Computation-Friendly Shape Grammars

With Application to Determining the Interior Layout of Buildings from Image Data

Kui Yue

PHD COMMITTEE

Ramesh Krishnamurti (Chair)

Martial Hebert

Gary Miller

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

School of Architecture
Carnegie Mellon University
Pittsburgh, PA 15213

September, 2009

UMI Number: 3382443

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI Microform 3382443
Copyright 2009 by ProQuest LLC
All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

CARNEGIE MELLON UNIVERSITY

School of Architecture

College of Fine Arts

Dissertation

Submitted in Partial Fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

TITLE:

Computation-Friendly Shape Grammars
With Application to Determining the Interior Layout of Buildings from Image Data

PRESENTED BY: **Kui Yue**

ACCEPTED BY ADVISORY COMMITTEE:

Professor Ramesh Krishnamurti Principal Advisor — DATE: 9/17/2009

Professor Martial Hebert Advisor — DATE: Sept. 11 2009

Professor Gary Miller Advisor — DATE: 9/11/09

I hereby declare that I am the sole author of this thesis.

I authorize Carnegie Mellon University, Pittsburgh, Pennsylvania to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Carnegie Mellon University, Pittsburgh, Pennsylvania to reproduce this thesis by photocopying or by other means, in total or in part, as the request of other institutions or individuals for the purpose of scholarly research.

05/17/2009

Copyright @ 2009 Kui Yue

Acknowledgements

I am indebted to Ramesh Krishnamurti, my advisor and thesis committee chairman, as well as mentor, for his invaluable vision, support, and encouragement. His enthusiasm and high expectation is essential to the success of this research. I also thank my thesis committee, Martial Hebert and Gary Miller, for their support and contributions.

I would like to express my special thanks to two sponsors supporting this PhD work: National Science Foundation (grant CMS-0121549 – the ASDMCon project) and US Army Corps of Engineers, Engineer Research and Development Center – Champaign (the AutoPILOT project).

A number of people have contributed to this thesis, directly or indirectly through their participation on the AutoPILOT project. Among others, Casey Hickerson, supervised under Ramesh Krishnamurti, developed the first Baltimore rowhouse grammar, and Ajla Aksamija, a team member of CERL, worked on the building ontology.

I also wish to thank all my friends for their help, in particular, my officemate and colleague, Tajin Biswas, for her valuable discussions; Daniel Huber and Pingbo Tang, for their input about extracting building features from image data.

Most of all, I thank my family for making this adventure possible. My wife, Zhilin Zhang, and my little daughter, Larissa, especially deserve my love and gratitude. Also, I would like to thank my parents and my parents-in-law for their indispensable support.

Abstract

A shape grammar is a formalism that has been widely applied, in many different fields, to analyzing designs. Computer implementation of a shape grammar interpreter is vital to both research and application. However, implementing a shape grammar interpreter is hard, especially for parametric shapes defined by open terms.

This dissertation explores the problem of implementing a shape grammar interpreter, which arose in the context of the AutoPILOT project, in which we were seeking an algorithm to determine the interior layout of a building given an input of building features and a shape grammar describing the building style. A general approach was adopted based on the fact that when applied exhaustively, a shape grammar can generate, as a tree, the entire layout space of the building style. The approach essentially requires a parametric shape grammar interpreter that caters to a variety of building types.

As extensions to the fact that shape grammars can simulate any Turing machine, three corollaries are found that significantly impact the implementation of a shape grammar interpreter. They are the following: a shape grammar may not halt; the language space of a shape grammar may be exponentially large; and parsing of a configuration against a shape grammar is generally unsolvable. The problem of interpreting a general parametric shape grammar is thus in general intractable; even parametric subshape recognition of two-dimensional, rectilinear shapes may require a high-degree polynomial time.

In reality, there are distinct but large classes of shape grammars, with differing underlying characteristics, for which interpreters are known to be computationally tractable. In this dissertation, I present a practical, 'general' paradigm for ensuring the tractability of designed shape grammars and implementing such shape grammars.

Even these tractable shape grammars may significantly differ from one another. A further way of classifying these tractable shape grammars, optimally in my belief, is by the types of data structure capable of carrying out their rule application. There are, of course, other factors that influence the computation of shape grammars; these include the description language and all adopted underlying manipulations. As a consequence, the paradigm is augmented so that every interpreter is supported by an application programming interface-

wise (API-wise) framework, which comprises an underlying data structure, basic manipulation algorithms, and a description meta-language. The paradigm specifies an overall framework comprising a series of sub-frameworks. The overall framework is capable of ensuring the computation for a specified shape grammar interpreter. Shape grammars, which follow such a framework, are termed as *computation-friendly*.

The concept of the overall framework is detailed by examining three sub-frameworks. These include one over 2D rectangular shapes (Rectangular sub-framework), one over 2D polygonal shapes (Polygonal sub-framework), and one for shapes describable by a graph structure (Graph sub-framework). The issue of how to develop a computation-friendly shape grammar is explained and illustrated by using the Baltimore rowhouse grammar as the exemplar.

The rectangular sub-framework, which has direct application to the AutoPILOT project, is examined in detail. This includes an investigation of estimating an initial interior layout from the feature input by constraints solution, and the application of spatial constraints from an estimated initial layout to prune the layout tree and 'fix' the open terms of the intermediate configurations. The building feature input for the AutoPILOT project is typically difficult to obtain. The technical feasibility of automatically extracting building features from image data is examined by comparing two pipelines, *an ideal pipeline* and *a realistic pipeline*.

In summary, in this dissertation, I develop a general approach for determining building interior layouts from exterior features with the aid of shape grammars. Central to the general approach, issues of implementing a shape grammars interpreter are formally investigated by complexity analysis. Subsequently, a practical 'general' paradigm is developed and demonstrated by sub-framework examples. The paradigm facilitates the development of a practical shape grammar interpreter and is readily extensible to future development.

Table of Contents

Chapter 1 Introduction 1

1.1 Motivation 4

1.2 Research question 7

1.3 Chapter overview 10

1.4 Assumptions 12

1.5 Summary 12

Chapter 2 Background review 13

2.1 Existing work on layout determination 13

2.2 Constraint-based techniques 14

2.3 Shape grammars 15

2.3.1 Shape, shape representation, and shape rules 15

2.3.2 Existing shape grammar applications and interpreters 21

2.3.3 Evolution of the definition of shape grammars 25

2.3.4 Trends in the development of shape grammars 42

2.3.5 Shape grammars vs. phrase structure grammars 43

Chapter 3 Building feature extraction from image data 51

3.1 Photo images and range images 52

3.1.1 Photo images 52

3.1.2 Range images 53

3.2 An ideal pipeline 55

3.3 State-of-art 55

3.3.1 Modeling-from-reality 56

3.3.2 Appearance-based object recognition 60

3.4 A realistic pipeline 60

3.5 Details of the realistic pipeline 61

3.5.1 Edge extraction of range images 62

3.5.2 Registration of range images 66

3.5.3 Edge extraction and annotation of photo images 67

3.5.4 Creating a co-located model 69

3.6 Remarks 74

Chapter 4 Computation-friendly shape grammars 77
4.1 Parametric subshape recognition 78
4.2 Determination of factors influencing tractability 80
4.2.1 A unified definition of shape grammars 80
4.2.2 Factors influencing tractability 82
4.3 A paradigm for a practical, 'general' interpreter 86
4.4 Classification of shape grammars 91
4.5 Augmented practical 'general' paradigm 92
Chapter 5 Three sub-framework examples 95
5.1 Rectangular sub-framework 96
5.1.1 A graph-like data structure 97
5.1.2 Transformations of the graph-like data structure 98
5.1.3 Common functions for the graph-like data structure 99
5.1.4 Meta-languages 104
5.2 Polygonal sub-framework 106
5.2.1 Data structure for polygonal sub-framework 108
5.2.2 Common functions of polygonal sub-framework 109
5.2.3 Meta-language for polygonal sub-framework 113
5.3 Graph sub-framework 115
5.3.1 Shape and graph grammars 115
5.3.2 Graph grammars as a sub-framework 120
5.3.3 Common functions for graph sub-framework 120
5.3.4 Meta-language for graph sub-framework 121
5.4 Discussion 123
Chapter 6 Development of computation-friendly shape grammars 127
6.1 The Baltimore rowhouse 128
6.2 Creation of a shape grammar 128
6.3 A traditional rowhouse grammar 130
6.3.1 Abstract shape representation 130
6.3.2 Variation in interior configuration 136
6.3.3 Patterns identified 138
6.3.4 The rowhouse grammar 140
6.4 A new computation-friendly rowhouse grammar 150

Chapter 7 Layout tree pruning and initial layout estimation 165
7.1 Layout tree pruning 165
7.2 Building feature constraints 167
7.3 Constraint satisfaction 168
7.4 CSP and Queen Anne houses 169
7.5 Other constraints on Queen Anne houses 174
7.6 Layout determination of Queen Anne houses 176
7.6.1 Implementation of the Queen Anne grammar 177
7.6.2 Layout tree pruning of Queen Anne houses 181
7.7 Layout determination of Baltimore rowhouses 183
7.7.1 Space subdivision tree and Baltimore rowhouses 184
7.7.2 Layout tree pruning of rowhouses 187
Chapter 8 Conclusion 189
8.1 Contributions 190
8.2 Future research 191
References 195

List of Figures

Figure 1-1: 519 Devonshire Street, Pittsburgh, PA........1
Figure 1-2: Stever House of Carnegie Mellon University........2
Figure 1-3: Warner Hall of Carnegie Mellon University........3
Figure 1-4: The Walt Disney Concert Hall by Frank Gehry........5
Figure 1-5: An example of a Queen Anne house........6
Figure 1-6: Example of Baltimore rowhouses........6
Figure 1-7: General approach to layout determination........8
Figure 1-8: Buildings in the research scope........10
Figure 1-9: Overview of the chapter structure and implementation........11
Figure 2-1: Sample decompositions of a shape........16
Figure 2-2: Emergent octagon from two overlapping gridline shapes........17
Figure 2-3: Some subshape examples........18
Figure 2-4: Sample shape rules from the Queen Anne grammar........19
Figure 2-5: Two examples of shape rule application driven by markers and labels........20
Figure 2-6: Example of shape rule application driven by subshape recognition........20
Figure 2-7: Example of a parametric shape grammar........21
Figure 2-8: Exemplar shape grammars........22
Figure 2-9: A serial shape grammar for the snowflake curve........30
Figure 2-10: Two simple shape grammars: SG1 and SG2........32
Figure 2-11: The combined shape grammar of SG1 and SG2........33
Figure 2-12: A shape grammar for Turing machines........45
Figure 2-13: Two shape rules simulating machine transitions........47
Figure 3-1: Measuring the 3D coordinates of points by triangulation........52
Figure 3-2: Range/intensity crosstalk and mixed pixels........54
Figure 3-3: An ideal pipeline........55
Figure 3-4: An ImageModeler demonstration........57
Figure 3-5: A PhotoModeler demonstration........57
Figure 3-6: Photorealistic scene reconstruction........59
Figure 3-7: A realistic pipeline........61
Figure 3-8: Grid pattern of laser beams and 8-neighborhood connectivity........63
Figure 3-9: Extraction of dihedral edges........65
Figure 3-10: Potential endpoint inaccuracy caused by mixed pixels.........66
Figure 3-11: Difficulty of registration by infinite lines........67
Figure 3-12: Edges detected from the picture in Figure 1-5 using a Canny edge detector....68
Figure 3-13: Subdividing a chain recursively, and extracting its straight-line segments.......69
Figure 3-14: Calculating the projection center by using three vanishing points........73
Figure 3-15: Results of initial camera pose estimation........74
Figure 4-1: Example of parametric subshape matching........79
Figure 4-2: A candidate with infinitely many matching transformations........84
Figure 4-3: Parametrically matching a convex quadrilateral to another........86

Figure 4-4: Rules and a derivation of the stylized sports figure grammar 88
Figure 4-5: Rules and a derivation of the emergent-fish grammar 88
Figure 4-6: A paradigm for a practical 'general' parametric interpreter...............................90
Figure 4-7: One sub-framework for each subclass...93
Figure 5-1: Examples of rectangular spaces and corresponding graph-like data structures...96
Figure 5-2: A layout represented by a set of graph units 98
Figure 5-3: Transformations of the graph-like data structures..99
Figure 5-4: Different cases for the north neighbor(s) of a room 101
Figure 5-5: The start and end nodes for finding neighbor room(s) 102
Figure 5-6: Finding *wStart* and *wEnd* 104
Figure 5-7: Two sample rules of the rectangular sub-framework and their meta-language.105
Figure 5-8: Relational painting No. 64, 1953, by Fritz Glarner 107
Figure 5-9: A shape rule of subdivision and its horizontal reflection 107
Figure 5-10: Dividing a simple polygon by intersection of two arbitrary polygons 110
Figure 5-11: Reshaping the cutting line 111
Figure 5-12: A simpler marching algorithm for polygon subdivision...................................112
Figure 5-13: Meta-language examples of picking up new points under constraints 114
Figure 5-14: A Sierpinski gasket..116
Figure 5-15: A collage grammar for the Sierpinski gasket 117
Figure 5-16: A shape grammar for the Sierpinski gasket..117
Figure 5-17: Implementing the ice-ray grammar as a graph grammar..................................118
Figure 5-18: Shape and corresponding graph rules of the ice-ray grammar 119
Figure 5-19: Subshape recognition in a grid figure...120
Figure 5-20: Meta-language description of the spiral collage grammar 122
Figure 5-21: A result of the spiral collage grammar with some sub-collages as elements ..123
Figure 5-22: The *point_face* operator..124
Figure 5-23: A rule for adding the second floor to a room 125
Figure 6-1: Identification of the better solution between two 129
Figure 6-2: The shape representation of rowhouse samples 134
Figure 6-3: Photos of sample rowhouses 136
Figure 6-4: Block configurations 137
Figure 6-5: Width configurations..137
Figure 6-6: Depth configurations 138
Figure 6-7: Stair configurations 138
Figure 6-8: The traditional rowhouse grammar...149
Figure 6-9: Derivation of 236 East Montgomery Street by the old rowhouse grammar......150
Figure 6-10: Derivation of 236 East Montgomery Street by the new rowhouse grammar ..151
Figure 6-11: Quantifying the shape rules generating staircases...163
Figure 7-1: Windows and doors constrain the height of each story 167
Figure 7-2: Window position constrains possible arrangements of a staircase...................168
Figure 7-3: Initial layout estimation of Queen Anne houses by CSP...................................171
Figure 7-4: Manual layout derivation of 719 Amberson Avenue 173
Figure 7-5: Results from the computer implementation of CSP 174
Figure 7-6: Determination of hallway types..176

Figure 7-7: Screenshot of layout determination of Queen Anne houses 177
Figure 7-8: Application of Rule 2 178
Figure 7-9: Interpretation of Rule 8 179
Figure 7-10: Screenshots of sample layouts generated by the Queen Anne grammar 181
Figure 7-11: Layout results of 5816 Walnut Street 182
Figure 7-12: Layout results of 719 Amberson Avenue 183
Figure 7-13: Screenshot of layout determination of Baltimore rowhouses 184
Figure 7-14: Results of the CSP algorithm on Baltimore rowhouses 185
Figure 7-15: Space subdivision tree of Baltimore rowhouses 186
Figure 7-16: The layout tree of the traditional Baltimore Rowhouse grammar 188
Figure 7-17: Layout results of Baltimore rowhouses 188

List of tables

Table 2-1: A classification of shapes 23
Table 2-2: Existing implementations of shape grammars 24
Table 4-1: Comparison of characteristics important for computer implementation 89
Table 6-1: Shape rules of the new computation-friendly rowhouse grammar 152
Table 6-2: Implicit conditions to make staircase rules exclusive 164
Table 7-1: Three types of hallways of Queen Anne houses 175

Chapter 1 Introduction

Imagine a gorgeous spring day, going out for a relaxed stroll, possibly, with your dog, when a striking window in the half-timbered wall on the second floor of a building catches your attention. Clearly, you can see that the building is a house (Figure 1-1).

Figure 1-1: 519 Devonshire Street, Pittsburgh, PA
(Source: photograph, Kui Yue)

Quite possibly, you recognize that the window belongs to a bedroom. Without any other evidence, it is perhaps harder to decipher much more, for example, whether or

not the room is the master bedroom. The red door (with a Christmas wreath) clearly indicates a main entrance; on the second floor, the small-sized window above the door suggests a bathroom or, perhaps, a staircase. The porch on the right side of the building implies that the room in the interior is a living room. This is somewhat confirmed by the presence of the larger window in the wall between the red door and the corner of the house.

You might run across another building on your way, such as the one shown in Figure 1-2.

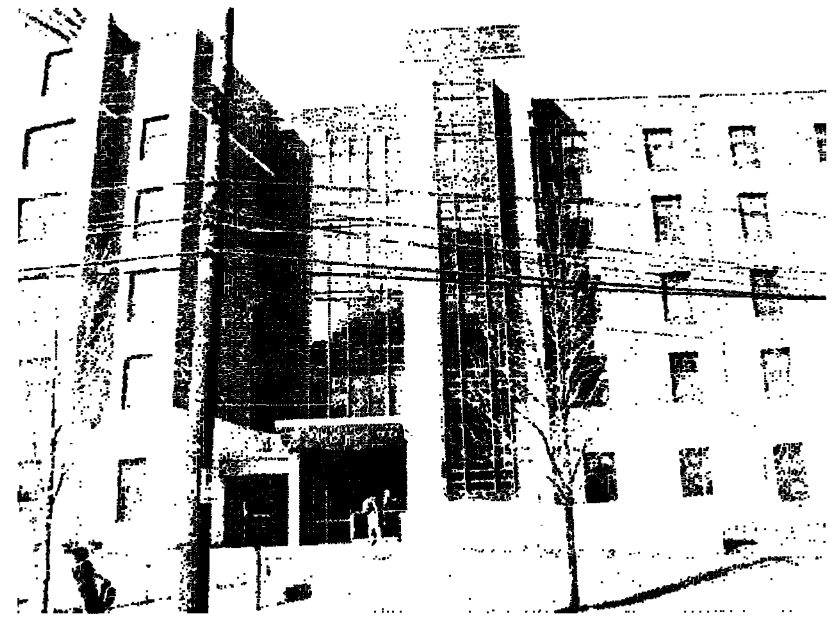


Figure 1-2: Stever House of Carnegie Mellon University
(Source: photograph, Kui Yue)

Although this could be the first time you see this building, you would have no difficulty in recognizing its entrance—an overhang clearly demarcates it as well as the long vertical curtain window. From the position and shape of the curtain window seen on the right, you might guess that the interior is a staircase; you probably even see some stairs through the glass. What about the space inside the curtain window right above the entrance? From your experience, most likely, you would conjecture it to be a public space. What makes you more confident is that the shapes on the left and right of the building suggest that this is a shared space; such a space is usually

public. You might also recognize that the dark box at the highest part of the building indicates an elevator—elevators usually have an equipment room on the roof, and close to the entrance. The exact function of the building might be harder to guess just from its exterior features. But ... if you happen to know that it is a dormitory, then you might guess that the interior of each rectangular window is likely to be a dorm room unit.

On the other hand, if the building that attracts your attention looks like the one shown Figure 1-3, you are probably going have a harder time. You might recognize the entrance as well as the positions of the central elevator and staircases. But the façade is so uniform that it says little about the interior except in this case some of the glass windows are translucent. This may not always be the case for buildings with similar façades.

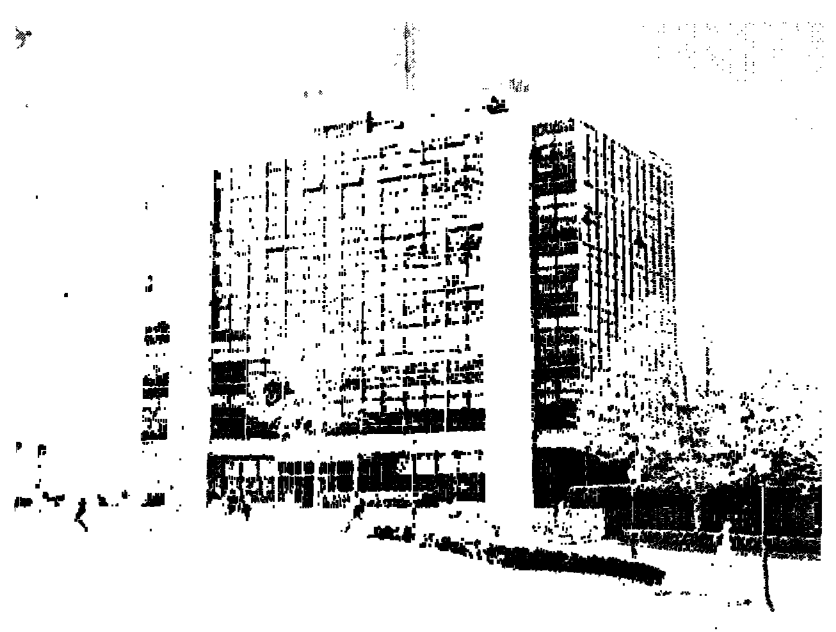


Figure 1-3: Warner Hall of Carnegie Mellon University
(Source: photograph, Kui Yue)

1.1 Motivation

As the above scenarios suggest, in general, it might not be difficult for a human to roughly 'predict' the layout of the interior of a building from observations of their exterior and surrounding features such as entrances, windows, ornaments, decorations, etc. This is, perhaps, particularly true of familiar buildings. On the other hand, it seems an extremely hard task for a machine to accomplish given present day technology.

Although the precise mechanism for human recognition remains unclear, yet, we may safely conclude that such ability relies on reasoning based on accumulated knowledge of the past. In the absence of such knowledge, for example, an individual in an unfamiliar, say different, cultural or vernacular, setting might still find it difficult to guess the interior layout of a building, or even the nature of the building. This is typical of the kinds of problems encountered in building restoration and preservation activities. This leads us to conjecture that if there are ways of providing a machine with the necessary knowledge, together with algorithms to reason against features as inputs, the machine might be able to generate possible layouts, at least, for buildings of certain special types.

Potentially, this technology is useful for a variety of practical applications. For instance, assessing the environment impact of demolition and salvage of building stock requires one to estimate the amount of renewable materials—the city of Baltimore is a case in point (Lund and Yost, 1997). Automating the process of interior layout determination would greatly assist this process.

Even for humans, it can be extremely hard to determine the interior layout of some buildings from their exterior features. The plain office building shown in Figure 1-3 is one such example. Another is Frank Gehry's Walt Disney Concert Hall (Figure 1-4), which does not conform to any normal architectural conventions. Clearly, it is unlikely that one can develop a general computational solution.

Figure 1-4: The Walt Disney Concert Hall by Frank Gehry
(Source: http://www.architectureweek.com/2003/1217/design_1-1.html; accessed Mar 2009)

On the other hand, many buildings follow a pattern book (Downing, 1981; Flemming et al., 1985; Hayward and Belfoure, 2005), and as such provide a handle on how to tackle the problem. That is, there are buildings that vary according to well-defined configurational patterns, and/or to certain well-established sets of regulations and dimensions. In this way, collections of buildings can be recognized as belonging to a particular style. Among these, the Frank Lloyd Wright's Prairie House (Koning and Eizenberg, 1981), Queen Anne House (Flemming, 1987) (Figure 1-5), and the Baltimore Rowhouse (Hayward, 1981) (Figure 1-6) are well-known examples of building styles. Employing knowledge about building styles might make the problem more tractable.

Figure 1-5: An example of a Queen Anne house
(Source: photograph, Kui Yue)

Figure 1-6: Example of Baltimore rowhouses
(Source: Hayward, 1981)

In terms of human designs, a pattern book might well suffice, whereas, in terms of programming, a computational mechanism is needed to represent style data encapsulated in a pattern book. I employ shape grammars (Stiny, 2006), although it is possible to use other rule-based or generative approaches to describing buildings that fall within a particular style. A shape grammar provides a remarkable facility for capturing the spatial and topological aspects of building styles and for generating such designs.

In principle, practical design layout determination should ideally start with images of building exteriors and surroundings, the counterpart to the human vision system. Current computer vision technologies offer the promise to automatically extracting building features from image data (Stamos and Allen, 2000; Frueh et al., 2004). In this dissertation, the feasibility of this approach is examined by comparing two pipelines: *an ideal pipeline*, based on the requirements of the research question, and *a realistic pipeline*, based on state-of-art computer vision technologies.

1.2 Research question

Assuming that an image has been appropriately preprocessed so that the desired feature inputs are available, my research question can be formalized as follows:

Is it possible to "reasonably accurately" determine the interior layout of a building from its exterior features and an "appropriate" representation of its building style?

My response to this question is given in the form of an algorithm, which takes as input, a collection of exterior building features, and employs a shape grammar to encapsulate the building style. The role of a shape grammar as a descriptor of style, as previously mentioned, has been richly documented in the literature (Stiny, 1977; Knight, 1980; Stiny, 1980b; Knight, 1981a; Knight, 1981b; Knight, 1983; Knight, 1989; Stiny, 1991; Stiny, 2006), and I will assume that, for the purpose of this dissertation, this body of work satisfies any requirement of proof. Building feature inputs include the footprint of each story, as well as a reasonably complete set of exterior features, e.g., windows, chimneys and surrounding buildings. I do not base

accuracy of determination on any statistically derived metric; instead, I will rely on visual comparison between generated outcome and ground truth as the basis for verification. My focus and emphasis in this dissertation is solely algorithmic.

I adopt a general approach (Figure 1-7) based on the fact that when applied exhaustively, a shape grammar can generate, as a tree, the entire layout space of a building style. The approach begins with an initial layout estimation employing constraints on the building feature input. Spatial and topological constraints from this estimate are then used to prune the layout tree and 'fix' the possible open terms in the current configuration. The layouts that remain correspond to the desired layouts. Note that both the initial layout estimation and the layout tree generation can be an exponential search. However, in this thesis, they are restricted in a way that both search becomes polynomial.

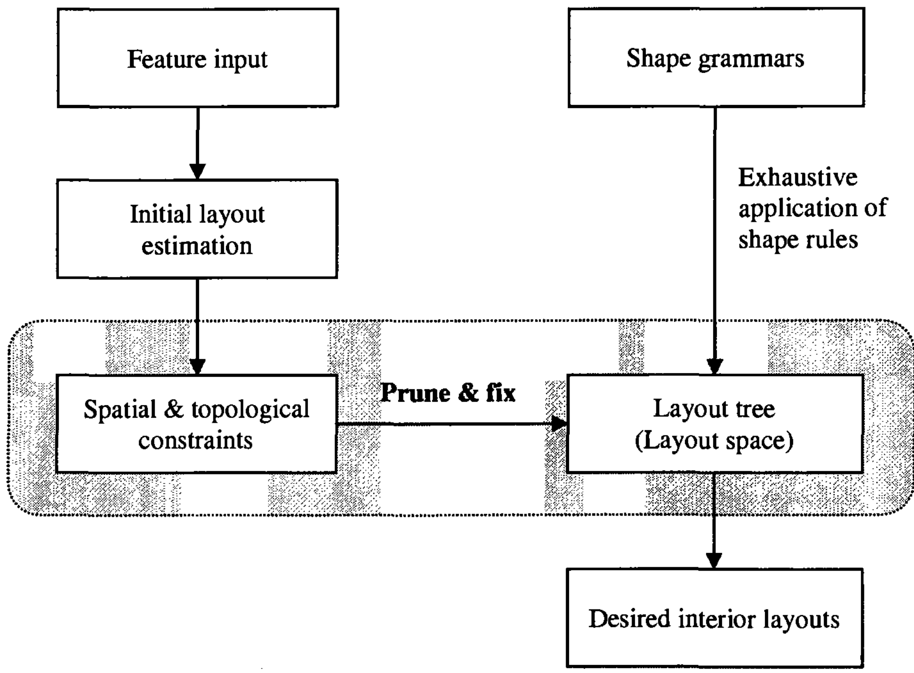


Figure 1-7: General approach to layout determination

Initial layout estimation is achieved by using various constraints from exterior features together with prior knowledge, namely, constraints resolution. Pruning a

layout tree, effectively, is to find a tree node with layout consistent with the partial layout estimate, 'fix' the open terms in the configuration if applicable, and continue to apply the subsequent shape rules. Such a node is typically internal, although, it could be, luckily, the root node. In each case, the approach essentially requires a parametric shape grammar interpreter that caters to a variety of building types; for layout estimation, it would be impractical to implement individual interpreters for each grammar.

A single general parametric shape grammar interpreter has long been a dream in the research community. However, implementing such an interpreter is considered difficult (Gips, 1999; Chau et al., 2004). But, the reasons have not always been apparent. This dissertation attempts to tackle this problem by a formal examination of tractability of the shape grammar formalism. The conclusion reached is that the problem of interpreting a shape grammar is, in general, intractable. This conclusion essentially negates the feasibility of a single general parametric shape grammars interpreter; whence, a practical 'general' paradigm that comprises a set of sub-interpreters is proposed. The paradigm is augmented by a set of API-like frameworks, one for each interpreter for each class of tractable shape grammars. I call such shape grammars as *computation-friendly*. Moreover, the paradigm is targeted at extant shape grammars so that, if need be, they can be modified to be tractable. The paradigm is demonstrated for the class of shape grammars that can be supported by a rectangular sub-framework, into which the majority of building grammars fall.

In this dissertation, I focus on buildings describable by shape grammars. Figure 1-8 indicates the scope of this research relative to the universe of all buildings. As there is no statistical data on how many buildings fall into each category, the assumption here is that there are a large number of buildings that follow pattern books; among these, there is a large proportion, which is describable by shape grammars. If we restrict the class of buildings considered to those that have been constructed, and further, if we were to reverse engineer the assembly of those processes, we might be able to provide a pattern book description for their designs. If B represents the set of all buildings, B_{PB}, the set of all buildings describable by

pattern books and B_{SG}, the set of all buildings describable by shape grammars, then we may safely state that $B_{SG} \subset B_{PB} \subset B$. However, whether $B_{SG} = B_{PB}$, or the unlikely $B_{PB} = B$ requires substantive analysis, which is beyond the scope of this dissertation.

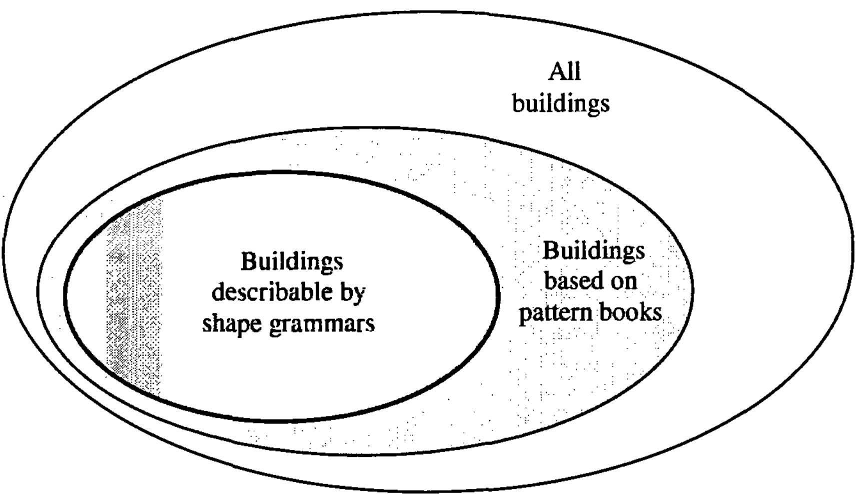


Figure 1-8: Buildings in the research scope

1.3 Chapter overview

Figure 1-9 shows how the chapter structure of this dissertation is organized against the proposed general approach to layout determination and which aspects have been actually implemented. Chapter 2 provides background review supporting all chapters except chapter 3, which also heavily relies on the literature, investigating the feasibility of using existing computer vision techniques to automatically extract building exterior features from image data. A large part of Chapter 2 is about shape grammars, including evolution of shape grammar definitions, trends of the development of shape grammars, as well as comparison of shape grammars and phrase structure grammars, where extra theoretical results as extensions to the fact that shape grammars can simulate any Turing machines are made.

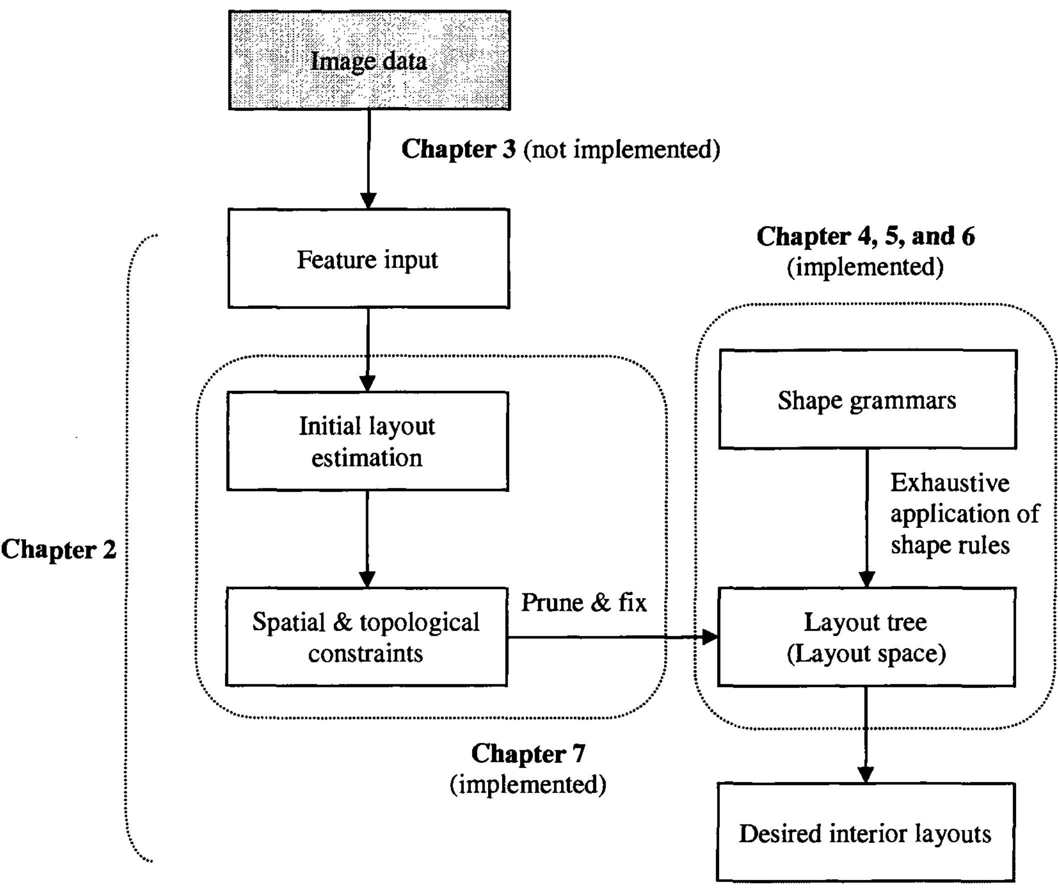


Figure 1-9: Overview of the chapter structure and implementation

The generation of a layout tree essentially needs the implementation of a parametric shape grammar interpreter, which is investigated in chapters 4, 5, and 6. Chapter 4 examines the computational characteristics of parametric shape grammars, and identifies factors influencing the tractability of shape grammars in general; a practical paradigm comprising a set of sub-frameworks is then proposed, whence the concept of computation-friendly shape grammars. Chapter 5 explains the paradigm in details through three examples of sub-frameworks. Chapter 6, using the Baltimore rowhouse grammar as an example, illustrates how to develop a computation-friendly shape grammar.

