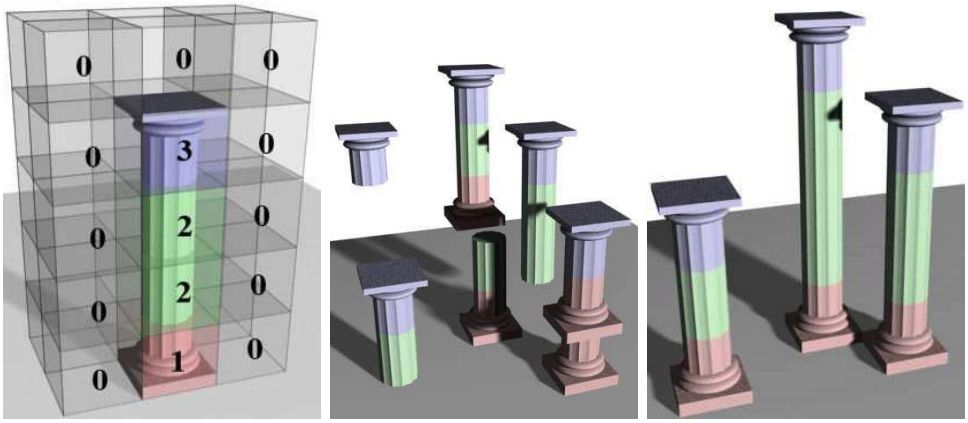
4. Video demonstration can be found at `http://www.youtube.com/watch?v=SncibzYy0b4`.



(a) (b)

Figure 2.8: The large procedurally generated model (b) is synthesised using the input model (a) created by an artist. Lines drown on figure (a) are used for highlighting subdivision of a model to model pieces. Source [7].

In the later work - *Continuous model synthesis* [8], which focused on a pattern mining from input model, a progress towards relaxing a restriction on input model properties was done - models no longer need to be axis aligned or fit a grid. Paper *Constraint-based model synthesis* [9] describes additional input data - restrictions which enable users to specify more precisely generated models in terms of dimensions or large-scale structure.



(a) The example of an input model. Every distinct model piece has its unique number.

(b) An inconsistent procedurally generated model.

(c) A consistent procedurally generated model.

Figure 2.9: Source [7].

Chapter 3

Implementation

This chapter is a summary of problems and challenges faced during development of the practical part of the thesis. The first part, which details CGA shape grammar implementation, is followed by the second part presenting two approaches on generating detailed models.

3.1 Rules of CGA shape grammar

In this section the syntax and semantics of basis rules are described. Code listings used for demonstrations are written in pseudocode to clarify the key idea and to avoid intricacies of the actual implementation.

3.1.1 Rule implementation

Shape, which is the key term of CGA shape grammar 2.1.4, is implemented as a class and its public interface methods are called *(simple) rules*. These rules are thoroughly described in subsections 3.1.2, 3.1.3, 3.1.4. For now it is important to highlight property of *(simple) rules* - all of them return a reference to the current object.

Complex rules are written using *fluent interface* [2] technique which is implemented by using of a method chaining. An example of *a (complex) rule* definition is given in listing 3.1 - *(complex) rule example* is defined by using a calling chain of shape's methods alias *(simple) rules* (*Push, Pop, T, S, I*). Visualization of this rule is shown in figure 3.1.

Listing 3.1: CGA shape *(complex) rule example* defined using *(simple) rules*

```
example = \
    Push().T(0,0,6).S(8,10,18).I(cube).Pop().\
    T(6,0,0).S(7,13,18).I(cube).\
    T(0,0,16).S(8,15,8).I(cylinder)
```

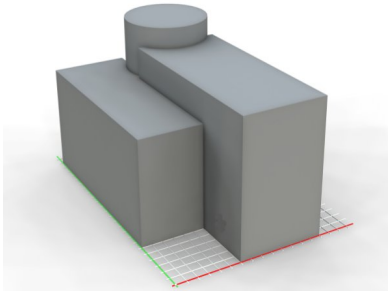


Figure 3.1: A simple model generated by the *example* rule. The image is taken from [11]

Rule selection probability While updating a rule set it is possible to associate one rule identifier with multitude of actual implementations. If this is the case then during derivation process one of implementations is picked up randomly with a uniformly distributed probability unless explicitly set otherwise as shows the following formula.

$$\frac{1 - \sum \{\, rule.chance \mid rule \in ruleset \wedge CHANCE\_SET(rule) \,\}}{|\{\, rule \mid rule \in ruleset \wedge \neg CHANCE\_SET(rule) \,\}|}$$

A simple example is illustrated in listing 3.2. The probability of selecting implementations A is the same as the probability of selecting B - 25 %. Probability of selecting implementations C is 50 %.

Listing 3.2: Setting selection probability of a rule

```
// implementation A
addRule('obj', I(cube))
// implementation B
addRule('obj', I(cylinder))
// implementation C
addRule('obj', I(sphere), chance = 0.5)
```

3.1.2 Split based rules

These rules serves mainly to divide a scope into smaller ones and generally respects the divide-and-conquer principle.

Split Split is the most important of rules as it harnesses the key concept of dividing scope into smaller ones which are further processed in a recursive manner. It takes three parameters - *axis, ratios and symbols*.

- *Axis* describes a scope axis along which division is performed.
- *Ratios* is a list of ratios used for controlling subdivision of scope along new shapes. Each element in ratios can be characterized as absolute or relative. Absolute values do not scale opposite to relative values which do scale.
- *Symbols* is a list of symbols identifying new shapes which will be created.

Listing 3.3: Rule simpleHouse

```
simpleHouse = Split(Z, [60%, 40%], [Box, Prism])
```

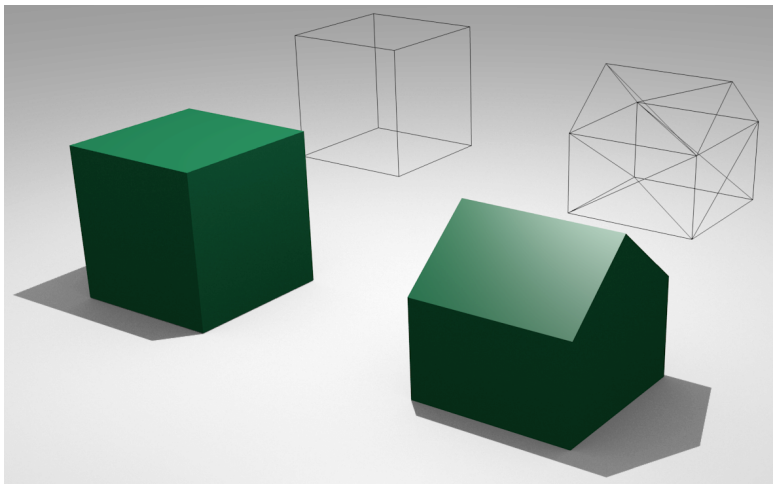


Figure 3.2: Demonstration of *simpleHouse* rule 3.3. The box on the left is a visualization of a scope for the initial shape. Objects on the right - a box and a prism are results of the rule application.

