Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Master's Thesis

A Stochastic Procedural Modelling Tool for Architectural Structures

submitted by
Himangshu Saikia

submitted on

October 29, 2012

Supervisor / Advisor
Dr.-Ing. Thorsten Thormählen

Reviewers

Dr.-Ing. Thorsten Thormählen
Prof. Dr. Hans-Peter Seidel

Acknowledgements

I would like to thank my supervisor Dr.-Ing. Thorsten Thormählen, Ph.D student Arjun Jain and all my friends at Max Planck Institute and Saarland University who helped and motivated me during the time of writing my thesis.

Contents

Abstract

Procedural modelling has been shown to be extremely useful in modelling naturally occurring patterns as well as man-made artifacts. In this thesis, we introduce CastleMaker - A stochastic procedural modelling tool for architectural structures, specifically intended for castles. CastleMaker uses a procedural modelling production rule system similar to L-systems called CastleMaker Rule Language (CRL) and is influenced by exisiting rule systems like CGA Shape. Added to its ability to define a tree based hierarchical structure (also known as the Model Component Tree or MCT) which can go upto any arbitrary depth - resulting in arbitrary level of detail - CRL also has facilities for stochastic rules, which help change the very structure of the MCT in a non-deterministic way. This enables construction of structurally different but semantically similar architectures with high levels of detail. The system comes replete with features such as auto-texturing, the ability to include small generated models into larger ones, a large set of terminal shapes and a GUI based interface with syntax highlighting. CastleMaker is a very lightweight implementation with regards to application size and memory requirements, yet can be used to build a variety of complex structures for fast 3D modelling, arenas for 3D games or just to tinker around and build something for fun.

Chapter 1

Introduction

Procedural modelling is a technique by which 2D/3D patterns and/or models are created using a set of rules or procedures. These rules can either be a set of algorithms to produce self-similar structures also known as fractals, or a hierarchical definition of production rules like in an L-system. L-systems are a kind of formal grammar, which are the set of production rules defining the form for strings in that grammar. Furthermore, we only deal with context-free grammars here, which says that there should be one and only one non-terminal symbol on the left hand side of a production rule. That is, the definition of a non terminal symbol is independent of the definitions of other non terminal symbols. For example, let us consider the production rule :

$$V \to w$$

Here V is a non-terminal symbol and w is a string of non-terminal or terminal symbols (Also can be empty). Context sensitive grammar on the other hand, works on a non-terminal symbol only if it satisfies certain criteria, also given as precursors on the left hand side of the production rule. Defining the grammar in a context free way yields a simple understanding of a structure in terms of a hierarchy and is the preferred standard for other L-system like grammars.

L-systems takes this idea of production rules for strings and converts these strings to geometric patterns in 2D or 3D resulting in astoundingly similar structures as found in nature. In [PPM96], the authors have come up with L-system generated plant models and explained the mathematical theory underneath. Figure 1.1 shows an example of one such fractal plant in 2D. Figure 1.2 shows examples of 3D trees generated using this system.

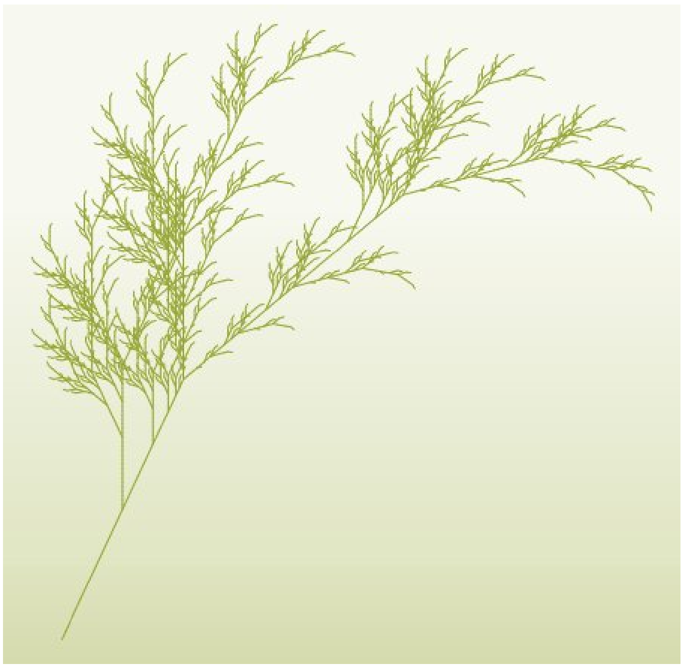


Figure 1.1: Fractal plant generated using the following rules : (X $\rightarrow$ F-[[X]+X]+F[+FX]-X), (F $\rightarrow$ FF), X is the start symbol, Number of iterations to achieve this model : 6 [Source : www.wikipedia.org/wiki/L-system]

In this thesis, we research and implement one such system aimed at architectural modelling, especially castles. We take forward ideas presented before and modify and/or add new features which logically fit in within the framework of such architectures. The idea, in simple terms, is to take in the dimensions of the building's (or structure's) bounding box, and build a rough geometric shape resembling it. And then, further breaking down this geometry into smaller and smaller segments and modifying these in turn, and keep going on in this fashion, resulting in enormous amount of detail (upto any arbitrary precision) in the final model. At any step, a segment may be further broken down, or a terminal element can be inserted. A terminal element being a small atomic part of a structure, for example a door or a window, or in case of a castle, a pointed tower roof.

Unlike L-systems though, CastleMaker does not follow a simple fractal pattern, and hence in principle, does not follow self similarity. Although it does have provisions for stochastic recursive rules similar to L-systems.

Several free and commercial software exist for procedural modelling. One of them being CityEngine which is a commercial software based on the production rule

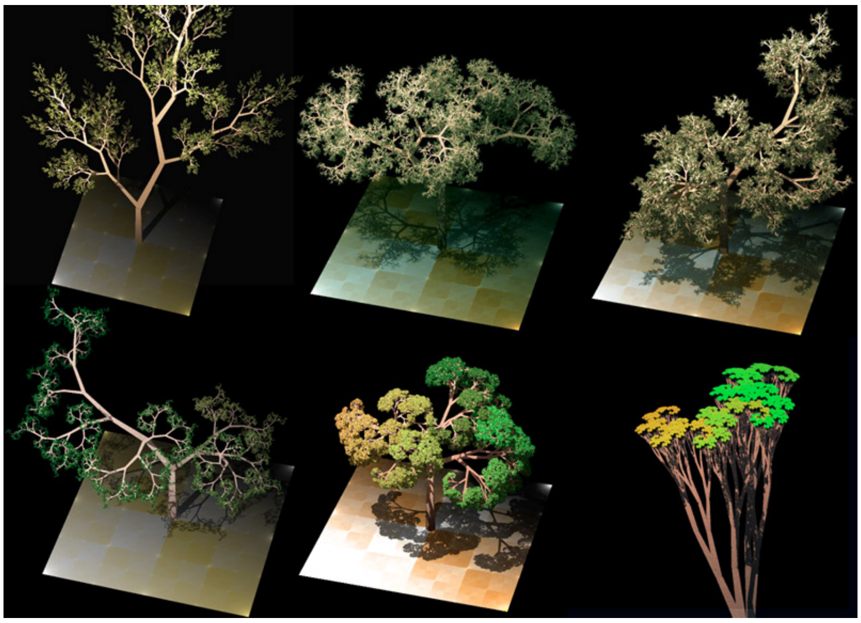


Figure 1.2: L-system trees [Source : www.wikipedia.org/wiki/L-system]

system called CGA shape developed by Müller et al. [PMG06] CGA shape has many similarities to the production rule system used in CastleMaker, viz. CRL (CastleMaker Rule Language, as explained in Chapter 3) and also a few differences.

Although it is possible to design almost any structure using CRL and can be extended to accommodate for complex rule patterns.

CastleMaker was originally intended to be a tool to interactively generate different variants of possibly medieval castles (hence the name) using a few clues about dimensions, and partial strutural data. This would allow children, for example, to marvel at how historical castles would have looked like, had they still existed today. Apart from this, the possible uses of CastleMaker can be much more. One being, generating large-scale models with arbitrary level of detail, using only a few production rules. The stochastic element which is a major feature of CRL ensures controlled randomness in parameters to functions, as well as in the entire hierarchy of structure (defined as the MCT, or Model Component Tree in Chapter 3) resulting in complex and varied structures, all produced from a single rule file. These large scale models can then be used in computer games, as arenas for example. CRL also comes with auto texturing, and a set of pre-built textured terminal shapes, especially aimed at modelling castles, which could be used directly in the production rules.

Chapter 2

Related Work

Procedural modelling has been used in several ways for different architectural design purposes. In [GCZ08] the authors have used tensor fields to create virtual street networks procedurally. Street networks as well as other aspects of large urban modelling can thus be modelled efficiently using procedural modelling.

The most well known contributions in this area has been done by Pascal Müller with the introduction of CGA Shape, a production rule system for procedural modelling of buildings[PMG06]. A production process in CGA Shape starts off with a crude volume structure which the authors refer to as the 'mass model', and goes deeper into splitting that element into its façades and adding more and more detail at every subsequent step. Just like CRL, the parameters passed to the production rules can be tweaked to generate slightly different but structurally similar components. This makes it possible to re-use existing design rules to come up with a large plethora of structurally similar architectures to populate an entire city.

A rudimentary form of procedurally modelling entire cities was presented in [PM01] wherein the authors had developed the first version of the CityEngine software. In this work they demonstrated how a 2D surface can be subdivided into building 'lots' and successively extrude buildings from these lots.

Different approaches to mass modelling of entire cities and/or larger structures from smaller ones can be observed in [Mer09] and [MM08]. These approches are different from conventional production rule based modelling in the sense that they involve modelling similar structures to the original model by slightly varying some structural attributes without violating the original meaning(structure) too much. The challenge here is to figure out the ways in which these attributes can be modified. In production rule-based modelling, on the other hand, the attributes are specified by the user in the form of production rules and the variety to modelling different complicated structures is virtually endless. Added to that is the possibility of adding very high levels of detail as we go down the hierarchical tree structure of the production process.