The constraints associated with the feature input have to been transformed into a form, which can be computationally applied to prune the layout tree. Chapter 7

employs two test cases, the Baltimore Rowhouse and Queen Anne House, to describe how to use constraint resolutions to derive an initial layout and examine how to use the constraints from the derived initial layout to prune the layout tree and fix the open terms so that the desired interior layout can be determined. Chapter 8 ends this dissertation with a conclusion, and a discussion of potential future research.

1.4 Assumptions

As computer implementation is restricted to certain specific topics covered in this dissertation, certain assumptions are made. I assume that the feature input has been made available; that is, it has been preprocessed in a way that is ready for desired usage. The criteria for choosing building features are that they are not difficult to obtain from image data and that they are visible from the exterior, e.g., doors, windows, building footprint, chimneys from overhead view (e.g., in Google map). The discussion given in this dissertation on building feature from image data is theoretical; it has not been implemented by the author.

1.5 Summary

The problem of determining the interior layout of building from exterior features originated in the US Army Corp of Engineer Research and Development Center, Champaign-funded project, AutoPILOT—**Auto**mated **P**rising of **I**nterior **L**ayouts **O**ver building **T**ypes. Conceptually, the project tackles the problem with the aid of shape grammars in conjunction with a building ontology system developed at CERL. The converse problem is used as a vehicle to investigate shape grammars in a practical context, in particular, the problem of implementing a parametric shape grammar interpreter. This leads to an examination of the tractability of shape grammars, developing a practical 'general' paradigm comprising a set of sub-interpreters each backed by its own sub-framework, and whence, the notion, considered in this dissertation, of computation-friendly shape grammars.

Chapter 2 Background review

There is not much literature relating directly to the problem of determining building interior layouts from exterior features, since this idea is quite unique. There are, however, three major methodologies that are applied in this research, namely, shape grammars, constraints, and object recognition from image data. Each methodology, in itself, specifies a broad field. Since extracting building features from image data (given in Chapter 3) relies heavily on existing techniques in objection recognition, only the relevant and related literature is reviewed there. Constraint-based techniques have been widely adopted in many fields, and there is a vast literature on the subject. The review given here is intended neither to be full-blown nor exhaustive; only those works closely related to the subject matter of this dissertation are listed. The formalism of shape grammars is central to the general approach to layout determination and will be elaborated upon in some detail.

2.1 Existing work on layout determination

The closest related work is found in the field of robotic mapping. Roberts et al. introduce an approach, which generates high-confidence layouts of building interiors from limited exterior and interior structural observables (Roberts et al., 2007). The approach starts with an algorithm, which generates interior layouts of urban buildings based on a rule set. The rule set is derived from engineering and

architectural practices for the particular building type and for the geographic region in which the building is located. Given the exterior footprint dimensions of the building, the rule set systematically divides areas of the building into sub-areas based on relative dimensions of these areas, relationships between sub-areas, ability to move from one sub-area to another, and typical component dimensions. The layouts so generated are then refined by rule-based inferences, which operate on limited interior structure observables.

2.2 Constraint-based techniques

Constraint-based techniques have been widely used in computational design, for example, space layout planning (Akin et al., 1992; Yoon, 1992; Damski and Gero, 1997; Lee and Lee, 2006; Narahara and Terzidis, 2006; Donath and Böhme, 2007). Design is viewed as process of problem solving. Design requirements, rules, relations, and natural laws can be further viewed as constraints. Design becomes a process of constraints exploration (Gross, 1986), completed when all constraints have been satisfied. Such a paradigm matches the model of constraint satisfaction problem (CSP) (Russell and Norvig, 2002), which has been widely studied in the field of Artificial Intelligence, and the many techniques developed there can be taken advantage of. Constraint satisfaction, however, is not easy. Design constraints are usually numerous, complex and highly non-linear. Satisfying a large set of arbitrarily complex equality and inequality constraints is, in essence, a non-linear programming problem (Krishnan, 1990). Techniques, like forward checking and constraint propagation, have been developed to speed up the searching process for a solution.

In the context of this research, there are constraints from the exteriorly observable features, while constraints, such as design requirements, are unknown. Consequently, only partial layout information can be obtained from the constraints of building exterior features. Nonetheless, this partial layout can be used to prune the tree of the layout space generated from the shape grammars.

2.3 Shape grammars

In computational architecture design, there are hard research questions that require algorithmic answers. For example, aesthetically, functionally, structurally or otherwise, how does a computer recognize the commonality shared by a corpus of design artifacts, which humans believe to be in the same style? How do we specify an algorithm so that new design artifacts in the same style can be generated?

In trying to answer such style-related questions, spatial grammars have been favored as the preeminent formal method, because, they bridge the gap between visual expressions of form preferred by designers, and symbolic encoding required by computers (Carlson, 1993; Heisserman, 1994; Stiny, 2006). However, the passage from one side of the gap, namely, visual expression, to the other, symbolic encoding, is far from trivial.

Of the various spatial grammar systems around, perhaps, the best known is shape grammar (Stiny, 2006)—a production system consisting of shape rewriting rules. Every design can be considered and represented as a shape that, itself, has been generated as a sequence of shapes. The sequence always starts with a given initial shape. Each shape in the sequence is produced from the previous shape by substituting a part of it for another. The two parts constitute a shape rule, conveniently, the left-hand side and right-hand side shapes. To ensure closure of the rule application sequence, a terminal rule is typically specified, which prevents other rules from being subsequently applicable. Shape rules are usually further classified, so that shape or design generation can be broken down into phases.

2.3.1 Shape, shape representation, and shape rules

In the purest sense, a shape grammarist views shapes from the standpoint of a designer. This may differ quite distinctly from how shapes are viewed in other disciplines. Here, a shape is an arrangement of a finite number of *spatial elements*, each with a definite boundary of limited but non-zero extent, for example, points, line segments, plane segments etc.—these spatial elements are said to be *embedded* in the shape. In Stiny's widely-quoted definition (Stiny, 1980a), shapes were limited

to two dimensions and being rectilinear. Elsewhere, shapes have been considered in three dimensions (Krishnamurti and Earl, 1992; Stouffs and Krishnamurti, 2006), or comprising of curves (Chau, 2002; McCormack and Cagan, 2003; Jowers, 2006).

In shape grammars, the shape representing a design is not fixed; rather, it changes dynamically throughout a computation. Any shape, except those composed of distinct isolated points, can be viewed, and hence, decomposed into spatial elements in indeterminately many ways. Each such decomposition provides a specific way of describing a shape. For example, Figure 2-1 shows, among the indefinitely many ways, a few distinct decompositions of the shape made up of two overlapping rectangles.

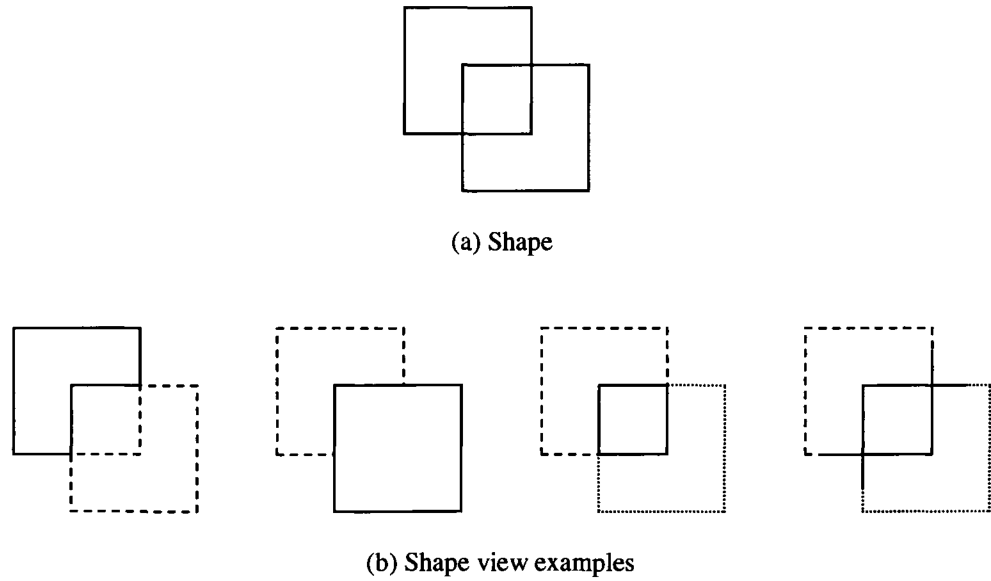


Figure 2-1: Sample decompositions of a shape

Figure 2-1 illustrates another phenomenon—that of emergent shapes. That is, when two or more shapes are brought together, new shapes *emerge* as a result of the interaction between the primary shapes. Figure 2-2 illustrates a situation where when two gridline shapes overlap, an octagon emerges (Krishnamurti and Stouffs, 1997).

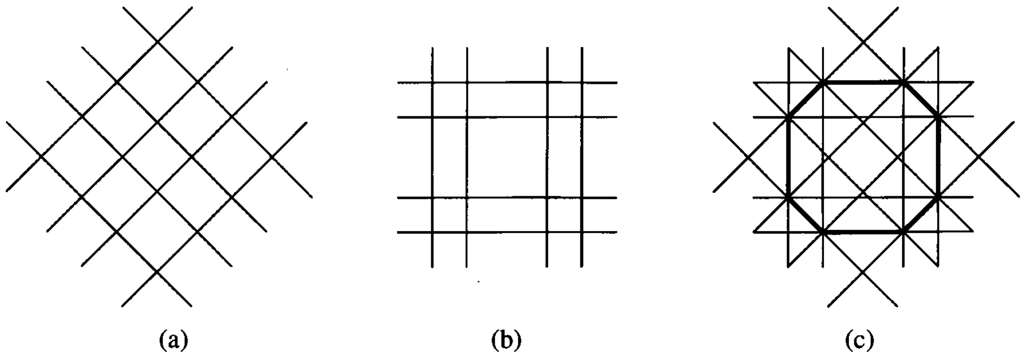
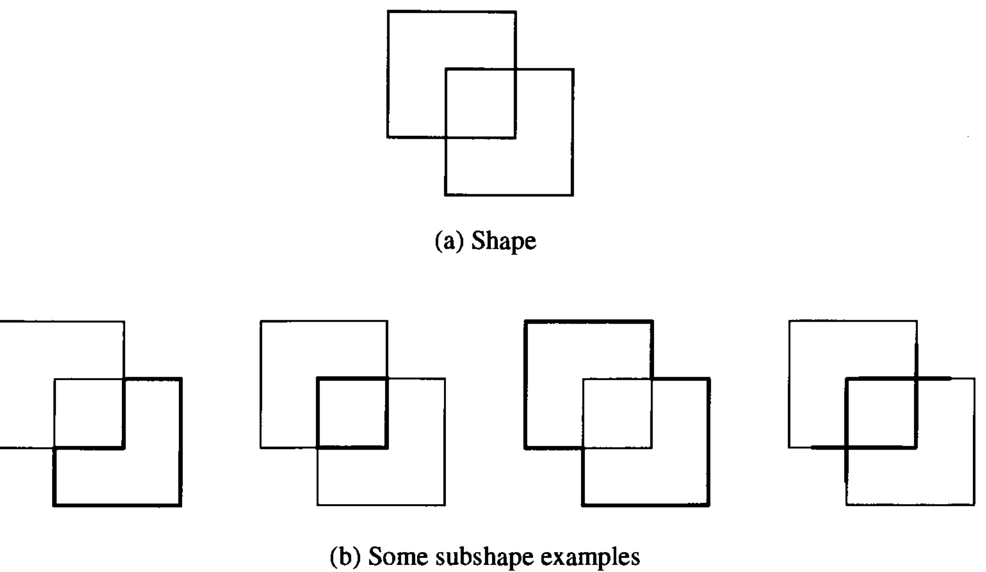


Figure 2-2: Emergent octagon from two overlapping gridline shapes
Adapted from (Krishnamurti and Stouffs, 1997)

In any rule–based computation involving shapes, it is desirable to provide a consistent representation for shapes in which all different views can be described in a unique, canonical way. This is achieved by representing a shape by its *maximal* spatial elements (Stiny, 1980a; Krishnamurti, 1992b). Here, collinear elements, which pairwise overlap or share a boundary (in the case of say, lines, an end point), get merged to form a single element that is maximal within the representation. In the literature, shapes under maximal representation are also called in their *reduced form* (Stiny, 1975).

The concept of *subshape* is important to shape grammars. A shape is said to be a *subshape* of a second shape whenever each element of the former is embedded in an element of the latter. For instance, the emboldened parts of the shapes in Figure 2-3b are subshapes of Figure 2-3a. Note, again, that there are infinitely many subshapes. Of these, the *empty shape* is a special shape, consisting of no spatial elements, and it is a common subshape to any and every shape.



(a) Shape

(b) Some subshape examples

Figure 2-3: Some subshape examples

In generating designs, shapes are often tagged with markers or labels, in order to deal with functional and other non-spatial design features. The subtle differences between markers and labels are explained in Section 2.3.3 under subsection headings, *SG-DEF-1974* and *SG-DEF-1977*. Unless stated otherwise, in this dissertation, both terms are used interchangeably with respect to guiding the application of shape rules, with the convention that labels are typically alphanumeric strings associated with a shape element, while markers are symbols used to control the status of the entire configuration. Markers and labels can be erased or added during the generation of a design. Figure 2-4 shows two example shape rules taken from the Queen Anne grammar (Flemming, 1987). Here, shape rule 0 replaces the initial shape by a hallway room *H*, changing the status marker from *H* to *R*. In shape rule 1, the left-hand side is a room with label *H*. In the rule, the room has a corner with label *B* and another with a parameterized label *X*, which could be either *B* (back) or *F* (front) depending on the context. If the left-hand shape exists in the currently generated shape, then, through this shape rule application, another room *R* is added, with labels and markers updated accordingly.

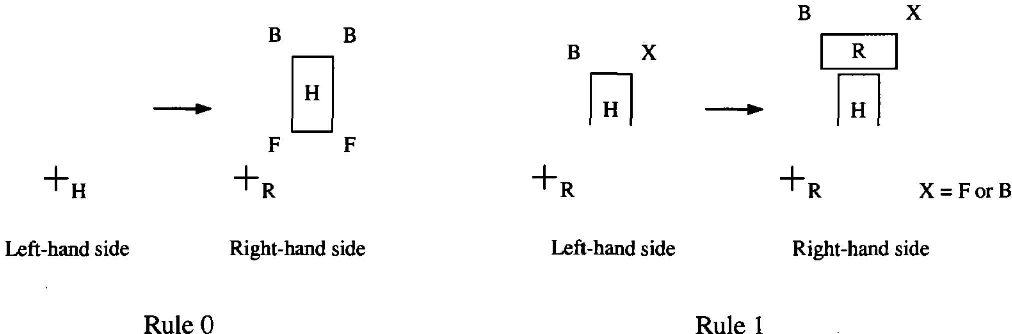


Figure 2-4: Sample shape rules from the Queen Anne grammar

In applying a shape rule, the left-hand shape has to match—that is, be equal to—a subshape of the target configuration under an allowable geometric transformation. Unless stated otherwise, the allowable transformation is commonly taken to be a similarity—that is, an isometry with uniform scale. It should be noted that under the subshape relationship, the associated markers and labels have to match their counterparts in the subshape.

There are a number of different ways of controlling or driving shape rule application. In some instances, markers and labels play an important role; in other instances, the shape itself may be of more importance. Typically, when employing markers and labels, these identify and limit which rules can or cannot be applied, and in each case, the shape is simply transformed as desired. A terminal rule typically removes all markers and labels. Figure 2-5 shows two examples using the shape rules in Figure 2-4.

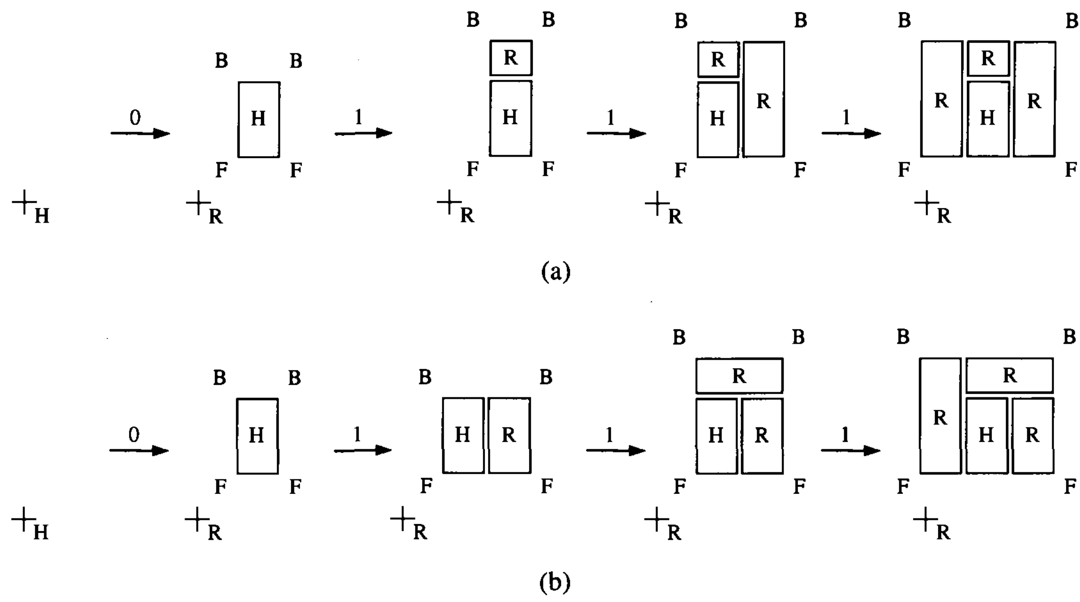
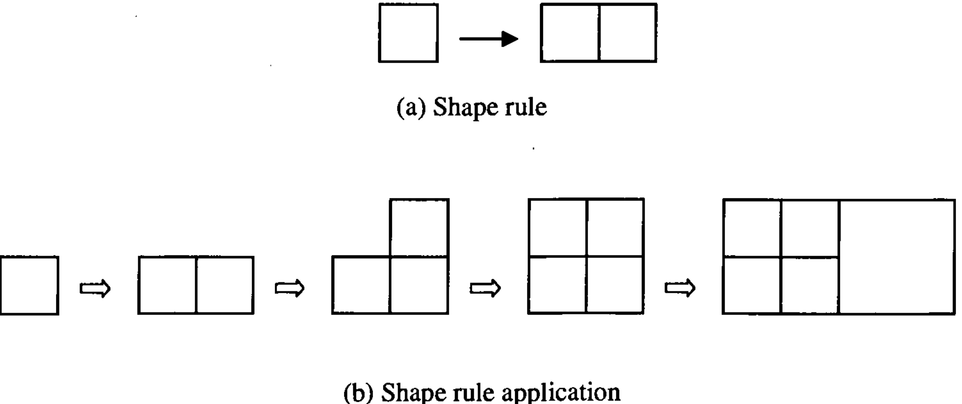


Figure 2-5: Two examples of shape rule application driven by markers and labels

Another way of controlling shape rule application is by recognizing appropriate subshapes. The left-hand shape is transformed, within allowable limits, so that a subshape of the given shape can be tested for a match; if successfully found, the shape rule is applicable to that subshape. Figure 2-6 is an example, taken from (Stiny, 2006), in which Figure 2-6b shows the repeated application of the shape rule in Figure 2-6a through recognition of a suitable rectangle as the selected subshape.



(a) Shape rule

(b) Shape rule application

Figure 2-6: Example of shape rule application driven by subshape recognition
Adapted from (Stiny, 2006)

In some cases, just allowing similarity transformations can restrict the shape grammar. Strictly, under similarity, rule 1 of Figure 2-4 cannot apply in some steps of Figure 2-5, as it needs an anamorphic scaling of a rectangle—in other words, by matching a rectangle to another rectangle. Figure 2-7, taken from (Stiny, 2006), illustrates an even 'wilder' example. Here, basically, a convex quadrilateral can match another. To handle such requirements, shape grammars are extended to *parametric shape grammars*, which allow open terms (variable coordinates of points) to be determined when applying the shape rule.

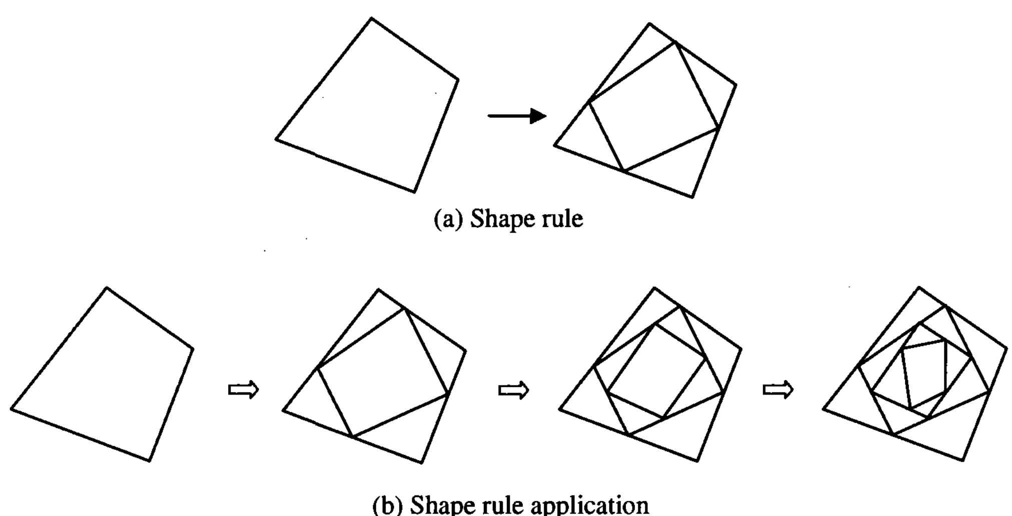


Figure 2-7: Example of a parametric shape grammar
Adapted from (Stiny, 2006)

2.3.2 Existing shape grammar applications and interpreters

Figure 2-8 is based on the review of Chau et al. for shape grammars in the literature (Chau et al., 2004). These shape grammars range from the field of painting, decorative arts, architectural plans, to engineering designs, with about half dealing with architectural plans, almost all with the intent of capturing a specific style.

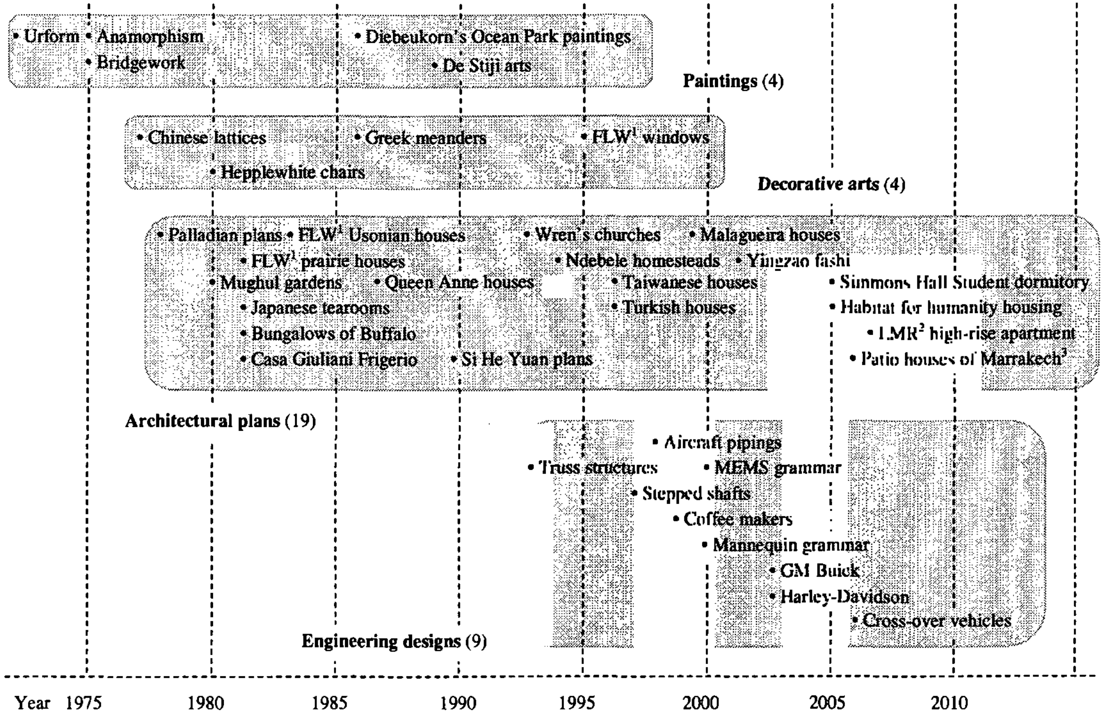


Figure 2-8: Exemplar shape grammars
Adapted and modified from (Chau et al., 2004)

Given the inherently computational nature of shape grammars, there have been significant efforts made toward their implementation. There are two desirable requirements for any general shape grammar interpreter. The first is support for shape emergence; the second, support for automatic subshape recognition[1]. Progress in meeting these two challenges depends on the complexity of the shape, which can be described in the following way. If we were to classify shapes as either 2D or 3D, rectilinear or with curves, parametric or otherwise, then, clearly, parametric 3D curved shapes are the most complex. For convenience and accuracy of discussion,

[1] The two features are not altogether independent. Automatic subshape recognition does not necessarily mean support for shape emergence. However, the converse is true; that is, interpreters supporting shape emergence are usually capable of automatic subshape detection. Most extant implementations do not support shape emergence.

we define shapes to be one of the eight types identified in Table 2-1. For example, non-parametric 2D rectilinear shapes are of *Type I.*

Table 2-1: A classification of shapes

	Rectilinear		**With curves**	
	2D	**3D**	**2D**	**3D**
Non-parametric	I	II	III	IV
Parametric	V	VI	VII	VIII

Table 2-2 is an extraction from (Chau et al., 2004) of shape grammar interpreters found in the literature. Gips implemented the first shape grammar interpreter (Gips, 1975). It handled non-parametric non-self-intersecting 2D polygonal shapes. Emergent shapes were not considered. Krishnamurti gave the theoretical basis for computing 2D shape grammar systems (shapes of Type I) and implemented the first interpreter capable of detecting emergent shapes and supporting automatic subshape detection (Krishnamurti, 1981; Krishnamurti, 1982). This implementation featured the maximal representation of line shapes and employed homogeneous coordinates. Krishnamurti and Giraud rewrote the two-dimensional interpreter in Prolog, a logic programming language (Krishnamurti and Giraud, 1986). Chase and Tapia developed grammar interpreters based on the algorithms developed by Krishnamurti (Chase, 1989; Tapia, 1999). Both were non-parametric interpreters that work with shape grammars comprised solely of line shapes. Tapia's GEdit is known for its interface design. The maximal element representation was extended to 3D (Krishnamurti and Earl, 1992; Krishnamurti and Stouffs, 2004; Stouffs and Krishnamurti, 2006). Stouffs implemented shape arithmetic in 3D (Stouffs, 1994). Algorithms for subshape recognition in 2- and 3-D shapes made up of different kinds of planar elements are given in (Krishnamurti and Earl, 1992; Krishnamurti and Stouffs, 1997); however, there are no corresponding practical implementations.

Table 2-2: Existing implementations of shape grammars
Adapted and modified from (Chau et al., 2004)

	Name	Reference	Tool(s) used	Shape emergence	2D/3D
1	Simple interpreter	(Gips, 1975)	SAIL[1]	No	2D
2	Shape grammar interpreter	(Krishnamurti, 1982)	FORTRAN	Yes	2D
3	Shape generation system	(Krishnamurti and Giraud, 1986)	PROLOG[2]	Yes	2D
4	Queen Anne houses	(Flemming, 1987)	PROLOG	No	2D
5	Shape grammar system	(Chase, 1989)	PROLOG	Yes	2D
6	Genesis (CMU)	(Heisserman, 1991)	C/CLP(R)[3]	No	3D
7	Grammatica	(Carlson, 1993)		No	
8	GRAIL	Krishnamurti and Stouffs 1992-6[4]	C		2D/3D
9	Genesis (Boeing)	(Heisserman, 1994)	C++/CLP(R)[3]	No	2D/3D
10	GEdit	(Tapia, 1996)	LISP[5]	Yes	2D
11	Shape grammar editor	(Shelden, 1996)	AutoLISP	Yes	2D
12	Implementation of basic grammar	(Simondetti, 1997)	AutoLISP	No	3D
13	Shape grammar interpreter	(Piazzalunga and Fitzhorn, 1998)	ACIS Scheme	No	3D
14	SG-Clips	(Chien et al., 1998)	CLIPS	No	2D/3D
15	3D Shaper	(Wang, 1998)	Java/Open Inventor	No	3D
16	Coffee maker grammar	(Agarwal and Cagan, 1998)	Java	No	2D/3D
17	MEMS grammar	(Agarwal et al., 2000)	LISP		2D
18	Shaper 2D	(McGill, 2001)	Java	No	2D/3D
19	U_{13} shape grammar implementation	(Chau, 2002)	Perl	Yes	3D
20	Shape grammar interpreter[6]	(Trescak et al., 2009)	Java	Yes	2D

[1] Stanford Artificial Intelligence Language
[2] SeeLog developed at EdCAAD
[3] IBM CLP(R) compiler
[4] Private communication
[5] Macintosh Common LISP
[6] http://www2.iiia.csic.es/~ttrescak/sgi.html

McCormack and Cagan have developed a parametric interpreter aimed at shapes consisting of curved shapes (Types III and VII) (McCormack and Cagan, 2003). The interpreter works by determining a straight-line equivalent of a curved shape, which is then used, with reference to the original curved shape, to detect emergent shapes.

It was used to implement the Buick brand shape grammar (McCormack et al., 2004). Chau implemented an interpreter targeted at the general shapes (Types IV and VIII) (Chau, 2002). But the examples shown are actually 2.5D (3D generated by extrusion).

There are a number of computer implementations listed in Table 2-2, which do not use a maximal element representation. For instance, Heisserman uses a graph-based boundary solid representation for shapes of Type II (Heisserman, 1991); Carlson applies a nondeterministic functional programming language as an interpreter of 2D shapes (Types III and VII) (Carlson, 1993); and Piazzalunga and Fitzhorn rely on ACIS, an extant graphical modeling kernel, to implement an interpreter targeted at 3D shapes of Types II and VI (Piazzalunga and Fitzhorn, 1998). None recognize subshapes, nor deal with shape emergence.

To summarize, implementation issues for shapes of Type I have been resolved. Theoretical investigations for shapes of Type II have been completed though, as yet, without full implementation. Implementations for other types of shapes, especially parametric shapes, have been attempted; most solve a special case; those claiming generality lack proof of completeness.

2.3.3 Evolution of the definition of shape grammars

A rigorous definition of shape grammars is essential to the theoretical analysis given in this dissertation. The foundation of the subject is given in the seminal article by George Stiny and James Gips (Stiny and Gips, 1971), which contains the first formal definition of a shape grammar. Since then, there have been several other versions appearing in the literature, each reflecting either the understanding at that particular time, or a specific research flavor. In the sequel, these definitions are reviewed chronologically, exploring important characteristics over the time, capturing the tendencies of development, and obtaining insights so that a definition of shape grammars appropriate for complexity analysis can be developed. For the convenience of discussion, each definition reviewed is named using the format, *SG-DEF-year*. Moreover, for the convenience of comparison, when appropriate,

different symbol representation systems used in the different definitions will be unified according to the convention of the very first definition.

SG-DEF-1971

The following is the very first definition (*SG-DEF-1971*) taken verbatim from the seminal article by George Stiny and James Gips, who follow an analogy to phrase structure grammars (aka. generative grammars):

"A *shape grammar* (SG) is a 4-tuple: $SG = <V_T, V_M, R, I>$ where

(1) V_T is a finite set of shapes.

(2) V_M is a finite set of shapes such that $V_T^* \cap V_M = \emptyset$.

(3) R is a finite set of ordered pairs (u, v) such that u is a shape consisting of an element of V_T^* combined with an element of V_M and v is a shape consisting of (A) the element of V_T^* contained in u or (B) the element of V_T^* contained in u combined with element of V_M or (C) the element of V_T^* contained in u combined with an additional element of V_T^* and an element of V_M.

(4) I is a shape consisting of an element of V_T^* combined with an element of V_M.

Elements of the set V_T^* are formed by finite arrangements of an element or elements of V_T in which any element of V_T may be used in a multiple number of times with any scale or orientation. Elements of V_T^* appearing in some (u, v) of R or in I are called terminal shape elements (or *terminals*). Elements of V_M are called non-terminal shape elements (or *markers*). Elements (u, v) of R are called *shape rules* and are written $u \rightarrow v$. I is called the *initial shape* and normally contains a u such that there is a (u, v) which is an element of R.

A shape is generated from a shape grammar by beginning with the initial shape and recursively applying the shape rules. The result of applying a shape rule to a given shape is another shape consisting of the given shape with the right side of the rule substituted in the shape for an occurrence of the left side of the rule. Rule application to a shape proceeds as follows: (1) find part of the

shape that is geometrically similar to the left side of a rule in terms of both non-terminal and terminal elements, (2) find the geometric transformations (scale, translation, rotation, mirror image) which make the left side of the rule identical to the corresponding part in the shape, and (3) apply those transformations to the right side of the rule and substitute the right side of the rule for the corresponding part of the shape. ... The generation process is terminated when no rule in the grammar can be applied."

In many ways this definition reflects the infancy of shape grammars in the sense that the definition is purely shape-based. Markers, which are used to guide the application of shape rules, are also shapes distinguishable from the principal shape, avoiding the use of any non-shape symbols. There are no restrictions on the types of shapes used; that is, in principle, shapes can be combinations of straight lines or curves, 2D or 3D, whatsoever. Analogous to phrase structure grammars, the * operator is defined over V_T, which is interpreted as a finite arrangement of elements of V_T under transformations of similarity, including the empty shape ø[2]. This enables one to define a shape vocabulary succinctly. Examining carefully, there is no * operator defined over V_M—in all probability, this is a typo— however, this results in a definition that is not completely consistent with the shape grammar examples (Urform I, II, and III) given in the paper. In particular, the marker for RULE 1 (pp. 1461) changes in size as well as orientation while the set V_M is defined to contain only a unique marker. Noticeably, here shape rules can, in effect, only add more terminal shapes, with no capacity for subtraction, although markers can be eliminated, revised, or exchanged during the application of shape rules. Implicitly, the application of a shape rule involves the shape operations of Boolean sum and difference; moreover, the recursive application of shape rules requires that both Boolean operations are closed over the types of shapes involved.