The example rule 3.3 generates a model of a very simple house which input scope is divided along axis Z in ratio 6 to 4. Newly derived scopes are associated with new shapes - Box and Prism which could be subjected

of further derivation but for sake of simplicity they are directly interpreted as geometry primitives. Visual results are shown in figure 3.2.

Computation of relative ratios An application of absolute values in a list of rations is straightforward. The computation of relative values is more demanding as definition 5 shows. To sum up the computation, it is needed to subtract all absolute values from a referred dimension and then divide the result by a sum of all relative values. Resulting number is the length of exactly one relative factor.

Definition 5. *Let* $\{r_1, ..., r_n, abs_1, ..., abs_m\}$ *be set of ratios used by split rule and* $\vec{s} = (s_x, s_y, s_z)$ *be a scale vector of the (parent) scope. The dimension of (a child) shape binded to* r_i *along axis* x *is then defined as* $r_i \frac{s_x - \sum_{i=1}^{m} abs_i}{\sum_{i=1}^{n} rel_i}$.

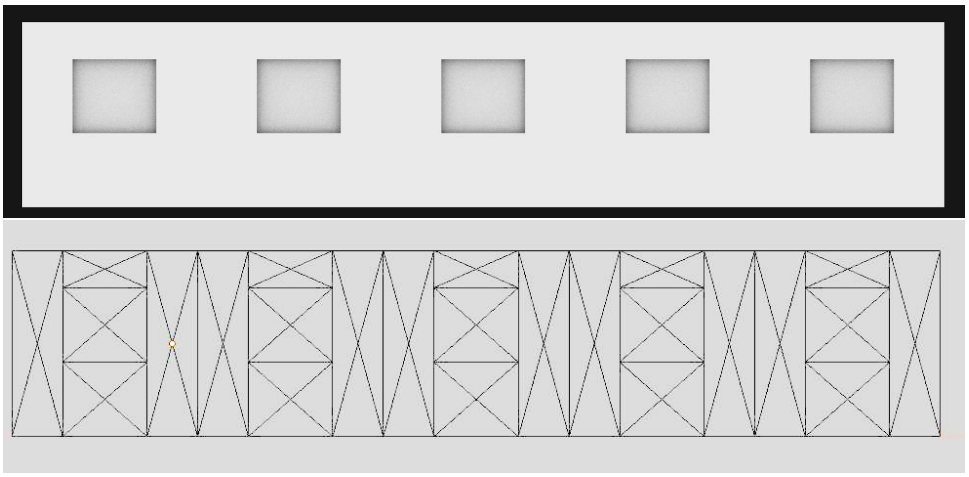


Figure 3.3: The top image is a visual result of the *simpleHouseWall* rule 3.4. The wire-frame is provided to demonstrate the influence of split rules on the underlying geometry.

Repeat The rule repeat is useful for describing patterns which should be repeated over a volume of arbitrary dimensions. A simple example could be a house wall with windows such as defined in listing 3.4 and shown on image 3.3. The repeated pattern is 'wall-window-wall' which is filling all area of the wall. Repeat rule takes four parameters - *axis*, *dimension*, *symbol and random*:

Listing 3.4: Rule simpleHouseWall

```
simpleHouseWall = Repeat(X, 2, wall-window-wall)

wall-window-wall = Split(X, [30%, 40%, 30%],\
    [Box, vertical-window-wall, Box])

vertical-window-wall = Split(Z, [40%, 40%, 20%],\
    [Box, Window, Box])
```

- *Axis* describes scope axis along which the repetition is to be performed.
- *Dimension* is an absolute dimension of pattern to be repeated.
- *Symbol* specifies new shapes which are to be created.
- *Random* optional parameter specifying randomness of newly created scopes dimensions.

Because the function of the Repeat rule is closely related to the function of the Split rule it is convenient to use in the Repeat rule underlying implementation the Split rule as it is done in listing 3.5.

Listing 3.5: Implementation of Repeat

Repeat(axis, dimension, symbol, random)
    n = $\lceil scope[axis]/dimension \rceil$
    if random
        tmp = $(\underbrace{rnd(1, random), \ldots, rnd(1, random)}_{\text{n}})$
        unit = $\frac{100}{\sum_{i=0}^{n-1} tmp_i}$
        ratios = $(tmp_0 * unit\ \%, \ldots, tmp_{n-1} * unit\ \%)$
    else
        ratios = $(\underbrace{\frac{100}{n}\%, \ldots, \frac{100}{n}\%}_{\text{n}})$
    return Split(axis, ratios, $(\underbrace{symbol, \ldots, symbol}_{\text{n}})$)

Component split Component split or *Comp* is a rule used for splitting a scope into scopes of lesser dimensions. An example is given in listing 3.6 and its visual result is in figure 3.4. The comp rule takes two parameters - *type and symbol*.

- *Type* describes new scopes. For example *side-faces* would yield all four side faces but *side-faces-x* would yield only one face (perpendicular to positive axis x).

- *Symbol* specifies new shapes which are to be created.