Another technique to generate similar models preserving semantic attributes is done in [MB12]. In this paper the authors present a new algebraic system of shape representation, in which they separate the variational parameters from the main structural attributes, and by varying these parameters can obtain similar structures. In case of CRL, this would mean randomness in parameters but no stochasticity i.e. structural elements left intact.

Although CGA shape and CRL are similar in some respects, CRL can be said to be a lighter version of CGA Shape but with powerful capabilities and great efficiency. CRL borrows from CGA Shape in the sense that the production process is similar, with a few subtle differences. Here we take a look at the noticeable differences.

CGA Shape starts off with a crude volumetric shape and subdivides it into façades and goes on further down adding new levels of detail at every step. CRL does almost the same except for the fact that it does some sort of *lazy evaluation*. This means that there is no 3D model or shape as such, until and unless some terminal symbols are added. So a simple lot production rule along with a extrude does not produce a 3D building. But only when the facades are filled with some terminal symbols does the building appear. This gives rise to a very subtle advantage. While traversing depth first into the hierarchical tree structure (defined as the Model Component Tree in Section 3) there is no need of passing the entire intrinsic information about the model but just a transformation matrix which converts the local terminal shape to its parents' co-ordinate system. This makes it simpler and faster, besides not having to deal with traversals to empty leaf nodes. The final geometric model is hence built bottom-up rather than top-down.

CRL does not however take into account of structural stability or stress factors in a structure. These factors are important to be taken into account for purposes of modelling real architectures from a virtual building design. For example, in [EWD09], the authors incorporate structural stability into their system and automatically tweak certain parameters to achieve this.

Similar work has been done in [Mos10] which was also a starting reference for CastleMaker. Here, the author develops his own procedural modelling system titled 'Building Maker' based on CGA Shape.

An interactive visual editor for procedural generation is provided in [MLW08] and CastleMaker can be extended in future to completely replace the rule editor to interactive drag and drop type functionalities for ease of use, and accessibily to a larger audience.

Chapter 3

The Procedural Modelling System

3.1 Overview

CastleMaker is a production rule based procedural modelling system built entirely from scratch. It consists of essentially three components. First, a parser, which extracts a heirarchical tree structure, hereby referred to as the **Model Component Tree *(MCT)***, from the given rule file. The second component, referred to as the modeller, which does the actual construction of the 3D model defined by the rule file. This takes in the tree structure parsed by the parser and constructs a 3D model by doing a depth first search of the tree. And the third component, also called the renderer or interface, which provides a openGL rendered view of the generated 3D model.

3.2 Rule Files

Rule files are specifications provided by the designer and/or architect about the underlying rules that conform to the 3D model. These may include building dimensions, internal layout of structures and their relative positioning with respect to each other. Rules are written in a pre-defined context free language, hereby referred to as the **CastleMaker Rule Language *(CRL)***, and files containing rules typically have the extension of *.rule*.

3.3 The Parser

The CastleMaker parser is written in **Flex** *(The Fast Lexical Analyzer)*- which parses rule files into tokens, and **Bison** *(The GNU Parser Generator)* which takes in a specification of a context free language and creates a parser which checks a particular token stream for conformance to this language.

3.4 The Modeller

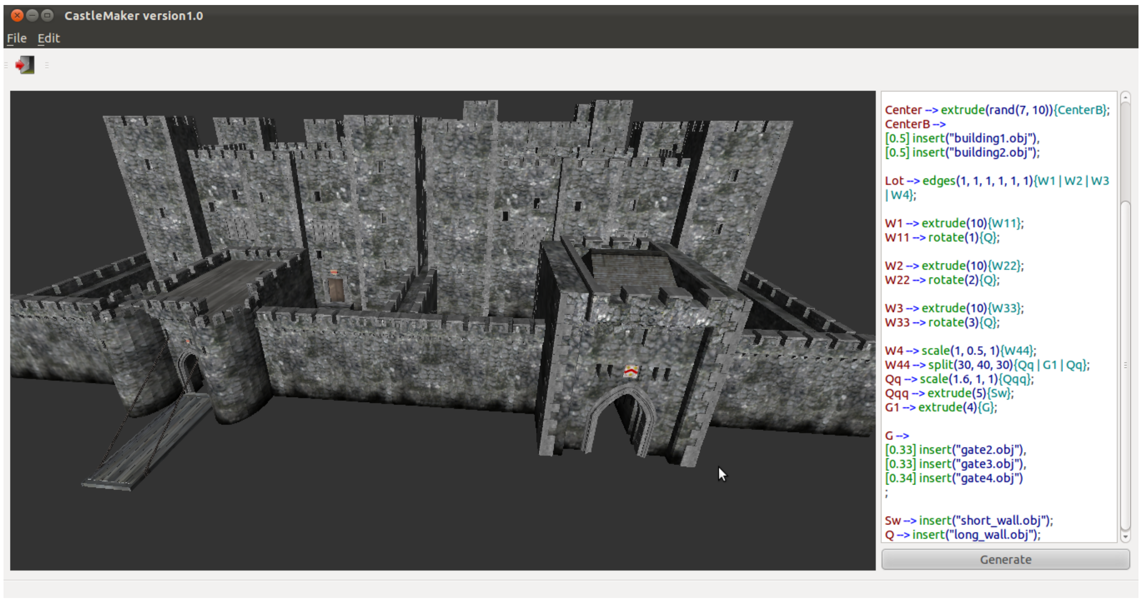
This component, being probably the most important component of the system, is responsible for creating the 3D model defined by the given rule file. It traverses the rule based specification tree top down and combines all child elements at any given root. This model data structure is then either written to a 3D Wavefront Object file *(.obj)* at the Command Prompt, or displayed as is, in the OpenGL viewer provided by the Interface to the system.

At any phase of traversal down the tree, the modeller does not store any information about the actual model, viz. the vertices, faces, texture etc. The only thing that is passed is a transformation matrix like structure also known as a *Local Transformation Node*. The model is actually evaluated only when a leaf node / terminal node is encountered. This lazy evaluation cuts down on a large chunk of memory needed, while doing a depth first traversal. All randomizations in the arguments to the rules are also done during every traversal of a node, making sure that visiting the same predecessor-successor node multiple times during the depth first traversal ensures a different value for random arguments, if any.

The modeller is written entirely in C++.

3.5 The Interface

The Interface, being the user front-end, has additional components for the user to interact with the system. Herein, is a rule-file editor panel, with built-in syntax highlighting enabled for CRL, and also, an OpenGL viewer window which displays the model generated as defined by the rules specified in the editor panel. This component is especially useful to view different variations that can be generated with one set of stochastic rules for example. Figure 3.1 shows how the Interface currently looks like.



```
Center --> extrude(rand(7, 10)){CenterB};
CenterB -->
[0.5] insert("building1.obj"),
[0.5] insert("building2.obj");

Lot --> edges(1, 1, 1, 1, 1, 1){W1 | W2 | W3
| W4};

W1 --> extrude(10){W11};
W11 --> rotate(1){Q};

W2 --> extrude(10){W22};
W22 --> rotate(2){Q};

W3 --> extrude(10){W33};
W33 --> rotate(3){Q};

W4 --> scale(1, 0.5, 1){W44};
W44 --> split(30, 40, 30){Qq | G1 | Qq};
Qq --> scale(1.6, 1, 1){Qqq};
Qqq --> extrude(5){Sw};
G1 --> extrude(4){G};

G -->
[0.33] insert("gate2.obj"),
[0.33] insert("gate3.obj"),
[0.34] insert("gate4.obj")
;

Sw --> insert("short_wall.obj");
Q --> insert("long_wall.obj");
```

Figure 3.1: A screenshot of CastleMakers' interface

3.6 Common Definitions

3.6.1 Lot

Lots are defined as the 2D surfaces upon which buildings are build upon. They are essentially the blocks in a 2D map of a city/town/building as seen directly from above. The first step to creating any model is to create the Lot on top of which the model will reside. There are different kinds of Lots defined in CRL.

LOT

A normal Lot defined directly using any of the two direct lot producing rules *rect* or *circ*.

CHILD LOT

A lot produced by any of the indirect lot producing rules, or as roof element of the *insert volume* terminal rule.

CIRCULAR CORNER LOT

Typically found in corners of rectangular lots, this is a special kind of lot which preserves equal scaling in both directions in the plane of the lot.

EDGE LOT

Produced as a result of the *edges* rule on a parent lot.

3.6.2 Handle

Handles are identifiers to components in the MCT. Every handle refers to a component which has been, or can be transformed into some other component or terminal shape. Every rule in CRL starts with only one handle on the left hand side, also known as the Predecessor, and one or more handles on the right hand side within curly braces, also known as the Successor(s). All components should have a unique handle in order for the MCT to be uniquely defined. Also all handles must start with a capital letter, and can be succeeded by any number of capital or small letters, numerals and/or underscores.

3.6.3 Leaf Nodes / Terminal Objects

Terminal Objects or Leaf Nodes of the MCT are 3D models which are simple enough to be inserted at the specified positions defined by the rule file. They are always **centered at the origin** and contained within a unit bounding box with a body diagonal running from $(-0.5, -0.5, 0)$ to $(0.5, 0.5, 1)$. This alignment is however performed by the system before a terminal object is inserted into the final model. There are two kinds of Terminal Objects :

- **Predefined Wavefront Object files (OBJs)** : These are small *.obj* files which can be inserted diectly. Typically these are thin shapes like doors, windows etc.
- **Terminal Shapes** : These are available in the system and can be modified accordingly with various parameters. These are defined later under Rule definitions.

3.6.4 Tranformation Matrix

This matrix is responsible for transforming a Terminal Object into the desired spatial co-ordinates.

3.6.5 Attributes

These are global variables which can be defined before any rule definitions and can then be used in place of any of the parameters to the rule functions. The attribute name is a string and can be a combination of lowercase letters, numerals and underscores. The attribute value can be an integer or real number. Attribute definitions are of the form :

attr attribute_name = attribute_value;

3.7 Rule Definitions

The following section will provide definitions for all rules that are implemented in this system, their syntax and working principle.