[2] Notationally, I distinguish between Ø, the empty set and ø, the empty shape.

SG-DEF-1974

In his dissertation (Gips, 1974), Gips adopts a new definition for shape grammars (*SG-DEF-1974*), which can be viewed as an improvement over the definition of *SG-DEF-1971*. Accordingly[3],

"A shape grammar, SG, is a 4-tuple: $SG = <V_T, V_M, R, I>$ where

(1) V_T is a finite set of shapes.

(2) V_M is a finite set of shapes such that $V_T \cap V_M = \varnothing$.

(3) R is a finite set of ordered pairs (u, v) such that u is a shape consisting of an element of V_T^* combined with an element of V_M^+ and v is a shape consisting of an element of V_T^* combined with element of V_M^*.

(4) I is a shape consisting of an element of V_T^* combined with an element of V_M^*."

Apart from the * operator over V_T, * and $^+$ operators are also defined over V_M, where $V_M^* = V_M \cup \{\emptyset\}$, where $\emptyset$ is the empty shape. Compared to *SG-DEF-1971*, this augmented definition supports much more powerful shape rules, including the effect of shape subtraction. This is achieved by defining u to be in $V_T^* \cup V_M^+$ and v to be in $V_T^* \cup V_M^*$. In other words, any element of V_T^* can appear on either side of a shape rule; for a shape rule, when the terminal shape of the right side happens to be the terminal shape of the left side with the removal of some elements, the effect is a subtraction. The above way of defining shape rules also corrects the inconsistency existing in the definition of *SG-DEF-1971*; markers (non-terminal shapes) can be manipulated in the same way as terminal shapes. However, there are still 'defects' with this definition. I should be a shape consisting of an element of V_T^* combined with an element of V_M^+ instead of V_M^*, since an initial shape is not allowed to be entirely empty. In addition, V_M should be defined as a finite set of shapes such that $V_T^* \cap V_M^* = \varnothing$ instead of $V_T \cap V_M = \varnothing$, as the latter does not necessarily imply the former. As a counter example, consider the situation where V_T is a horizontal line

[3] The subscripts t and m in Gips' original formulation have been changed to T and M to be consistent in usage with *SG-DEF-1971*.

segment of unit length and V_M is a vertical line segment of unit length, then $V_T^* = V_M^*$ although $V_T \cap V_M = \varnothing$.

Additional to the definition, Gips explains the importance of markers and shows their flexible usage. Markers restrict the application of a shape rule to specific parts of a shape and help determine the transformation needed to apply the rule. Markers are more shape-like than symbol-like; apart from position, their orientation and geometrical characteristics can be equally important. For example, in the serial shape grammar generating a snowflake curve (SG7), asymmetry of the marker is used to force the generation to always proceed in the counter-clockwise direction (Figure 2-9). As a matter of fact, all the shape grammars in Gips' dissertation are designed in such a way to be entirely driven by markers. Moreover, markers are designed in such a way that the determination of both the applicability of shape rules as well as the corresponding transformations can be easily computed. Once a transformation is decided, the application of a shape rule simply becomes the operations of Boolean sum and difference. Such Boolean operations are computationally tractable for most frequently used shapes. In other words, shape grammars based on this definition are largely computation-friendly.

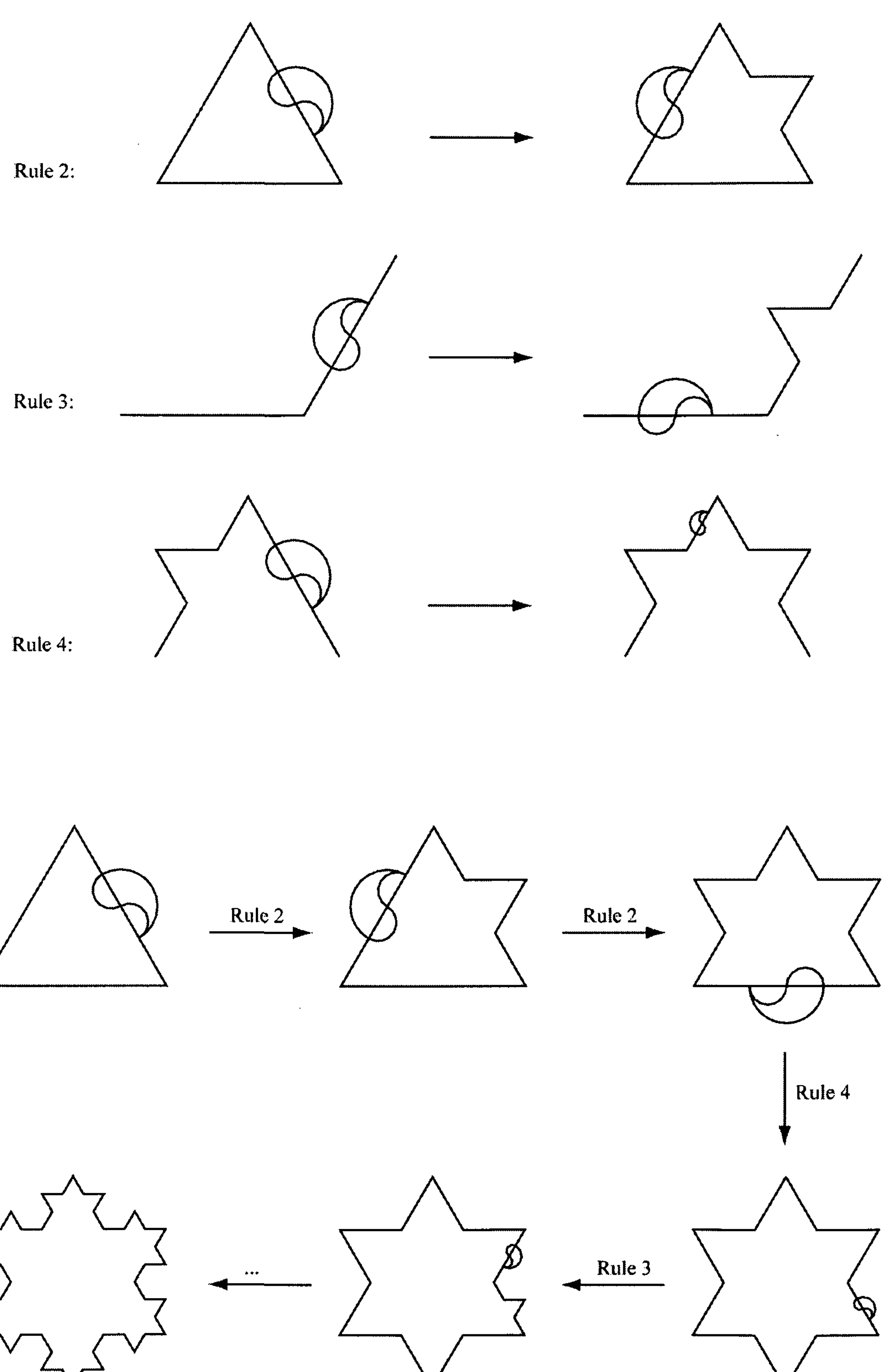


Figure 2-9: A serial shape grammar for the snowflake curve
Adapted from (Gips, 1974)

SG-DEF-1975

In his dissertation (Stiny, 1975), Stiny investigated the concept of shape grammars in terms of two models, pictorial and formal. The definition of shape grammars in the pictorial model is essentially the same as Gips' definition, *SG-DEF-1974*. However, the definition for the formal model (*SG-DEF-1975*) is 'custom-designed' for analysis, analogously to phrase structure grammars. Here, a shape is restricted to consist of only line segments. Such shapes are most common, and have certain nice properties. For example, all shapes belongs to the set defined by a unit line segment under the * operator. In contrast, this is not the case for shapes made out of arcs. The following is Stiny's definition:

"A shape grammar of index n is a 4-tuple $SG = <V_T, V_M, R, I>$ where

(1) V_T is a finite set of shapes.

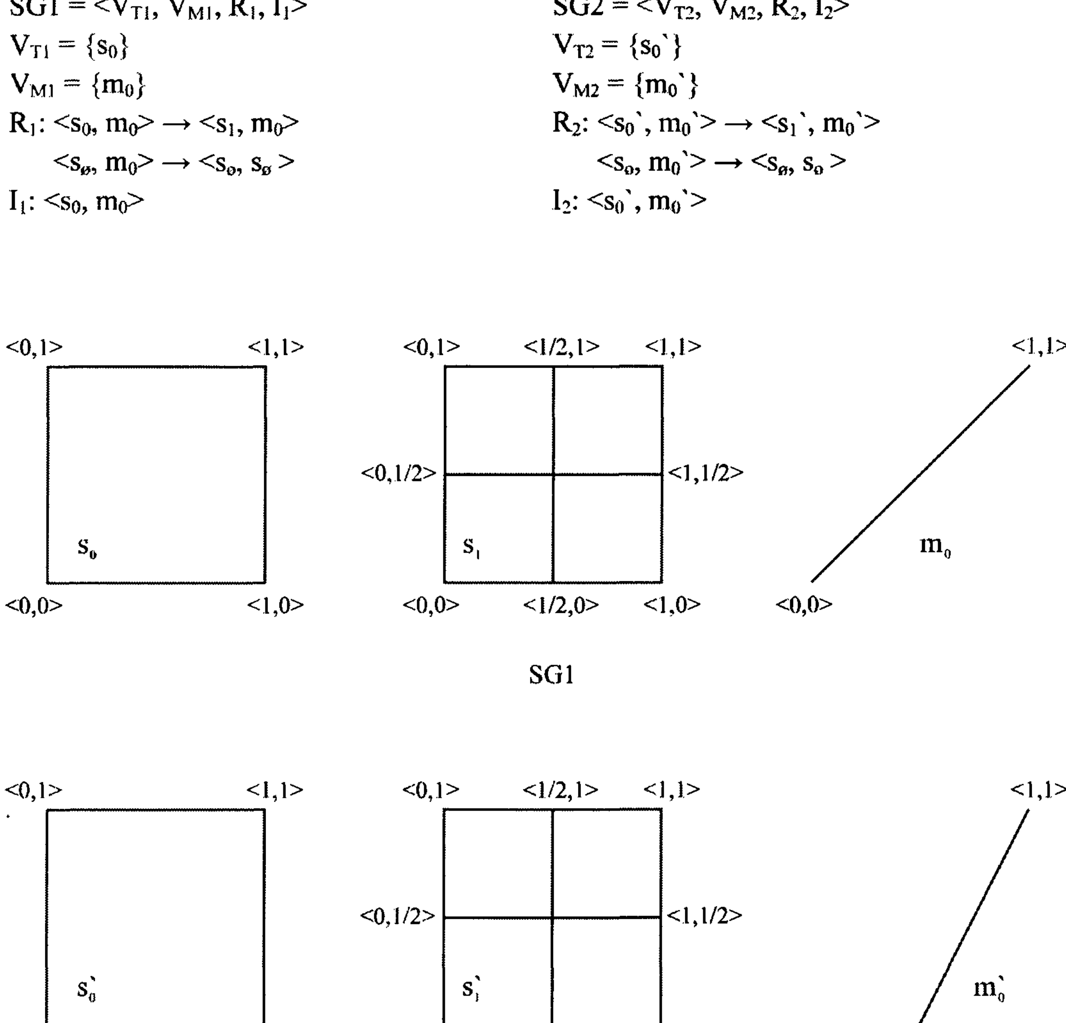
(2) V_M is a finite set of shapes.

(3) R is a finite set of shape rules of the form $<\sigma, u_1, \ldots, u_n> \rightarrow <\sigma', u_1', \ldots, u_n'>$ such that (a) $\sigma, \sigma' \in V_T^{*R}$; (b) for all i, $1 \leq i \leq n$, $u_i \in V_T^{*R}$, or $u_i = e$, for all i, $1 \leq i \leq n$, $u_i' \in V_M^{*R}$; and (c) there is an i, $1 \leq i \leq n$, such that $u_i \neq S_{\emptyset}$ and $u_i \neq e$.

(4) I is an $n+1$ tuples of shapes $I = <s_0, m_{0_1}, \ldots, m_{0_n}>$ such that (a) $s_0 \in V_T^{*R}$; (b) for all i, $1 \leq i \leq n$, $m_{0_i} \in V_M^{*R}$; and (c) there is an i, $1 \leq i \leq n$, such that $m_{0_i} \neq S_{\emptyset}$."

New to this definition is the ^{R} operator, which enforces shapes to be in a reduced form (maximal lines). The restriction of shapes made out of straight lines makes it nearly impossible to distinguish V_T^* from V_M^*. The technique to deal with such difficulty is to use shapes with multiple tuples; shapes on different tuples are on different 'channels,' which do not interfere with one another. The use of $n+1$ tuples of shapes, together with the symbol e (which behaves as the empty shape, but helps to avoid using the symbol $S_{\emptyset}$), enables shape grammars to be combined to form a new shape grammar. Figure 2-10 shows two simple shape grammars SG1 and SG2, and Figure 2-11 shows a new combined shape grammar of SG1 and SG2. In the new

shape grammar SG3, the left hand side and right hand side of a basic grammar are treated as markers, which are put in different tuples so that multiple shape grammars can be combined. This mechanism provides for the definition of composite languages of shapes from simple languages of shapes. However, to my knowledge, no further work along these lines can be found in the subsequent literature.



SG2

Figure 2-10: Two simple shape grammars: SG1 and SG2
Adapted from (Stiny, 1975)

$SG3 = \langle V_{T3}, V_{M3}, R_3, I_3 \rangle$

$V_{T3} = \{s_0^{\prime\prime}\}$

$V_{M3} = \{m_0^{\prime\prime}\}$

R_3: $\langle s_\emptyset, s_0, m_0, s_\emptyset, s_\emptyset \rangle \rightarrow \langle s_\emptyset, s_1, m_0, s_\emptyset, s_\emptyset \rangle$

$\langle s_\emptyset, s_\emptyset, m_0, s_\emptyset, s_\emptyset \rangle \rightarrow \langle s_\emptyset, s_\emptyset, s_0, s_\emptyset, s_\emptyset \rangle$

$\langle s_\emptyset, s_\emptyset, s_\emptyset, s_0^{\prime}, m_0^{\prime} \rangle \rightarrow \langle s_\emptyset, s_\emptyset, s_\emptyset, s_1^{\prime}, m_0^{\prime} \rangle$

$\langle s_\emptyset, s_\emptyset, s_\emptyset, s_\emptyset, m_0^{\prime} \rangle \rightarrow \langle s_\emptyset, s_\emptyset, s_0, s_\emptyset, s_\emptyset \rangle$

$\langle s_\emptyset, m_0^{\prime\prime}, s_\emptyset, m_0^{\prime\prime}, s_\emptyset \rangle \rightarrow \langle s_0^{\prime\prime}, s_\emptyset, s_\emptyset, s_\emptyset, s_\emptyset \rangle$

I_3: $\langle s_\emptyset, s_0, m_0, s_0^{\prime}, m_0^{\prime} \rangle$

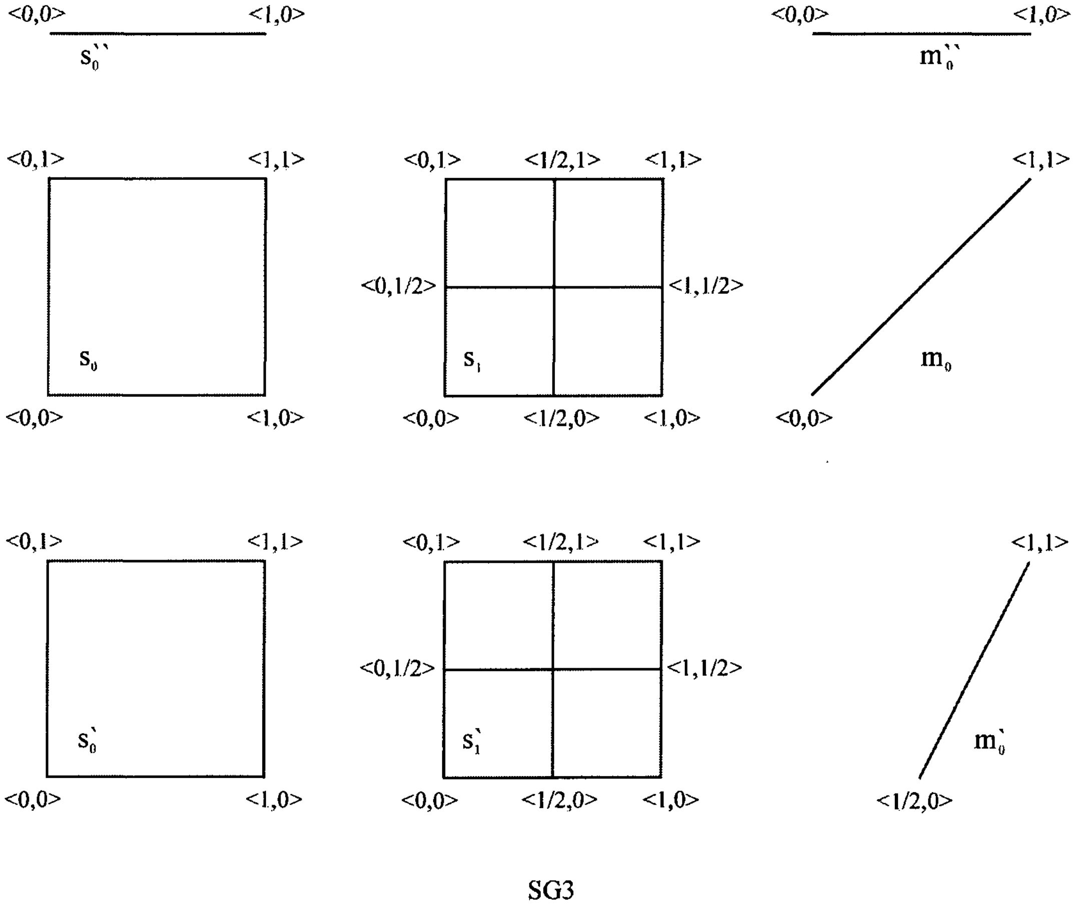


Figure 2-11: The combined shape grammar of SG1 and SG2
Adapted from (Stiny, 1975)

This definition also distinguishes a special case of shape rule application. When the union of the left hand side of a shape rule has fewer than two points of

intersection, there are potentially infinitely many ways to apply such a shape rule. It seems that Stiny, at the time, regarded the 'infinitely many ways' unfavorably, defining the transformation to be the one, which transforms the left hand side in such a way that each element has an identical, rather than a subshape, counterpart in the configuration.

Stiny also showed that shape grammars so defined are as equally powerful as Turing machines. Algorithms for shape rule application, and Church's thesis demonstrate that Turing machines can simulate any shape grammar. Likewise, a shape grammar can be constructed to simulate any Turing machine. The details are given in the Section 2.3.5, where I compare and contrast shape grammars and phrase structure grammars.

SG-DEF-1977

In his now famous paper in which he introduces the Chinese ice-ray lattice grammar (Stiny, 1977), Stiny gives the first definition for labeled shapes and parametric shape grammars, briefly explaining the corresponding definition for non-parametric shape grammars in the appendix to the paper. As Stiny further elaborates on the definition of parametric shape grammars in (Stiny, 1980a), the discussion on parametric shape grammars is postponed to section *SG-DEF-1980*, and the focus here is on labels and markers. The following is a brief paraphrase of the definition, using, as much as possible, Stiny's original language:

> « A *shape* is a finite arrangement of straight lines of limited but nonzero length in two or three dimensions. Shapes are specified by *drawing* them in a Cartesian coordinate system. The coordinate system is usually not given explicitly, its origin, axes, and units being understood.
>
> A family of shapes can be defined by associating parameters or parametric expressions satisfying certain conditions with a limited number of points coincident with lines in a given shape. A particular member of this family is specified, by giving an *assignment* of real values to parameters that satisfies the

conditions. The result of applying an assignment g to a *parameterized shape* s is the shape denoted by $g(s)$.

A *labeled point* $p{:}m$ is a point p with a symbol m associated with it. Two labeled points $p_1{:}m_1$ and $p_2{:}m_2$ are equivalent if and only if $p_1 = p_2$ and m_1 is identical to m_2. A transformation t of a labeled point $p{:}m$ is the labeled point $t(p){:}m$, where $t(p)$ is the image of p under t.

A *labeled shape* consists of a shape and an unordered set of labeled points. More precisely, a labeled shape σ is given by the ordered pair $\sigma = <s, l>$, where s is a shape and l is an unordered set of labeled points. Labeled points in l may be coincident with lines in s, but this need not necessarily be the case. A labeled shape $\sigma = <s, l>$ is specified by drawing s in a Cartesian coordinate system and placing the labels in l next to the points with which they are associated. A shape s is the labeled shape $<s, \emptyset>$, where $\emptyset$ is an empty set.

A *labeled parameterized shape* σ is given by $\sigma = <s, l>$, where s is a parameterized shape and l is an unordered set of labeled parameterized points. An assignment g to the parameters in s and l specify a particular labeled shape $g(\sigma) = <g(s), g(l)>$ in the family of labeled shapes defined by σ.

A parametric *shape grammar* has five parts: (1) S is a finite set of shapes. (2) L is a finite set of unordered sets of labeled points. (3) R is a finite set of *shape rules* of the form $A \rightarrow B$, where A and B are labeled parameterized shapes, $A = <u, i>$ and $B = <v, j>$. Any assignment g to the parameters in the parameterized shapes u and v, and the unordered sets of labeled parameterized points i and j, results in shapes $g(u)$ and $g(v)$, that is in S^*, and unordered sets of labeled points $g(i)$ and $g(j)$, that are in L^+ and L^* respectively. (4) I is a labeled shape such that $I = <w, k>$, where w is a shape in S^*, and k is an unordered set of labeled points in L^+. The labeled shape, I, is called the *initial shape.* (5) T is a set of transformations.

A shape is generated by a shape grammar by beginning with the initial shape I, and recursively applying the shape rules in the set R. A shape rule $A \rightarrow B$

applies to a labeled shape C when there is an assignment g and a transformation t such that $t[g(A)] \subseteq C$. The result of applying the shape rule $A \rightarrow B$ to the labeled shape C under g and t is another labeled shape given by $[C - t[g(A)]] \cup t[g(B)]$. The shape generation process terminates when no shape rule in the set R can be applied. The *language* defined by a shape grammar is the set of shape s generated by the shape grammar, that is, labeled shapes of the form $<s, \emptyset>$.

The definition of standard shape grammars is obtained from the definition of parametric shape grammars by deleting part (5) and replacing part (3) with this statement: R is a finite set of shape rules of the form $A \rightarrow B$, where A and B are labeled shapes. A shape rule $A \rightarrow B$ applies to a labeled shape C when there is a Euclidean transformation t such that $t(A) \leq C$. The result of applying $A \rightarrow B$ to C under t is the labeled shape $[C - t(A)] \cup t(B)$. »

This definition of shape grammars uses labeled points instead of markers, as opposed to *SG-DEF-1971*, *SG-DEF-1974*, and *SG-DEF-1975*. As explained in the Appendix of (Stiny, 1977), labeled points function in the same way as markers to guide the shape generation process; however, labels are invariant under the Euclidean transformations whereas markers are not. However, this distinction might be construed as overly simple, and depends on the design of shape rules. For example, in *SG-DEF-1974*, Gips employs certain geometrical characteristics, such as asymmetry, of the markers to control shape rule application. By replacing markers with labels, the only important geometry information is position. However, as most grammars do not rely on the geometric characteristics of markers or labels beyond their position, markers and labels, can be and are used interchangeably in this dissertation unless otherwise stated, for instance the phrases 'marker-driven' and 'label-driven' mean exactly the same thing.

Knight provides an extensive discussion on the usage of labels (Knight, 1983). Labels in a shape rule normally supply additional information not provided by the shapes themselves, and indicate (1) how, (2) where, and (3) when a shape rule may applied to the design being generated. For case (1), labels specify under which

Euclidean transformations a rule can apply (usually by altering the symmetry). For case (2), labels specify to which subshape(s) in the design a shape rule can be applied. For case (3), labels are associated with the design instead of with any particular point(s). This last kind of labeling is most frequently used to indicate successive stages in the generation of a design. Here, labels serve as status markers, regulating the sequence and repetition of rule applications. For the first two cases, labels are spatial as their location is important. For the third case, labels are non-spatial as their presence rather than location is more important. In this dissertation, I tend to call non-spatial labels *as markers* to distinguish them from the spatial *labels*, although, more often than not, both are used interchangeably, as discussed in last paragraph.

SG-DEF-1980

In 1980 Stiny published a paper (Stiny, 1980a) to explain the shape grammar formalism, where concepts of labeled shapes, and non-parametric and parametric shape grammars are elaborated. This version of the shape grammar definition (*SG-DEF-1980*) has subsequently become standard, and is the most widely quoted. The following is a brief description, again using, as much as possible, Stiny's original language:

> « A *shape* is a limited arrangement of straight lines defined in a Cartesian coordinate system with real axes and an associated Euclidean metric. A shape is specified by *maximal* line representation. A shape is a *subshape* (part) of another shape whenever every line of the first shape is also a line of the second shape. A *labeled shape* consists of two parts: a shape and a set of labeled points. A *parameterized shape* is obtained by allowing the coordinates of the end points of the maximal lines in a given shape to be variables. A *parameterized labeled shape* σ is given by $\sigma = <s, P>$, where s is a parameterized shape, and P is a finite set of labeled parameterized points. A *labeled parameterized point* is a labeled point p where the coordinates of p are variables.

A *shape grammar* has four components: (1) S is a finite set of shapes; (2) L is a finite set of symbols; (3) R is a finite set of *shape rules* of the form $A \rightarrow B$, where A is a labeled shape in $(S, L)^+$, and B is a labeled shape in $(S, L)^*$; and (4) I is a labeled shape in $(S, L)^+$ called the *initial shape*.

In *non-parametric shape grammars*, a shape rule $A \rightarrow B$ applies to a labeled shape C when there is a transformation t such that $t(A)$ is a subshape of C. The labeled shape produced by applying the shape rule $A \rightarrow B$ to the labeled shape C under the transformation t is given by $[C - t(A)] + t(B)$.

Parametric shape grammars are extensions of non-parametric shape grammars in which shape rules are defined by filling the open terms (point variables) of a general schema. A *shape rule schema* $A \rightarrow B$ comprises parameterized labeled shapes, A and B, where no member of the family of labeled shapes specified by A is the empty labeled shape. When specific values are given to the variables of A and B by an assignment g, to determine specific labeled shapes, a new shape rule $g(A) \rightarrow g(B)$ is defined. This shape rule can then be used to change a given labeled shape into a new shape in the usual way. That is, shape rule application is expressed as $[C - t(g(A))] + t(g(B))$. »

In comparison to *SG-DEF-1975* and *SG-DEF-1977*, this definition is much more succinct and allows for more flexible shape rules. In *SG-DEF-1975*, markers are just shapes on different channels from the principal configuration. In *SG-DEF-1977,* labels replace markers. In this definition, shape rules without symbols are supported; subshape matching drives shape rule application rather than markers or labels, whence, shape emergence becomes an important factor to be considered during shape rule application. While this allows new types of shape rules, there is a price to pay. Computationally, determining the applicability of shape rules as well as the corresponding transformations becomes much more complicated. Accordingly, in comparison to early definitions, this definition becomes less computation-friendly.

Note that, in this definition, the allowable transformations can be restricted to special kinds, although this facility seldom features in the subsequent literature. The

restriction on the transformations in the case of the infinitely many ways of applying a shape rule, which appears in *SG-DEF-1975*, is not singled out here.

The introduction of parametric shape grammars basically extends the scope of allowable transformations. While providing for more flexible and natural design of shape rules, function g for assigning parameters implicitly implies computational difficulty. Such functions are those allowing the points of a shape as variables (open terms) and the space of such functions is infinitely large. This means searching an infinite space. Indeed, Stiny states that devising an algorithm to find the transformations under which a parametric shape rule applies to a configuration is an open question (Stiny, 2006: pp 280). In terms of practical computer implementations, function g needs to be defined more specifically when designing a parametric shape grammar so that a case-based efficient algorithm can be devised.

SG-DEF-1991

An obvious deficiency of *SG-DEF-1980* is the limitation on shapes requiring them to be composed of straight lines. Shapes, in general, are formed as arrangements of points, lines, planes, solids, and even exotic curves and surfaces. In (Stiny, 1991), Stiny generalizes the definition of *SG-DEF-1980* in terms of shape algebras.

> "In a shape grammar, any pair of objects A and B defines a rule $A \rightarrow B$. The rule applies to an object C in a two-stage process involving a transformation t. The transformation is used in both stages, once with the relation $\leq$ to distinguish some part of C, and then again with the operations + and − to replace the part that has been picked out."

> "Mathematically, if there is a t such that $t(A) \leq C$ is satisfied, then an object is produced according to the formula of $[C - t(A)] + t(B)$. t, $\leq$, +, and − are operators defined over a shape algebra, where t is a transformation function over a shape and can be generalized as a being alike function, $\leq$ is a partial order relation in terms of subshape, and + and − are Boolean sum and difference. All these operators are applied recursively until reaching the basic elements, on which these operators are directly defined."

Under this definition, shapes are readily extensible. A shape can be *simple* —formed from basic elements of a single kind; or *compound* —a mix of various elements, optionally augmented in some way, for example, by colors. The only condition is that the operators of any shape algebra are defined on all its elementary objects, are recursively applicable, and are closed. In contrast to definition *SG-DEF-1980,* indeterminacy, that is, the infinitely many ways of applying a shape rule, is encouraged rather than restricted. While this causes little trouble for designers, indeterminacy is a tough issue for computer implementations. Additionally, shape emergence is regarded as a way of producing novel designs.

SG-DEF-1992

In the following year, Stiny formally extends the definition of shape grammars to include labels and weights, which he describes in algebraic terms (Stiny, 1992). Briefly, the algebra U_{ij} contains shapes, each of which contains a finite but possibly empty set of basic elements that are maximal with respect to one another. U_{ij} can be augmented with labels or weights to form, respectively, new algebras, V_{ij} and W_{ij}. Labels can classify shapes as collections, or introduce other additional information. Weights are properties of shapes, for example, the thickness of lines. The shape grammar formalism in algebras U_{ij} is extended to algebras V_{ij} and W_{ij} without modification. With this background, we have the following definition for a shape grammar:

> "In the algebras U_{ij}, V_{ij} and W_{ij}, and in the algebras formed by combining them, shape grammars contain rules that are followed to carry out computations with shapes. Rules are defined with shapes, and apply recursively to a given initial shape and then to shapes produced from shapes to determine a series of shapes that forms a computation.
>
> In exact detail, a rule $A \rightarrow B$ establishes a relation between two shapes A and B. The rule applies to another shape C whenever there is a transformation t such that $t(A) \leq_X C$. A new shape is then produced according to the formula $[C -_X t(A)] +_X t(B)$. The shapes A, B, and C are taken from the algebra in which

the rule is defined; they may be individual shapes in U_{ij}, V_{ij} or W_{ij}, or, more generally, they may be compound shapes formed when theses algebras are combined. The part relation $\leq_X$, and the operations of sum $+_X$ and difference $-_X$ also depend on the algebra in which the rule is defined.

Schemata are sometimes used to extend these devices. A shape schema $A(x)$ is a finite but possibly empty set of variables x that describes a family of shapes. If $A(x)$ is empty, then a shape is given automatically. Otherwise, a function F assigns values to the variables that depend on the algebra in which $A(x)$ is defined.

...

Schemata are also used to define families of objects in Cartesian products, either families of compound shapes or families of rules in shape grammars. For example, any two shape schemata $A(x)$ and $B(x)$ — with or without shared variables — that describe shapes in the same algebra can be joined in a rule schema $A(x) \rightarrow B(x)$. A function F assigns values to all of the variables in this schema, so that any required conditions are satisfied. And once the sets defined in this way are used to obtain shapes, as in the procedure for an algebra W_{ij}, a rule is formed that may be applied in the manner established above. In effect, this allows for shapes and their relationships to vary within rules, and extends the transformations under which rules apply."

SG-DEF-2006

In his monograph, *Shape: Talking about seeing and doing*, Stiny discusses shape grammars in terms of drawing shapes and calculating by seeing. The historical analogy of shape grammars to phrase structure grammars is re-examined, with the conclusion that the analogy is inappropriate; it implies a lot more than it should. As a matter of fact, during the design process, a designer's vocabulary of shapes is typically not prescribed; instead, new types of shapes are defined on the fly. Noticeably, in this book, the definition of a shape grammar is never mentioned and

only alluded to informally, with the basic formalism remaining the same as *SG-DEF-1992*.

2.3.4 Trends in the development of shape grammars

From the above definitions, it is clear that the evolutionary development of shape grammars fall into two stages: marker-driven and subshape-driven. Definitions *SG-DEF-1971*, *SG-DEF-1974*, *SG-DEF-1975* and *SG-DEF-1977* belong to the former category in the sense that shape grammars, so defined, are controlled by markers. It is the markers, which play a pivotal role in determining both the applicability of shape rules as well as their corresponding transformation. Markers can be designed in a way that the determination of applicability and transformation is relatively straightforward to compute. In later developments of shape grammars, markers evolve as alphanumeric symbols, which make determination even simpler (albeit while losing some power). All other definitions belong to the subshape-driven category, during which marker-driven (aka. label-driven) and subshape-driven rule application can coexist. In other words, the definitions support both marker-driven and subshape-driven shape grammars. In comparison to marker-driven shape grammars, there are harder computational issues involved with subshape-driven shape grammars, in particular, parametric subshape recognition and indeterminacy.

Chronologically, the above definitions exhibit backwards compatibility. That is *SG-DEF-1971* << [*SG-DEF-1974*, *SG-DEF-1975*] << *SG-DEF-1977* << *SG-DEF-1980* << *SG-DEF-1991* << *SG-DEF-1992* << *SG-DEF-2006*, where the right side of << is more general than the left side. There is, however, a discrepancy between *SG-DEF-1974* and *SG-DEF-1975,* which were developed independently by the two principal authors of shape grammars, for very distinct research purposes, from the same root, *SG-DEF-1971*.

The evolutionary development shows a trend from 'rigid' to 'soft'. 'Rigid' here means that the shape grammars are defined in a way that is closer to phrase structure grammars. Such shape grammars are more machine-bound in the sense that they are relatively easy to carry out (compute) on a computer, but harder to use to generate

novel designs. As a matter of fact, there is very limited novelty involved. On the other hand, 'soft' being more human-centered, shows more concern and consideration on how to use shape grammars to generate novel designs. This explains, in part, the importance of subshape-driven grammars, concepts of indeterminacy and shape emergence, and the support for ambiguity in shape grammar research. Humans have little trouble handling such concepts. Moreover, human designers actually benefit from them. However, these concepts are problematical when considering computer implementation. In other words, the earlier definitions are more computation-friendly and the latter ones are less.