Listing 3.6: Rule simpleHouseBody

```
simpleHouseBody = \
    Comp('side-face-x', wall-window-wall).\
    Comp('side-face-y', wall-door-wall).\
    Comp('side-faces-nx', wall-window-wall).\
    Comp('side-faces-ny', wall-window-wall)
```

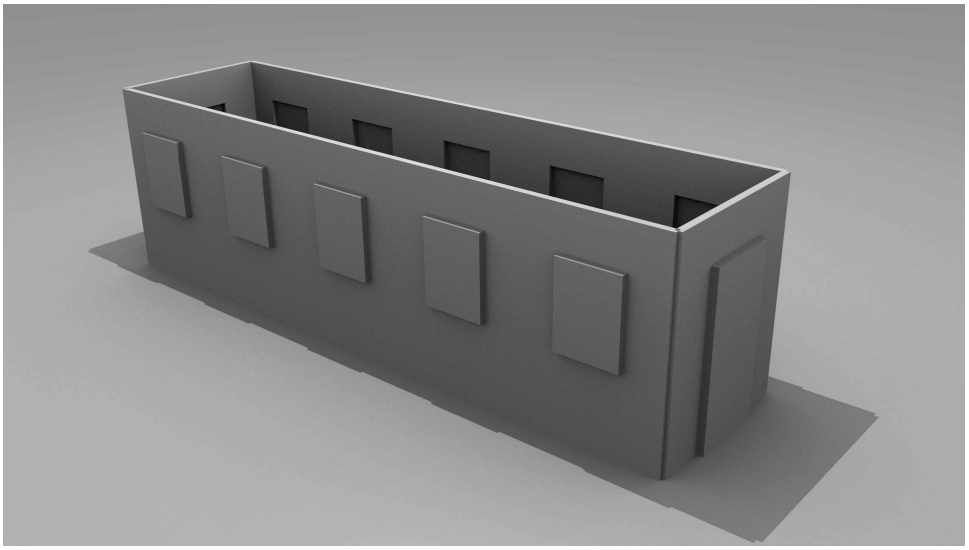


Figure 3.4: Visual result of *simpleHouseBody rule* 3.6.

3.1.3 Scope modification rules

This subset of rules was designed to modify attributes of scope such as position, orientation and scale. The functionality of presented rules is not orthogonal because the main aim of the grammar design was rather on a concise and apt form of developed rules.

Translate A translate rule is noted T and it takes three coordinates as parameters describing a desired translation of a scope along its local axis.

Scale and Size *Scale* and *Size* rules are used for modifications of scope dimensions. *Scale* works in a traditional way - it multiplies a scope scale vector by an input vector. On the other hand, *Size* is used to directly assign dimension values of a scope. It is convenient to define auxiliary rules such as *SizeX*, *SizeXadd* etc. to operate on a desired coordinate only and to perform just updates of current values.

Yaw, Pitch and Roll These rules update orientation of a scope by updating its Euler angles.

Positional Rules *Left*, *Right*, *Top*, *Down*, *Front*, *Back* are used to move a local origin of a scope to its border face which is specified as the rule input datum.

3.1.4 Other rules

Set of rules that provide other functionality from scope manipulation.

Push and Pop These rules stores/loads a current state of a scope on a stack. Any changes of the scope done between calling Push and Pop are lost after Pop is called. Using these rules is convenient if defying complex structures such as balcony (see section 3.2). Example is shown in listing 3.8 where it shades *'stone-window-frame'* from translation applied on *'iron-grid'*.

Instantiate This rule is noted I and it is used to instantiate a model specified by an input identifier. Other input parameters may include a flag to enable occlusion scope testing or attributes which are controlling a process of a procedural generation of a model. An example code is shown in listing 3.7 which generates top battlement in figure 3.11a.

Listing 3.7: Parameterized *I* rule

```
I(battlement, params = {'crenel-width' : 0.2,\
                        'merlon-width' : 0.2,\
                        'merlon-height' : 0.4})
```

Module This rule calls rule specified as its input parameter. This concept enables reusing rules and provides a way of making a structure of rules more readable. An example is shown in listing 3.8 which is a high level rule used in the process of generating the window shown in picture 3.5.

Listing 3.8: Module rule

```
stone-window-obj = \
    Push().\
    T(0,-window_pillar_dim/2,0).Module('iron-grid').\
    Pop().\
    Module('stone-window-frame')
```

3.2 Detailed models defined using CGA shape grammar

Geometry of some models is possible to be described and consequently generated directly using CGA shape grammar rules. An advantage of this approach is the uniformity of a descriptive representation opposite to ad hoc definitions used for models described in section 3.3. A disadvantage may be a computational overhead caused by the derivation process and the cumbersome formulation of a description for some shapes.

Bared window A window used in half-timbered houses as shown in figure 3.5 is an example of this technique. A pseudocode presenting concepts of its generation is given in listing 3.9. The window consists of two parts - an iron-grid and a frame. The definition of the grid is straightforward but the definition of the frame is more tricky since every generated frame differs from each other. The frame consists of four pillars and every pillar is constructed as irregular repetition (explained in 3.1.2) of randomly sized and translated bricks.

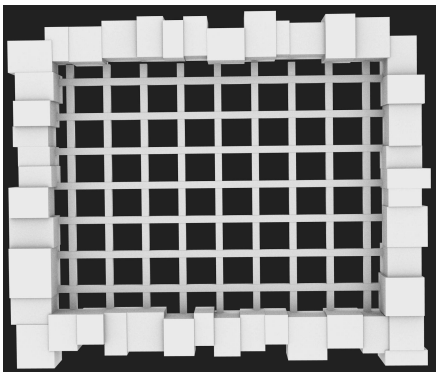


Figure 3.5: Procedurally generated window using CGA shape grammar.

Figure 3.6: Procedurally generated balcony using CGA shape grammar.

Balcony Another example of a detailed model defined only by using CGA shape grammar rules is a balcony shown in figure 3.6. The randomness of a balcony is delivered by making every plank different from the others in size and texture coordinates. A part of the wall to which balcony is attached is random in the sense that the underlying non-terminal start symbol is selected from a number of previously prepared combinations like door-window, window-door-widow, wall-door-window etc.

Tower Tower is an example of an object generated by using CGA shape grammar which sides are not mutually perpendicular. The construction of

Listing 3.9: window

```
//translation is used to avoid Z-fighting
iron-grid = \
    Repeat(X,0.1,barX)).T(0,0.001,0).Repeat(Z,0.1,barZ)
barZ = SizeZ(0.02).SizeY(0.02).I(cube)
barX = SizeX(0.02).SizeY(0.02).I(cube)

// left part of window frame
window-pillar-irr-left = \
    Repeat(Z,window_pillar_dim,\
    stone-small-brick-pillar-left,0.5)

// brick of pillar
stone-small-brick-pillar-left = \
    ResizeAligned(-X, prnd(1, 0.6)).\
    ResizeAligned(-Y, prnd(1, 0.6)).\
    T(rnd(0.01,1), rnd(0.01,1), 0).I(stone)
```

the tower is created using the *Comp* rule which is called to generate side faces of polyhedron with an n-sided polygonal base. Every side designates a scope of a new shape which type is selected depending on an order number of generated face. For example towers used in figure 3.8 and in detail in figure 3.7 have an 8-sided polygonal bases and every 3rd face determines a scope for a 'window-wall' shape while the other faces are scopes for 'simple-wall' shapes.