3.7.1 Lot Rules

These rules have the common format of :

1. $LotHandle = function(parameters)$; // Direct Rule
2. $ParentLotHandle \rightarrow function(parameters)\{DerivedLotHandle\}$; //Indirect Rule

CIRC

This rule is used to create a circular lot of radius *Radius* units and with center $(Center_X, Center_Y)$. Notice however that this actually does not create anything, because of the underlying lazy evaluation. *Circ* is one of the two direct lot producing rules in CRL.

Syntax : $CircularLot = circ(Center_X : real, Center_Y : real, Radius : real)$;

The transformation matrix which conforms to this rule can be written as follows :

$$T = \begin{bmatrix} Radius & 0 & 0 & Center_X \\ 0 & Radius & 0 & Center_Y \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

RECT

This rule is used to create a rectangular lot with length *Length* and width *Width* units with the bottom left corner at $(Position_{X1}, Position_{Y1})$. *Rect* is the second direct lot producing rules in CRL.

Syntax : $RectangularLot = rect(Position_{X1} : real, Position_{Y1} : real, Position_{X2} : real, Position_{Y2} : real);$

where $Position_{X2} = Position_{X1} + Length$ and $Position_{Y2} = Position_{Y1} + Width$.

The transformation matrix which conforms to this rule can be written as follows :

$$T = \begin{bmatrix} Length & 0 & 0 & \frac{Position_{X1}+Position_{X2}}{2} \\ 0 & Width & 0 & \frac{Position_{Y1}+Position_{Y2}}{2} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

CENTER

This function takes in two arguments (dimensions) and one successor handle, and results in a rectangular lot of the given dimensions, centered at the parent lots' center. Center is one of the three indirect lot producing rules in CRL.

Syntax : *ParentLot* $\rightarrow$ *center(length : real, width : real)*$\{$*CenteredLot*$\}$*;*

CORNERS

This function results in producing upto four special lots of the class **CIRCULAR_CORNER_LOT** which regardless of the scaling in either direction always maintains a circular alignment. The function takes in six parameters : a relative x-dimension (w.r.t the parent lot), a relative y-dimension and four integers which could be either 1 or 0 depending upon whether the particular corner should have a lot or not. Corners are considered in order anti-clockwise. All four corner lots can have different successor handles. This rule is typically used to insert towers at the corners of rectangular structures and is the second of the three indirect lot producing rules in CRL.

Syntax : *ParentLot* $\rightarrow$ *corners(rel_length : real, rel_width : real,* $isCorner_1$ *: bool,* $isCorner_2$ *: bool,* $isCorner_3$ *: bool,* $isCorner_4$ *: bool)*$\{CornerLot_1 \mid \ldots \mid CornerLot_n\}$*;*

EDGES

This function takes in six arguments : relative x dimension (w.r.t the parent lot), relative y dimension, and four integer values which denote the number of lots to be created along the particular edge of the parent lot in order anti-clockwise. All four edge lots can have different successor handles. But all lots lying on one particular edge will have the same successor handle. This rule is the third and last of the three indirect lot producing rules in CRL.

Syntax : *ParentLot* $\rightarrow$ *edges(rel_length : real, rel_width : real,* $howManyInEdge_1$ *: bool,* $howManyInEdge_2$ *: bool,* $howManyInEdge_3$ *: bool,* $howManyInEdge_4$ *: bool)*$\{EdgeLot_1 \mid \ldots \mid EdgeLot_n\}$*;*

3.7.2 Transformation Rules

These rules have the common format of :

$$PredecessorHandle \rightarrow function(parameters)\{SuccessorHandle\};$$

and in case of terminal objects :

$$TerminalHandle \rightarrow function(parameters);$$

EXTRUDE

This rule *extrudes* a 2D Lot into a 3D structure by a specified *Height* units. This is always the second step that has to be performed to a Lot.

Syntax : *Lot* $\rightarrow$ $extrude(Height : real)\{Building\}$;

The corresponding transformation matrix can be written as :

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & Height & 0 \end{bmatrix}$$

COMPONENT SPLIT

This rule splits a 3D volume into its 2D components, viz. the top face and the side faces. Since evaluation is lazy, the system requires the user to provide the number of side faces she wants to split the volume into.

Syntax : *Building* $\rightarrow$ *comp*$(NumberOfFaces :$ *integer*$)\{TopFaceHandle$ $|SideFaceHandle_1| \dots |SideFaceHandle_n\}$;

where the side face handles $SideFaceHandle_1$ to $SideFaceHandle_n$ are repeatedly assigned to the faces in order until all $NumberOfFaces$ faces are assigned a Handle. Thus, if only one handle is provided, all faces are assigned the same Handle.

SUBDIVIDE / VERTICAL SPLIT

This rule splits a component into a number of child components vertically. There are three flavours to this rule.

- **Normal Subdivide** : Takes in N percentage values and N successor handles and subdivides the parent into N children in the ratio of height suggested by the corresponding percentage value in order. Note that all percentage values must add up to 100.

  Syntax : *Building* $\rightarrow$ *subdiv(*Val_1 *: real,* $\dots$*,* Val_n *: real)*$\{Succ_1 \mid \dots \mid$ $Succ_n\}$*;*

- **Uniform Subdivide** : Takes in one argument *NumberOfParts* and uniformly segments the parent into *NumberOfParts* children. There can be one (mandatory) or more successor handles and assignment is done in a round robin fashion.

  Syntax : *Building* $\rightarrow$ *uniform_subdiv(NumberOfParts : integer)*$\{Succ_1 \mid \dots \mid$ $Succ_n\}$*;*

- **Repeat Subdivide** : Takes in *NumberOfParts + 1* arguments where the first argument is *NumberOfParts* and the rest being precentage values, and *NumberOfParts* successor handles. It then divides the parent into *NumberOfParts* parts and applies normal subdivision to each part according to the percentage values and assign the corresponding successor handle in order.

  Syntax : *Building* $\rightarrow$ *repeat_subdiv(NumberOfParts : integer,* Val_1 *: real,* $\dots$*,* Val_n *: real)*$\{Succ_1 \mid \dots \mid Succ_n\}$*;*

HORIZONTAL SPLIT

This rule splits a component into a number of child components horizontally. As with the vertical split operation, this rule also has three equivalent flavours. Note that correct results are only obtained when this operation is performed **after a component split**.

- **Normal Split** : Works exactly the same as normal subdivision except for in the horizontal direction.

  Syntax : *Building* $\rightarrow$ *split(*Val_1 *: real, ..., *Val_n *: real)*$\{$*Succ*$_1$ $|$ ... $|$ *Succ*$_n\}$*;*

- **Uniform Split** : Exactly the same as the vertical counterpart. Except that this one splits horizontally.

  Syntax : *Building* $\rightarrow$ *uniform_split(NumberOfParts : integer)*$\{$*Succ*$_1$ $|$ ... $|$ *Succ*$_n\}$*;*

- **Repeat Split** : Again, works like the vertical repeat split but in the horizontal direction.

  Syntax : *Building* $\rightarrow$ *repeat_split(NumberOfParts : integer, *Val_1 *: real, ..., *Val_n *: real)*$\{$*Succ*$_1$ $|$ ... $|$ *Succ*$_n\}$*;*

SCALE

This rule takes in three parameters s_x, s_y, s_z and scales a component in the X, Y and Z axes by the amounts suggested by the parameters in order.

Syntax : *Building* $\rightarrow$ *scale(*s_x *: real, *s_y *: real, *s_z *: real)*$\{$*ScaledBuilding*$\}$*;*

TRANSLATE

This rule takes in three parameters t_x, t_y, t_z and translates a component along the X, Y and Z axes by the amounts specified by the parameters in order.

Syntax : *Building* $\rightarrow$ *translate(*t_x *: real, *t_y *: real, *t_z *: real)*$\{$*TranslatedBuilding*$\}$*;*

SIMPLE LOT SPLIT

Takes in two parameters $fraction_{XSplit}$ and $fraction_{YSplit}$ in the range [0, 1] and four successor handles to produce four rectangular lots within the parent lot splitting it into the ratios $fractionXSplit$ / $(1 - fractionXSplit)$ in the X-direction and $fractionYSplit$ / $(1 - fractionYSplit)$ in the Y-direction.

Syntax : $ParentLot \rightarrow lot\_split\_simple(fraction_{XSplit}, fraction_{YSplit})\{Lot_1|Lot_2|Lot_3|Lot_4\};$

LOT SPLIT

Takes in a flavor of parameters and splits a lot either uniformly or according to the ratios specified. The uniform split, takes in two parameters $XSplits$ and $YSplits$ and splits the parent lot into $(XSplits + 1) * (YSplits + 1)$ child lots. The successor handles are assigned in a round robin fashion.

Syntax : $ParentLot \rightarrow lot\_split(XSplits, YSplits)\{Lot_1| \ldots |Lot_n\};$

The non-uniform variant takes in the first two parameters viz. $Divisions_x$, $Divisions_y$ and successively $Divisions_x$ and $Divisions_y$ parameters specifying the ratios of the splits (in percentages). The successor handles are assigned in a round robin fashion.

Syntax : $ParentLot \rightarrow lot\_split(Divisions_x, Divisions_y, Part_{x1}, \ldots, Part_{xn}, Part_{y1}, \ldots, Part_{yn})\{Lot_1| \ldots |Lot_n\};$

RANDOM LOT SPLIT

Takes in two parameters $NumberOfSplits_X$ and $NumberOfSplits_Y$ and a successor handle and splits a rectangular lot into $(NumberOfSplits_X+1)*(NumberOfSplits_Y+1)$ lots with random lenghts and widths for each child lot, assigning the successor handle to each lot. These lots can then be used with stochastic rules for example, to form different structures.