2.3.5 Shape grammars vs. phrase structure grammars

It is hard to completely negate the connection between shape grammars and phrase structure grammars (aka. generative grammars), especially at the early stage of the development of shape grammars, although recent work shows that such analogy between them is inappropriate (Stiny, 2006). The connection is two-sided. On one hand, it does help in understanding the computational properties of shape grammars, for example, issues such as computing power, decidability, and computational complexity. On the other hand, the analogy may confuse computer scientists and mislead designers.

For the computer scientist, one cause for the confusion lies in the dual role that shapes play. In some context, shapes may behave just like symbols; you can move them around, and rearrange them with different orientation and scale, but they do not interact with one another. In fact, a symbol is something atomic with a fixed shape. In computing terms, these can be represented by some fixed internal binary number(s). However, in other contexts, shapes may behave quite differently. For example, when two shapes are placed together in some arrangement, new shapes may emerge, visible to most designers (see Section 2.3.1). Moreover, as shown later, shape grammars can simulate Turing machines. Similarly, some well-known grammars in computer science such as string grammars, graph grammars, etc., can also be simulated by shape grammars. In this sense, shape grammars are 'super' to string grammars and graph grammars. Because of this, shape grammars may show

strong similarities to other grammars, for example, graph grammars, although their fundamental problems remain different and distinct (See Section 5.3.1).

Designers distinguish themselves by their creativity, and generally do not believe rules can help them create novel designs. Drawing a close connection between shape grammars and the much 'drier' phrase structure grammars might make them feel even less convinced of the efficacy of grammars.

Widely known as generative grammars, phrase structure grammars are an integral part of the theory of formal languages (Harry and Christos, 1997). Thus, any connection between shape grammars and phrase structure grammars can be extended to a connection between shape grammars and formal languages, so that theories of formal languages can be leveraged to understand the properties of shape grammars.

Indeed, using definition *SG-DEF-1975*, Stiny has shown that shape grammars can simulate any Turing machines (Stiny, 1975). That is, shape grammars are at least as powerful as Turing machines. The conclusion is two-sided. On one hand, it shows that shape grammars are a powerful formalism. On the other hand, it shows that shape grammars share similar technical difficulties with Turing machines in terms of a computer implementation. For the latter reason, the simulation of a Turing machine is briefly outlined.

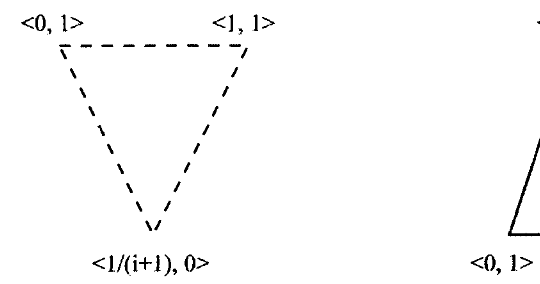
In order to show that a shape grammar can be constructed to simulate any given Turing machine, it is essential to show five aspects. The following explains in detail each aspect as well as the corresponding simulation.

(i) The states of a Turing machine can be encoded as shapes in reduced form, such that no two similar shapes represent distinct states. The set of shapes corresponding to the set of states of the Turing machine will form the main part of the set of markers for the constructed shape grammar.

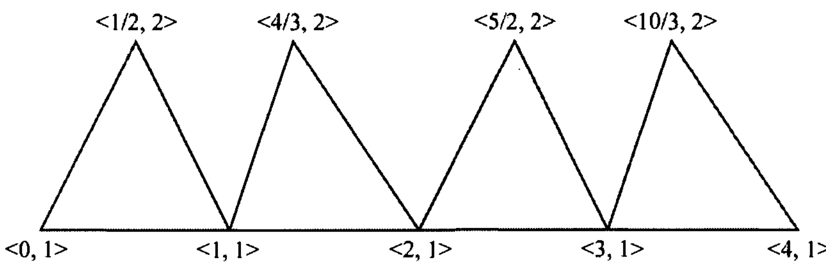
Consider a Turing machine with states given by the set $K = \{q_i \mid 0 \leq i \leq n\}$. Each state q_i in K can be encoded uniquely by a triangle shape s_i with points

$\{\{<0,1>,<1,1>\},\ \{<1,1>,<\frac{1}{i+1},0>\},\ \{<0,1>,\ <\frac{1}{i+1},0>\}\}$ where $0 \leq i \leq n$.

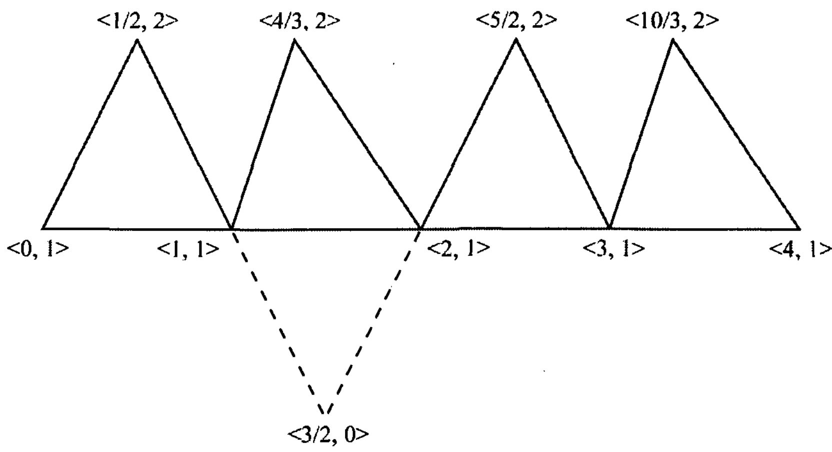
Notice that for states q_i and q_j, if $q_i \neq q_j$, then s_i is not similar to s_j. For the shape rules simulating transitions, the states serve as markers. Figure 2-12a shows an example of one such state.



(a) A shape encoding the state q_i (b) A shape encoding the symbol a_i



(c) A shape encoding the tape $a_1a_2a_1a_2$



(d) A shape encoding a configuration

Figure 2-12: A shape grammar for Turing machines
Adapted from (Stiny, 1975)

(ii) The tape symbols, including the blank tape symbol, of a Turing machine can be encoded as shapes in reduced form such that no two similar shapes represent different tape symbols. The set of shapes corresponding to the set of tape symbols of the Turing machine form the main part of the set of terminals for the constructed shape grammar.

The tape symbols can be defined in a way similar to state symbols. Let the Turing machine have the set of tape symbols $\Sigma = \{a_i \mid 1 \leq i \leq m\}$. Let the blank symbol be given by a_0. Each symbol in the set, $\Sigma \cup \{a_0\}$, can be uniquely encoded by a triangle with points in the set $\{\{<0,1>, <1,1>\}, \{<1,1>, <\frac{1}{i+1}, 2>\}, \{<0,1>, <\frac{1}{i+1}\ 2>\}\}$ where $0 \leq i \leq m$. Figure 2-12b is an example of such a symbol.

(iii) Turing machine tapes and configurations can be represented by shape grammars.

Consider the Turing machine tape $a_{i_0} \dots a_{i_k}$ where all symbols to the left of a_{i_0} and to the right of a_{i_k} are the blank tape symbol a_0. This tape can be presented by the shape $t_{i_0} \cup trans(t_{i_1}, 1) \cup \dots \cup trans(t_{i_k}, k)$, where $trans(t, x)$ means translating shape t by x along the X-axis. Figure 2-12c is an example of such a tape.

Now assume that the Turing machine is in state q_i and is scanning the tape symbol a_{i_j} occurring in the tape $a_{i_0} \dots a_{i_j} \dots a_{i_k}$. This configuration can be represented by the pair of shapes $<T, trans(s_i, j)>$ where T is the shape representing the tape $a_{i_0} \dots a_{i_j} \dots a_{i_k}$. Figure 2-12d is an example of such a configuration.

(iv) Turing machine transitions can be represented as shape rules. The set of shape rules corresponding to the set of transitions of the Turing machine form the main part of the set of shape rules for the constructed shape grammar.

A transition $<q_i, a_j, a_{j'}, q_{i'}, L>$, which reflects a Turing machine in state q_i scanning symbol a_j, replacing it by symbol $a_{j'}$, subsequently, going into state $q_{i'}$, and moving its tape one tape cell to the left, can be represented by the shape rule $<t_j, s_i> \rightarrow < t_{j'}, trans\ (s_{i'}, 1)>$. Figure 2-13 shows two shape rules, which simulate such transitions.

(v) A Turing machine computation can be simulated as a derivation in the constructed shape grammar.

With the above setup, it is easy to see that the constructed shape grammar simulates the computation of a Turing machine by derivation.

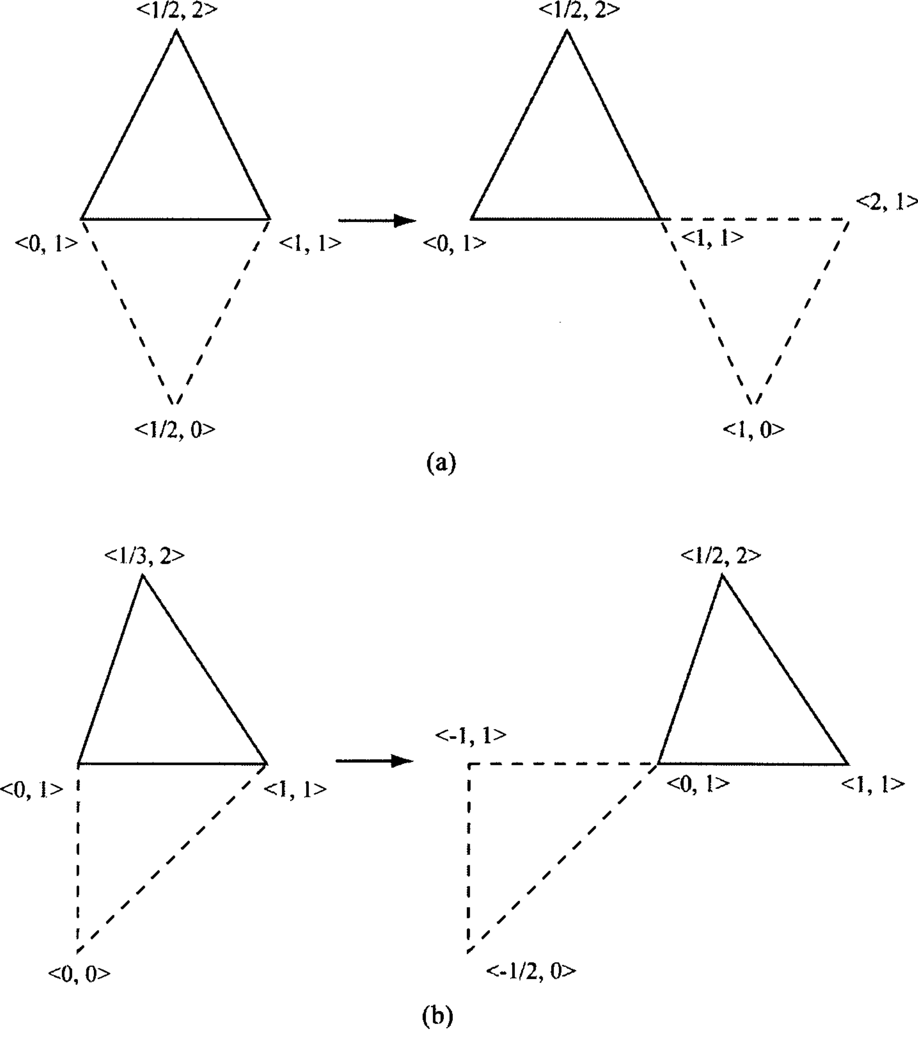


Figure 2-13: Two shape rules simulating machine transitions
Adapted from (Stiny, 1975)

The following results are direct extensions from the theory of formal languages (Harry and Christos, 1997). These results, which have an impact on computer implementation of shape grammars, have received no attention in the literature.

For instance, it is well known that a Turing machine may not halt. Assume there is a computer program, which recursively applies the shape rules of a shape grammar until no shape rules can apply. As a result, this computer program will not halt for any shape grammar that simulates a non-halting Turing machine. In other words, shape grammars may not halt and an interpreter of shape grammars needs a way of handling such shape grammars.

It is also known that a simulation of a non-deterministic Turing machine (abbreviated as NTM) with n steps by a deterministic Turing machine (abbreviated as DTM) requires exponentially many steps in n. Naturally, a shape grammar can be designed in a non-deterministic fashion, for example, the sports figure grammar (Carlson et al., 1991) (and also Figure 4-4). Thus, a shape grammar can be designed to simulate any NTM in a fashion similar to simulating a DTM. This is equivalent to the problem of simulating a NTM by a DTM; exploring the entire language of such a shape grammar might take an exponential number of steps, making exhaustive interpretation impractical. Therefore, the language space of a shape grammar can be exponentially large and a shape grammar interpreter has to devise a way of handling such cases.

Lastly, another well-known theorem for unrestricted string grammars is that the membership problem — that is determining whether a string belongs to the language defined by a grammar or not — is undecidable. A shape grammar can be designed to simulate a string grammar in a similar manner to simulating a Turing machine. For such a shape grammar, the membership problem is equally undecidable — the proof, by contradiction, is trivial. In other words, in general, determining whether a configuration (aka. shape) belongs to the language defined by the shape grammar is unsolvable; that is, the problem of parsing a configuration against a shape grammar is unsolvable in general. Whether it is possible to restrict shape rules, to restriction categories similar to those defined for string grammars, for example, context-free

string grammars, is an open research problem, and is beyond the scope of this dissertation.

Chapter 3 Building feature extraction from image data

The general approach developed for the AutoPILOT project assumes the availability of the exterior features of a building. There is a practical difficulty in this assumption, namely, that the geometry of an arbitrary target building is usually unknown, and time-consuming to generate. Automatic generation of building feature input is, of course, desirable. Recent progress in computer vision offers the promise to automate the generation procedure. In this chapter, related computer vision research is examined. In particular, two pipelines from processing image data to building features are compared: *an ideal pipeline*, based on the requirements of the AutoPILOT project, and *a realistic pipeline*, based, mainly, on the works of (Stamos, 2001), (Frueh et al., 2004) and (Fulkerson et al., 2008). As in the AutoPILOT project, the particular focus here is on conventional building types, that is, buildings composed of rectilinear spaces and components, or approximated as such. Moreover, the goal of building feature extraction here is not a full-blown model; instead of, only those building features important for initial layout estimation are under consideration, for example, footprints, windows, doors, chimneys, etc. As a caution, the investigation here is purely theoretical; the implementation is left as a future research.

3.1 Photo images and range images

Photo and range images are two types of image data commonly employed in computer vision research. Their basic characteristics are briefly reviewed below.

3.1.1 Photo images

Quite simply, a photo image records the world through color or brightness information, and photo-imaging systems have become cheaper and ubiquitous. Points are the basic units for describing the geometry of an object. Measuring points using photo images is the precise goal of traditional *photogrammetry* (Mikhail et al., 2001). Modern computer technology relieves photogrammetry from reliance upon specific physical devices.

The basic approach to measuring point coordinates is triangulation (Figure 3-1). By taking photographs from at least two different locations, 'lines of sight' are developed from each camera to points on the object. The lines of sight are mathematically intersected to produce the 3D coordinates.

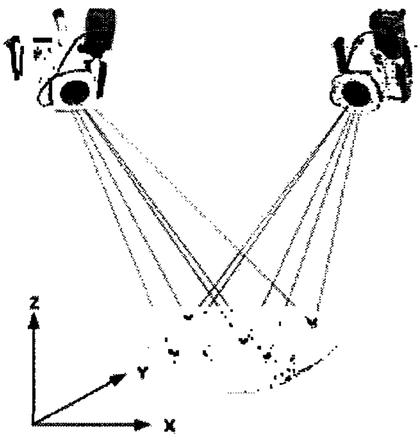


Figure 3-1: Measuring the 3D coordinates of points by triangulation (http://www.geodetic.com/Whatis.htm: accessed May 2009)

This approach necessarily implies a sub-procedure, namely, that of establishing correspondences between pixels in different views. Automatic pixel matching is

difficult and costly; many solutions have been proposed, including the famous RANSAC algorithm (Fischler and Bolles, 1981).

A major drawback is that photogrammetric measurements are inherently dimensionless. For instance, there is no way of distinguishing between pictures of full-sized cars and their matchbox models. Mathematically, an extracted model is correct to scale; for exact dimensions, we need, at least, one known measurement. As discussed later, compared to range images, photo images are relatively noiseless, although rectification and correction of radial distortion are often required.

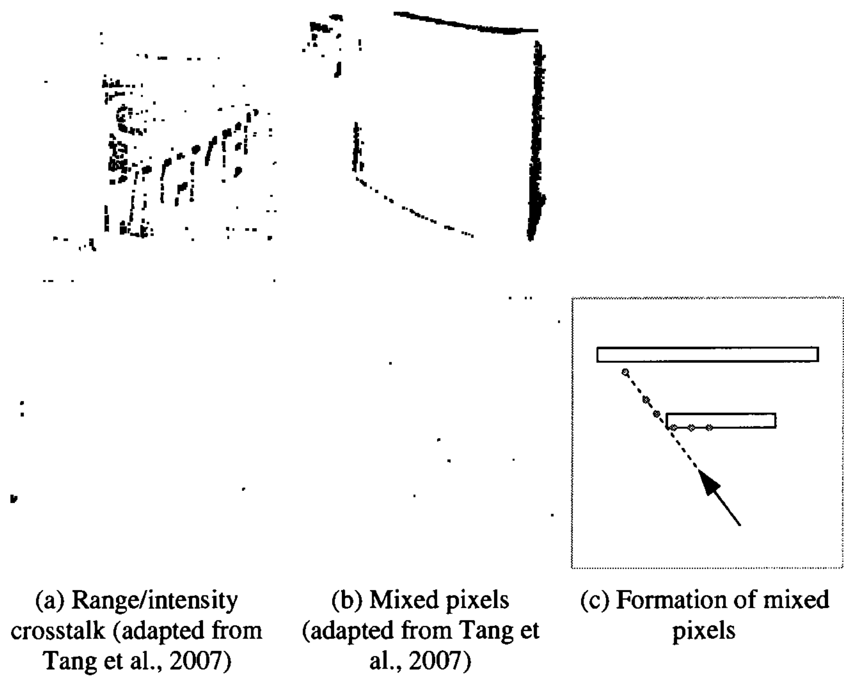
3.1.2 Range images

Range images store the depth at which the ray associated with each pixel first intersects the scene as observed by a range sensor. A Cartesian transformation converts range pixels to points in space, resulting in a *3D point cloud.* Range images are 'easier' in that the image data points explicitly represent scene surface geometry. The incident mesh is virtually ready for varying uses, for example, monitoring the progress of construct sites (Shih and Wang, 2004). However, to extract the geometry as basic shapes, such as lines, planes, cylinders, etc., most low-level problems that exist for photo images remain the same, such as filtering, segmentation, and edge detection (Paul, 1988).

Range images are captured mainly using 3D laser scanners, which typically have limits on the view in terms of horizontal and vertical angles. To capture a given scene, multiple scans are required. These scans have to be further aligned, also known as *registration*, and optionally merged together, as each scan is represented in a local coordinate system relative to the laser scanner. Many algorithms had been proposed to automate the registration of a large number of individual scans; for instance, Huber and Hebert describe fully automated registration based on spin-images (Huber and Hebert, 2001).

Many laser scanners determine the range by measuring the shift in phase between an amplitude-modulated continuous-wave emitted beam and its reflection. This principle leads to two detrimental effects, namely range/intensity crosstalk

(Figure 3-2a) and mixed pixels (Figure 3-2b) (Herbert and Krotkov, 1992; Tang et al., 2007). Range/intensity crosstalk is due to the fact that a range measurement is not independent of the reflective properties of the observed surface. The influence is so significant that useless range data can be produced. Mixed pixels happen when a laser beam partially hits the front surface and then hits another surface behind (Figure 3-2c). The fact that the range is measured by integrating over the entire projected spot leads to the result that the measured range can be anywhere along the line of sight. The implication is that occluding edges of scene objects are often unreliable.



(a) Range/intensity crosstalk (adapted from Tang et al., 2007) (b) Mixed pixels (adapted from Tang et al., 2007) (c) Formation of mixed pixels

Figure 3-2: Range/intensity crosstalk and mixed pixels

Although there are algorithms that have been proposed to eliminate mixed pixels in special cases (Tuley et al., 2005; Tang et al., 2007), simple general cost-effective remedies do not seem to exist at the moment (Herbert and Krotkov, 1992). As a result, these two effects greatly reduce accuracy, down to centimeter from the advertised millimeter level.

3.2 An ideal pipeline

The desired building feature input, for example, windows and doors, has to be given as typed objects, that is, the type and geometry of a particular object. Notably, we are able to distinguish between the geometry representing a window and that of a door. This requires that the object recognition algorithm is capable of both extracting object geometry, as well as annotating its type.

Figure 3-3 shows an ideal pipeline based on the above discussion. The pipeline first builds a co-located model of range and photo images; that is, we know which pixel in the photo image corresponds to a point in 3D space. In the next step, geometries are extracted and photo images are automatically annotated. Each basic shape in the extracted geometries is typed by using the annotation of the corresponding pixels in the photo images. In this way, an annotated 3D model is created. The desired features of the model are outputted into an XML file, to serve as input to the layout determination program.

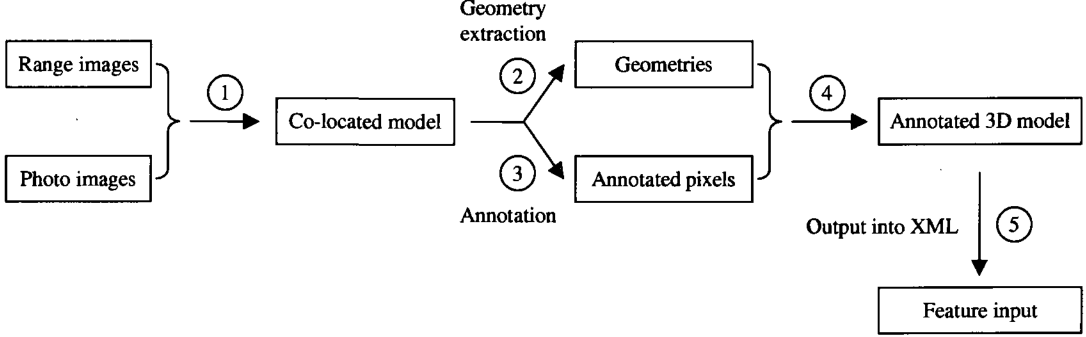


Figure 3-3: An ideal pipeline

3.3 State-of-art

The ideal pipeline is related to both modeling-from-reality and appearance-based object recognition in computer vision research. The former aims at photorealistic reconstruction of scenes; the latter identifies the existence of an object in a given photo image, as well as its (rough) location.

3.3.1 Modeling-from-reality

Modeling-from-reality is a challenging, but well-studied problem in computer vision. Mainstream techniques have been developed using photo images or range images or a combination of the two. Techniques for photorealistic reconstruction typically use both photo and range images.

In the literature, image-based modeling refers to modeling from multiple photo images. The determination of object geometry from multiple views is not solely in the domain of computer vision; photogrammetry (Mikhail et al., 2001), which dates back to the mid-19th century, also attempts to precisely recover quantitative geometric information from multiple photo images. There are commercial software based on computer vision and photogrammetric technology. ImageModeler (http://usa.autodesk.com/adsk/servlet/index?id=11390028&siteID=123112: accessed May 2009) developed by Autodesk, and PhotoModeler developed by Eco System (http://www.photomodeler.com/index.htm: accessed May 2009) are among the two better known products. ImageModeler (Figure 3-4) relies on marker points specified by the user to calibrate camera position and parameters. Once calibrated, modeling is a manual procedure using polygonal primitives. Likewise, modeling in PhotoModeler (Figure 3-5) is mainly manual, although it can automate many sub-procedures, such as, automated marking and matching.

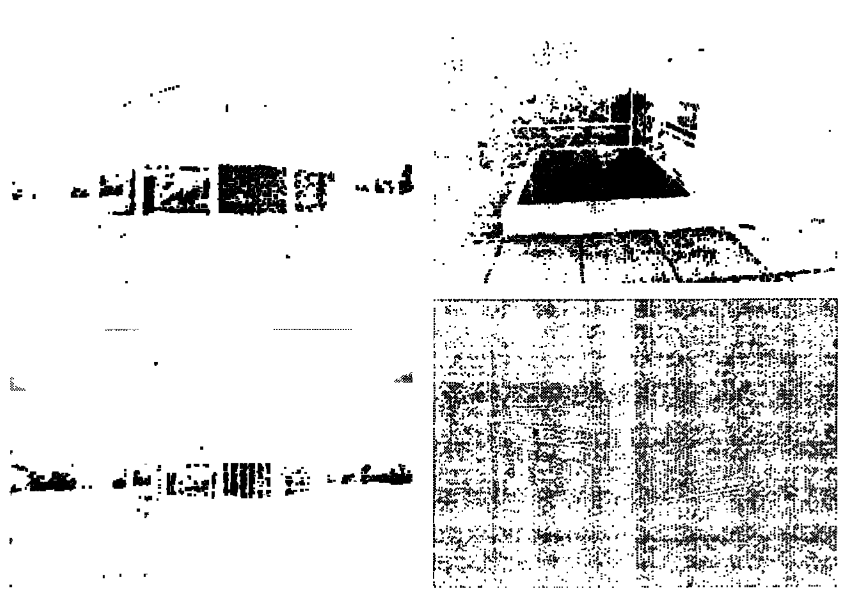


Figure 3-4: An ImageModeler demonstration
(http://usa.autodesk.com/adsk/servlet/index?siteID=123112&id=11983371: accessed May 2009)

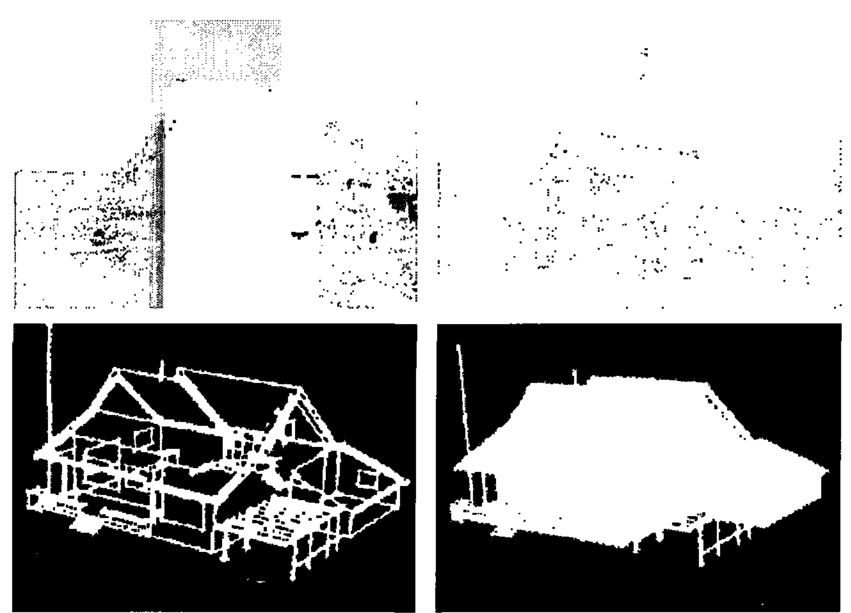
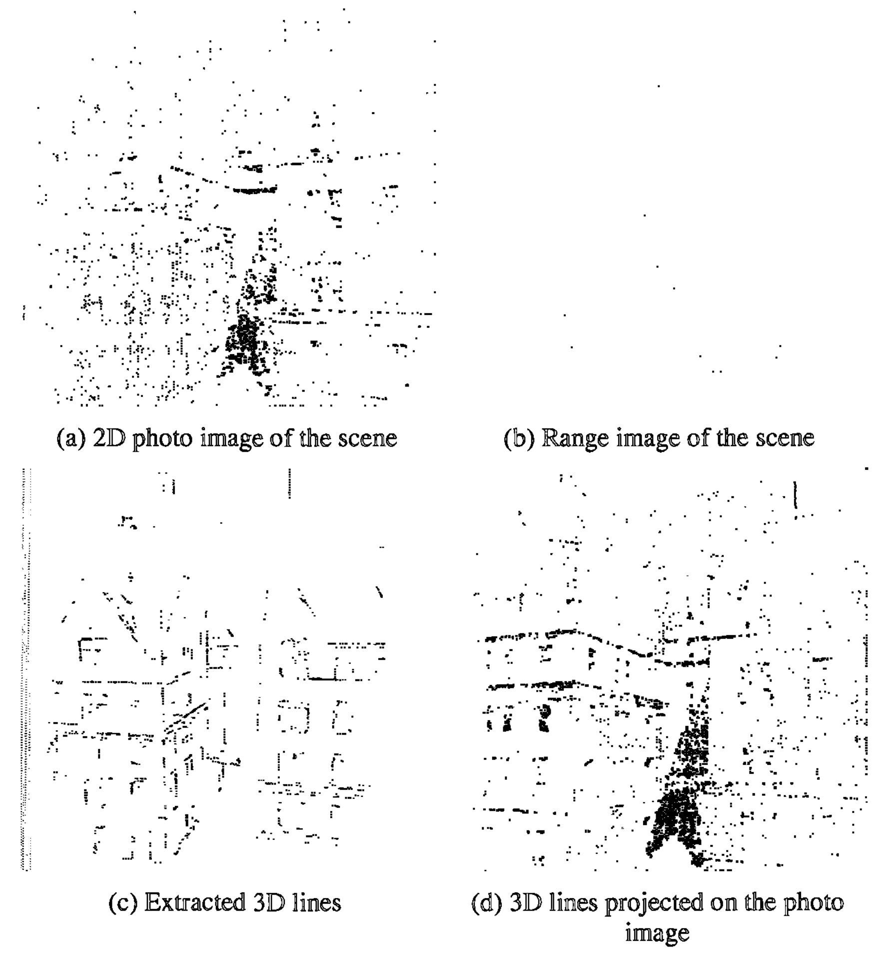


Figure 3-5: A PhotoModeler demonstration
(http://www.photomodeler.com/applications/architecture_and_preservation/examples.htm: accessed May 2009)

There are models that directly use meshes incident with registered range images, for various objects, for example, statues (Levoy et al., 2000), heritage sites (Ikeuchi et al., 2003) and underground mines (Huber and Vandapel, 2006). The basic modeling procedure comprises capturing of range images, image alignment, merging range images into a mesh object, and optionally texture-mapping. As geometry information is given as meshes, this kind of technique does not meet our requirements.

Without a priori knowledge of the type of the objects in a scene, it is generally hard to extract object surfaces from range images. This is because a range image treats an entire scene as an entity; thus, it is difficult to automatically determine which subset of points belongs to an object. Various techniques for geometry extraction have been developed. These fall into two categories: those that segment a point cloud based on such criteria as proximity of points or similarity of locally estimated surface normals, and those that directly estimate surface parameters by clustering and locating maxima in a parameter space. The former obtain the geometry as meshes, the latter, though, more robust, is only used for shapes that are described by a few parameters such as planes or cylinders. For our purpose, these latter methods provide more appropriate geometry. Examples include Faber and Fisher who use knowledge-based architectural models as constraints to build geometric models with the quality of CAD models (Faber and Fisher, 2002), and Vosselman et al. who explore techniques for recognizing objects as planes, cylinders or spheres in industrial plant and urban landscape contexts (Vosselman et al., 2004).



(a) 2D photo image of the scene

(b) Range image of the scene

(c) Extracted 3D lines

(d) 3D lines projected on the photo image

Figure 3-6: Photorealistic scene reconstruction
Adapted from (Stamos and Allen, 2002)

Stamos and Allen develop a systematic approach to the problem of photorealistic 3D model acquisition from a combination of range and photo images (Stamos and Allen, 2002). Their approach utilizes parallel and orthogonal constraints, which abound in urban environments. As a result, this approach works well on urban scenes consisting of conventional buildings. The system takes a set of 3D range images from different viewpoints and a set of 2D photo images of the scene, creating first a 3D solid model, which describes the geometry of the scene, then recovering the positions of the 2D cameras with respect to the 3D model, and finally, photo-realistically rendering the scene by texture-mapping the associated photographs on the model. Figure 3-6 shows results.

3.3.2 Appearance-based object recognition

Appearance-based object recognition can be used to annotate objects. With a co-located model, we can determine the object type of the extracted geometry by detecting the object type to which the corresponding pixels belong.

Appearance-based object recognition is still active research. The earliest work utilized global descriptions such as color or texture histograms. The main drawback to this was sensitivity to real world variability, such as viewpoint and light changes, clutter and occlusion. Global methods have been gradually supplanted by part-based methods in the last decade. Part-based object models combine appearance descriptors of local features with a representation of their spatial relations. While part-based models offer a satisfying way to representing many real-world objects, learning and inference problems for spatial relations remain complex and computationally intensive, especially in a weakly supervised setting where the location of the object in a training image has not been marked by hand. The bag-of-features model (Csurka et al., 2004) has the advantage of simplicity and computational efficiency, though it fails to represent the geometric structure of the object class. Various approaches have been developed to overcome this; examples are SIFT descriptors (Sivic et al., 2005), novel kernels (Zhang et al., 2007), etc. Overall, the current techniques are still far from being capable of recognizing object types of an image on the level of pixels; instead, only the rough position can be determined. This makes it difficult to accurately determine the object types of extracted geometry elements.

3.4 A realistic pipeline

Using commercial software such as ImageModeler and PhotoModeler offers a practical option. Photo images at different angles are input to such software, with a 3D annotated model manually created, and the desired building features then output to XML. This approach is, however, time-consuming, as there is limited automation involved.

On the other hand, the technique, developed by (Stamos and Allen, 2002) in reconstructing photorealistic texted 3D model, can be fully automated. However, for

our purposes, there are potential problems. For instance, there is no way of guaranteeing that the desired geometry information can be extracted; moreover, it is unlikely that the photo-image pixels are correctly annotated. The extracted geometry is typically *loosely* connected—it would be hard to automatically convert such geometry information to annotated objects even if annotations were available. Consequently, it demands manual operations. However, on the motto that there is presently nothing better, a pipeline based on this technique is still preferable to the manual approach; at least, there is some automation involved. Accordingly, one such pipeline is chosen as the *realistic* pipeline (Figure 3-7). Note that the textured 3D model also serves as a co-located model.