3.3 Ad hoc procedurally defined models

This section describes ad hoc procedurally created 3D models which are instantiated by grammar rules as described in chapter 2.1.4. An advantage of preferring these models to artist-made is that generated models fit precisely into the intended scope and there is no need to rescale and suffer a visual quality decline caused by a proportional deformation or badly mapped textures. All the models described in the following paragraphs could be generated in same manner as objects in the previous section but this approach would not be suitable due to its inefficiency. It is simple to provide a tailored routine capable of addressing specific problems of a given object. Among objects generated in this manner belong basic geometry primitives such as

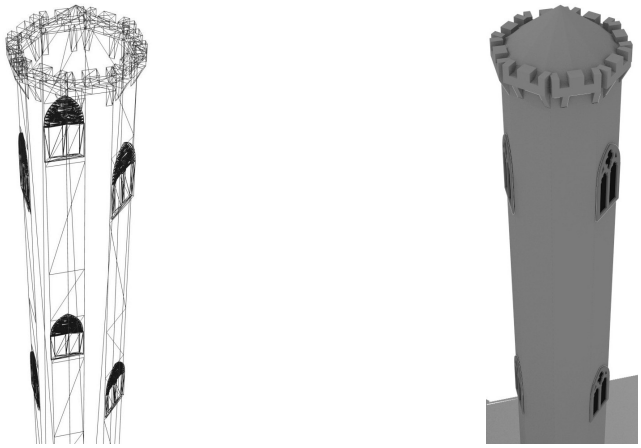


Figure 3.7: Tower with 8-sided polygonal base.

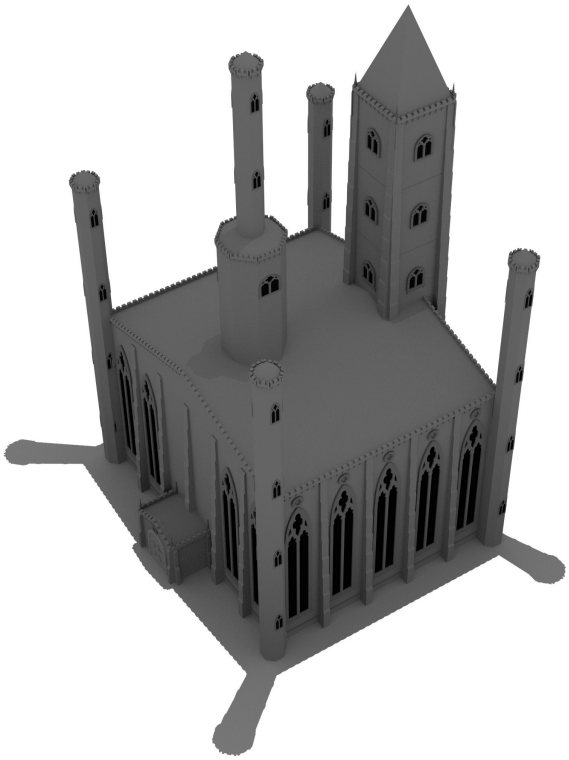
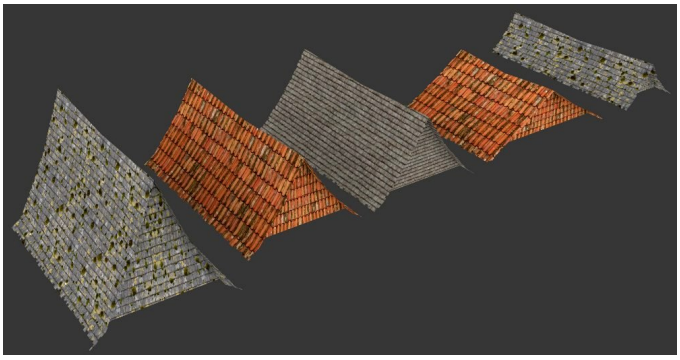
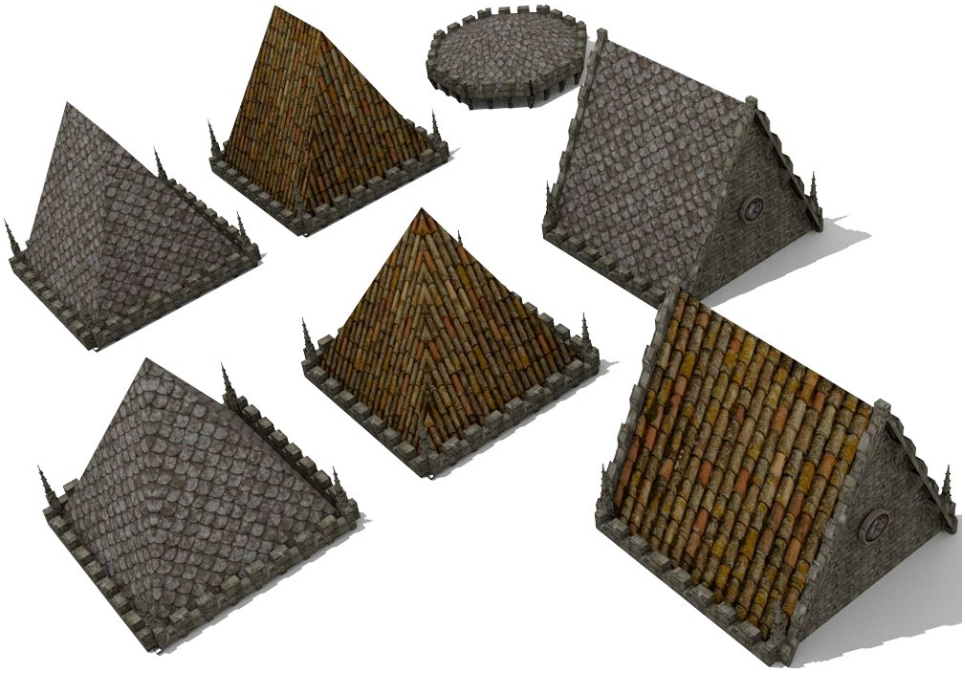


Figure 3.8: Palace with several towers.

a cube, a cone and a cylinder used as temporary substitution for a more complex object before the final solution is ready or they are used as basic elements from which the more complex structure is built.



(a) Procedurally generated roofs of half-timbered houses of various sizes. Each roof has its specific edges.



(b) Procedurally generated roofs of decorated buildings.

Figure 3.9: Procedurally generated roofs.