Syntax : $CityLot \rightarrow lot\_split\_rand(NumberOfSplits_X : integer, NumberOfSplits_Y : integer)\{BuildingLot\};$

ADD MATERIAL

Takes in four parameters, a material handle, a diffuse red coefficient, a diffuse green coefficient and a diffuse blue coefficient and a successor handle and assigns a diffuse material to the component having the particular successor handle. This rule can only be used just before a terminal rule. Thus the effect is to assign a material with properties as described to the terminal object following this rule. As an example a cyan ($R = 0, G = 0.5, B = 0.5$) material can be assigned to a $WindowHandle$ handle and then a terminal operation can be called on the successor i.e. the $CyanWindowHandle$ handle, such as an insert, resulting in a cyan colored window.

Syntax : *WindowHandle → add_material(material_handle : string,* $R_{diffuse}$ *: real,* $G_{diffuse}$ *: real,* $B_{diffuse}$ *: real){ColoredWindowHandle};*

where the red, green and blue coeffiecients all lie in the range [0, 1].

ADD TEXTURE

This function is used to add a texture to a terminal object. Any textures or material properties associated with the terminal object are over-ridden by this call. It takes in four parameters : A texture handle, a valid texture filename, an x-stretch factor and a y-stretch factor. The stretch factors are used to manually stretch the texture in certain cases. This is made possible by an auto-scaling mechanism which makes sure no matter how stretched structures are due to parameters specified in the rules or due to randomness, the associated texture always auto-scales to fit the overall look. For example a brick texture always looks the same size on small walls as well as big walls even though both walls inherit the same terminal symbol, in this case, a plain rectangular object. The successor handle obtained in this way can now be used to invoke any terminal operation.

Syntax : *Handle → add_texture(texture_handle : string, texture_map : string,* S_x *: real,* S_y *: real){TexturedHandle};*

INSERT

This rule inserts built-in terminal objects into the bounding box specified by the parent component. It takes in two mandatory arguments and two optional arguments. The first mandatory argument is a valid filename for a terminal OBJ file available in the DATA_DIR for the project. The second mandatory argument is a boolean flag which indicates whether the object is pre-textured or not. All OBJs to be inserted into side faces must follow the following configuration – It must lie upright in the ZX plane with the front face having normal direction (0, -1, 0). Appropriate translation and scaling is done by the system to centralize the model before insertion.

Syntax : *TerminalHandle* $\rightarrow$ *insert(terminal_obj_filename.obj : string, is_textured : bool, keep_materials_local : bool (optional), not_scale_z : bool (optional));*

The second mandatory argument *isTextured* is used to maintain child textures even if the parent texture gets modified. If this option is turned on, the textures of the the inserted model will be preserved. The first optional argument *keepMaterialsLocal* is used to enable changing of textures present in objects imported from included rule files from the parent rule file. This option is turned off by default. If turned on, changing the same material handle from within the parent rule file also changes the texture of the object included from a child rule file but maintaining its texture scaling co-ordinates. The second optional argument *notScaleZ* is used to insert 3D volumes in recursive stochastic rules for example, where an explicit extrude is not required. If this option is turned on, the height of inserted shape is not scaled to unity but kept as it is.

3.7.3 Terminal Shapes

These are common geometric shapes made available in the system which can be tweaked using various parameters.

INSERT VOLUME

This rule is used to insert a terminal volume element, which is typically a roof in most cases. It takes in 6 parameters : A relative X-dimension, a relative Y-dimension, a relative Z-dimension, an X-offset, which denotes by how much is the top face offset on both the X and -X sides, a Y-offset, which is defined similarly for the Y-dimension and an integer term which is 1 or 0 depending upon whether the volume element should be made visible or not. Combining all these parameters, several different common roof structures can be obtained.

This rule can also take in a successor handle which can then be used as a lot on the top face of the volume. Considering that it is realistic (large) enough to be used as a lot.

Syntax : *VolumeHandle* $\rightarrow$ *insert_volume(rel_length : real, rel_width : real, rel_height : real,* $offset_x$ *: real,* $offset_y$ *: real, visibility : int)*$\{[TopFace]\}$*;*

INSERT CYLINDER

This rule, as the name implies inserts a terminal cylindrical element, which are typically tower structures for example. It takes in four arguments : A (relative) bottom radius, a (relative) top radius, a (relative) height, and an integer denoting the number of divisions of the cylinder. The larger the fourth parameter, the finer the cylinder.

Syntax : *CylinderHandle* $\rightarrow$ *insert_cylinder(*rad_{bottom} *: real,* rad_{top} *: real, height : real, NumberOfDivisions : int);*

INSERT CONE

This rule is used to insert a conical element, which are typically characteristic of tower roofs for example. It takes in three arguments. A (relative) radius, a (relative) height, and an integer denoting the number of divisions of the cone. The larger the third parameter, the finer the cone.

Syntax : *ConeHandle* $\rightarrow$ *insert_cone(radius : real, height : real, NumberOfDivisions : int);*

INSERT PARABOLIC CONE

This rule is used to insert a parabolic cone element, which are typically used for more realistic looking tower roofs. It takes in four arguments. A (relative) radius, a (relative) height, an integer denoting the number of divisions along the circumference, and another integer denoting the number of divisions along the height. The larger the third and fourth parameters are, the finer the cone.

Syntax : *ParabolicConeHandle* $\rightarrow$ *insert_parabolic_cone(radius : real, height : real, NumberOfHorizontalDivisions : int, NumberOfVerticalDivisions : int);*

3.7.4 Stochastic Rules

Stochastic rules are a nice feature of CastleMaker whereby the MCT can be changed non-deterministically at any node. Currently use of guard expressions is not implemented and hence the course of traversal is altered using a probability value, which denotes the fraction of importance of that branch. All probability values for each branch should sum up to 1.

Syntax :

$$\begin{gathered}
Predecessor \rightarrow \\
[p_1]func_1(< arguments >)\{Successor_{11}, ..., Successor_{1n}\ \}, \\
[p_2]func_2(< arguments >)\{Successor_{21}, ..., Successor_{2n}\ \}, \\
\ldots \\
[p_n]func_n(< arguments >)\{Successor_{n1}, ..., Successor_{nn}\ \}, \\
;
\end{gathered}$$

where $p_1 + p_2 + \ldots + p_n = 1$

Chapter 4

Using CastleMaker : Designing a Simple Toy Castle

In this chapter, we will start from the basics and try building a simple toy castle from scratch. We will only be using basic building blocks and the simplest of terminal shapes available to us, in order to illustrate the power of the application.

To start of, as always, we initialize a lot.

$$ToyCastleLot = rect(-10, -30, 10, 30);$$

This implies that the lot is initialized and lies in the rectangular region between the points (-10, -30), (10, -30), (10, 30) and (-10, 30) on the horizontal (XY) plane.

$$ToyCastleLot \rightarrow lot\_split(0, 2)\{Left|Center|Right\};$$

The idea is to split the lot into three equal divisions along the Y-direction. Notice that there are no splits in the X-direction. The three successor handles are assigned in order.

$$\begin{gathered} Left \rightarrow extrude(20)\{LeftBuilding\}; \\ Right \rightarrow extrude(20)\{RightBuilding\}; \\ Center \rightarrow extrude(50)\{CenterBuilding\}; \end{gathered}$$

Now all the three lots are extruded upwards by the specified amounts. The center lot is extruded higher than the side lots.

$$LeftBuilding \rightarrow corners(0.13, 0.13, 1, 0, 0, 1)\{Tower\};$$
$$RightBuilding \rightarrow corners(0.13, 0.13, 0, 1, 1, 0)\{Tower\};$$
$$CenterBuilding \rightarrow corners(0.13, 0.13, 1, 1, 1, 1)\{Tower\};$$

Castle Towers are placed in appropriate positions. Notice that the lot produced is a circular corner lot and the radius of this lot is 13% of that of the parent lot.

$$LeftBuilding \rightarrow comp(4)\{Top|SideFace1\};$$
$$RightBuilding \rightarrow comp(4)\{Top|SideFace1\};$$
$$CenterBuilding \rightarrow comp(4)\{Top|SideFace1|SideFace2\};$$

The buildings themselves are split into their top and side face components. Several of the split operations will be illustrated now as operations on these façades or side faces. The center building assigns two different side faces to alternate faces. We will see how they differ later.

$$Top \rightarrow add\_material(''blue'', rand(0, 0.5), rand(0, 0.5), rand(0.5, 1))\{BlueTop\};$$
$$BlueTop \rightarrow insert\_volume(1.1, 1.1, 0.5, 1, 1, 1);$$

Now we see some randomness in action. We add a material to the Top face and assign it a high blue coefficient and a low red and green coefficient (As a reminder, the order of the diffuse co-efficients is red, green and blue). Then we add a pyramidal volume to this handle, resulting in a bluish pyramidal roof.

$$SideFace1 \rightarrow insert(''simple\_window.obj'');$$
$$SideFace2 \rightarrow subdiv(70, 30)\{Door|SideFace1\};$$
$$Door \rightarrow uniform\_subdiv(7)\{VerticalBars\};$$
$$VerticalBars \rightarrow uniform\_split(5)\{SideFace1\};$$

There are four rules that follow. Let us go through each of them one by one. The first rule is a terminal rule which inserts a simple terminal shape (in this case a colored bordered window object) and assigns it to the handle SideFace1. This results in all sides of the left and right parts of our toy castle having this window object. The second rule goes a bit further and subdivides SideFace2 in the ratio 7:3 and assigns a Door handle to the first part and a SideFace1 handle to the second part. So the top 30% of SideFace2 will contain the window object as defined by SideFace1 in the first rule. Notice that SideFace2 appears on the front and back of the center building. The third rule uniformly subdives the Door element into

7 vertical bar elements. In the fourth rule, each of these bar elements are further split horizontally into 5 segments. Each of these segments is assigned a SideFace1 handle, resulting in the same window object appearing in each of these positions. Hence the door object is essentially a combination of 35 simple window elements aligned uniformly.

Notice that the order of writing the rules did not matter and any successor definition can also appear before. As long as the MCT is consistent the order does not matter. Although it makes it much more readable if the rules are written following a consistent order.