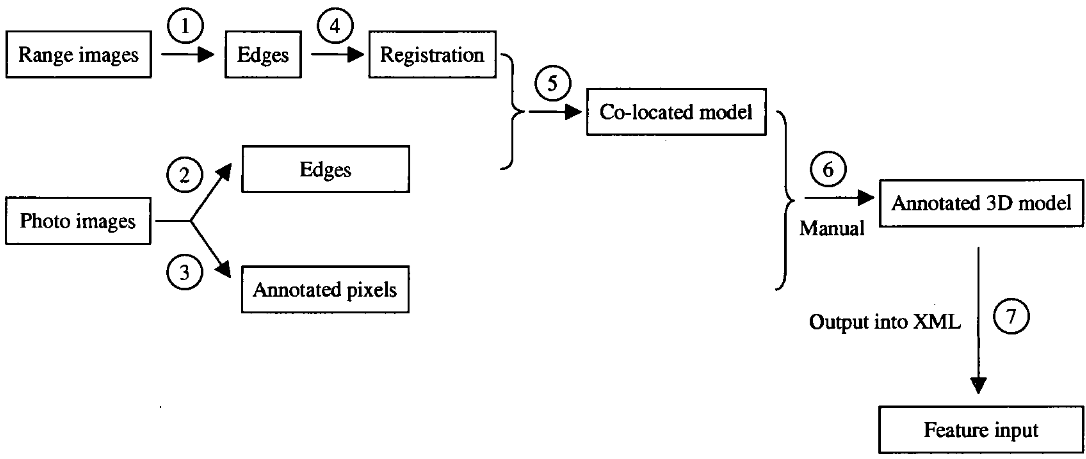


Figure 3-7: A realistic pipeline

3.5 Details of the realistic pipeline

The realistic pipeline is designed to generate building features as automated as possible. The automated part is based mainly on the works of (Stamos, 2001), (Frueh et al., 2004) and (Fulkerson et al., 2008), including i) edge extraction of range images, ii) registration of range images using the extracted edges, iii) edge extraction and annotation of photo images, and iv) creation of a co-located model by aligning photo images to range images. The manual part includes filling the missing geometries and constructing objects with the help of the annotation information. A

system similar to AutoDesk's ImageModeler will greatly assist the manual tasks. Algorithms for the automated part are detailed in the sequel.

3.5.1 Edge extraction of range images

Straight-line edges are fundamental to the geometry of conventional buildings. However, as discussed early, occluding edges in range images are often unreliable due to mixed pixels. This makes direct edge extraction impractical. On the other hand, a fitted plane feature is less sensitive to such noises, as it reflects the votes of a large number of points. Taking advantage of this fact, an edge can, instead, be computed by intersecting its two adjacent planes.

Plane extraction via segmentation

The plane measured by a set of points $\{p_1, \dots p_n\}$ can be computed through matrix A= $\sum_{i=1}^{n}((p_i - p)^T \cdot (p_i - p))$, where p is the centroid of points $\{p_1, \dots p_n\}$ and a point on the plane. The normal of the plane is the smallest eigenvector of A. The smallest eigenvalue d measures the quality of the fit (least squared deviation).

In reality, however, the mapping of a range point and the plane it measures is usually unknown. The process to determine such a mapping is known as *segmentation*. Informally, segmentation is of labelling all the points in a range image, so that points whose measurements are of the same surface are given the same label. For our purpose, the only type of surface concerned is plane.

A number of different segmentation algorithms have been developed (Hoover et al., 1996; Han et al., 2004) and, surprisingly, segmentation remains an active research. As suggested by the experiment conducted by (Hoover et al., 1996), it is hardly to distinguish a perfect one. Here, we examine the one adopted in (Stamos, 2001), which is a process of iteratively region growing by taking advantage of the 8-neighbourhood connectivity incident with a range image.

The laser beams emitted by 3D laser scanners typically follow a fixed pattern, vertically emanating laser beams incremented by a constant angle α, horizontally

rotating a constant angle β, and then vertically emanating another column. Conceptually, the laser beams can be indexed on a 2D rectangular grid (Figure 3-8). This leads to a mathematical representation of a range image as a set $\{r(i, j), i = 1 \ldots N, j = 1 \ldots M\}$, where $r(i, j)$ is a range pixel. Neighborhood connectivity relationships can be easily defined on the 2D grid, for example, 8-neighborhood connectivity. Moreover, the connectivity relationships are the exact same for the corresponding 3D points in space, assuming these points are on the same plane.

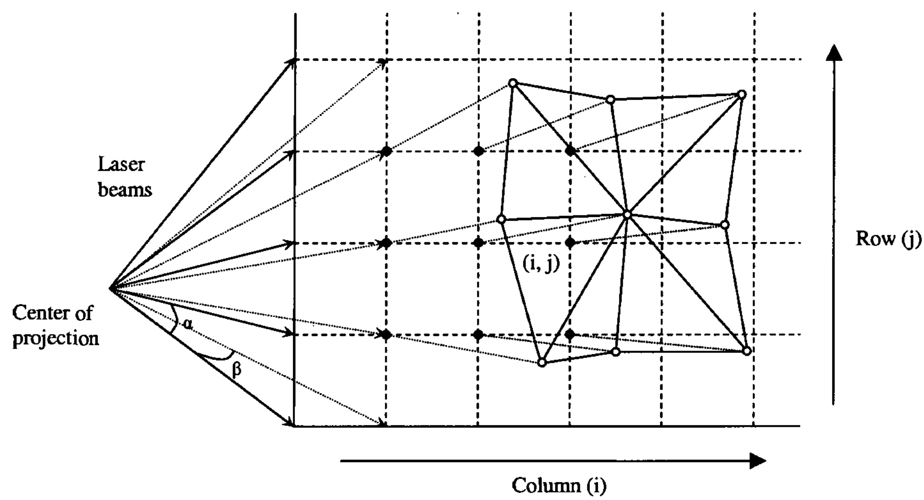


Figure 3-8: Grid pattern of laser beams and 8-neighborhood connectivity
Adapted from (Stamos, 2001)

Formally, segmentation is to divide a range image $\{r(i, j), i = 1 \ldots N, j = 1 \ldots M\}$ into a set of clusters $\{C_{null}, C_1, \ldots, C_n\}$, such that i) $\bigcup \{C_{null}, C_1, \ldots, C_n\}$ is the entire range image, ii) no two clusters overlap, iii) each cluster C_i, $i \geq 1$, contains points of a connected planar region with area larger than $T_{minArea}$, a user-defined threshold representing the minimum area of the desired building components, and, iv) the special cluster C_{null} contains points which cannot be classified to any planar region.

The segmentation algorithm first attempts to fit a local planar patch on the $k \times k$ neighborhood of every 3D point. The fit can be either acceptable or inacceptable,

which is guarded by the smallest eigenvalue d of the corresponding matrix A constructed. The fit is inacceptable when d is larger than a threshold value d_{fit}, meaning that the $k \times k$ neighbor is not on a plane. d_{fit} should be small and on the order of the square of the expected noise level (Siciliano and Khatib, 2008). The point with unacceptable fit is put in the cluster C_{null}.

A list of clusters is initialized, by creating a cluster for each point with acceptable fit. This initial list is iteratively merged by the metrics of conormality and coplanarity. Two adjacent local planar patches, (p_1, n_1) and (p_2, n_2), are considered as *conormal* when the angle $\alpha = \cos^{-1}(n_1 \cdot n_2)$ is smaller than a threshold α_{merge}, and *coplanar* when the distance $dist = \max(|\overline{p_1 p_2} \cdot n_1|, |\overline{p_1 p_2} \cdot n_2|)$ is smaller than a threshold value $dist_{merge}$.

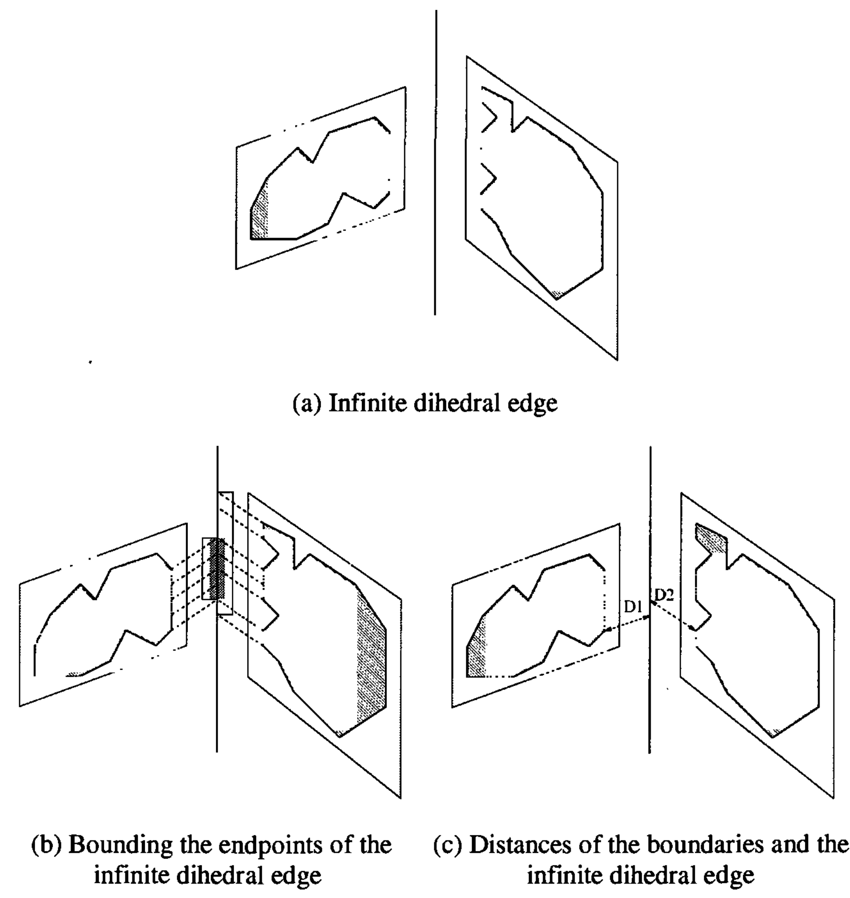
The merging process stops when the list of clusters becomes stable; that is, no two clusters can be further merged. A plane is fitted to each stable cluster. The boundaries of each plane are also extracted. Typically, each plane has an outer boundary and potentially multiple inner boundaries (for example, a wall surface with window openings). A point is on the boundary if at least one of its 8-neighbors is not in the cluster.

The above cluster merging process can be efficiently implemented by visiting each acceptable fit in a raster-scan manner along the range image grid and by maintaining a cluster equivalence table. For acceptable fit of (i, j), the acceptable fits at $(i+1, j)$, $(i, j+1)$, and $(i+1, j+1)$ are tested and labeled. The real merging happens in another run of the raster scan. This implementation has complexity $O(N)$ where N is the total number of range points.

Edge extraction

The parameters of a dihedral edge (Figure 3-9a) with two known adjacent planes, (p_1, n_1) and (p_2, n_2), can be computed in two steps: firstly computing the incident infinite line l, and then determining its endpoints. An infinite line l is specified by a point and its direction vector. The direction vector is given by $n_1 \times n_2$. A solution p_0

of $\begin{bmatrix} n_1^T \\ n_2^T \end{bmatrix} p = \begin{bmatrix} n_1^T p_1 \\ n_2^T p_2 \end{bmatrix}$ is a point on l. The endpoints can be bounded by point projection (Figure 3-9b). The points of the corresponding clusters of the two adjacent planes, whose distances to l are less than a threshold value d_{edge}, are projected on the line. The line segment bounded is the final result.



(a) Infinite dihedral edge

(b) Bounding the endpoints of the infinite dihedral edge

(c) Distances of the boundaries and the infinite dihedral edge

Figure 3-9: Extraction of dihedral edges
Adapted from (Stamos, 2001)

In reality, which two planar regions form a dihedral edge is usually unknown. The following is a process to determine such pairs of planar regions: Pairs of planar regions whose bounding boxes are close to each other (under threshold d_{bound}) are first chosen. The dihedral infinite lines are then computed. To filter out factious candidates, a distance criterion is used. The distance is defined as the 2D minimum distance of a finite line and every 'smaller' edge of the 'used' boundaries of the

adjacent planar regions. This distance can be computed fast by an algorithm described in (Lumelsky, 1985). Pairs, whose have a distance larger than a threshold d_{bound}, are discarded (Figure 3-9c).

Mixed pixels have the potential to influence accuracy of the endpoints. For example, there may be mixed pixels on surface S_1 near the lower-left corner (see Figure 3-10). The boundary used to constrain the endpoint will be influenced by those mixed pixels.

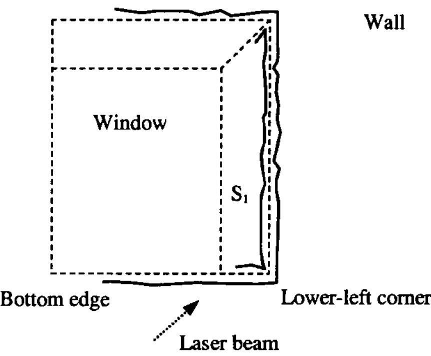


Figure 3-10: Potential endpoint inaccuracy caused by mixed pixels.

3.5.2 Registration of range images

Essentially, registration of range images involves computing the transformation matrix, including a rotation matrix and a translation vector, between two range images. The extracted dihedral edges can be employed to determine the transformation matrix. There are two options to constructing the equations: either using three pairs of non-parallel infinite lines incident with edges, or using three pairs of edge endpoints.

Between these two options, we may be tempted to use infinite lines, since mixed pixels potentially impact the accuracy of the endpoints. However, it may be impossible to find such pairs due to the way edges are extracted. For example, as shown in Figure 3-11, all the vertical dihedral edges in Scan1 is either occluded or become non-dihedral in Scan2, with only horizontal edges left as candidates. As a

result, in general, edge endpoints have to be employed when computing the transformation matrix.

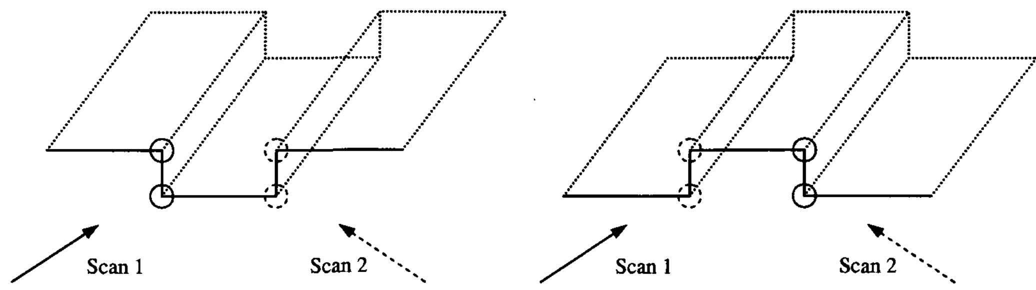


Figure 3-11: Difficulty of registration by infinite lines

Following the above argument, a RANSAC algorithm on three pairs of edge endpoints can be used to determine the transformation matrix between two range images, with a rating equation similar to the one to be described in Section 3.5.4—a candidate transformation matrix is determined by the randomly selected three pairs of edge endpoints, the extracted edges in one range image corresponds to the 2D line segments found in the image, the extracted edges of another under the candidate transformation matrix corresponds to the projection of 3D line segments, and the desired transformation matrix is found when the rating equation achieve a maximum value.

3.5.3 Edge extraction and annotation of photo images

Edges of photo images are required in creating a co-located model. The following describes the algorithm to extract edges:

1) Apply Canny edge detector (Canny, 1986) with *hysteresis thresholding*. The results are chains of 2D edges, one pixel each in size. Figure 3-12 shows the edges obtained by applying the Canny edge detector implemented by Tom Gibara (http://www.tomgibara.com/computer-vision/canny-edge-detector: accessed July 2009) on the picture of Figure 1-5.

2) Segment each chain into linear parts. Here, an algorithm similar to the one described in (Lowe, 1987) is adopted. Each chain is recursively subdivided at the point with maximum deviation from a line connecting its endpoints (Figure 3-13a, b). This process is repeated until each sub-chain is no greater than a threshold l_{min} in length, or the least square deviation of its points is no greater than a threshold dev_{max}.
3) Fit and compute the parameters of the line segment for each sub-chain (Figure 3-13c). Merge two collinear line segments if the minimum distance of endpoints is less than a threshold value d_{min}.

Figure 3-12: Edges detected from the picture in Figure 1-5 using a Canny edge detector

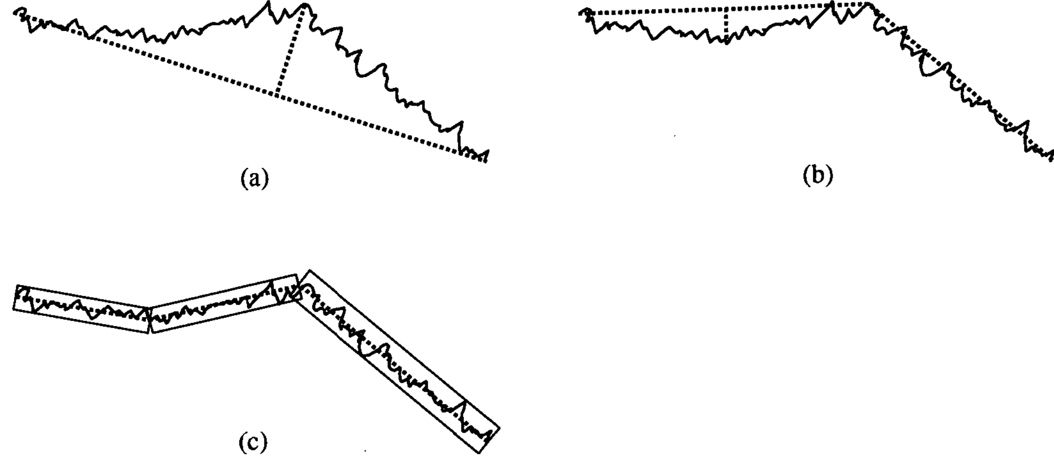


Figure 3-13: Subdividing a chain recursively, and extracting its straight-line segments
Adapted from (Lowe, 1987)

Automatic annotation of photo images is still active research. In the realistic pipeline, the results returned by annotation just serve as hints. The approach developed in (Fulkerson et al., 2008) is adopted. It uses bag-of-features as a 'black box' to perform pixel-level category recognition. There are three key developments making this brute-force way feasible: reducing the size of a large generic dictionary to low hundreds; extending the concept of integral images to the computation of local histograms; and computing SIFT descriptor densely in linear time. See (Fulkerson et al., 2008) for further details.

3.5.4 Creating a co-located model

The creation of a co-located model is to align each photo image with the range image model so that the particular pixel corresponding to a 3D point in space can be determined. Essentially, this is the problem of camera pose determination; that is, determining the camera projection matrix.

Ignoring lens distortion, the general camera projection matrix M is a 3×4 matrix with 6 extrinsic and 5 intrinsic parameters. When three exact-matching pairs of 3D and 2D straight-line segments are known, all the 11 unknown parameters can be recovered by minimizing $|Lm|^2 = m^T L^T Lm$, subject to $|m| = 1$, where

$$L = \begin{bmatrix} P_1^T & 0 & -u_1 P_1^T \\ 0 & P_1^T & -v_1 P_1^T \\ \vdots & \vdots & \vdots \\ P_N^T & 0 & -u_N P_N^T \\ 0 & P_N^T & -v_N P_N^T \end{bmatrix}.$$

In practice, however, exact matching is seldom known. On the other hand, it is reasonable to assume that the 3D and 2D images significantly overlap. If an initial pose estimate were known, we could rate a given pose by projecting the 3D lines onto the target image and comparing against the 2D lines extracted from the image. This rating is a non-linear minimizing process.

Here we adopt a rating equation based on slope and proximity, which is proposed in (Frueh et al., 2004):

$$Q_{pose} = \sum_{i=0}^{M} \sum_{j=0}^{N} \| l_i \| \cdot S(l_i, L_j) \cdot D(l_i, L_j).$$

l_i is the ith 2D line segment found in the image. L_j is the 2D projection of the jth 3D line segment. $||l_i||$ is the length of the ith 2D line segment found in the image.

$S(l_i, L_j)$ is a function of the slopes of lines l_i and L_j such that

$$S(l_i, L_j) = \begin{cases} 0 & for \ \langle l_i, L_j \rangle < S_{max} \\ \langle l_i, L_j \rangle & for \ \langle l_i, L_j \rangle \geq S_{max} \end{cases},$$

where $\langle l_i, L_j \rangle$ is the dot product of the normals of l_i and L_j; S_{max} is a threshold value.

$D(l_i, L_j)$ is a function of the proximity of lines l_i and L_j such that

$$D(l_i, L_j) = \begin{cases} 0 & for \ d(l_i, L_j) < D_{max} \\ \dfrac{D_{max} - d(l_i, L_j)}{D_{max}} & for \ d(l_i, L_j) \geq D_{max} \end{cases},$$

where $d(l_i, L_j)$ is the sum of the minimum distance of the endpoints of lines l_i and L_j; D_{max} is a threshold value.

The rating equation is designed to achieve a maximum value when both slope and position of the projected 3D model lines best match the 2D image lines. To compute exact camera pose, Q_{pose} needs to be optimized over the 11 dimensional parameter space.

Initial camera pose estimation

The initial camera pose can be obtained from GPS (Global Position System) or INS (Inertia Measurement System). This needs the relative position of the range sensor and the camera sensor to be rigidly fixed.

The algorithm designed by (Stamos, 2001) removes the above restriction. In particular, it can estimate the rotation, translation, focal length, and principle points of a camera. Note that for a general camera project matrix, 90° is a good initial estimate for the camera axis skew, and 1 for the axes scales between world and image coordinates. Putting this all together, the initial camera position estimate is complete. The following describes Stamos's algorithm in details:

Let F_{3D} and F_{2D} be the 3D and 2D line segments extracted from the range and image data.

1) *Group the 3D and 2D line segments into clusters of parallel 3D lines* L_{3D} *and converging 2D lines* L_{2D} *(intersecting at vanishing points).*

   In a building environment, the number of vanishing points is almost always 3 (may be at infinity). Vanishing points can be determined, by first computing all pair-wise intersections of F_{2D}, then constructing a 2D histogram of the intersection points, and lastly, extracting the first three peak values. With vanishing points, F_{2D} can be clustered into $L_{2D} = \{L_{2D1}, L_{2D2}, L_{2D3}\}$ by point-on-line testing. Figure 3-15b shows a sample result. Note that constructing such a 2D histogram can be a subtle task, for example a vanishing point can be at infinity. Related issues have been widely examined in computer vision research, with common solutions employing a *Gaussian sphere* or applying

the *Hough transformation.* See (Shufelt, 1999; Rother, 2002) for further details.

An iterative merging process clusters the 3D lines. Initially, each 3D line defines its own cluster. Based on similarity between the average angles of the lines contained in a cluster, the two most similar clusters are merged. This process is repeated until only three clusters are left. At the end, F_{3D} are clustered into three groups of parallel lines $L_{3D} = \{L_{3D1}, L_{3D2}, L_{3D3}\}$, along with the average direction of each cluster $U_{3D} = \{U_{3D1}, U_{3D2}, U_{3D3}\}$. Figure 3-15a shows a sample result.

2) *Estimate the focal length, principle point, and rotation matrix.*

As a fact, vanishing points and the center of projection of the camera form a proper tetrahedron if and only if the three scene line directions are orthogonal with respect to each other (Becker, 1997). Thus three vanishing points computed previously can be used to calculate the center of projection of the camera, whence the focal length and principle point (Figure 3-14).

Knowing the center of the projection *CoP*, the line segments of connecting CoP and three vanishing points are the directions of the three orthogonal axes in the camera coordinate system. Together with the three orthogonal axes directions in the coordinate system of 3D range images, the rotation matrix can be computed (Figure 3-14).

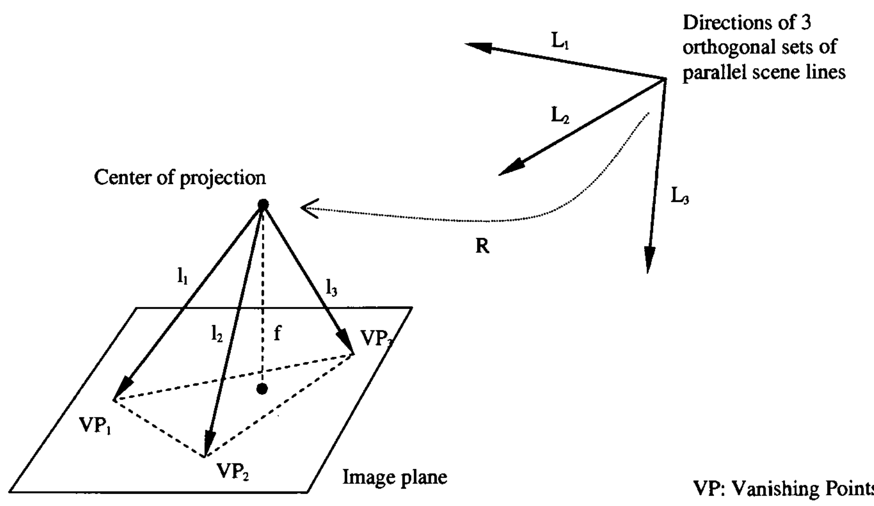
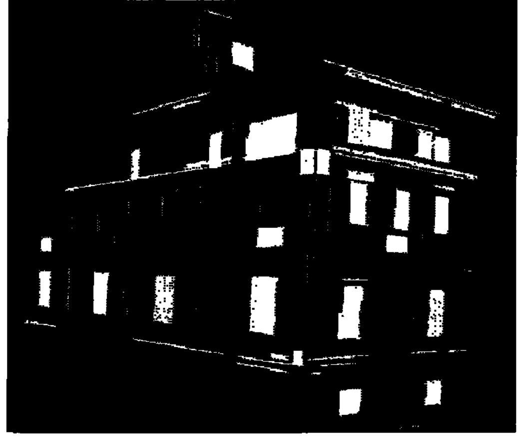


Figure 3-14: Calculating the projection center by using three vanishing points
Adapted from (Stamos, 2001)

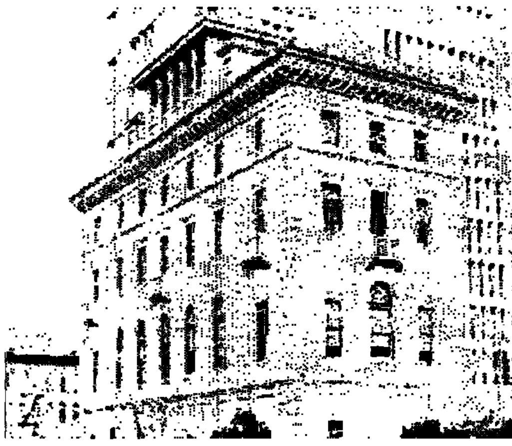
3) *Estimate the translation.*

Theoretically, executing the RANSAC algorithm directly on the extracted edge features can compute translations, but the search space will be combinatorially explosive. To reduce search space and increase rate of valid matches, 3D and 2D lines can be further grouped into 3D and 2D rectangle-shapes. To extract 2D rectangles from photo images, quadrangles are first found, and then the perspective effect is cancelled by projecting back into the world coordinate systems. Extraction of 3D rectangles involves checking for the coplanarity of the border line segments. See (Stamos, 2001) for details of the grouping procedure. After grouping, the RANSAC algorithm can run on these 'high-level' rectangle features to compute translation. Figure 3-15a shows the 3D rectangles extracted from a range image, Figure 3-15c shows the 2D rectangles extracted from a photo image, and Figure 3-15d shows the match of some 3D and 2D rectangles.

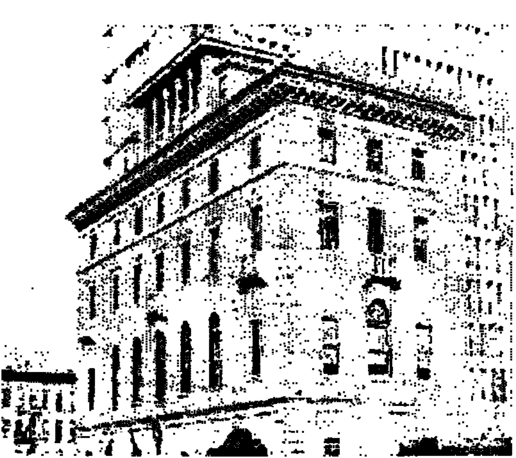


(a) Clusters of 3D lines (color encodes different directions) and extracted 3D rectangles (rendered as solids of different colors)

(b) A photo image and clusters of 2D lines extracted (color encodes different directions)



(c) Extracted rectangles in the photo image



(d) Extracted rectangles in the photo image and matched 3D rectangles projected on the photo image under the estimated camera pose

Figure 3-15: Results of initial camera pose estimation
Adapted from (Stamos, 2001)

3.6 Remarks

Although significant progress has been made in computer vision research, there is still a noticeable gap between the ideal and reality. Commercial software developed speed up manual generation of building feature input. State-of-art research partially automates manual tasks, while leaving much to be desired.

Existing approaches on scene reconstruction are still mainly purely geometry-based, with little concern to the specific type of the underlying object, without

knowledge of which, makes extraction of complete geometry extremely difficult. To know the object type, it is necessary to improve current object recognition algorithms so that object types are accurately identified. This seems to be a kind of chicken-and-egg situation. However, this cycle can be broken, with some promise, by using a combination of range and photo images, the former primarily for geometry and the latter for annotation.

Chapter 4 Computation-friendly shape grammars

It was shown in Section 2.3.5 that any Turing machine can be simulated by a shape grammar, and that, as a computing model, shape grammars are at least as powerful as Turing machines. Moreover, shape grammars share some of the same computing difficulties as Turing machines, including that a shape grammar may not halt, the language space of a shape grammar may be exponentially large, and the membership problem is unsolvable in general. These provide formal evidence for the awareness that implementing a shape grammar interpreter is difficult, in particular for parametric shape grammars (Gips, 1999; Chau et al., 2004); shape grammars are subject to ambiguity, combinatorial explosion and indefinite emergent possibilities, which make certain computation intractable (Chase, 1997).

On the other hand, it is known that in practice, many shape grammars are implementable. As a result, it would be advantageous to examine the *tractability* of shape grammars in a way that determinant factors can be identified and whence, serve as a guide to avoiding those intractable cases. Formally, problems are deemed *tractable* whenever there is a polynomial algorithm, and *intractable* when they require super-polynomial time.

The approach used to trace down the determinant factors of the tractability of shape grammars is based on the observation that interpreting a shape grammar is through the application of its shape rules. A shape grammar contains a finite set of shape rules. This fact underlies the fundamental basis for a shape grammar, namely, that of using a small number of shape rules to realize many, potential infinitely many, design possibilities (Stiny, 2006). According to the number of steps of applying shape rules at the time of termination, shape grammars can be divided into two categories: those need super-polynomial steps or even infinitely many, and those need polynomial steps. The former becomes intractable automatically. For the latter, the tractability is actually determined by the application of each shape rule.

In this chapter, factors influencing tractability are identified by exploring the formalism for shape rule application. The existence of both tractable and intractable shape rules, together with other computation difficulties mentioned at the beginning of this chapter, negates the possibility of a single general shape grammar interpreter. Instead, I propose a paradigm for a practical, 'general' interpretation of shape grammars, comprising a collection of sub-interpreters, one for each class of tractable shape grammars. Then, an optimal way to classifying shape grammars is discussed, with the conclusion that classification is reliant upon the underlying data structure.

There are other, mainly external, factors that influence computability of extant shape grammars, which can be resolved by recasting—in essence, codifying—their shape rules. That is, pragmatically, the paradigm is augmented to comprise a set of sub-frameworks; each, in actuality, is an application programming interface (API) built on top of an underlying data structure, a basic set of manipulation algorithms, and a meta-language.

4.1 Parametric subshape recognition

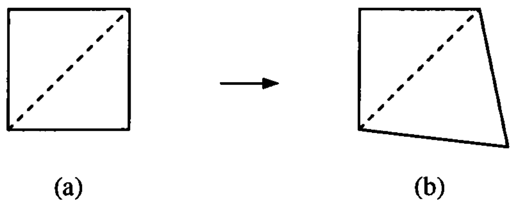
Parametric subshape recognition, a central step to the application of any parametric shape grammars, is known to be difficult. In this section, I will use the problem of parametric subshape recognition of two-dimensional rectilinear shape to show the

potential computational difficulty. The discussion is based on the shape grammar definition of *SG-DEF-1980* (see Section 2.3.3 for details).

For non-parametric subshape recognition of two-dimensional rectilinear shapes, the transformation t can be determined by matching three distinguishable points of A to three distinguishable points of C (Krishnamurti, 1981). However, for parametric subshape recognition, this is not necessarily the case.

It is possible that a parametric shape A has a certain number of fixed points (non-open terms). If there are more than three fixed points (distinguishable by definition), the above 3-point algorithm is still applicable, with $\binom{n}{3}$ possibilities to test against.

For shapes with 1 or 2 fixed points, this is identical to the situation when all points are open as similarity is subsumed by the assignment. When all points are open, the shape transformation may not be describable by a homogeneous transformation matrix. For example, Figure 4-1a matches Figure 4-1b under a parametric shape rule, but there is no 3×3 homogeneous matrix which describes the transformation. As a result, open terms have to be determined, point-by-point, for each candidate subshape in C.



(a) (b)

Figure 4-1: Example of parametric subshape matching

In general, when there are k open terms, there are $\binom{n}{k}$ possibilities. Even assuming that testing against each possibility costs unit time (typically, this is much

more expensive in reality), when k is close to $n/2$, the time complexity is a high-degree polynomial or even super-polynomial. To give a concrete example, the possible number of tests is 7.5×10^7 when $k=5$, $n=100$; 1.7×10^{13} when $k=10$, $n=100$; and 1.0×10^{29} when $k=50$, $n=100$. It takes a computer, with performance of thousands of millions of instructions per second, several minutes to test all the possibilities when $k=10$, $n=100$. Note that when k is more than $n/2$, the number of possible tests begins to decrease.

4.2 Determination of factors influencing tractability

The analysis of factors influencing tractability is preferred to be general; that is, it is applicable to different kinds of shape grammars. For such an analysis, it is necessary to have a general definition of shape grammars, which covers all known types of shape grammars and, preferably, the new types in the future.

4.2.1 A unified definition of shape grammars

As reviewed in Section 2.3.3, the development of shape grammars is evolutionary, while the basic formalism remains the same. The only factor actively evolving is the scope of the allowable basic shape elements as well as various augmentations.

Initially, the focus is two-dimensional shapes, which are made out of a finite straight lines in terms of maximal representation (Stiny, 1980a). Already, in the kindergarten grammar (Stiny, 1980b), the basic shapes were essentially three-dimensional rectilinear solids, albeit drawn as line shapes. In another paper, Stiny states that shapes made up of points, lines, planes, or solids provide the main objects for shape grammars (Stiny, 1991). Krishnamurti examined shape arithmetic for shapes made up of finite planes (Krishnamurti, 1992a) and considered subshape recognition for three-dimensional shapes under linear transformations (Krishnamurti and Earl, 1992). Along with Stouffs, he extended the arithmetic to higher-dimensional shape algebras (Krishnamurti and Stouffs, 2004), described algorithms for three-dimensional shape arithmetic and analyzed their computational complexity (Stouffs and Krishnamurti, 2006) and considered subshape recognition over the Cartesian products of differently dimensioned shapes (Krishnamurti and Stouffs,

1997). A three-dimensional shape grammar implementation based on a commercial solid modeling kernel is described by (Piazzalunga and Fitzhorn, 1998), and grammars over curves have been considered by several authors (Chau, 2002; McCormack and Cagan, 2003; Jowers et al., 2004; Prats et al., 2004).