Roof Roofs are procedurally generated to map a texture correctly and independently on desired dimensions. A variety of generated roofs is increased by making edges irregular (picture 3.10) and applied textures are randomly selected from a set of textures. Examples of generated roofs are shown in figure 3.9.

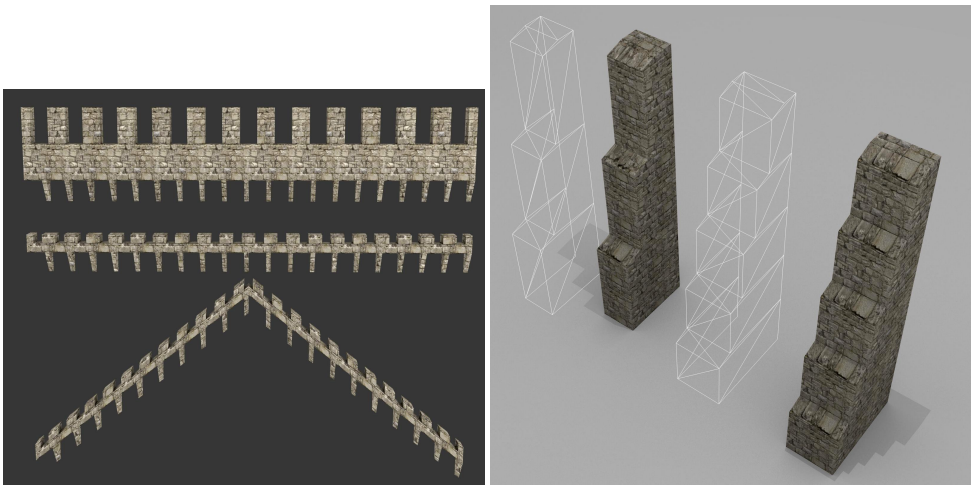
Figure 3.10: Detail of roof edge.

Battlement A battlement is a defensive structure commonly present on castle walls and towers providing a protection for defenders against attackers' arrows and bolts. A battlement consists of crenels and merlons which alternate regularly. Merlons are gaps in the wall allowing defenders to see below and crenels are a solid part of walls providing protection. Later a battlement became a decorational part of a building and its scale was changed to reflect its new function.

Procedurally generated battlements can be parameterised along with a bounding box volume by setting a *saddle* flag on to generate battlements suitable as decoration for house gables and by setting dimensions of crenels and merlons. Examples of battlements generated with distinct parameters are shown in figure 3.11a.

Buttress A buttress is a common architectural structure used since ancient times to support walls on large buildings and it also serves decoration purposes.

A geometry and texture coordinates of a buttresses model are generated from its input data consisting of a bounding volume information and a 'floor-height' which is a parameter influencing the ornate appearance of structure and thus providing a variety of possibilities to enhance the diversity of buildings. The example is shown in figure 3.11b.



(a) The battlement at the top has proportions of a defensive structure and the other two are rather ornate.

(b) Wire-frame and solid models of two procedurally generated buttresses which differ only in the value of the 'floor-height' parameter.

Figure 3.11: Procedurally generated characteristic elements of medieval buildings.

3.4 Artist made models

Models described in this section were modeled using traditional techniques. Usage of these models greatly improved a visual quality of generated models and any further expansions of this set would be very beneficial. All used models are rendered in figure 3.12.

Figure 3.12: A list of models used in the plug-in ordered from left to right: an arc door, an arc window, a pinnacle, a rose window.

Chapter 4

Results

As a functionality demonstration of the implemented plug-in two rule sets for distinct groups of procedurally generated objects were developed. The first one describes simple buildings which derivation is sufficiently seeded by just an input bounding volume.

The second group is more complex and requires a little more user collaboration. There are multiple types of building facades and several varieties of roofs which can be combined together to generate a vast number of buildings. Furthermore, the diversity is increased by the possibility to create bounding volumes with a non empty mutual intersection by the virtue of occlusion testing (section 2.1.4) used by some of the rules (currently applied only on arc windows and round windows).

(a) A non textured version of a house. (b) A textured version of the same house.

Figure 4.1: A medium multi-story half-timbered house with several balconies, a variety of framing along walls and randomly wide open shutters. A bigger version of a textured model is given in figure D.1.

4.1 Simple procedurally generated buildings

This group of buildings is represented by half-timbered houses. Half-timbered houses are residential buildings designed in medieval Europe. The structure of a building consists of exposed wood framing and the space between timbers is filled with plaster, stones or bricks. In 19th century imitations of half-timbers houses became fashionable so even today the characteristic appearance of its wood framing is not rare to be seen.

Figure 4.2: Half-timbered multi-story houses.

Figure 4.3: Simple one-story cottages.

Houses shown in figures 4.1, 4.2 and in figures D.2, D.3 are generated by using an identical set of rules, which consists of more than 100 rules and are seeded with the very same starting non-terminal symbol. The heterogeneity of generated buildings stems from two different sources. The first source of diversity are input bounding volumes, which define footprint and a number of stories. The second source are a random selection of rules (section 3.1.1) in a process of derivation and a random selection of applied textures. By performing nominal effort on updating a few rules a new type of house was delivered - *cottage* as shown in figure 4.3.