It is also possible to randomize the number of vertical and horizontal splits by writing something like this :

$$Door \rightarrow uniform\_subdiv(rand(3,7))\{VerticalBars\};$$
$$VerticalBars \rightarrow uniform\_split(rand(2,5))\{SideFace1\};$$

This results in a random number of vertical bars (3 to 7 in this case) and a random number of horizontal segments (2 to 5) for *each* of these vertical bars due to lazy evaluation.

$$Tower \rightarrow scale(1,1,1.4)ScaledTower;$$
$$ScaledTower \rightarrow subdiv(80,20)\{TowerBody|TowerRoof\};$$

The Tower handle at the corners of the building lots are defined now. First, the towers are scaled to be a bit higher than the building height. And second, the scaled tower is subdivided into a body and a roof, in the ratio 8:2.

$$TowerBody \rightarrow$$
$$add\_material("green", rand(0,0.5), rand(0.5,1), rand(0,0.5))\{GreenTowerBody\};$$
$$GreenTowerBody \rightarrow insert\_cylinder(1,1,1,20);$$
$$TowerRoof \rightarrow$$
$$add\_material("orange", rand(0.5,1), rand(0,0.5), 0)\{OrangeTowerRoof\};$$
$$OrangeTowerRoof \rightarrow insert\_parabolic\_cone(1.5,1.5,20,10);$$

A greenish material is assigned to the tower body and a orange one to the tower roof. The tower body is essentially a cylinder with 20 horizontal divisions along the curved surface. The tower roof is a parabolic cone with 20 horizontal and 10 vertical divisions. Figure 4.1 shows how our Toy Castle looks at the moment.

So far so good. The randomizations in the colors can be seen very well. Let us now explore some stochastic rules. To this end, we design a simple toy border around our castle.

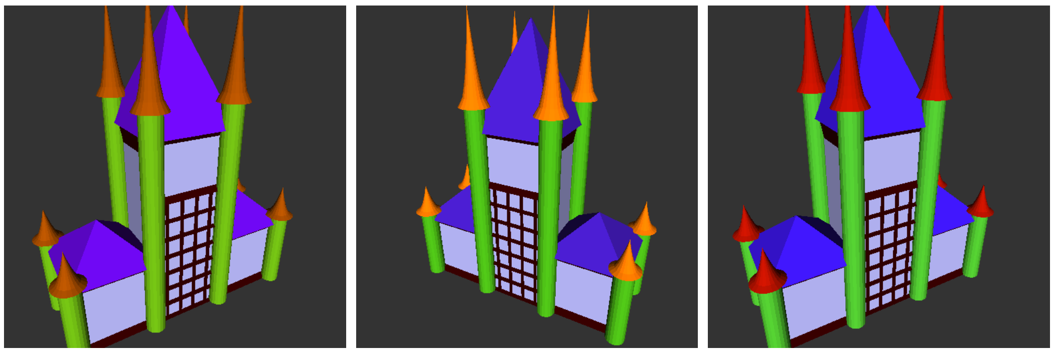


Figure 4.1: This is how our Toy Castle looks at the moment. All three models are generated from the same set of rules described above.

$$BorderLot = rect(-60, -60, 60, 60);$$

We call this lot the BorderLot. It is a square lot encompassing the entire area of our castle.

$$\begin{gathered} BorderLot \rightarrow add\_material("grass", 0, 0.4, 0)\{Lawn\}; \\ Lawn \rightarrow insert\_volume(1, 1, 1, 0, 0, 1); \end{gathered}$$

Here some green color is added to the entire border lot and a thin volume is inserted. This gives the feel of a grassy outdoor lawn surrounding the castle.

$$BorderLot \rightarrow edges(0.03, 0.03, 25, 25, 25, 25)\{BlockLot\};$$

Now 25 new lots are created at the edges of the border lot. They occupy 3% of the dimensions of the parent lot. These 100 new lots will be used to showcase the stochastic modelling feature of CastleMaker.

$$\begin{gathered} BlockLot \rightarrow extrude(rand(3, 8))\{Stone\}; \\ Stone \rightarrow \\ add\_material("walls", rand(0, 0.5), rand(0, 0.5), rand(0, 0.5))\{ColoredStone\}; \end{gathered}$$

First the lots are extruded by a random amount between 3 and 8 height units upwards. We assign a Stone handle to each of these volumes. These stones are all then assigned a random dark color.

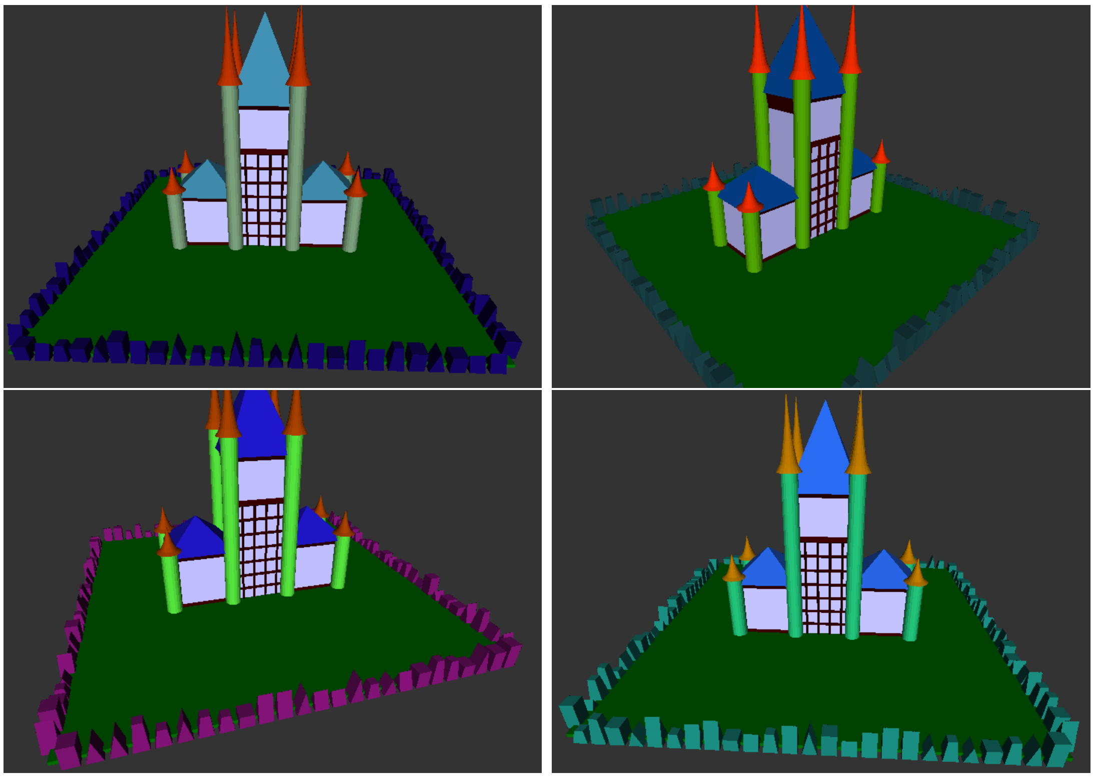


Figure 4.2: Final toy castle with borders. All of these models are generated from the same set of rules above.

$$\begin{gathered}ColoredStone \rightarrow [0.25]insert\_volume(1,1,1,0,0,1),\\ [0.25]insert\_volume(1,1,1,0,1,1),\\ [0.25]insert\_volume(1,1,1,1,0,1),\\ [0.25]insert\_volume(1,1,1,0.4,0.4,1)\\ ;\end{gathered}$$

Here the stochastic rules come into play. Each colored block is assigned a one out of four different volumes with an equal probability (25%). The first one being a regular cuboid, the second one a gabled roof element with straight faces in the X direction, the third, a gabled roof element with straight faces in the Y direction and the fourth a mansard roof element with an offset of 0.4 on both sides.

Figure 4.2 shows how the final castle models look like. Here we demonstrated some of CastleMaker's features at a very basic level. The level of detail that can be added in this way can be arbitrarily large, and is given by the depth of the MCT. Although it is suggested that recursive rules (Same predecessor and successor handle) with a

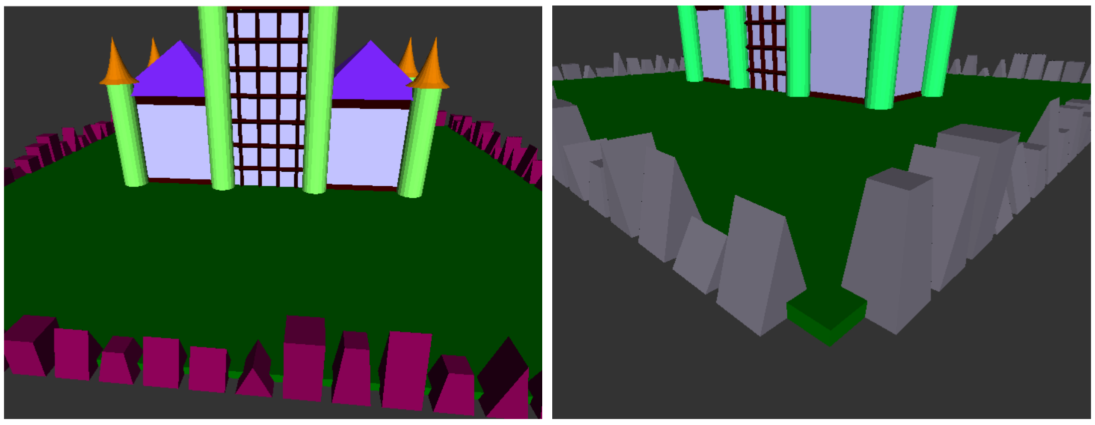


Figure 4.3: A closer look at the stone border. All positions take up one out of the four possible volume structures at random

high probability should be avoided. Recursive rules which are not stochastic should never be used. Figure 4.3 shows the results of the stochastic rules from a closer angle.