Geometric shapes can be augmented by symbols, numbers, attributes, in general, weights (Stiny, 1992), in this way, connecting shapes of various kinds (Stiny, 1991). Shapes so augmented can be further extended, by using the schemata so that a family of such shapes can be defined. A shape schema $A(x)$ is a finite but possibly empty set of variables, for example, the coordinates of points, that describes a family of shapes. If x is empty, then a shape is given automatically. Otherwise, a shape can be 'fixed' by using a function g to assigns values to variables x as $g[A(x)]$. Shape grammar with schemata is historically called *parametric shape grammars*. To distinguish, shape grammars without schemata is called *non-parametric shape grammars*.

Traditionally, the non-parametric shape grammars formalism is defined as: for a shape rule $A \rightarrow B$ and a configuration C, if $t(A) \leq C$, then the result of applying the shape rule on C is $[C - t(A)] + t(B)$, where t is a transformation of similarity, $\leq$ is a part relation, $-$ is the operation of Boolean difference, and $+$ is the operation of Boolean sum. Note that, the operations of Boolean sum and difference implicitly involve an operation of reduction R (See *SG-DEF-1975* of Section 2.3.3), which is used to maintain the maximal representation (Krishnamurti, 1992b).

For parametric shape grammars, the formalism is defined as: for a shape rule schema $A(x) \rightarrow B(x)$ and a configuration C, if $t[g(A(x))] \leq C$, then the result of applying the shape rule on C is $[C - t[g(A(x))]] + t[g(B(x))]$, where g is a function which makes an assignment to the open terms (aka. variables) of the schema.

Since t can be generalized as a being-alike function (Stiny, 1991), the function g can be combined with, thus subsumed by, t to form a new being-alike function. In this way, the formalisms of non-parametric and parametric shape grammars are unified. By explicitly pulling the reduction operation R as the last step of applying a shape rule, the general formalism for all shape grammars becomes:

For a shape rule $A \rightarrow B$ and a configuration C, if $t(A) \leq C$, then the result of applying the shape rule on C is $R[[C - t(A)] + t(B)]$, where t is a being-alike function, $\leq$ is a part relation, $-$ is the operation of Boolean difference, $+$ is the operation of Boolean sum, and R is the reduction operation to maintain a maximal representation.

In the following discussion, the above definition is used as the general definition of shape grammars. Following this definition, the scope of basic shape elements can be extended arbitrarily, such as lines, curves, labels, weights, whatsoever; the bottom line is that all operators of t, $\leq$, $-$, $+$, and R are well defined. In particular, elements are implicitly typed in a way that operators of $\leq$, $-$, $+$, and R only operates on two elements of the same type (that is, co-equal); For example, for two line segments, these operators are only meaningful when two has the same slope.

4.2.2 Factors influencing tractability

For those halting shape grammars with a polynomial language space, there are at most polynomial steps of shape rule application. For such shape grammars, the tractability is actually determined by the application of each shape rule. By definition (of shape grammars), application of a shape rule involves the operations of t, $-$, $+$, $\leq$, and R on elementary objects. If any of these operations takes super-polynomial time or even undecidable, shape rule application becomes intractable. In everyday design practice, the computational complexity of these operations may seem trivial, as these operations are not so difficult for rectilinear shapes. As concluded in (Stouffs and Krishnamurti, 1993; Stouffs and Krishnamurti, 2006), the asymptotic upper bounds of comparing two co-equal spatial elements, the fundamental operations of $-$ and $+$, with maximum boundary size n is polynomial for elements in a d-dimensional space, $0 \leq d \leq 3$. In particular, for $d = 0$ and 1, the upper bound is a constant; for $d = 2$, it is $\Theta((m+n) \log n)$, with $m = O(n^2)$; and for $d = 3$, $\Theta((Km + kn) \log n)$, with $K = O(k)$, $k = O(n)$ and $m = O(n^2)$.

However, for certain kinds of shape objects, some of these operations can be difficult, even intractable. An example is the Boolean operation on two solids with

rational curved surfaces, which involves finding the intersection of two rational surfaces. The intersection of two smooth surfaces is one of the following: i) empty; ii) a collection of points; iii) a collection of smooth curves; iv) a collection of smooth surfaces; or, v) any combination of ii), iii), and iv) (Barnhill et al., 1987). Traditionally, analytical approaches by variable elimination have been the means to solving this kind of intersection problem. However, the degree of the resulting polynomial can be too high to solve. For instance, two generic bicubic patches can intersect in a curve of degree 324. Moreover, it has been shown that the intersection curves cannot be exactly represented by parametric equations even of degree 324 (Katz and Sederberg, 1988). Therefore, numerical methods have to be used and only curves under certain approximations are obtained. Although surface-to-surface intersection is still an active area of research (Patrikalakis et al., 2004; Hur et al., 2009), as highlighted in the book, *Numerical Recipes: The Art of Scientific Computing*, by (Press et al., 2007), there are no good, general solvers for solving systems of multivariate polynomial equations — the equivalent problem to surface-to-surface intersection.

The implication of this is that one cannot arbitrarily expand the scope of shapes. Basic operations of certain shape elements can become so complicated as to make them intractable. As a guideline, in order to design tractable shape grammars, the basic operations of the allowable elementary objects are required to be in a polynomial time. In the following discussion, we assume that this is the case.

The application of a shape rule $A \rightarrow B$ to shape C involves two steps: searching the configuration C for applicable regions according to the left-hand side A, and rewriting the configuration with the right-hand side B. Rewriting a configuration involves two steps: subtracting $(-)$ the left-hand shape under a known t, and adding $(+)$ the right-hand shape under the same t. By our previous assumption, the operations of $-$, $+$, t, and R for each allowable elementary objects are in polynomial time. As there are a fixed number of elementary objects involved, the overall time complexity of rewriting still has an upper bound in polynomial time. It should be noted that the algorithm here is brute force, given simply for the purpose of deriving

a polynomial upper bound—seeking efficient, uniform algorithms for rewriting is still a valid research (Stouffs, 1994; Jowers, 2006). In our discussion, however, rewriting is 'easier', in the sense that there is always a brute-force polynomial algorithm.

On the other hand, searching a configuration for possible rule applications can be much 'harder'. In effect, the searching procedure includes two steps: using certain criteria to identify possible matching candidates, and then verifying the exact matching of each candidate under all allowable *t*. Even with the optimal searching criteria, the number of matching candidates can be high-degree polynomial.

In the verification step for exact matching of a candidate, it is possible that there are infinitely many *t*'s, which are impossible to compute in finite time. For example, for the candidate shape found in Figure 4-2 (marked with a dashed circle), the possible transformations, up to scale, are infinitely many. This phenomenon is known, in the literature, as *indeterminacy*, and viewed as an advantage where unexpected variations can be introduced (Stiny, 1991).

Figure 4-2: A candidate with infinitely many matching transformations

However, it is hard for a computer implementation to appreciate this advantage. The basic question then is: which is the best way to choose one or a subset of possible candidates from the infinitely many? Random choice provides an answer, but relying upon randomness to create novel designs is probably not always a good idea. Manual selection is another option, although this is counter to the goal of a computer implementation. What is for certain is that it is impossible to elaborate all

the infinitely many possibilities; we have to assume that the grammar designer specifies a way of selecting a finite subset so that the implementation is tractable.

To conclude, there are three factors which influence the tractability of a shape grammar: i) the complexity of basic operations of t, $-$, $+$, $\leq$, and R; ii) the number of matching candidates; iii) indeterminacy—the number of possible t's for each matching candidates.

Factor i) is the most controllable in terms of computer implementation; the system only supports basic elementary objects, for which there are efficient algorithms (at most a polynomial time) for these operations.

Factor ii) probably only occurs when the number of open terms is large. In practice, the number of open terms is usually a small number.

Factor iii) is somewhat controllable. As stated in (Stiny, 1991), the detailed conditions for indeterminacy are more complicated and vary from algebra to algebra and from dimension to dimension; Cartesian products are recommended as a useful way to avoid indeterminacy in general.

In practice, chances for indeterminacy are much less. Shape grammars are seldom designed based on purely geometry—typically, they are imbued with semantics in the form of labeled points or elements. The semantics are usually so rich to permit only a limited number of possible transformations. For instance, in Chapter 5, I consider graph-like structure designed to present the layouts of conventional buildings (Figure 5-1, Figure 5-2), the allowable t's belong to one of only eight possible cases (Figure 5-3).

As another example, consider matching a convex quadrilateral to another, as shown in Figure 4-3, there are only 8 possibilities in total. There are 4 possible positions for point 1, viz., p_1, p_2, p_3 or p_4; once it is fixed, there are only two positions for point 2, either the previous point or adjacent to the new position of point 1; after fixing point 2, then the entire shape is fixed.

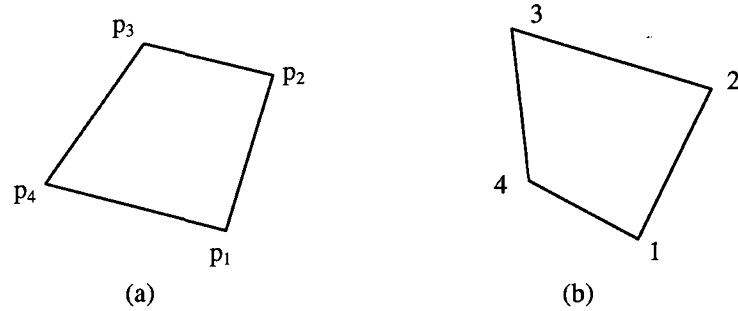


Figure 4-3: Parametrically matching a convex quadrilateral to another

4.3 A paradigm for a practical, 'general' interpreter

The existence of intractable shape grammars implies that algorithms, which deal with parametric shapes, fall into two categories. The first category handles special shapes; the second category is more general, which is only practical for shapes of small sizes. The implication for practice is that the best we can do is to design and implement a parametric shape grammar interpreter, which is capable of handling a subset of grammars.

Even for tractable shape grammars, their characteristics vary significantly. One possible explanation for this is that the shape grammar formalism is so rich that it covers a wide spectrum of designs, which stem from different disciplines. The rich variety is shown indirectly by a number of well-known categories such as subshape-driven vs. marker-driven, non-parametric vs. parametric, rectilinear vs. curvilinear, etc. The variety can also be observed from details of the basic operations of t, $-$, $+$, $\leq$, and R; their diversity forms different algebras, for example U_0, U_1, U_2 and U_3 (Stiny, 1991), on top of which shape grammars can be further defined. There is also, in the literature, an examination of variety by the criteria of different kinds of restrictions on rule format and ordering; in (Knight, 1999), six types of different shape grammars are distinguished, namely, basic grammar, nondeterministic grammar, sequential grammar, additive grammar, deterministic grammar, and unrestricted grammar.

The varieties are tangibly noticeable even when we focus on grammars for a specific subset of the tractable grammars, for example, 2D, rectilinear (with limited curved) shapes. The following are three such examples:

The Baltimore Rowhouse grammar (see Chapter 6 for details) is an example of a shape grammar that captures a specific building style (see Figure 6-8 for all the shape rules and Figure 6-9 for a sample derivation). Other examples in this type include the Queen Anne (Flemming, 1987) and Frank Lloyd Wright's Prairie House grammars (Koning and Eizenberg, 1981). These are parametric shape grammars, in which shape rule application does not depend on emergent shapes. Markers drive shape rule application, and configurations are rectangular or can be approximated as such. Moreover, parameterization is often limited to the height or width of a room, or to the ratio of a room split. The central manipulation unit is a room (or space). Shape rules typically relate to adding a room, to splitting a room, or to refinements such as adding windows, doors, etc.

Figure 4-4 shows rules and a sample derivation of a stylized sports figure grammar (Carlson et al., 1991), which is an example of a *structure grammar*, an augmented variation of a formalism known as a *set grammar* (Stiny, 1982). The kindergarten grammar is another example in this type (Stiny, 1980b). These grammars treat designs as symbolic objects; designs are enforced to be an element of the sets from which they are formed. Thus, the integrity of the compositional units in designs is preserved, as these parts cannot be recombined and decomposed in different ways. This is in contrast to those grammars, where shape elements are decomposed and recombined freely so that new shapes can emerge, for example, the grammar in Figure 4-5. We are essentially manipulating symbols in a 2D space, thus making these grammars amenable to computer implementation. The resulting shapes are simply replacements of internal symbols that occur at the final stage, for the purpose of visualization.

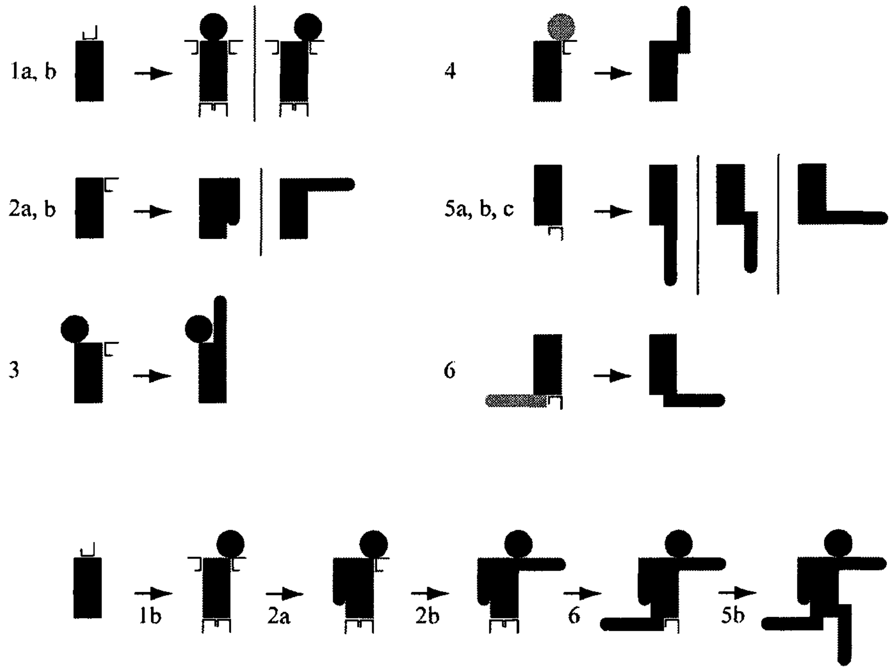


Figure 4-4: Rules and a derivation of the stylized sports figure grammar
Adapted from (Carlson et al., 1991)

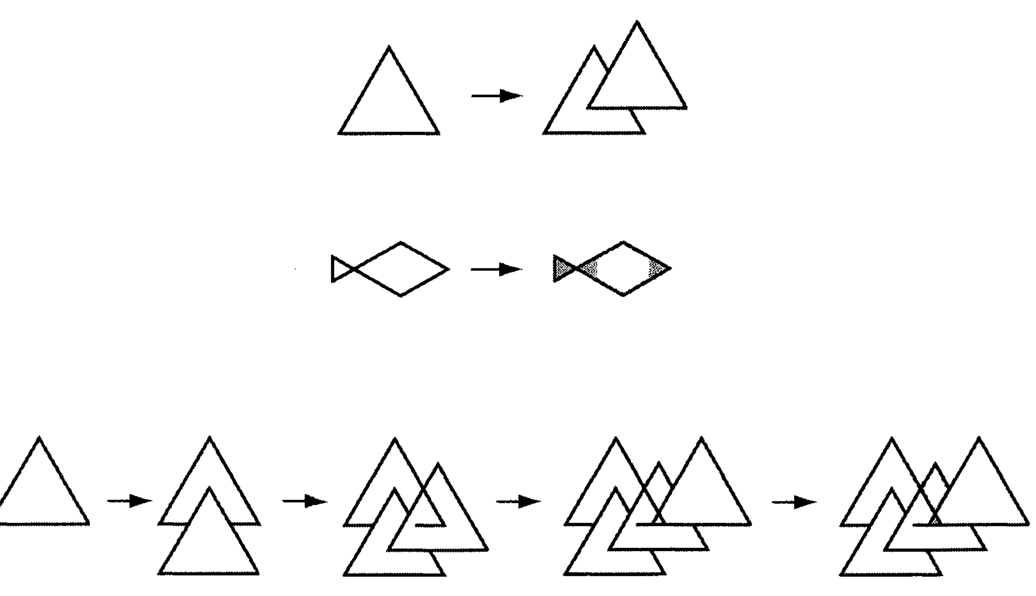


Figure 4-5: Rules and a derivation of the emergent-fish grammar
Adapted from (Knight and Stiny, 2001)

Figure 4-5 shows rules and a sample derivation of a grammar, where a fish-shape emerges during the application of shape rules. This is example of those shape

grammars defined in (Stiny, 1980a), where shape elements are free to be decomposed and recombined so that shape emergence is an important feature during shape rule application. Computation here is non-classical (Knight and Stiny, 2001), which may or may not cause difficulty for computer implementation, depending upon the factors listed in Section 4.2.2.

Table 4-1 shows a comparison of several characteristics of the three grammar examples discussed above on concerns, which are of importance to computer implementation. The variety in this table shows that, even for tractable shape grammars, it is still difficult to come up with the design of a single uniform interpreter.

Table 4-1: Comparison of characteristics important for computer implementation

Grammar	Driver	Emergence	Manipulation unit	Parametric	Context
Rowhouse	Marker	No	Room	Yes	Sensitive
Sports figure	Marker	No	Symbol	No	Sensitive
Emergent-fish	Subshape	Yes	Shape element	No	Free

In contrast, more often than not, it is relatively straightforward to implement an interpreter for a special class of shape grammars, for example, grammars that capture building styles. As we cannot handle intractable shape grammars, why not focus on dealing with as many tractable shape grammars as possible, employing a concept, in spirit, comparable or similar to approximation algorithms (Cormen et al., 2004). Following this idea, I propose a paradigm for practical, 'general' parametric shape grammar interpreters, as shown in Figure 4-6. The paradigm is comprises a set of sub-interpreters, each for a class of tractable shape grammars. In this way, collectively, most parametric shape grammars can be covered.

This paradigm is a perfect subject for applying techniques of object-oriented design, in particular, modularity, polymorphism and inheritance (Grady et al., 2007). The top-level formalism of shape grammars can be implemented as abstract classes

and methods, which are materialized in the sub-interpreter for each class. The shared functionalities—for example, interfaces, can be implemented as part of the top-level infrastructure such that developers for the subclass interpreters are free from unnecessary redundant work.

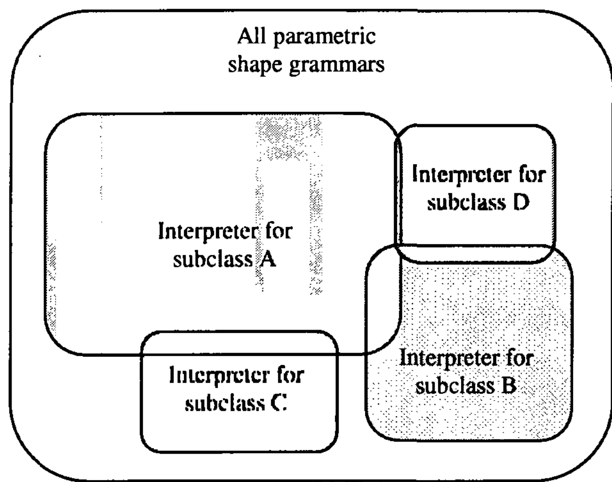


Figure 4-6: A paradigm for a practical 'general' parametric interpreter

The paradigm can be promoted to follow the successful model of the Eclipse project (www.eclipse.org: last accessed 07/09/09), which aims at an open development platform comprising extensible frameworks, tools and runtimes for building, deploying and managing software across the lifecycle. By building a similar platform backed up by the above paradigm, researchers geographically dispersed around the world can collaboratively work on the same platform; each freely developing their idea as an Add-in, thus contributing to their effort. Designers can freely download and exploit up-to-date grammar systems, testing new design ideas, suggesting new features and reporting bugs. Such a platform fundamentally changes the past discrete structure of the research of implementing a shape grammar interpreter (Chau et al., 2004); duplicated work is significantly reduced, and the scope of the users, greatly expanded.

It should be noted that the proposed paradigm depends on a classification of shape grammars into subclasses. Moreover, the classification is considered to be

'better' when the number of subclasses is smaller, and when, simultaneously, the scope covered, collectively, is larger. Here, a research question immediately emerges: *what is the most optimal way of classifying shape grammars*?

4.4 Classification of shape grammars

As mentioned previously, in the literature, there exist ways of classifying shape grammars from various perspectives. Some are based on relatively 'obvious' properties, such as, 2D vs. 3D, rectilinear vs. curvilinear, etc. Others go deeper, for example, structure grammars (Carlson et al., 1991). Classification is also made based on the properties of shape rules as well as their application. For example, non-parametric vs. parametric based on how shape rules are specified, marker-driven vs. subshape-driven based on how to drive the application of shape rules, context-free vs. context-sensitive based on neighborhood dependence when applying shape rules, etc. Knight's six types of shape grammars falls into such a classification (Knight, 1999). Classification from the field of formal linguistics can be introduced too, for example, finite vs. infinite based on the size of the underlying language, that is, the design space.

However, none (of the above) is really appropriate for the purpose of implementation. This is because that each such category is still too broad in that the grammars covered still have enormous variety. In other words, the classification criteria are not fundamental with respect to concerns of implementation.

The fundamental elements for a computer program are algorithms and data structures; this is evident from the title of Niklaus Wirth's classic textbook *Algorithms + Data Structure = Programs* (Niklaus, 1978). This is equally true for computer implementations of shape grammars. An implementation is essentially a computer program that manipulates the internal representation of a design—data structure—by a set of operations. The basic operations of t, $-$, $+$, $\leq$, and R operate on data structure, and their details vary from one data structure to another. The exact procedure of searching for matching candidates depends on the data structure, so do the exact match verification. The underlying data structure in turn determines how to

carry out these operations, and how efficient they are. Moreover, the data structure pre-fixes the power of the shape rules built on top of them; as stated in (Stiny, 1994), *"the antecedent definition of meaning parts and units limits the subsequent possibilities for inquiry... Descriptions fix things in computations, and nothing is ever more than its description anticipates explicitly."*

The argument can also be conducted from a cognitive aspect. The design of a data structure is simply a particular view of the underlying subject, which is present-at-hand. As stated in (Winograd and Flores, 1986), "*Whenever we treat a situation as present-at-hand, analyzing it in terms of objects and their properties, we thereby create a blindness.*" Things covered by any current data structure correspond to those that are seen; the blind parts are left to other data structures.

In line with this argument, the underlying data structure used to support algorithms for the implementation fundamentally characterizes the corresponding class of shape grammars. Assuming that there is always a power difference between any two data structures adopted in the paradigm, and if no other data structure subsumes any of the adopted data structures, then we have reached an optimal classification.

4.5 Augmented practical 'general' paradigm

The 'general' paradigm comprises a set of sub-interpreters, one for each class of shape grammars. Moreover each class is backed up by a data structure, which reflects the internal characteristics of the corresponding subset of shape grammars.

Apart from the internal characteristics of shape grammars, there are other factors that influence computational tractability, for example, how shape grammars are designed and described. Traditionally, a shape grammar is designed to simply and succinctly describe an underlying building style, with little consideration on how the grammar can be implemented. For example, as is often found in the literature, such descriptions of the form "If the back or sides are wide enough, rule 2 can be used..." are inherently counter-computable. As a result, in order to translate this into programming code, shape rules have to be specified in a *computation-friendly* way:

that is, shape rules need to be quantitatively specified; furthermore, there is enough precision in the specification to disallow generation of ill-dimensioned configurations.

Closer examination also shows that there may be more than one way to describe a particular shape rule; it is possible that one way is easy to compute, and the other, might be computationally intractable. As a result, it is desirable to design an API-like framework to support the design of shape grammars; then, shape grammars that follow the framework are guaranteed to be computationally tractable. Such a framework is built on top of an underlying data structure, and basic manipulation algorithms. Moreover, for the ease of code translation, a meta-language built on top of the basic manipulation algorithms should also be developed. As grammars in different classes typically have differing underlying structures, the appropriate underlying data structure for the framework will be different. Consequently, the overall framework comprises a series of sub-frameworks, one for each class of shape grammars, as shown in Figure 4-7. As the overall framework is capable of ensuring computability, we term shape grammars following such a framework as *computation-friendly*.

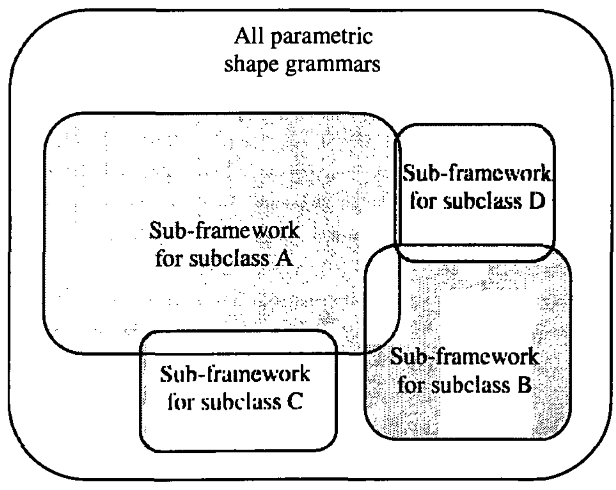


Figure 4-7: One sub-framework for each subclass

Chapter 5 Three sub-framework examples

In this chapter, the paradigm for a practical, 'general' interpretation of shape grammars discussed in last chapter will be detailed by examining three sub-framework examples.

In choosing these sub-frameworks, it was deemed advantageous to initially select a sub-framework for a subclass with the largest population. It turns out that a shape grammar that captures building styles happens to be good choice. Of the shape grammar applications reviewed by (Chau et al., 2004) (Figure 2-8), about half deal with architectural plans. Moreover, conventional buildings, namely, buildings with rectangular spaces or dominated by such, are often the subject matter. Consequently, a sub-framework for shape grammars capturing corpora of conventional building types, namely, the rectangular sub-framework, is chosen.

Two-dimensional polygons are another type of shape widely used in existing shape grammars, for example, Chinese ice-ray lattices (Stiny, 1977) and Hepplewhite-style chair back grammars (Knight, 1981a). Thus, a sub-framework for two-dimensional polygonal shapes is also chosen. From the appearance, such a sub-framework can be viewed as an extension of the rectangular sub-framework. Yet, as

to be shown in this chapter, the extension is not straightforward. In fact, both the application context and the basic manipulations are quite different.

The rectangular sub-framework relies on a graph-like data structure. This leads to a concern of the relationship between shape and graph grammars; the former is mainly investigated in the design field, in particular, architectural design, while the latter is widely studied in computer science. The comparison in this chapter will show that both differ significantly although there is a noticeable intersection. Graph grammars are most useful when dealing with those shape grammars, which are dimensionless and context-free. Accordingly, we consider graph grammars as a sub-framework for implementing dimensionless, context-free shape grammars.

5.1 Rectangular sub-framework

Conventional buildings are buildings with rectangular spaces or dominated by such. A rectangular space is specified by a set of walls in such a way that the space is considered rectangular by the human vision system. In Figure 5-1, among other variations, a space can be specified by four walls jointed to one another, four disjoint walls, three walls, or framed by four corners.

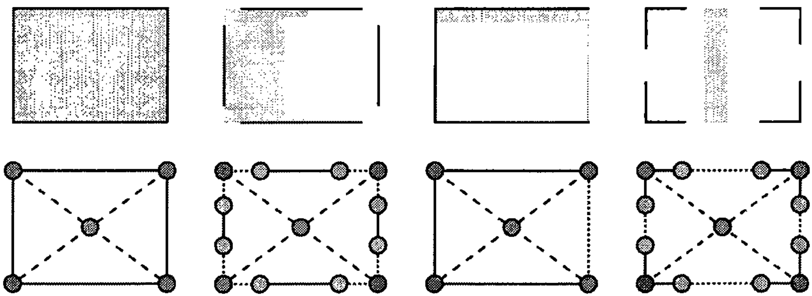


Figure 5-1: Examples of rectangular spaces and corresponding graph-like data structures

Spaces (rooms) are central to buildings—whence, to shape grammars that describe building styles. For shape grammars capturing corpora of conventional building types, shape rules are parametrically specified in such a way that parametric subshape recognition is typically of searching a special room under certain constraints, which is actually label matching. Such grammars generally start with a

rough layout; details, such as openings and staircase, are added at a subsequent stage. There are two main ways of generating a layout: space subdivision and space aggregation. Combination of the two ways is also possible.

5.1.1 A graph-like data structure

The interpreter needs a data structure to represent layouts with rectangular spaces; that is, a data structure that contains both topological information of spaces as well as concrete geometry (in this dissertation, 2D) data of a layout including walls, doors, windows, staircase, etc. It needs to support viewing a layout as a whole, viewing a layout from a particular room with its neighborhood, or simply focusing on a particular room itself. Moreover, the data structure needs to support Euclidean transformations augmented by both uniform and anamorphic scaling.

A graph-like data structure (Figure 5-1) has been designed to specify such rectangular spaces. There is a boundary node for each corner of the rectangular space, as well as a node for each endpoint of a wall. These nodes are connected by either a wall edge (solid line) or an empty edge (dotted line). A central node represents the room corresponding to the space, and connects to the four corners by diagonal edges (dashed lines). It is needed for manipulating boundary nodes of room units, such as dividing a wall through node insertions, creating an opening in a wall by changing the opening's edge type to *empty*, and so on. More information about a room is recorded in the room node, e.g., a staircase within the space. Windows and doors are assigned as attributes of wall edges. Further, unlike traditional graph data structures, the angle at each corner is set to be right angle. A node has at most eight neighbors. A set of such graph units can be combined to represent complex layouts comprising rectangular spaces (Figure 5-2).

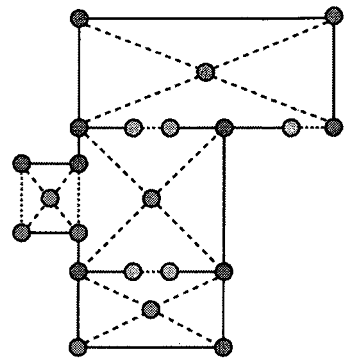


Figure 5-2: A layout represented by a set of graph units

5.1.2 Transformations of the graph-like data structure

The target layout is assumed to comprise only rectangular spaces, and the allowable transformations are Euclidean with uniform and anamorphic scaling. As shape rule application is label-driven, translation is automatically handled. The graph-like data structure is capable of easily handling uniform and anamorphic scaling, by firstly matching room names, then labels on corner nodes, and lastly, by comparing possible room ratio or dimension requirements.

As a result, only rotations and reflections remain to be considered. Note that we can always pre-process a graph-like data structure so that the rectangular spaces are either horizontal or vertical. As the spaces are rectangular, rotations are limited to multiples of 90° and reflections are either horizontal or vertical. Moreover, a vertical reflection can be viewed as a combination of a horizontal reflection and a rotation. Hence, any combination of reflections and rotations is equivalent to a combination of horizontal reflections and rotations. Consequently, the following transformations are all we actually need to consider:

- *R0*: default; no rotation, with possible translation and/or scale.
- *R90, R180, R270*: a rotation of 90°, 180°, and 270°, respectively, with possible translation and/or scale.
- *RR0, RR90, RR180, RR270*: (first a rotation of 0°, 90°, 180°, and 270°, respectively, followed by a horizontal reflection) horizontal reflection,

vertical reflection, or their combination, with possible translation and/or scale.

As shown in Figure 5-3, transformations can be implemented on the data structure by index manipulation. Each of the eight possible neighbors of a node is assigned an index from 0 to 7; indices are then transformed simply by modulo arithmetic. For example, index+2 (modulo 8), counterclockwise rotates neighbor vertices through 90°. Other rotations and reflections are likewise achieved. By viewing the original neighbor relationship for each node with the transformed indices, we obtain the same transformation of the whole graph. By taking advantage of this fact, we need to manipulate only the interior layout instead of the left hand side of every shape rule. Consequently, we only need to consider how to apply shape rules with the default transformation, which is automatically applicable to the configuration under different possible transformations. This gives the same results, but is much simpler to achieve.

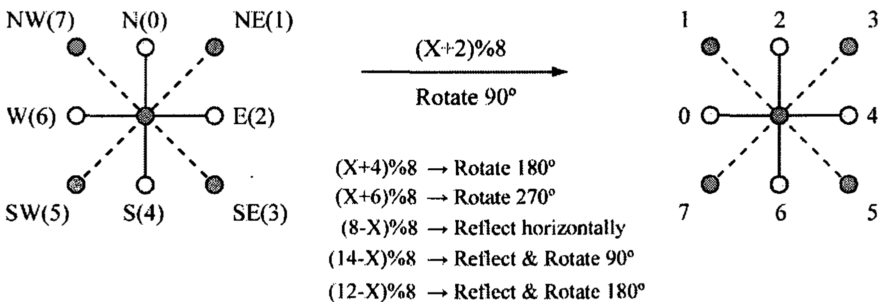


Figure 5-3: Transformations of the graph-like data structures

5.1.3 Common functions for the graph-like data structure

With the graph-like data structure, a layout is represented by an eight-way doubly linked list formed by nodes and edges. Shape rule application manipulates this structure, and a set of common functions shared by the shape rules can be identified. The functions are implemented in an object-oriented fashion.

Design of classes

LNodeCorner and *LNodeRoom* classes represent a corner node and a room node, respectively. The *LNode* class represents all other nodes. Edges are represented by the *Edge* class, with an attribute to represent the different edge types. Theoretically, in order to traverse the entire layout, just knowing the handle to a node or an edge is sufficient. For easy manipulation, the *InteriorLayout* class is defined to represent an interior layout configuration. There are several different ways to view an *InteriorLayout* object: i) as a layout with certain status marker, ii) as a list of rooms (room nodes), and iii) as a list of nodes and edges. Different views are useful under different contexts. For example, it is convenient to use views iii) to display the underlying layout: drawing all edges as well as the associated components first, and then drawing all nodes as well as associate components. To accommodate these different views, the *InteriorLayout* maintains the following fields:

- A status marker
- Name: for display and debugging purpose
- A hash map of a room name to a list of room nodes: for fast retrieval of one or more room nodes with a given name
- A list of room nodes for the entire layout
- A list of all nodes for the entire layout
- A list of all edges for the entire layout
- A hash map of attributes to values for other status values particular for a special shape grammar

Examples of common functions

Some common functions are relatively easy to carry out, for example, splitting a room into two and merging two rooms into one. Others are more complicated; examples include finding a room with a given name, finding the north neighbor(s) of a given room, finding the shared wall of two given rooms, etc. The sequel describes the algorithm and pseudo code for these examples.