(a) Original sample image taken from `http://paradiseintheworld.com`

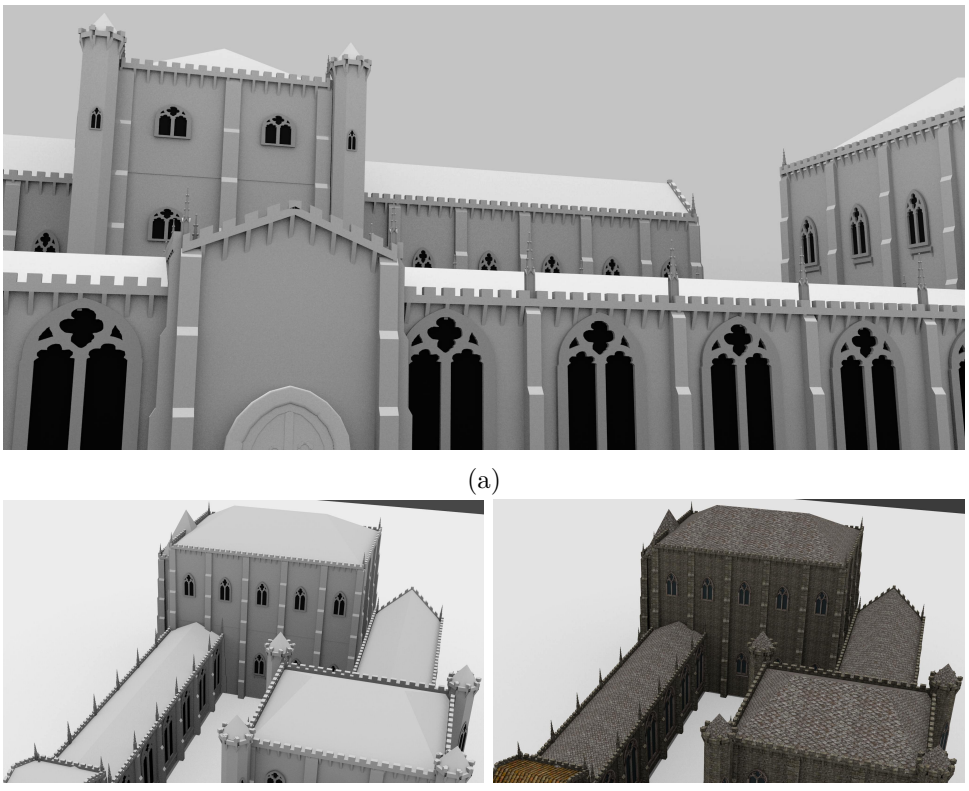
(b) Shaded geometry of generated objects.

(c) Textured generated objects.

Figure 4.4: Images of St John's College for comparison.

4.2 Complex procedurally generated buildings

These objects imitate superior structures of the medieval architecture. In this context a superior architecture is denoted as an architecture designed for noble purposes such as sacral buildings or nobility residencies. The medieval appearance has been achieved by a synthesis of characteristic medieval building elements such as buttresses 3.3, arc windows, pinnacles and battlements 3.3. Textures mimicking medieval materials were used to push up the level of fidelity. A rule set describing this class of buildings has approximately 110 rules.



(a)

(b) (c)

Figure 4.5: Detailed views on a generated model based on St John's College. A frontal view (a). An image of one of the building wings (b) - a non-textured model is provided to highlight generated geometry. An image is the same part of the building as shown in (b), but this time a textured version of model is used (c).

To provide a reader with bases for an objective quality assessment of the results derived by the implemented plug-in two samples of real world medieval style buildings were selected as a model to follow. Renders of generated *copycat* buildings are provided for a comparison with original samples.

The first example of a complex medieval building is St John's College in Cambridge[1] shown in figure 4.4. The geometry of generated models consist of more than 379,000 polygons. More detailed renders are shown in figure 4.5.

Figure 4.6: A detailed view on a generated building inspired by St Bartholomew's church in Appleby. A bigger version of this image is in image appendix D.4.

The second example is St Bartholomew's church in Appleby[2] (figure 4.7a). There is a commercial model (figure 4.7c) of this church, which was made by a professional artist utilizing traditional modeling techniques, sold on the

1. http://en.wikipedia.org/wiki/St_John%27s_College,_Cambridge
2. http://commons.wikimedia.org/wiki/Category:St_Bartholomew%27s,_Appleby

internet website TurboSquid[3] for $75.00. This model has 85,600 polygons. A generated model using a developed plug-in shown in figure 4.7b composes of approximately 72,000 polygons. To bring the generated model near the original sample as close as possible simple additional adjustments were performed using a standard modeling environment of Blender. Adjustments were mainly simple transformations (translation, scale) and duplications of some models (ledges on tower) taking only a few minutes to be done. A more detailed image is given in figure 4.6.

3. http://www.turbosquid.com/3d-models/church-appleby-max/556923

(a) Photography of St Bartholomew church. Source `wikipedia.org`.

(b) Model generated by the implemented plug-in. A bigger version is in the figure D.5.

(c) Professional artist made model. Source `http://www.turbosquid.com`.

Figure 4.7: Comparison of visual quality.

Chapter 5

Conclusion

The developed plug-in is an effective way to produce medieval style buildings which can be easily adjusted by Blender build-in tools. The amount of time taken to generate objects spans from fraction of seconds to several minutes depending on magnitude and complexity of an input scene and on a hardware configuration (in particular RAM size) of the machine where the task is run. The number of generated polygons is comparable to high poly models completely done by seasoned artists. Visual quality of generated buildings may be sufficient for an application where it is not exposed for too long or too close to a critical eye of an observer.

Further development of the plug-in would be necessarily focused on editing grammar rules and easy parametrization of ad hoc defined models. There are two means of reaching this goal. The first one is based on editing of a plain text and would require development of new language for CGA shape rules including syntax checking tools. The second approach is based on visual editing of grammar rules. Although Blender environment provides great help with customization and presentation of generated buildings incorporating either way of grammar rules adjustment is an open problem, which in my opinion might not be solved in reasonable time investment especially in the case of visual editing grammar rules.

Bibliography

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

[2] Martin Fowler. Fluentinterface. `http://martinfowler.com/bliki/FluentInterface.html`, 2005. [Online; accessed 1-May-2013].

[3] J Gips. Computer implementation of shape grammars. `http://www.shapegrammar.org/implement.pdf`, 1999. [Online; accessed 7-March-2013].

[4] Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. Procedural content generation for games: A survey. `http://www.st.ewi.tudelft.nl/~iosup/pcg-g-survey11tomccap_cr.pdf`, 2012.

[5] Brendan Lane and Przemyslaw Prusinkiewicz. Generating spatial distributions for multilevel models of plant communities. In *Proceedings of Graphics Interface*, pages 69–80, 2002.

[6] Markus Lipp, Peter Wonka, and Michael Wimmer. Interactive visual editing of grammars for procedural architecture. *ACM Transactions on Graphics*, 27(3):102–110, 2008. Article No. 102.

[7] Paul Merrell. Example-based model synthesis. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, I3D '07, pages 105–112, New York, NY, USA, 2007. ACM.

[8] Paul Merrell and Dinesh Manocha. Continuous model synthesis. In *ACM SIGGRAPH Asia 2008 papers*, SIGGRAPH Asia '08, pages 158:1–158:7, New York, NY, USA, 2008. ACM.

[9] Paul Merrell and Dinesh Manocha. Constraint-based model synthesis. In *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*, SPM '09, pages 101–111, New York, NY, USA, 2009. ACM.

[10] MIT. Course of computational design. http://ocw.mit.edu/courses/architecture/4-520-computational-design-i-theory-and-applications-fall-2005/lecture-notes/, 2005. [Online; accessed 7-March-2013].