Chapter 5

Stochastic Rules

5.1 Definition

As described in previous chapters, stochastic rules help change the very structure of the MCT in a non-deterministic way. A drawback of the system worth mentioned here is the absence of conditional statements to determine which path a particular branch should take. The stochastic rules make this decision depending upon a certain fixed probability measure which is hardwired into the rule file. Modifying these values can result in quite different final models and almost equally distributing the weights (values) across all branches results in the maximum stochasticity.

5.2 Usage

Stochastic rules, however, should be used with some caution. For example, having a recursive (non-deterministic) loop in the MCT is often dangerous if the stochastic weights assigned to the branches making the loop are high. For example consider this self loop syntax:

$$\begin{gathered} A \rightarrow \\ [0.8]insert\_volume(1,1,1,0,0,1)\{A\}, \\ [0.2]insert\_volume(1,1,1,1,1,1) \\ ; \end{gathered}$$

This might result in an incredibly large number of recursions and the MCT going really deep. Also dangerous are high weighted rules which branch into a number

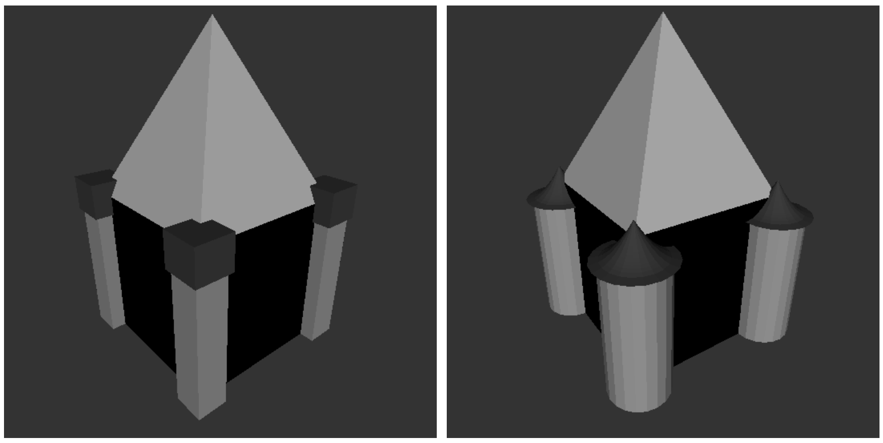


Figure 5.1: Two models generated from the same rule file using stochastic rules

of child components, each going into a recursive loop again. In certain cases the system may crash due to insufficient memory. Unfortunately, there is no way to curb such situations in the present implementation of CRL and it is left to the user to choose the branch probabilities carefully if at all the need to use recursive rules arise.

5.3 Example

In the example below a stochastic rule written like this will generate one of the two models with equal probability. The results can be seen in Figure 5.1.

$$\begin{gathered} OriginalLot \rightarrow \\ [0.5] corners(sizeX, sizeY, 1, 1, 1, 1)\{RoundTower\}, \\ [0.5] corners(sizeX, sizeY, 1, 1, 1, 1)\{SquareTower\} \\ ; \end{gathered}$$

Chapter 6

Other Features

CastleMaker has a couple of other useful features which make models look realistic as well as provides a very handy method of reusing existing rule definitions.

6.1 Auto correction of textures

The system keeps track of texture offsets and dimensions at every split/subdivision which it later uses to determine texture co-ordinates for polygons. A manual texture resize option provided with the *add_texture* function helps in manually suggesting how the texture should scale to the object at hand. Figure 6.1 shows this feature in action.

For terminal shapes like a volume element or a cylinder, texture continuity is maintained at the edge boundaries. Care should be maintained to specify manual resizing in such a way that the texture looks realistic. for example for a 4 sided component split, scaling in u should be 4 times that in v. And for cylinders scaling in u should be around π times that in v. Figure 6.2 shows an example.

6.2 Rule file includes

Stochasticity is useful but is sometimes better avoided. For example in case of a castle building with four towers at each of its corners, it is visually preferable in most cases, that all four towers be of the same height, structure and texture. If the

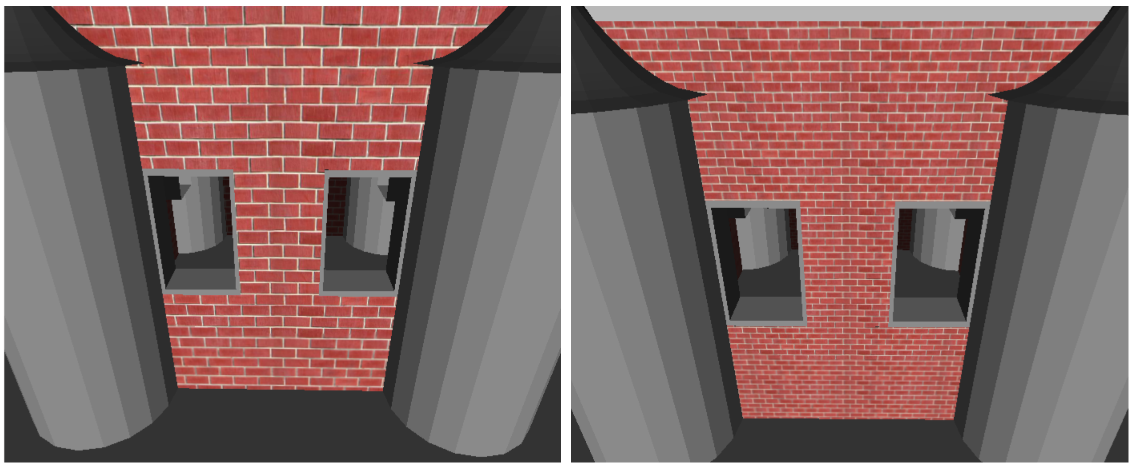


Figure 6.1: (a)Texture auto corrects itself at subdivision boundaries (b)With 2X scaling on both u and v directions. Auto-scaling preserves continuity.

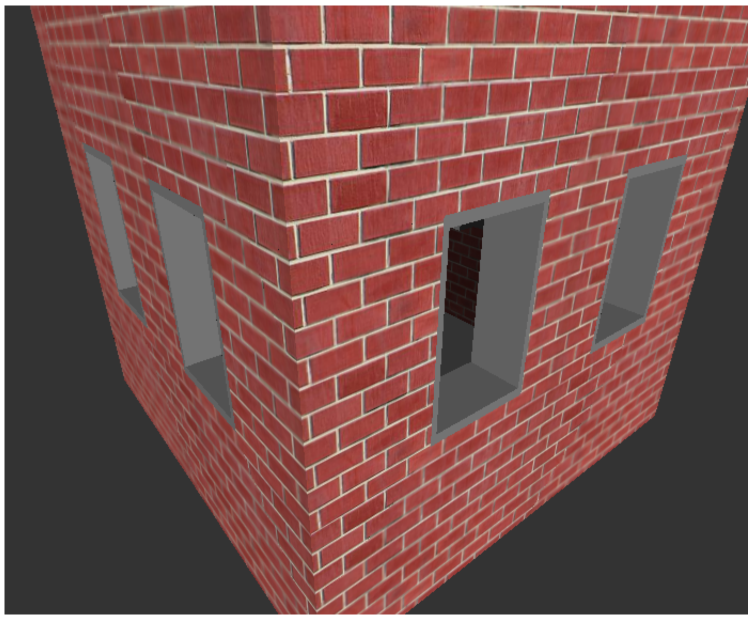


Figure 6.2: Auto correction of textures preserves continuity at edge boundaries.

towers themselves use stochastic rules, it is impossible to guarantee this uniformity and a lot of branches need to be evaluated and specified. This can be avoided using rule file includes.

This feature enables a user to invoke a child rule file from a parent rule file, thereby using models designed previously using the system to be used in another, larger, model. In this way several useful parts like doors and windows can be modelled once and reused at several places. This reduces too much uncontrolled randomness resulting from stochastic rules. Figure 6.3 demonstrates an example.

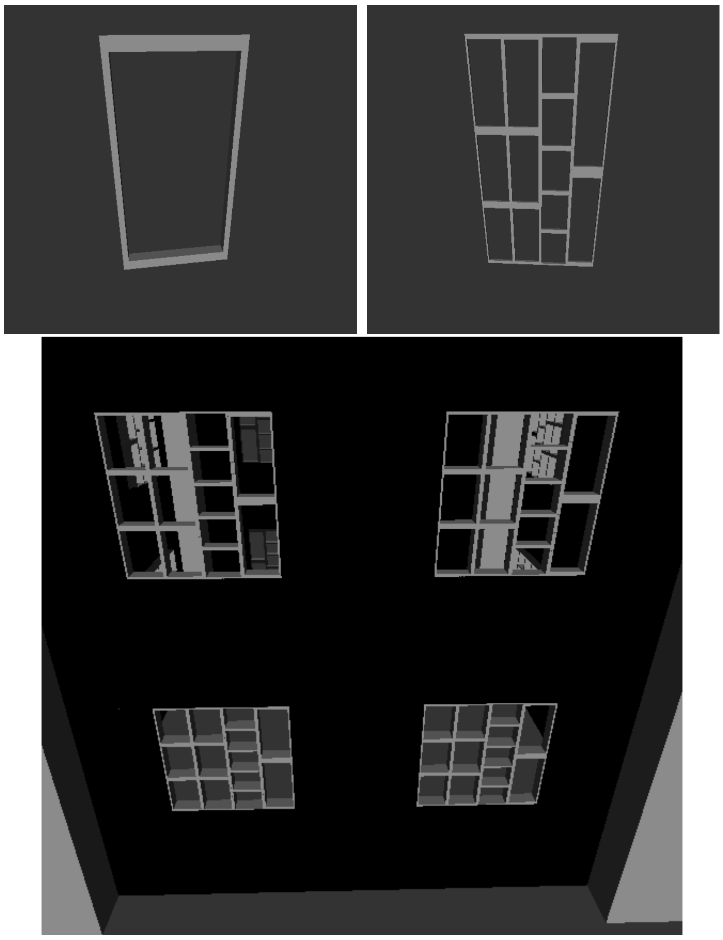


Figure 6.3: (a) The original basic model available as a terminal object (b) A rule file using some randomness to create a window using the basic object in **6.3(a)** (c) Including rule file used to create **6.3(b)** and reusing the same object at multiple places.