Finding room(s) with a given name

In the data structure, a room node represents a room. An *InteriorLayout* object maintains a hash map of room names to lists of room nodes. Thus, finding room(s) with a given name is simply to query the hash map with the room name as input.

---

findRoomNodes(*Name*)

Query the name-to-rooms hash map with parameter *Name*.

Finding the north neighbor(s) of a given room

Finding the north neighbor(s) of a given room is a special case of finding neighbor(s) of a given room. It turns out all that finding neighbor functions in the other three directions can be implemented as finding the north neighbor(s) under a certain transformation. For example, the east neighbor(s) of a given room is the same as the north neighbor(s) of the given room under a *R90* transformation.

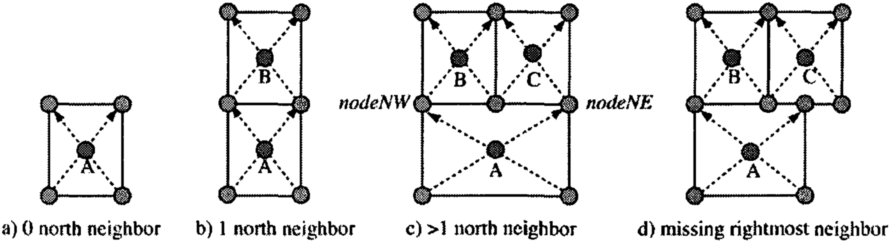


Figure 5-4: Different cases for the north neighbor(s) of a room

A room may have zero, one, or more north neighbors (Figure 5-4), which can be represented by a list of room nodes. Intuitively, to find the north neighbor(s) of *A*, we start by finding *A*'s north-east corner node, *nodeNE,* and north-west corner node, *nodeNW*. Then, we traverse through each corner node from *nodeNE* (inclusive) to *nodeNW* (exclusive) along the westerly direction to find its north-west neighbors. All north-west neighbors found are desired room nodes. For example, in Figure 5-4c, the north neighbors found are *B*, and *C*. However, as shown in Figure 5-4d, this intuitive algorithm will miss the rightmost neighbor room, that is, when two neighbor rooms only partially overlap so that *nodeNE* is on the south edge of that neighbor room, and

is not the desired end node. Therefore, we need to modify the intuitive algorithm to have the correct start and end nodes to loop through.

It can be proven that *nodeNW* is always the correct end node as a north neighbor *B* has to overlap with room *A*, which means room *B* must has a south-east corner node, *nodeSE*, at the right side of *nodeNW* (Figure 5-5a), or is *nodeNW* (Figure 5-4c). Otherwise, *B* is not a north neighbor of *A*.

The starting node can be either *nodeNE*, or a node to the right of *nodeNE* (Figure 5-5b). If *nodeNE* is not the start node, then it neither has a north-west nor a north-east neighbor, since having either neighbor means that *nodeNE* is the correct start point (Figure 5-5c), which is a contradiction. However, the converse is not true; as shown in Figure 5-5d, *nodeNE* neither has a north-west nor a north-east neighbor, but *nodeNE* is still the correct start node. That is, the only condition for a node, *nodeSE*, to the right of *nodeNE,* to be the correct start node is that it must have a north-west neighbor. Therefore, under the condition that *nodeNE* has no north-west and north-east neighbor, the algorithm searches for the first node, which is to the right of a *nodeNE* with a north-west neighbor. If such a node is found, it is the real start node. If a *null* neighbor is found, *nodeNE* is still the correct start node. The pseudo code is given below.

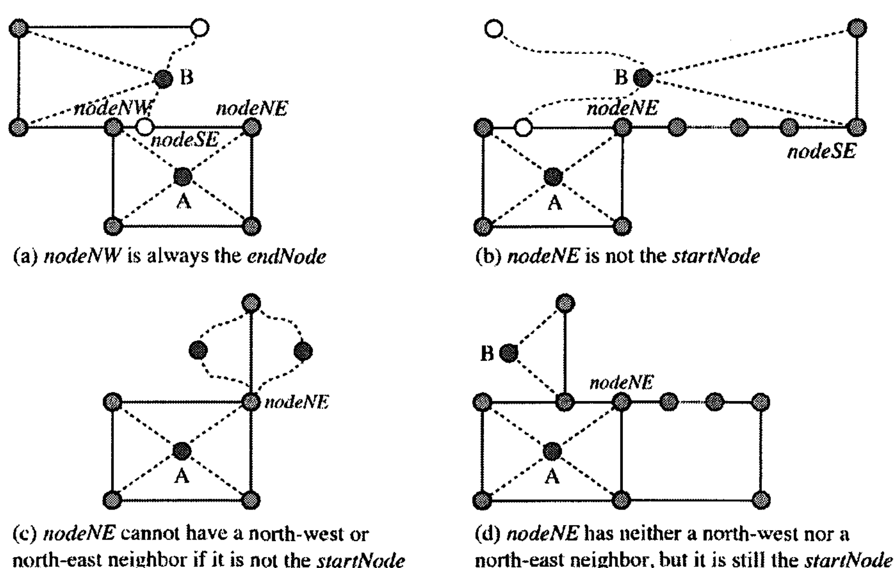


Figure 5-5: The start and end nodes for finding neighbor room(s)

```
findNorthNeighbors(A, T)
    (all operations related to directions are under transformation T)
    endNode ← north-west neighbor of A
    nodeNE ← north-east neighbor of A
    startNode ← nodeNE
    if nodeNE has neither north-west nor north-east neighbor
        search for a right neighbor, node, of nodeNE, with a north-west neighbor
        if found, startNode ← node
            go through each node in between startNode (inclusive) and endNode
             (exclusive), and get all north-west neighbors, neighbors
    return neighbors

findEastNeighbors(A)  // Other neighbors are likewise defined
    return findNorthNeighbors(A, R90)
```

Finding the shared wall of two given rooms

In the data structure, the shared wall of two given rooms is represented as a list of nodes connected by edges; the simplest form of a shared wall is given by two nodes connected by an edge. For two given input room nodes, *A* and *B*, in general, *A* and *B* may not be neighboring rooms at all. If, however, *A* and *B* are real neighbors, *B* can be in any one of four directions from *A*. Therefore, it is necessary for the algorithm to test all four sides of *A*; for each particular side, it is simply to test whether *B* is in the north neighbors under a given transformation *T*. If *B* is determined as a neighbor of *A* at a given side, the exact start node, *wStart*, and end node, *wEnd*, need to be further determined. The edge from the north-east node, *nodeNE*, to the north-west node, *nodeNW*, of room *A* under transformation *T* is guaranteed to be the wall of room *A*, but not necessarily the wall of room *B* (Figure 5-6a). As a result, *wStart* may be actually a node to the right of *nodeNE*. This node is found by traversing from *nodeNE* to *nodeNW*, testing whether *B* is its north-west neighbor or not. Similarly, *wEnd* may be actually a node to the left of *nodeNW*. This node is found by traversing from *nodeNW* to *nodeNE* and testing whether *B* is its north-east neighbor or not. The pseudo code is given below:

```
findWallShared(A, B)
    transformations ← {R0, R90, R180, R270}
    for each transformation in transformations
        results ← findNorthWallShared(A, B, transformation)
        if results is not null
            return {results, transformation}
    return null

findNorthWallShared(A, B, transformation)
    if B not in neighbors ← findNorthNeighbors(A, transformation)
        return null
    nodeNE ← north-east neighbor of A
    nodeNW ← north-west neighbor of A
    wStart ← null
    wEnd ← null
    for each node, node, from nodeNE to nodeNW
        if north-west neighbor of node is A
            wStart ← node, and break
    if wStart is null, then wStart ← nodeNE (Figure 5-6b)
    for each node, node, from nodeNW to nodeNE
        if north-east neighbor of node is B
            wEnd ← node, and break
    if wEnd is null
        wEnd ← nodeNW (Figure 5-6b)
    return {wStart, wEnd}
```

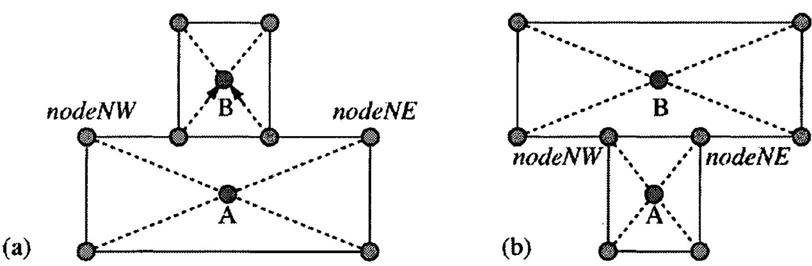


Figure 5-6: Finding *wStart* and *wEnd*

5.1.4 Meta-languages

All common functions collectively form an API (Application Programming Interface), which expresses the capability of its underlying data structure. Such an

API facilitates the design of shape rules, in a way similar to how the Java API helps to build Java applications. Moreover, grammar designers can apply the API to ensure the computability of their designed grammars. Further, the API supports describing grammars in a meta-language; in this way, shape rules are designed in a rigorous way so that they can easily translated into pieces of code, the ultimate format to interpret shape rules. Figure 5-7 shows two such examples. The meta-language is in the form of an *if-then* statement; the *if*-part determines whether the rule is applicable or not; the *then*-part specifies how to do the rewriting. Essentially, the meta-language is a set of function calls, which are predefined in the API.

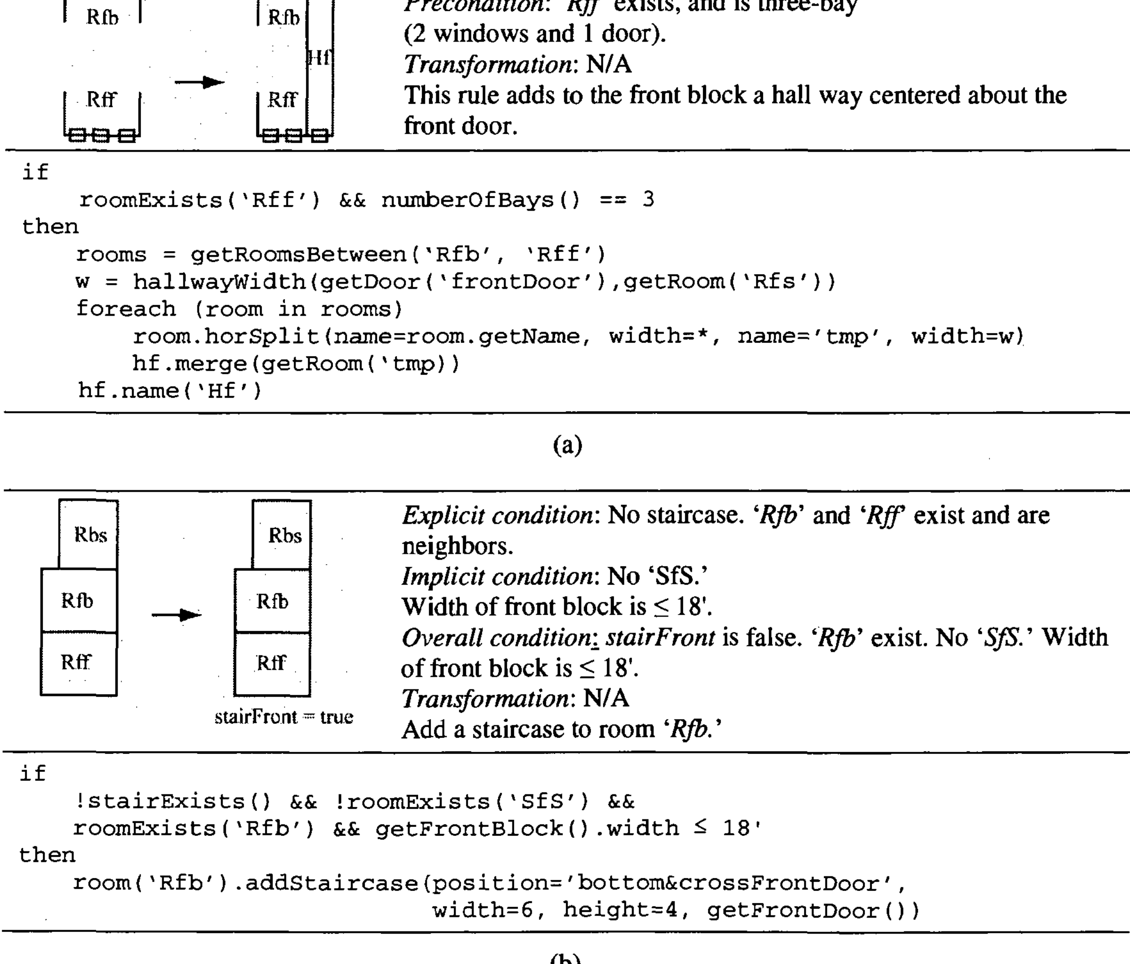


Figure 5-7: Two sample rules of the rectangular sub-framework and their meta-language

5.2 Polygonal sub-framework

Geometrically, the polygonal sub-framework may appear, quite simply, to be an extension of the rectangular sub-framework. Closer examination actually tells a different story. Technically, for example, the nice property of performing geometric transformations by indicial change, which works for the rectangular sub-framework, is not generally applicable to the polygonal sub-framework. The basic elements of the rectangular sub-framework are rectangles, which always have four sides, while the basic elements of the polygonal sub-framework are arbitrary polygons, for example, the triangle, quadrilateral, pentagon, hexagon and so on, each having a different number of sides. This makes it impossible to conduct modulo operations uniformly.

It is possible to extend transformation-by-indicial-change to those shapes composed of polygons of same type, for example, hexagon. However, an endeavor along these lines is not interesting due to the fact that the typical application context and basic manipulations of this polygonal sub-framework are very different from the rectangular sub-framework.

While the rectangular sub-framework works for shape grammars describing building layouts, the polygonal sub-framework does not. The reason is that majority of building spaces are rectangular rather than polygonal. Instead, shape grammars involving polygonal shapes are more common in describing other kinds of designs, for example, Chinese ice-ray lattices (Stiny, 1977), Hepplewhite-style chair backs (Knight, 1981a), as well as abstract paintings (Knight, 1989), see for example, the nonrepresentational paintings of Fritz Glarner (Figure 5-8). Such shape grammars are typically parametric and marker-driven. The central manipulation is subdivision, which is the theme selected for the polygonal sub-framework. Besides subdivision, there are other auxiliary manipulations, such as filling colors, inscribing to the initial shape with a shape of triangle, pentagon, hexagon, etc (Stiny, 1977). Such auxiliary manipulations cannot be generated by subdivision and are handled in a special way, by adding extra functions, or by other means, for example, treating the shape to be inscribed as part of the initial shape.

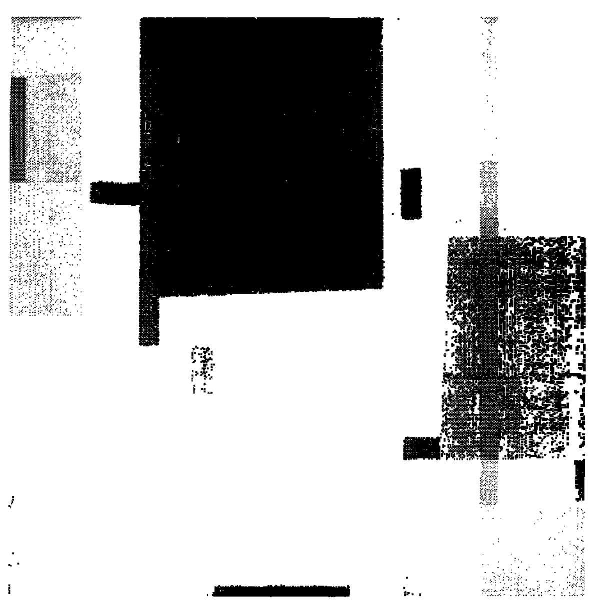


Figure 5-8: Relational painting No. 64, 1953, by Fritz Glarner (http://www.metmuseum.org/toah/hd/geab/ho_1983.579.htm: accessed May 2009)

Subdivision is a procedure for dividing a polygon into two smaller ones by a 'cutting' line, which is a straight-line segment, a joint line of two segments, or a polyline of multiple line segments. As a result, transformations become unnecessary, since an equal effect can be achieved by changing the coordinates of the endpoints of the cutting line. For example, Figure 5-9a shows a shape rule which subdivides a triangle into a smaller triangle and a quadrilateral. Figure 5-9b shows the horizontal reflection of the shape rule of Figure 5-9a.

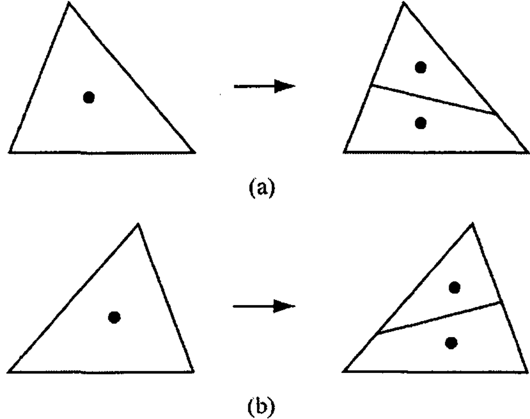


Figure 5-9: A shape rule of subdivision and its horizontal reflection

The same effect can be achieved by modifying the endpoints of the cutting line. Note that the shape rule is parametric so that by varying the parameters of the shape and the endpoints of the cutting line, the shape rule of Figure 5-9b can become exactly the same as Figure 5-9a.

The determination of the position of a cutting line starts with inserting a point or multiple points in the interior of a polygon or on its boundary; then the cutting line is generated by connecting new inserted points to other existing points of the polygon, possibly involving line extensions and intersections, or by simply interconnecting the new inserted points. There are typically constraints over the candidate position of a new point. The constraints can be a fixed position like the centroid of a polygon, an interval on a line, or a particular region. This means that there are generally infinitely many possibilities to position a new point. Two ways frequently used to position the new point are *manual pickup* and *random selection.*

Noticeably, a 'subdivision' of the polygonal sub-framework is different from a 'splitting' of a rectangular sub-framework. The former is typically oblique while the latter is always horizontal or vertical. Moreover, a cutting line of the former often has infinitely many possibilities while the position of a splitting line of the latter is usually uniquely 'fixed'.

5.2.1 Data structure for polygonal sub-framework

In terms of the data structure for the polygonal sub-framework, a shape consists of a list of polygon objects, each of which further consists of a list of straight-line segment objects. Note that, here, polygons are simple; that is, there is no self-intersection. For the convenience of manipulation, the line segments of a polygon are counter-clock-wise arranged and indexed by indices. Operations like the modification of the shared line segment of two polygons change the statuses of multiple related objects. Objects are used as references to ease such operations. By this way, the underlying data structure is able to support three different views: i) all the polygons as a single shape, ii) individual polygons with markers, and iii) individual labeled line segments. In addition, hash maps of markers to polygons,

labels to points, and point labels to line segments are used to simplify the search of polygons, line segments and points.

5.2.2 Common functions of polygonal sub-framework

Key common functions include the function of dividing a simple polygon into two by a cutting line, and those determining the positions of the new points.

Dividing a simple polygon into two by a cutting line

Here, we consider a more general function, dividing a simple polygon G into multiple sub-polygons by a cutting line C (Figure 5-10a).

This problem can be solved by converting the problem to finding the intersection of two arbitrary (may not be simple) polygons, which has been well studied (Stouffs, 1994; Greiner and Hormann, 1998; O'Rourke, 1998; Stouffs and Krishnamurti, 2006). This is done by first finding a rectangle, which is larger than the bounding box of polygon G, and then forming two simply polygons, G_{C1} and G_{C2}, by extending the starting and ending line segments of the cutting line (Figure 5-10b); the desired results will be $(G \cap G_{C1}) \cup (G \cap G_{C2})$ (Figure 5-10c).

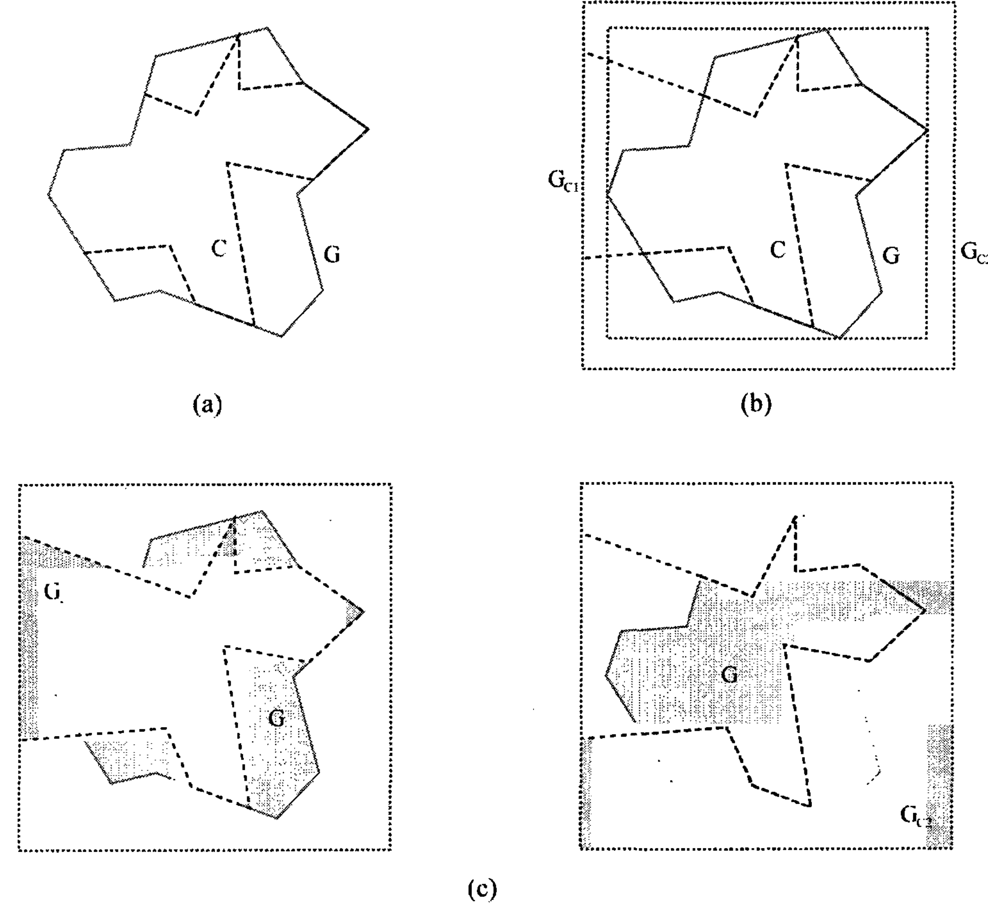


Figure 5-10: Dividing a simple polygon by intersection of two arbitrary polygons

In the following, I describe a simpler algorithm, which is inspired by (Greiner and Hormann, 1998). This algorithm takes advantage of the special properties of cutting lines in the polygonal sub-framework. A cutting line is always interior or on the boundary of the polygon G and has no self-intersection. Moreover, the start point P_s and end point P_e of the cutting line are on the boundary of polygon G (Figure 5-12a). In fact, any cutting line can be reshaped to satisfy these conditions (Figure 5-11).

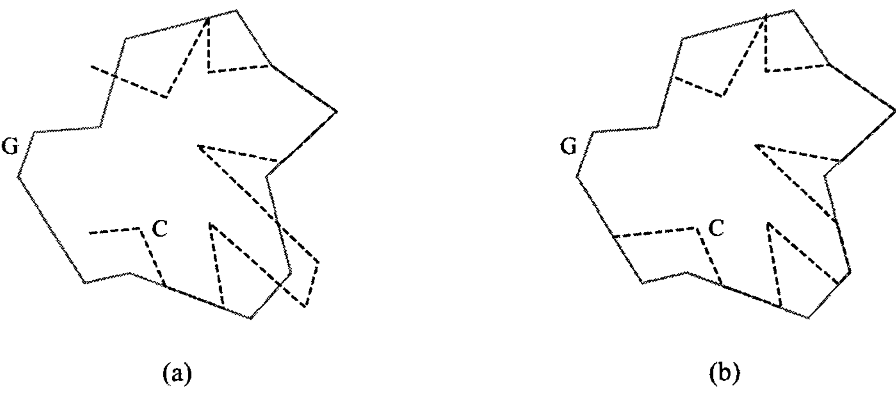


Figure 5-11: Reshaping the cutting line

The algorithm starts from the start point P_s of the reshaped cutting line, marching to the end point P_e, segment by segment. Each segment is tested for whether intersecting the polygon G or not by testing whether the other endpoint falls on the boundary of polygon G, for example, endpoint P_1 for segment S_1, P_2 for segment S_2 (Figure 5-12a). When an intersection is found, two new polygons are created by using the cutting line matched so far and continuously matching right and left, respectively, along the polygon G until going back to the start point. The segments marched are then removed from the cutting line so that a new cutting line is formed for the next step (the dashed line in Figure 5-12). If the new cutting line is empty, then both new polygons are the desired results. Otherwise, by using the point-inside test on the two new polygons with the point P next to the start point of the new cutting line, the one which P does not fall inside (dark shaded polygons in Figure 5-12) is the desired result, and the other (light shaded polygons in Figure 5-12), together with the new cutting will be used as the input for the next step. The above procedure is repeated and the entire algorithm stops when the cutting line becomes empty (Figure 5-12f). Figure 5-12 shows an example of applying the marching algorithm.

Note that there are possible degenerate cases that some segments matched overlaps with the some segments of the polygon G (Figure 5-12c, d, and f). In such cases, the number of segments in one of the two new polygons must be two; this can be easily tested and ignored. Another issue with such cases is that the segment

coming from the cutting line is co-linear and connected to the next segment coming from the polygon input, and these two should be merged into one (Figure 5-12f).

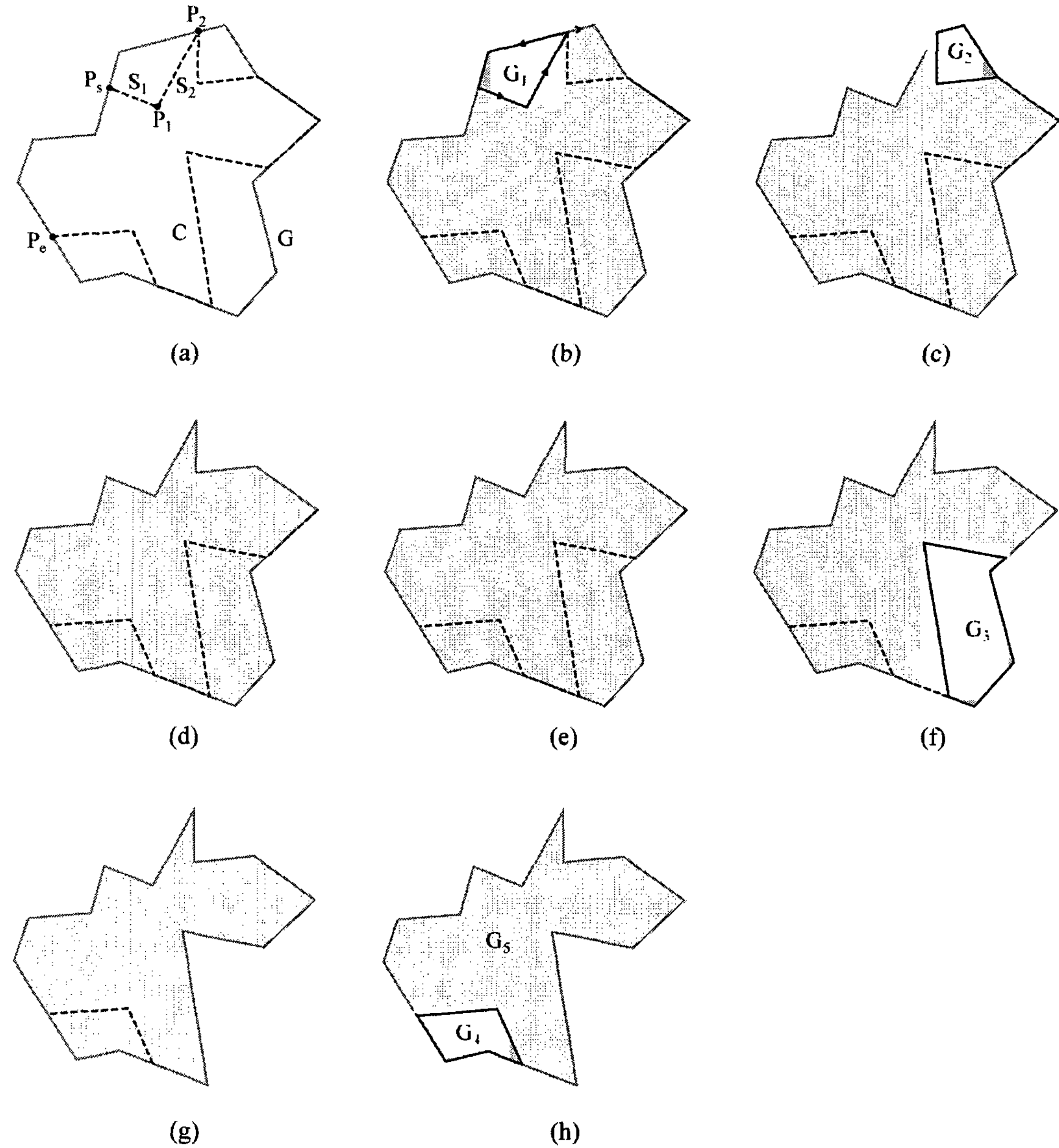


Figure 5-12: A simpler marching algorithm for polygon subdivision

The intersection test dominates the running time of the marching algorithm. The total number of intersection tests for marching along the cutting line is *mn*, where *m* and *n* is the number of segments in *C* and *G*, respectively. As a result, the complexity is *O(mn)*. The pseudo code of the marching algorithm is given below:

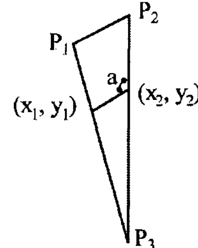
```
dividePolygon(G, C, newPolygons)
    S = get first segment from C
    while (the other endpoint of S not falling on G)
        marchedSegments.add(S)
        S = get next segment from C

    marchedSegments.add(S)
    I = the other endpoint of S
    C = C.remove(marchedSegments)
    divide the intersected segment of G into two if I is not its endpoints
    newPolygon1 = marching along marchedSegments and turning right at I
        until reaching the start
    newPolygon2 = marching along marchedSegments and turning left at I
        until reaching the start

    if (C is empty)
        newPolygons.add(newPolygon1)
        newPolygons.add(newPolygon2)
    return

    P2 = the point next to the starting point in the new cutting line
    if (P2 falls inside newPolygon1)
        newPolygons.add(newPolygon2)
        dividePolygon(newPolygon1, C, newPolygons)
    else
        newPolygons.add(newPolygon1)
        dividePolygon(newPolygon2, C, newPolygons)
```

Determining the positions of the new points

Determining the positions of the new points can be done in two ways, randomly or manually. Manual determination needs the help of an interface, for example, highlighting the candidate regions, and enforcing further constraints for the next new point after a new point has been picked. Random determination requires computing all candidates of intervals and regions, and randomly selecting a point.

5.2.3 Meta-language for polygonal sub-framework

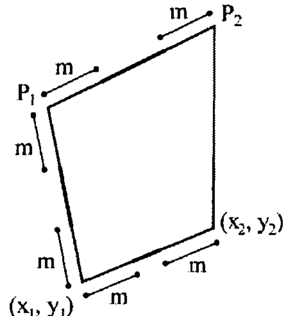
Figure 5-13 shows the meta-language for two examples taken from (Knight, 1980).



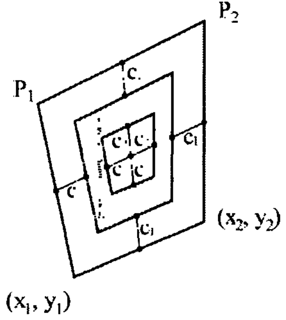
The point (x_1, y_1) is any point between 5/8 and 3/4 the distance from point P_3 to point P_1.
The point (x_2, y_2) is any point between 5/8 and 3/4 the distance from point P_3 to point P_2.
The angle a must be $\geq 90^o$.

```
(x2,y2)=randomPick(interval(getPoint('P3'),getPoint('P2'),5/8,3/4))

line=perpendicular((x2,y2),getLine('P2','P3'))
(x,y)=intersection(line, getLine('P1','P3'))
(x1,y1)=randomPick(interval(getPoint('P3'),getPoint('P1'),5/8,3/4)&
                   interval((x,y), getPoint('P3')))

dividePolygon([(x1,y1),(x2,y2)], [P1,P3,P2], [])
```

(a)



The point (x_3, y_3) can be any point on the lines with endpoints P_1 and P_2, or P_1 and (x_1, y_1), or (x_1, y_1) and (x_2, y_2) in the intervals *m* units away from the endpoints of these lines where *m* is a fixed constant.



The point (x_3, y_3) can also be any point within the area defined by constants c_1 and c_2. This area is inside the quadrilateral with vertices at (x_1, y_1), (x_2, y_2), P_2, and P_1, and its boundaries are parallel to the boundaries of the quadrilateral.

```
(x3,y3)=manualPick(interval(getPoint('P1'),getPoint('P2'),m) |
                   interval(getPoint('P1'),(x1,y1),m) |
                   interval((x1,y1),(x2,y2),m))

or
(x3,y3)=manualPick(area(getPolygon('P1','P2','x2'+'y2','x1'+'y1'), c1, c2))

dividePolygon([(x1,y1), (x3,y3), P2], [P1,(x1,y1),(x2,y2),P2], [])
```

(b)

Figure 5-13: Meta-language examples of picking up new points under constraints

Figure 5-13a is an example of randomly selecting two points from candidate intervals. The meta-language enforces the constraint of angle $a \geq 90^{\circ}$ by determining (x_2, y_2) first, creating a line passing through (x_2, y_2) and a perpendicular to line P_2P_3, computing the intersection (x, y) of the new line with line P_1P_3 to obtain a new interval $[(x, y), P_3]$, and using the intersection of the interval $[(x, y), P_3]$ with the interval $[P_3, P_1, 5/8, 3/4]$ as the interval for (x_1, y_1). Figure 5-13b is an example of manually picking up points from candidate intervals and regions. All candidate intervals and regions are computed and combined as a candidate pool. The interface highlights the candidate pool to help users to pick up a point.

5.3 Graph sub-framework

The rectangular sub-framework may give the impression that shape grammars are just special cases of graph grammars (Brouno, 1990; Rozenberg, 1997), which have been widely studied in the computer science. The following discussion will show that both significantly differ from one another. However, graph grammars can be used as a sub-framework to solve those dimensionless, context-free shape grammars.