[11] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 614–623, New York, NY, USA, 2006. ACM.

[12] Pascal Müller, Gang Zeng, Peter Wonka, and Luc Van Gool. Image-based procedural modeling of facades. In *ACM SIGGRAPH 2007 papers*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[13] Yoav I H Parish and Pascal Müller. Procedural modeling of cities. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, pages 301–308. Press, 2001.

[14] Audri Phillips. Interview with stewart mcsherry of xfrog and more. http://software.intel.com/en-us/blogs/2011/10/14/interview-with-stewart-mcsherry-of-xfrog-and-more, 2011. [Online; accessed 29-January-2013].

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

[16] T. Roden and I. Parberry. Procedural level generation. In *Game Programming Gems 5*, pages 579–588. Charles River Media, 2005.

[17] Adam Runions, Brendan Lane, and Przemyslaw Prusinkiewicz. Modeling trees with a space colonization algorithm. In *Eurographics Workshop on Natural Phenomena*, 2007.

[18] G. Stiny and J. Gips. Shape Grammars and the Generative Specification of Painting and Sculpture. In C. V. Friedman, editor, *Information Processing '71*, pages 1460–1465, Amsterdam, 1972.

[19] George Stiny. Spatial relations and grammars. *Environment and Planning B: Planning and Design*, 9(1):113–114, 1982.

[20] Chris White. King kong: the building of 1933 new york city. In *ACM SIGGRAPH 2006 Sketches*, SIGGRAPH '06, New York, NY, USA, 2006. ACM.

[21] Wikipedia. Fractal — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Fractal&oldid=540828203, 2013. [Online; accessed 2-March-2013].

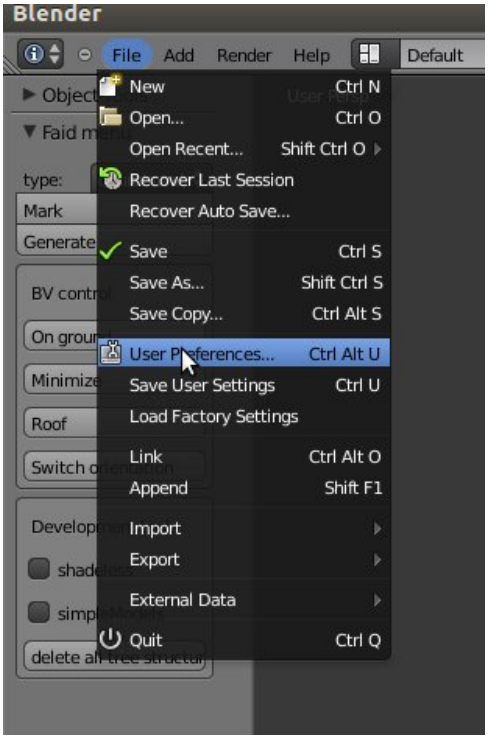
[22] Wikipedia. Turtle graphics — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Turtle_graphics&oldid=540957506, 2013. [Online; accessed 6-March-2013].

[23] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. Instant architecture. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 669–677, New York, NY, USA, 2003. ACM.

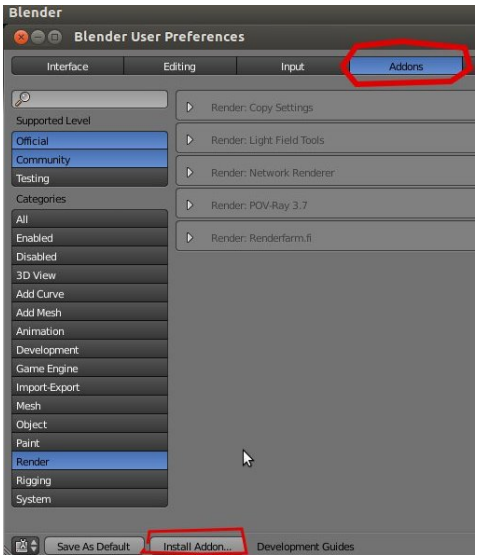
Appendix A

Installation

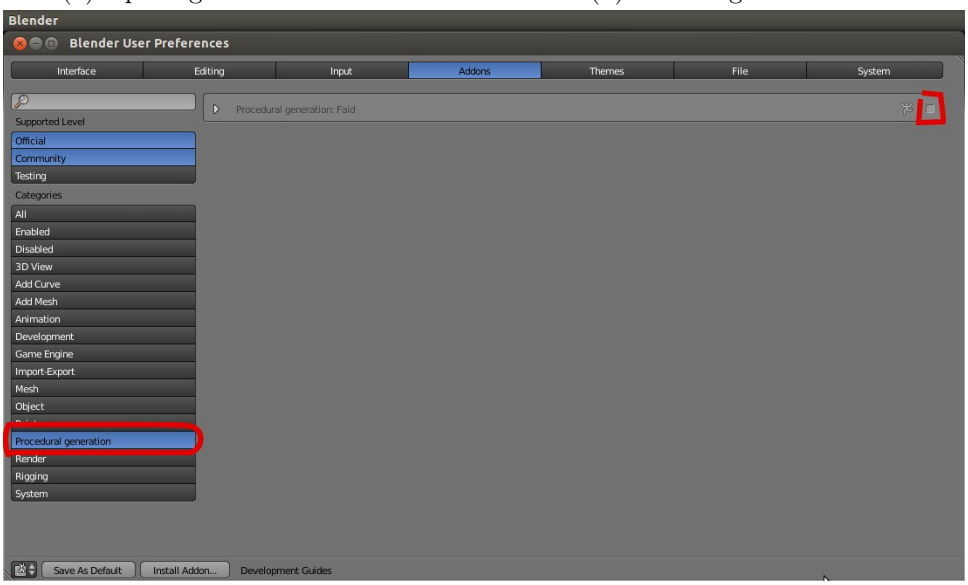
1. Save *faid-addon.zip* file, which is in an electronic archive of this thesis, locally on your computer.
2. Install Blender version 2.62 - installation files for Windows, Linux, OS X and FreeBSD are available at `http://download.blender.org/release/Blender2.62/`.
3. Run Blender.
4. From menu bar *File* select *User Preferences...* as is shown in picture A.1a.
5. From the list of buttons in the upper part of the window select the button *Addons* and then click on the button *Install Addon...* - relevant buttons are highlighted in picture A.1b.
6. In the popped up dialog window set path to *faid-addon.zip* and click on *Install Addon...*.
7. Select category Procedural generation and switch on the toggle button as illustrated in A.1c.
8. It may be helpful to click on the *Save As Default* button to make the addon available every time Blender is run.