Chapter 7

Results

7.1 Resulting Models

As was seen in the previous chapters, CastleMaker can be extremely powerful and can be used to design a myriad of structures. To come up with something grandiose though, one must write the rules carefully and with much thought. This may take time but with increased definition for detail, really complex and realistic models can be generated and using randomness and stochastic rules wisely, a large number of similar stuctures can be additionally created. In this chapter we present some of the results that were obtained using CastleMaker.

7.2 Stochastic Tower

Stochastic rules can be used in a variety of ways to create similar strctures. Figure 7.1 shows different versions of towers generated from a single rule file.

7.3 Toy Skyscrapers

A minimalistic toy skyscraper model consisting of over a million triangles, having no external textures and defined by only *nine* rules is shown in Figure 7.2.

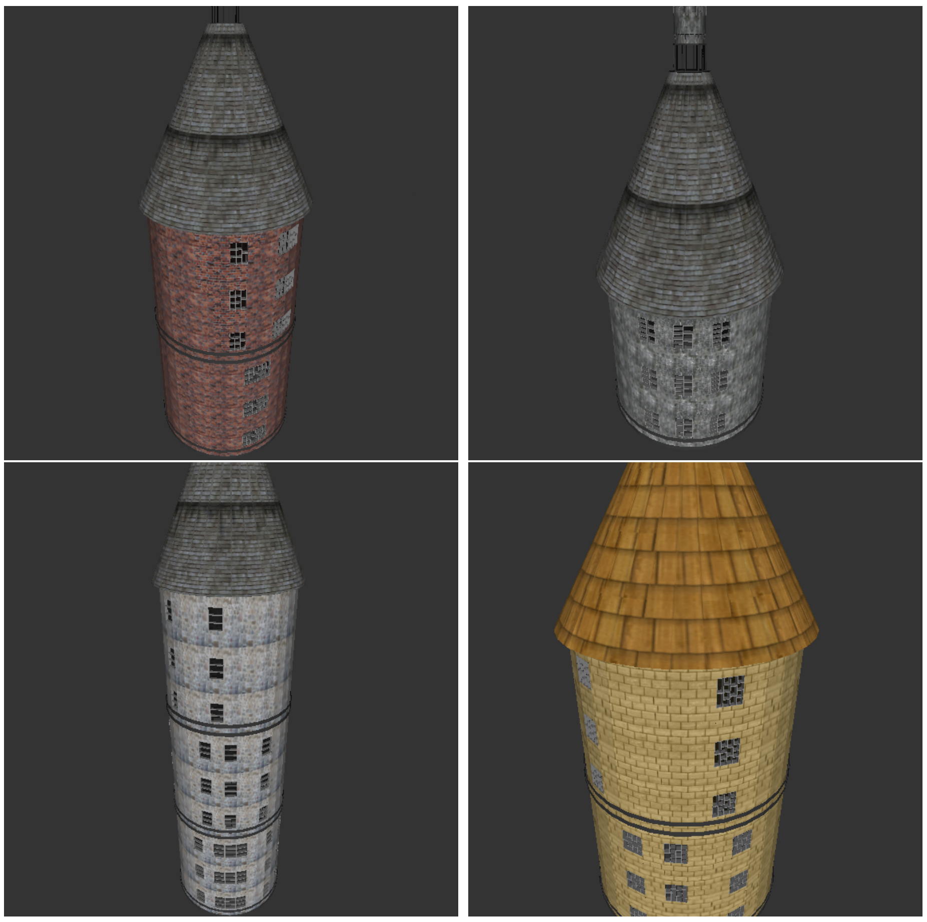


Figure 7.1: Several versions of the same tower definition.

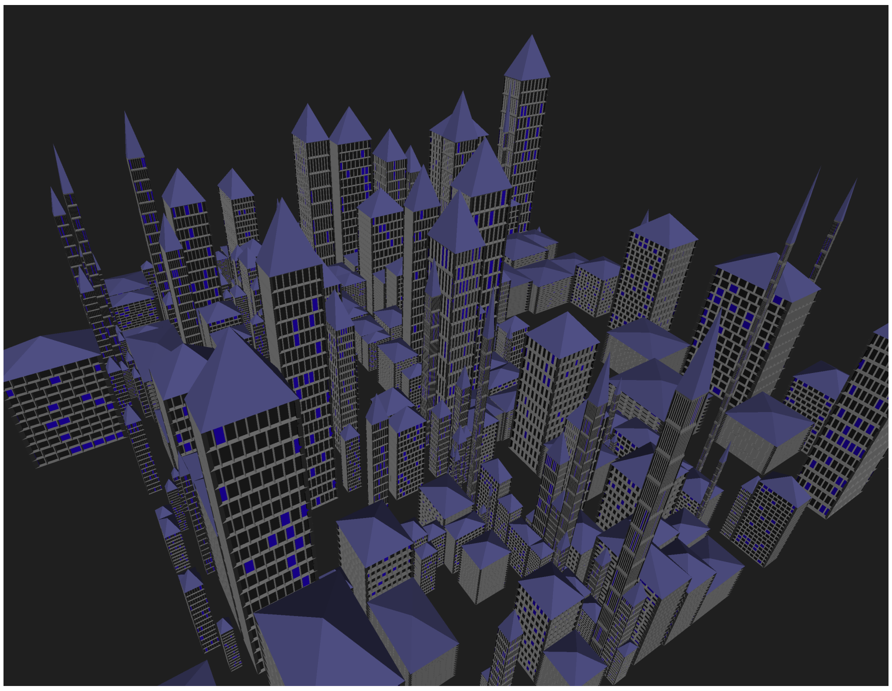


Figure 7.2: A particular instance of a random toy skyscraper definition.

7.4 Clone of the Château de Chambord

This is an attempt to define the rules for a clone of the 'Château de Chambord' of France. All models shown in Figure 7.3 through 7.6 are generated using the same rule file and showcase several different features provided by CastleMaker.

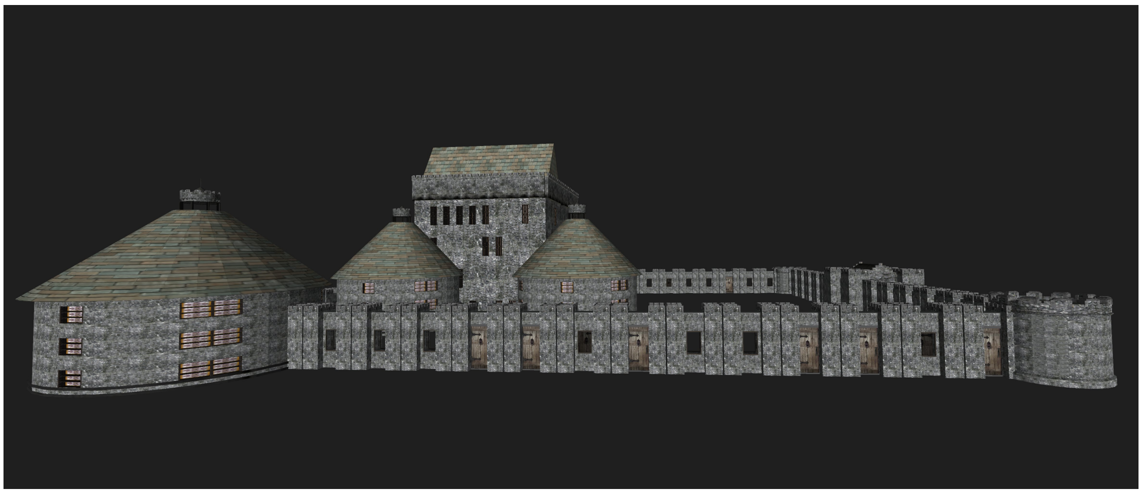


Figure 7.3: (a)

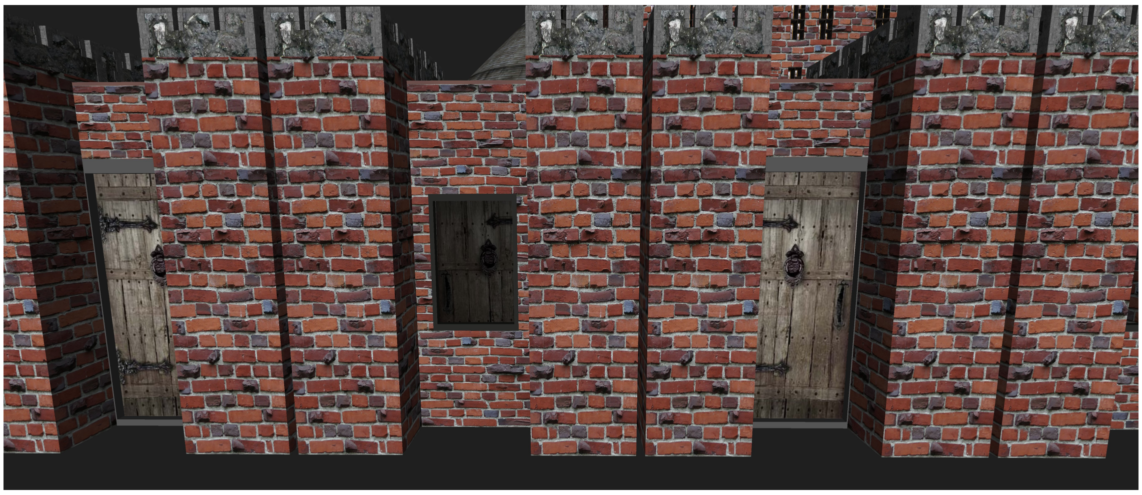


Figure 7.4: (b)