(a) Opening *User Preferences...*



(b) Installing the addon



(c) Enabling the addon

Figure A.1: Guiding images for the addon installation

Appendix B

Control

Controls of the plug-in are accessible at the panel labeled Faid in the 3D model editor at the tools region as is shown in figure B.1.

- *Type* property sets current non-terminal.
- *Mark* creates a new bounding volume or changes a non-terminal of selected bounding volume to the current non-terminal. Bounding volumes are visible only in the layer number one. Switching current layers is done using the layer control which is shown in figure B.2.
- *Generate* (re)generates geometry for all bounding volumes in scene. Geometry is generated at the layer number two.
- BV control (bounding volume) layout groups commands which manipulate with selected or create a new bounding volume.
  - *On ground* place any selected bounding volume on ground.
  - *Minimize* sets minimal dimensions of the bounding volume (the only one named boundingVolume) regarding current non-terminal.
  - *Roof* moves and updates dimensions of the bounding volume (the one named boundingVolume) above the currently selected bounding volume. Useful for a roof placing.
  - *Switch orientation* switches the depth and the width of a bounding volume and then rotates right angle about height axis. It is useful for generated objects like saddled roofs.
- Development tools
  - *Shadeless* switches a shadeless property of all materials in a scene.
  - *Delete all tree structures* deletes a bounding volume including all child models.

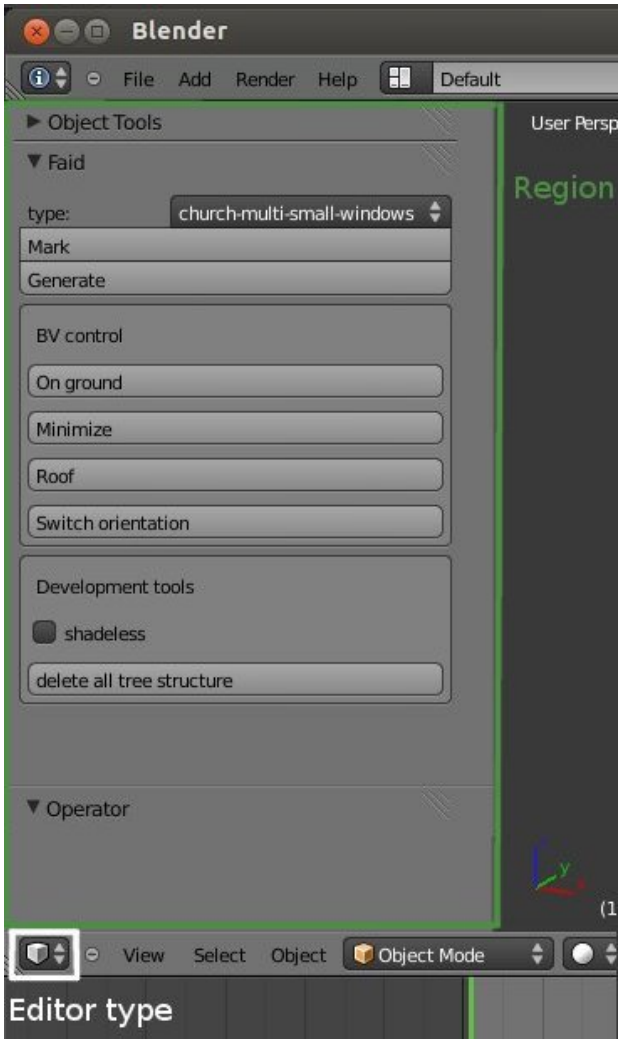


Figure B.1: A screen-shot of Blender GUI with highlighted areas of the editor type and the region.

Figure B.2: A screen-shot of Blender GUI with the highlighted layer control. The current state of the control element implies that the 1st layer is active and the 2nd layer contains some objects. Selecting is done by clicking on the layer boxes and by holding the shift key multiple layers can be activated.

Appendix C

Electronic attachment

Electronic archive of this thesis contains a file named *faid-addon.zip*, which is the developed plug-in Faid. The archive is composed of source files implementing CGA shape grammars technique, textures and models. An installation of this plug-in is the topic of the appendix A.

Appendix D

Images

Figure D.1: A half-timbered house

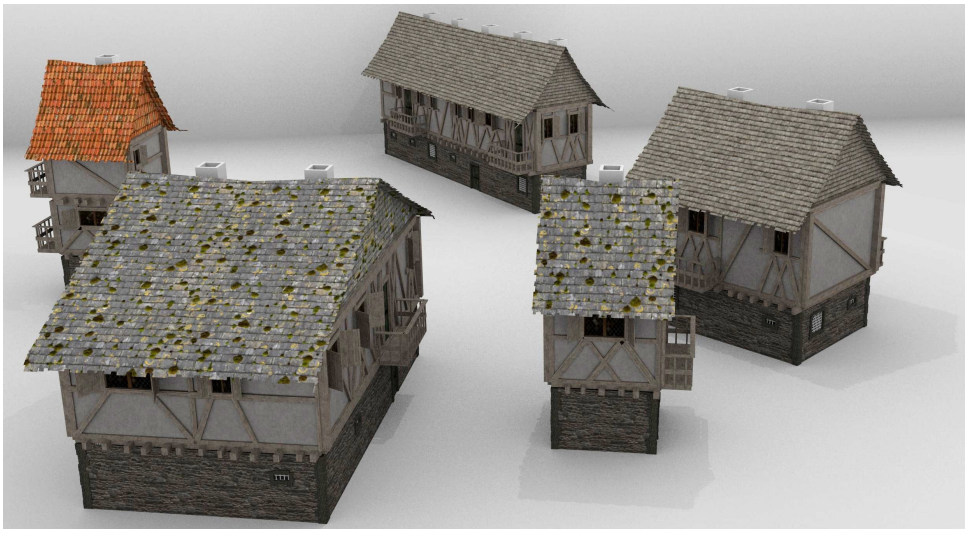


Figure D.2: A few half-timbered houses

Figure D.3: A few half-timbered houses

Figure D.4: High-resolution detail view on a procedurally generated building inspired by St Bartholomew's church in Appleby.

Figure D.5: High-resolution detail view on a procedurally generated building inspired by St Bartholomew's church in Appleby.

Figure D.6: High-resolution detail view on a procedurally generated building loosely based on Letohrádek Hvězda.