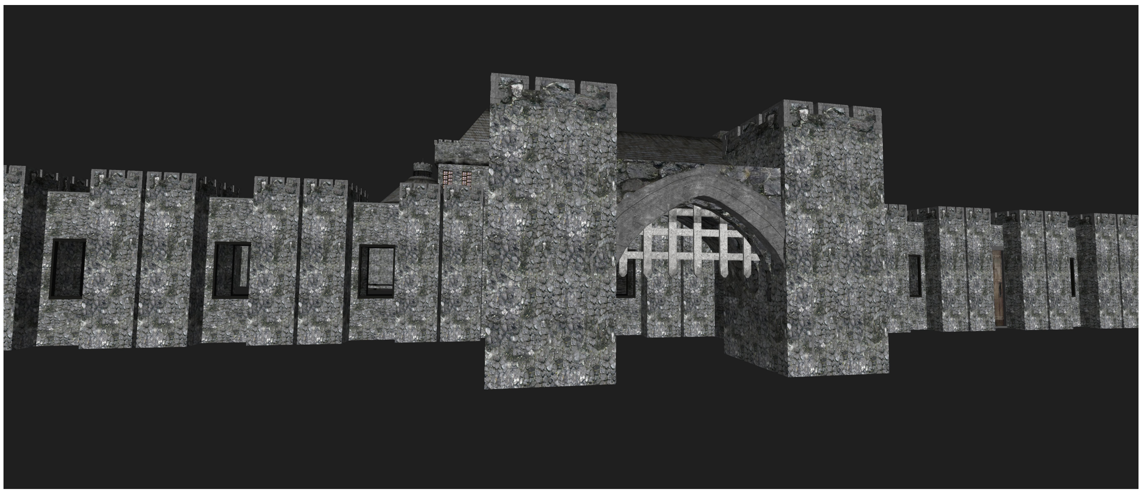


Figure 7.5: (c)

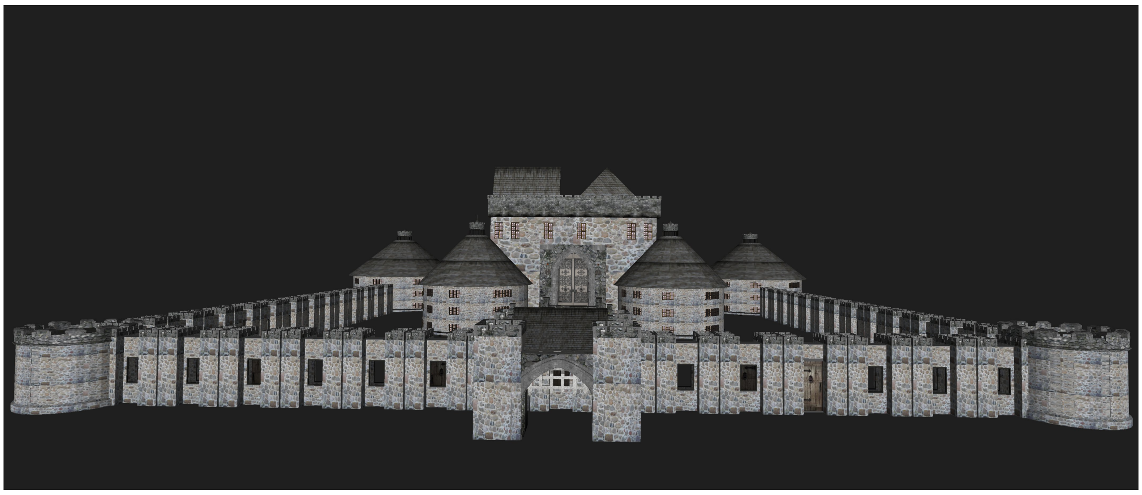


Figure 7.6: (d)

Chapter 8

Conclusion

We have thus seen how Procedural modelling can be used to generate different architectural structures and models fast and efficiently. The presence of fractal patterns have been proven to exist in a multitude of naturally occurring patterns like mountains and ferns to tiny entities like snowflakes and mollusk shells. It is indeed true that the seemingly random nature of the world around us is not all that random but is governed by a set of, possibly simple, mathematical transformation rules.

Leaving natural and biological patterns aside, man-made artifacts are also seen to adhere to certain rules in design, and seemingly different structures also have some basic attributes in common. This is what stochastic procedural modelling can simulate and quickly at that. Keeping the basic attributes same, and varying the parameters binding these basic attributes, results in complex yet beautiful geometry.

CastleMaker can be used to simulate variations of castle like structures with the help of random attributes and stochasticity. Starting with the parent lot, the 2D top view of the castle's bounding box, we can iteratively add subsequent child lots to the corners or edges, or maybe subdivide the parent lot, either by random or specifying exact parameters, and then go on building the elements from there using extrusions, component splits, and face subdivisions. Insertion operations can be greatly enhanced by adding more terminal shapes to the system, or by writing smaller rule files for frequently occurring terminal shapes and then including them in the parent rule file. Owing to the simplicity of the hierarchical tree definition of an architectural structure, it is possible to come up with such definitions for almost any structure.

A modern city, for example, can be simulated using a few gaussian like measures. Let us say, the heights of all skyscrapers follow a gaussian pattern. This pattern can be a random attribute. The presence or absence of a skyscraper at a particular position can also be a random attribute. The ground dimensions of a building can be yet another. Taking all of these attributes into consideration and applying randomness to them, we can come up with a realistic looking city design with just *nine* rules using CastleMaker (refer to section 7.3).

CastleMaker can hence be used as a tool not only to build castles but varying other architectural structures. Although several workarounds need to be applied in some cases for designing structures with random elements. For exact specifications though, CastleMaker can exactly simulate the structure.

An inherent feature of CRL (CastleMaker Rule Language) is that all distances (lengths) are relative. Except for the lots which are the 2D surfaces on which the structure is based, all other parameters are specified as percentages or ratios of the parent's dimensions. Though this works well mostly, the issue with absolute lengths arises in certain cases. Like the case of randomly varying the height of a building depending upon the absolute dimensions of a single floor, for example, is impossible to simulate, owing to the absence of absolute lengths. But there is a clever workaround to achieve this, using recursive stochastic rules. Although a given number of floors is still impossible to simulate exactly, this workaround results in even floors of the desired height.

Another feature which can be said to be missing in the current system is the availability of non-axis aligned lots. In the current implementation rotations which are not multiples of a right angle are unavailable. But the idea can be easily extended into the language. Terminal shapes which are not axis aligned, however, are perfectly allowed to be imported into a model.

Another feature which also could be easily extended is the presence of guard conditions using mathematical relations on attributes. All randomness in the production system is availed via a range of random numbers between two given boundaries, and/or using fixed probality measures in stochastic rules. Although this is quite powerful, it is not as flexible as guard conditional expressions. For example, let us say we want to subdivide a façade into a number of floors depending upon the height of the façade. This can just be guessed and tweaked manually, or a random attribute to the split function can be passed, which is somewhat guaranteed to lie between the acceptable range for the number of divisions. Apart from this approximation, there is no real solution for this problem. More complicated cases may be, presence or absence of a particular structure component depending upon what other components are present and/or having certain dimensions. This somewhat takes us into the realm of context sensitive grammars.

As demonstrated in examples from the previous chapter, CastleMaker can be used to design extremely complicated and detailed structures. Although CRL does not have conditional expressions at present, hard wired weights to drive stochastic rules work pretty well if tuned correctly. Also the availability of rule file *includes* help to maintain composition and at the same time make use of stochasticity. Manual texture scaling can be used to fine tune the appearance of different neighbouring structures making up a composite model. The interface can be extended to provide for a more intuitive GUI based drag-and-drop kind of behavior in the future. Moreover, there is room for improvement in optimizing the mesh structure before export, and taking care of issues like duplicate vertices etc. Also the system will be extremely powerful with arithmetic and logical operations built in for attributes, parameters as well as conditional expressions. Introduction of absolute lengths would violate some of the principles on which CRL is based on, but can be implemented with some effort. The eventual results seem quite appealing, and the possibilities to come up with absolutely grandiose models is restricted only to the designer's creativity. Although getting a particular design to work can have a complicated series of operations and more than one correct way may be possible to achieve it. With practice, CRL is easy to comprehend and the series of operations follow a consistent logical approach.

Bibliography

[EWD09] John Ochsendorf Emily Whiting and Fredo Durand. Procedural modeling of structurally-sound masonry buildings. In *SIGGRAPH Asia*, 2009.

[GCZ08] Peter Wonka Pascal Müller Guoning Chen, Gregory Esch and Eugene Zhang. Interactive procedural street modelling. In *ACM SIGGRAPH*, 2008.

[MB12] Hans-Peter Seidel Vladlen Koltun Martin Bokeloh, Michael Wand. An algebraic model for parameterized shape editing. In *ACM SIGGRAPH*, 2012.

[Mer09] Paul Merrell. Example-based model synthesis. In *Ph.D. Dissertation, University of North Carolina at Chapel Hill*, 2009.

[MLW08] Peter Wonka Markus Lipp and Michael Wimmer. Interactive visual editing of grammars for procedural architecture. In *ACM Transactions on Graphics, Vol. 27, No. 3, Article 102*, 2008.

[MM08] P. Merrell and D. Manocha. Continuous model synthesis. In *ACM Transactions on Graphics*, 2008.

[Mos10] Oussama Moslah. Procedural modelling. In *Ph. D. Dissertation, Towards Large-Scale Urban Environments Modeling from Images, University of Cergy - Pontoise*, 2010.

[PM01] Yoav I H Parish and Pascal Müller. Procedural modeling of cities. In *ACM SIGGRAPH*, 2001.

[PMG06] Simon Haegler Andreas Ulmer Pascal Müller, Peter Wonka and Luc Van Gool. Procedural modeling of buildings. In *SIGGRAPH*, 2006.

[PPM96] Jim Hanan Przemyslaw Prusinkiewicz, Mark Hammel and Radomír Měch. L-systems: from the theory to visual models of plants. In *Proceedings of the 2nd CSIRO Symposium on Computational Challanges in LifeSciences, CSIRO Publishing*, 1996.