5.3.1 Shape and graph grammars

Graphs provide a natural way of describing complex situations on an intuitive level. At certain level, this characteristic caters to the advantage that visual languages (that is, shapes) possess. Graph grammars are rule-based modification of graphs through graph rule application. Graph grammars have been developed as an extension to graphs of formal string grammars (aka. generative grammar, or phrase structure grammars). Among string grammars, context-free grammars are the best understood; they have proven extremely useful in practical applications and powerful enough to generate a wide spectrum of interesting formal languages. Analogously, most research focuses on 'context-free' graph grammars, which typically means local modifications of graphs without 'global' constraints. Rule application on graphs is, typically, label driven. There are two basic choices for rewriting a graph: *node replacement* and *hyperedge replacement*.

Shape grammars are rule-based rewriting systems of shapes. In many ways, these can be viewed as an extension of formal string grammars to shapes. Their shared roots imply a close connection between graph and shape grammars. As an example, Drews and Kreowski investigated the properties of collage grammars, a special case of graph grammars, and applied them to generate pictures, e.g., Sierpinski gasket (Drewes and Kreowski, 1999) (Figure 5-14 and Figure 5-15). Likewise, such pictures can be also succinctly described by shape grammars (Stiny, 1977; Piazzalunga and Fitzhorn, 1998) (Figure 5-16). This suggests that there is an intersection between graph and shape grammars.

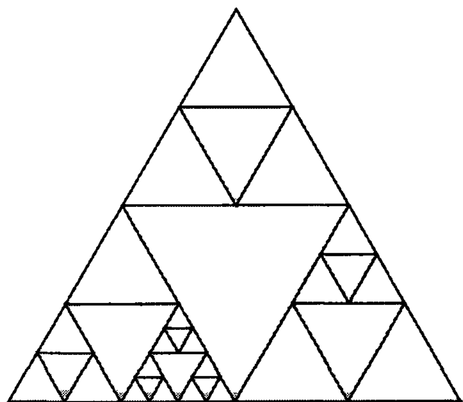


Figure 5-14: A Sierpinski gasket

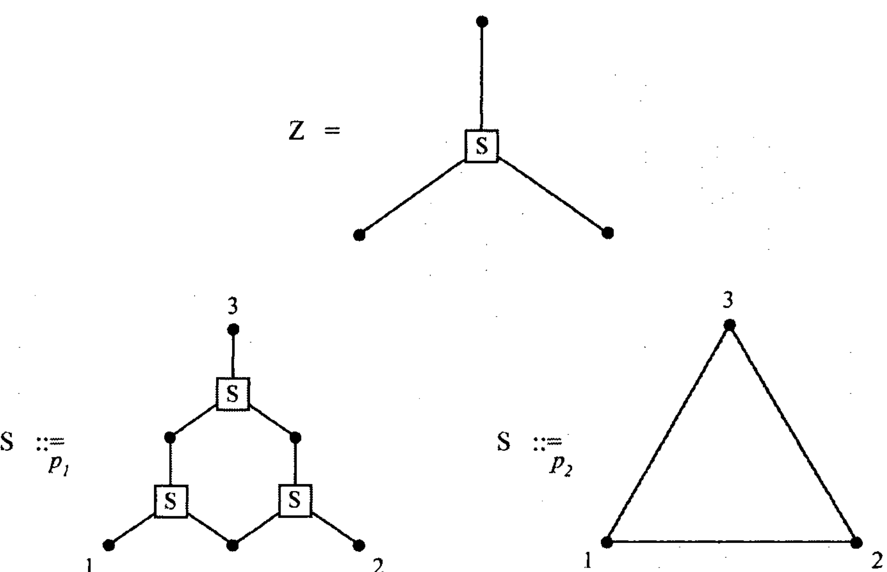


Figure 5-15: A collage grammar for the Sierpinski gasket
Adapted from (Drewes and Kreowski, 1999)

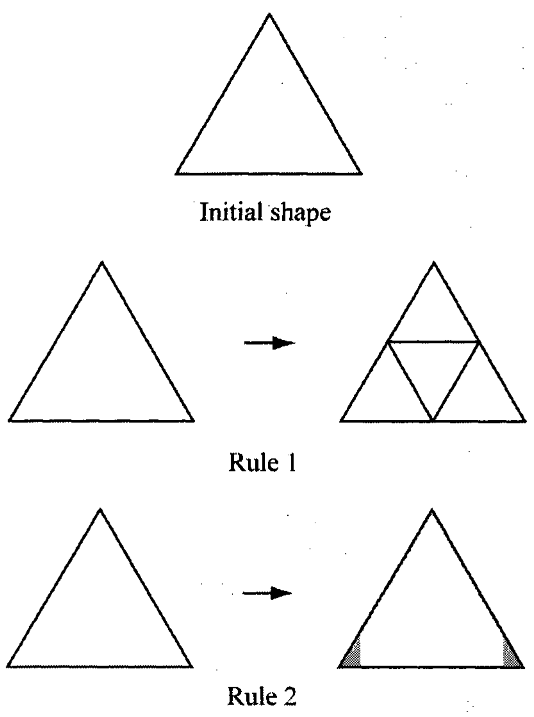
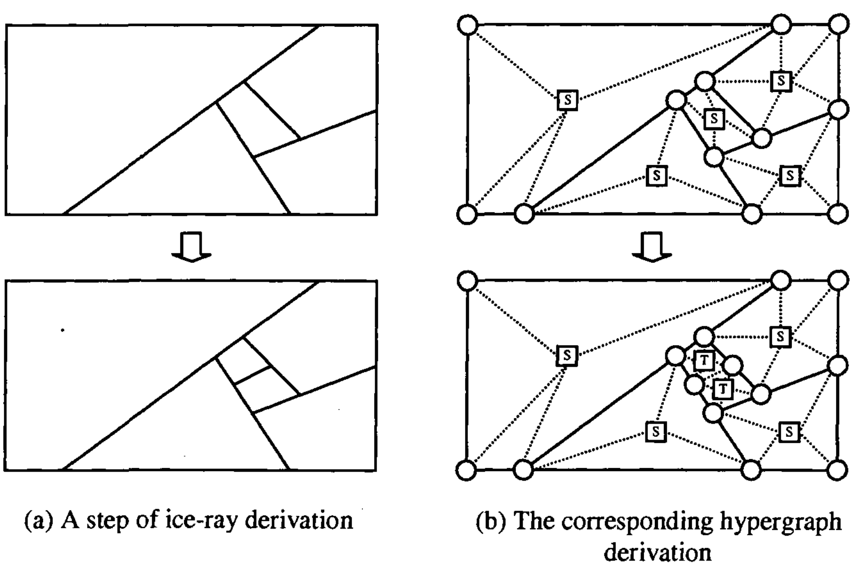


Figure 5-16: A shape grammar for the Sierpinski gasket

Consequentially, shape grammars can take advantage of graph grammar research results, especially for 'context-free' shape grammars; that is, when shape rewriting happens locally. For example, as shown in Figure 5-17, ice-ray grammars (Stiny, 1977), which essentially describes a process of polygon subdivision, can be implemented as a graph grammar. Each point corresponds to a vertex and each polygon is decorated with a hyperedge (the vertices drawn in squares together with dashed tentacles). Figure 5-18 shows shape rules vs. corresponding graph rules of ice-ray grammars: the right-hand hyperedges are labeled either *S* as candidates for further rule application, or *T* for no further rule application; the choice is based on certain criteria, for example, the area of the underlying polygon. Rule 3 of graph rules is applied in Figure 5-17. Note that there is a necessary step to convert graphs to figures when using graph grammars to generate designs; depending on the details of the conversion, such graph grammars may show different appearances (Figure 5-15 and Figure 5-18).



(a) A step of ice-ray derivation

(b) The corresponding hypergraph derivation

Figure 5-17: Implementing the ice-ray grammar as a graph grammar

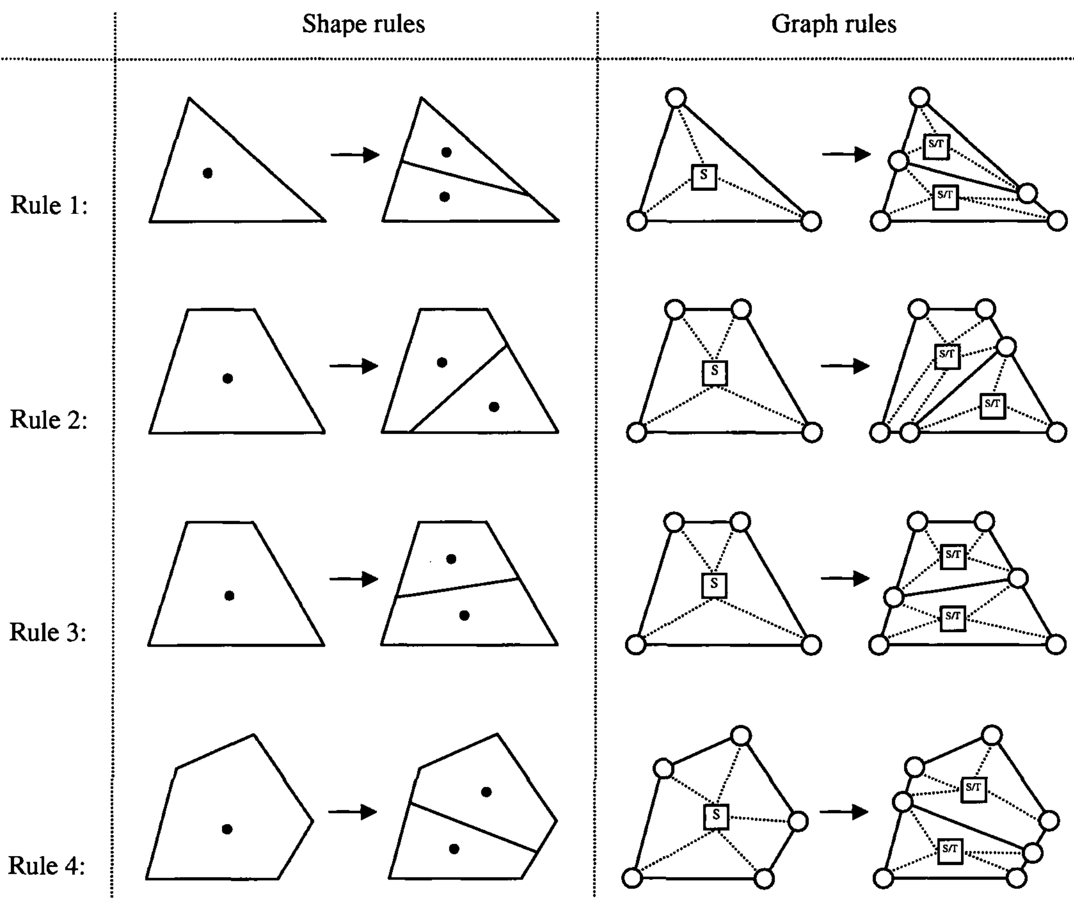
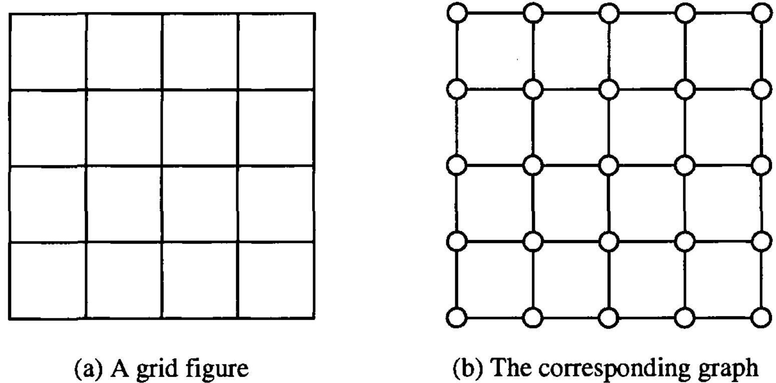


Figure 5-18: Shape and corresponding graph rules of the ice-ray grammar

On the other hand, shapes differ significantly from graphs and so do their grammars. Shape grammars do not deal solely with pure pictures; they are usually imbued with semantics, and represent designs in reality. In this respect, dimensions become typically important. Graph grammars, however, are inherently dimensionless. Moreover, semantics make most shape grammars context-sensitive; this greatly limits whatever advantages are provided by those nice theorems of graph grammars (on the assumption that the grammars are context-free).

Graph grammars are essentially label-driven; this puts further restrictions in helping solve the fundamental problem of subshape recognition in shape grammars. As a classical example (Figure 5-19), there are many, potential uncountable, number

of square subshapes in a grid figure. Converting the grid figure to a graph does not change the basic characteristics of the problem.



(a) A grid figure (b) The corresponding graph

Figure 5-19: Subshape recognition in a grid figure

5.3.2 Graph grammars as a sub-framework

Research on *collage grammars* (Drewes et al., 1996; Drewes and Kreowski, 1999) shows how graph grammars can be used as a sub-framework in the 'general' paradigm of a shape grammar interpreter. Such graph grammars are essentially parametric and label-driven. The underlying data structure is obviously a graph, typically undirected in the context of generating designs.

5.3.3 Common functions for graph sub-framework

The central step in using graph grammars to generate designs is the iterative application of a set of graph rules, which is known as graph transformation in the literature (Heckel, 2006). Moreover, the manipulation of the underlying graph is also achieved through graph transformation. Thus, the key common function is the application of a graph rule.

Graph rule application

Graph rule applications (that is, graph transformation) are central to graph grammars, and many different approaches have been investigated (Rozenberg, 1997). In general, a graph rule r is defined by six tuples (L, R, K, *glue*, *emb*, *appl*): i) L and R are left hand side and right hand side graph, respectively; ii) K is a subgraph of L called

interface graph; iii) *glue* is an occurrence of K in R, relating the interface graph with the right hand side; iv) *emb* is an embedding relation, relating nodes of L to nodes of R; and v) *appl* is a set specifying the application conditions for the rule (Andries et al., 1999). It is possible that K, *glue*, *emb*, or *appl* is empty—certain combination of emptyness forms rules with special properties, for example, rules without application conditions and with empty embedding relation corresponds to single-pushout rules.

The application of r to a graph G replaces an occurrence of the left hand side L in G by the right hand side R. This is done through three stages: i) removing a part of the occurrence of L from G, ii) gluing R and the remaining graph D, and iii) connecting R with D via the insertion of new edges between the nodes of R and those of D. Note that the left hand side matches all isomorphic graphs and this subsumes geometry transformations, which are usually important in the application of shape grammars. The pseudo code of applying a graph rule is given below, which is adapted from (Andries et al., 1999):

---

applyGraphRule (G, r=(L, R, K, *glue*, *emb*, *appl*))

- Choose an occurrence of the left hand side L in G
- Check the application conditions according to *appl*
- Remove the occurrence of L up to the occurrence of K from G as well as all *dangling edges*. This yields the *context graph* D of L which still contains an occurrence of K.
- Glue the context graph D and the right hand side R according to the occurrences of K in D and R. That is, construct the disjoint union of D and R and, for every item in K, identify the corresponding item in D with the corresponding item in R. This yields the *gluing graph* E.
- Embed the right hand side R into the context graph D according to the embedding relation *emb*. For each removed dangling edge incident with a node v in D and the image of a node v' of L in G, each node v'' in R, a new edge incident with v and the node v'' is established in E provided that (v', v'') belongs to *emb*.
- Return G

5.3.4 Meta-language for graph sub-framework

The meta-language for the graph sub-framework is mainly to call the common function of applying a graph rule by specifying the details of the graph rules, with

auxiliary functions to convert the final graph to shapes. Figure 5-20 shows the style of a collage grammar as well as the corresponding meta-language by using the spiral collage grammar, which is an example taken form (Drewes et al., 1996). Figure 5-21 showing the result of applying the spiral collage grammar by replacing *any* hyperedge with some sub-collages at the final step.

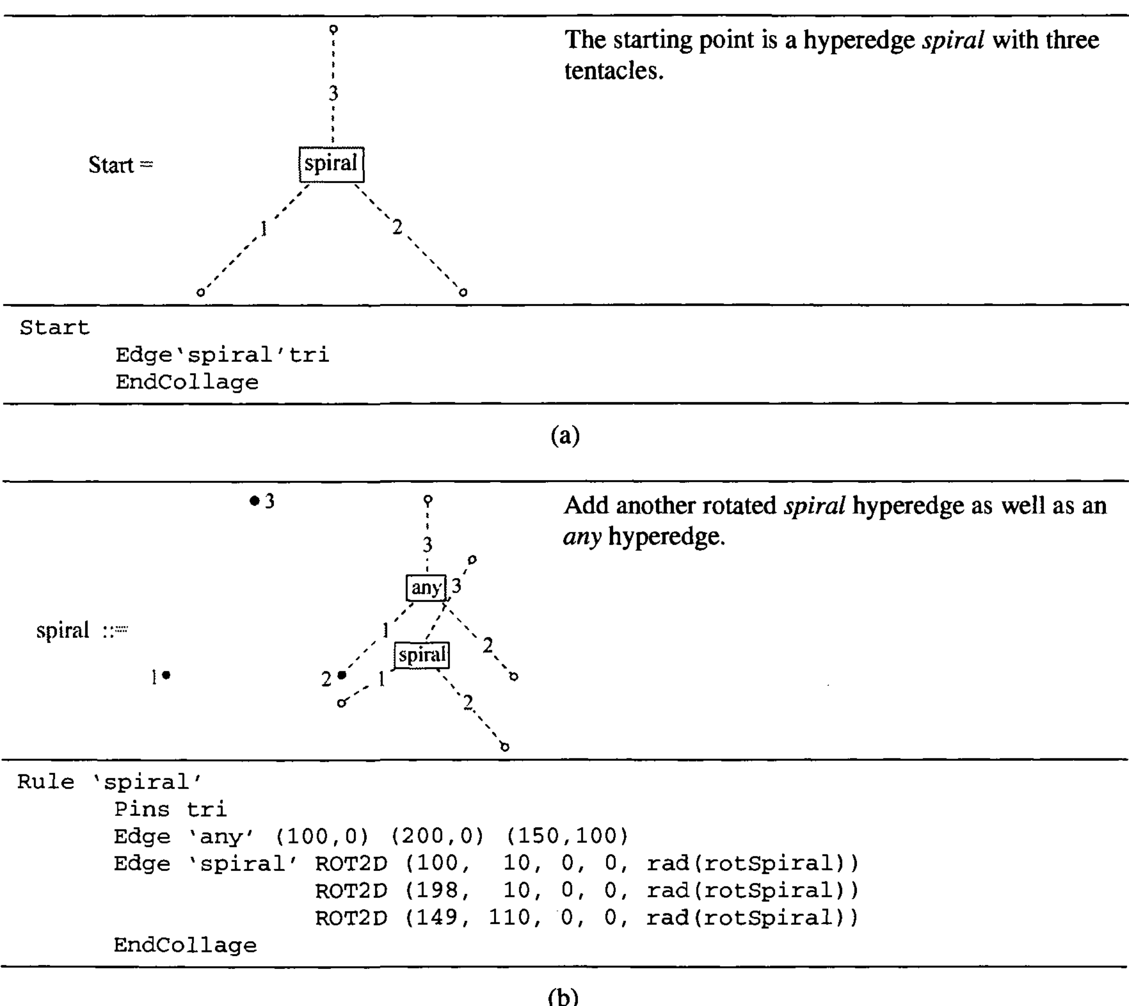


Figure 5-20: Meta-language description of the spiral collage grammar
Adapted from (Drewes et al., 1996).

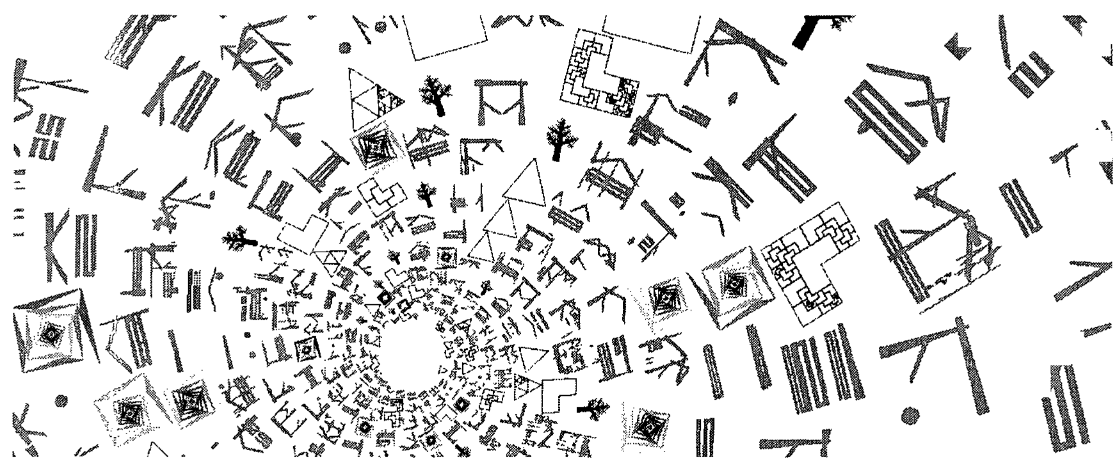


Figure 5-21: A result of the spiral collage grammar with some sub-collages as elements Adapted from (Drewes et al., 1996).

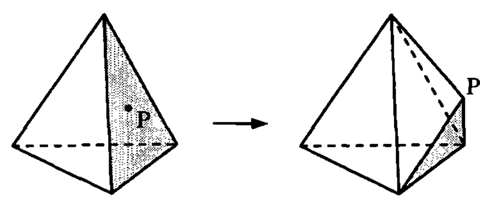
The latest development along this line is a Java implementation of a system called TreeBag. See http://www.informatik.uni-bremen.de/theorie/treebag/ (accessed 07/09/09) for further information. It includes a manual, which describes the syntax of a meta-language for specifying such grammars.

5.4 Discussion

Each sub-framework discussed in this chapter specifies a way of implementing a subclass of shape grammars. In terms of language space, the language covered by a sub-framework is equal to the language of a subclass of shape grammars. However, each sub-framework takes advantage of the special characteristics of the corresponding subclass of shape grammars so that implementation is manageable. That is, although the language spaces are equal, the implementation does not truly implement the shape grammar formalism as described in the formal definition.

The three illustrated sub-frameworks are all two-dimensional. There is nothing intrinsic in the paradigm to prevent a sub-framework from being three-dimensional. In fact, Heisserman's boundary solid grammar is really such an example (Heisserman, 1994). The representation of solid objects is composed of two parts: topology and geometry. The topology is represented as a graph composed of nodes and arcs — the nodes are topological elements, and the arcs represent the adjacencies

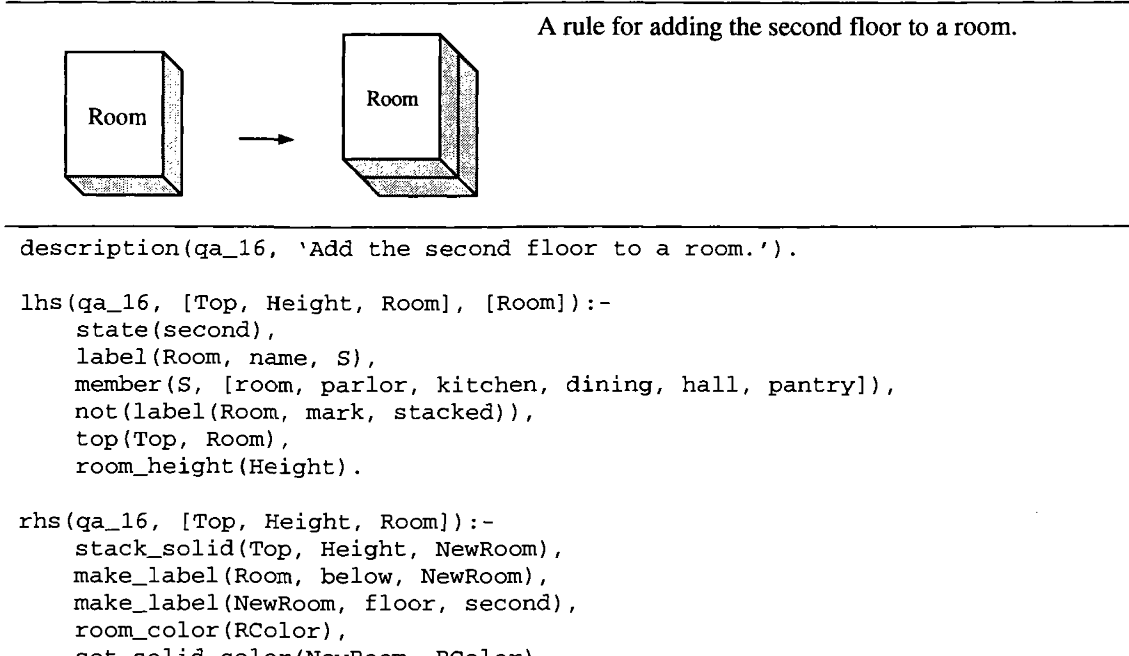
between such elements. The geometry contains vertex coordinates for polyhedral solids. The topology together with the geometry forms a boundary representation. The basic operations are the Euler operators for modifying the topology, vertex coordinate assignment for modifying the geometry, label addition and removal, and state change to indicate the current status; these are all specified as predicates in the declarative programming language, CLP(R). Figure 5-22 shows one such example of a basic operation. Supported by a set of basic operations, boundary solid grammars specify a subclass of shape grammars, which, in the context of this dissertation, specifies a sub-framework. Accordingly, the boundary representation is the underlying data structure, the algorithms are those for the basic operations, and the meta-language are expressions in CLP(R), and the subclass of shape grammars contains those describable by the basic operations. Figure 5-23 shows an example of a shape rule.



point_face operator: pulling a surface out into a point, correctly modifying the number of edges and vertices of the face.

```
point_face(Face, Height):-
    face_eh(Face, Eh),
    ccw_eh(Eh, LastEh),
    edgeh_v(Eh, V),
    face_normal(Face, Normal),
    face_center(Face, Center),
    mev(V, LastEh, VTop, EhBt),
    other_eh(EhBt, EhTb),
    scalar(Height, Normal, Direct),
    vecplus(Center, Direct, CTop),
    set_vertext(VTop, CTop),
    point_face_1(LastEh, Eh, VTop, EhTb).

point_face_1(EndEh, EndEh, _, _).
point_face_1(EndEh, Eh, VTop, EhTb):-
    cw_eh(Eh, NextEh),
    edgeh_v(NextEh, V),
    v_coord(V, C),
    mefl(V, Eh, VTop, EhTb, NewEhBt, _, _),
    other_eh(NewEhBt, NewEhTb),
    point_face_1(EngEh, NextEh, VTop, NewEhTb).
```

Figure 5-22: The *point_face* operator
Adapted from (Heisserman and Woodbury, 1993)



A rule for adding the second floor to a room.

```
description(qa_16, 'Add the second floor to a room.').

lhs(qa_16, [Top, Height, Room], [Room]):-
    state(second),
    label(Room, name, S),
    member(S, [room, parlor, kitchen, dining, hall, pantry]),
    not(label(Room, mark, stacked)),
    top(Top, Room),
    room_height(Height).

rhs(qa_16, [Top, Height, Room]):-
    stack_solid(Top, Height, NewRoom),
    make_label(Room, below, NewRoom),
    make_label(NewRoom, floor, second),
    room_color(RColor),
    set_solid_color(NewRoom, RColor).
```

Figure 5-23: A rule for adding the second floor to a room
Adapted from (Heisserman and Woodbury, 1993)

Chapter 6 Development of computation-friendly shape grammars

The development of a computation-friendly shape grammar is tied to a particular sub-framework in the general paradigm; that is, the shape grammar can be described by the meta-language supported by the underlying data structure of the corresponding sub-framework. In this chapter, I will use an example showing the details of developing a computation-friendly shape grammar. The sub-framework chosen is the rectangular sub-framework. The grammar to be designed is for the Baltimore Rowhouse (Hayward, 1981; Hayward and Belfoure, 2005), which is a target environment of the AutoPILOT project. To show contrast, this chapter starts with a shape grammar developed in the traditional way; this part is based on the work of Casey Hickerson (a team member on the AutoPILOT project). Then, the grammar will be modified (improved) to be computation-friendly. As a matter of fact, such a development procedure can serve a pattern (or strategy) for developing computation-friendly shape grammars in general; the traditional development stage focuses on capturing the target style, and computation-friendly modification focuses on the underlying computational characteristics.

6.1 The Baltimore rowhouse

The rowhouse become the dominant house type of Baltimore since its adoption in the eighteenth century (Hayward and Belfoure, 2005). Rowhousing exists in other American cities like Boston, Philadelphia, New York, Richmond, and St. Louis; but few are like Baltimore in that the city's spirit and identity are closely tied to this architectural form, whence the name — the Baltimore Rowhouse. The rowhouse had been persistently and tirelessly developed across two centuries, blossoming with the prosperity prior to World War II, suffering from the discrimination of postwar planners, and their recent redemption as humanely scaled housing. The two-story, three-bay house was an English invention in the beginning, plain in design without useless ornamentation, representing an efficient development policy that proved viable over decades of use. Then, this house form had been modified across time to meet the needs of different population groups of the city. Those for the wealthy were architect-designed; those for everyone else were built on speculation and, for the most part, designed by the builders themselves. To attract customers, and to make their product stand out among the thousands of rowhouses available, builders kept up with the latest styles, making modifications to cornice designs, window treatments, and the brick façade itself, adding bay windows, peaked roofs, stick-style porches, and carved or modeled embellishments. Across two centuries, the rowhouse history of Baltimore involves both changes and lack of changes; the changes relate the development of the city, and the lack of changes forms the style of the Baltimore Rowhouse.

6.2 Creation of a shape grammar

The process of creating a shape grammar to describe an existing set of designed objects involves three fundamental tasks: i) identifying the set of design patterns that most succinctly constitutes those objects, ii) formalizing those patterns as a set of shape rules, and iii) organizing the shape rules so that the grammar generates as many valid designs as possible while producing as few invalid design as possible.

To find patterns within a set of designed objects, one must look toward examples. Still, one cannot understand the process used to design a set of objects from examples alone. One first needs to understand the factors that motivate the design process. Having this information helps to identify the minimal set of design patterns that characterizes the design process.

After identifying the patterns, one translates these patterns into a set of shape rules. The goal of this step in the process is to use as few rules as possible to create as many valid designs as possible while keeping the rules as simple as possible. Although how well a grammar meets this goal could be a subjective matter, it is usually not so hard to identify the better one from candidate solutions. For example, to generate a configuration with front and back divisions and a stair (solid grey) in a connecting hallway (Figure 6-1a), the rule of Figure 6-1b is obviously too 'heavy', which is usually discouraged. On the other side, using rule of Figure 6-1c first, and then rule of Figure 6-1d, is a much better solution; the former splits the blocks, and the latter adds a staircase. Note that the rule of Figure 6-1c may still need to be decomposed into 'lighter' rules.

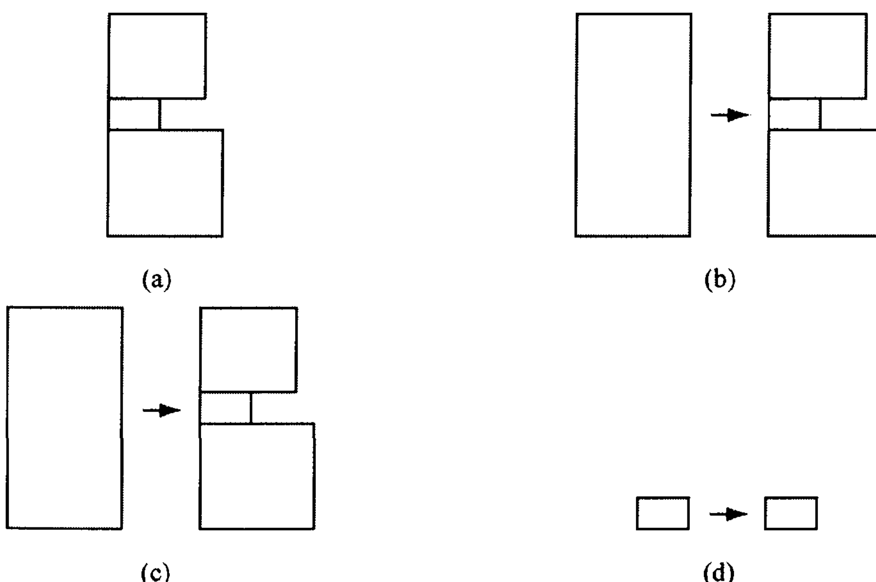


Figure 6-1: Identification of the better solution between two

Generative rules are created to build 'on top of one another' so that a starting shape, e.g., the undivided building block, can be transformed into a possible configuration using a sequence of rule applications. At present, one cannot always determine a priori whether a grammar correctly generates all valid objects of a type; likewise, one cannot always determine a priori whether a grammar creates valid design objects. As a result, one generally evaluates the validity of a shape grammar through trial and error: applying every possible sequence of rules to the initial shape. In reality, the number of configurations generated by a sufficiently powerful grammar is so large that one cannot test every possible design. Still, an astute observer (or a trained grammarist) might reasonably determine whether a grammar is likely to be valid. Note that the above observation agrees with the conclusion made in Section 2.3.5 that the problem of parsing a configuration against a shape grammar is computationally unsolvable in general.

6.3 A traditional rowhouse grammar

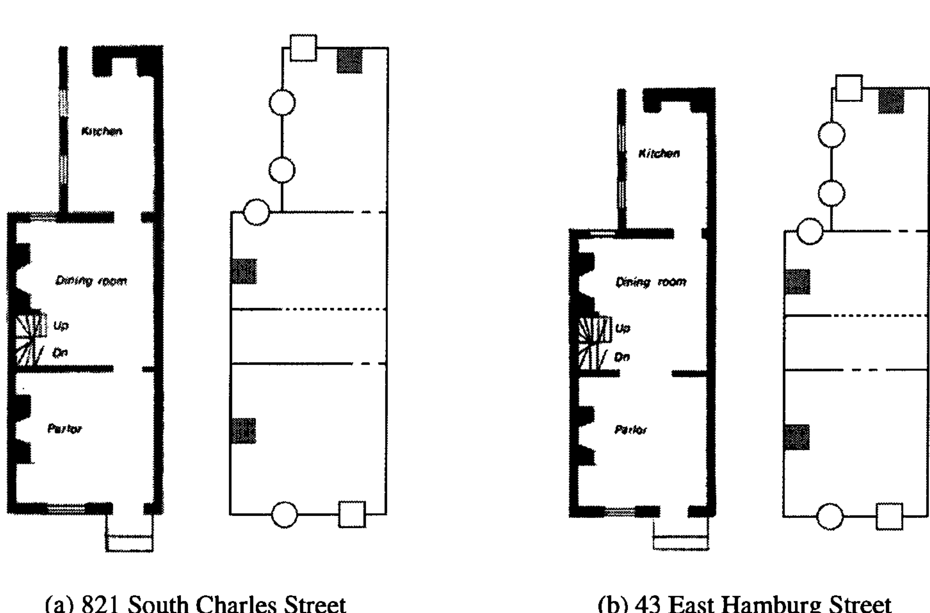
There are two main resources used to develop the rowhouse grammar: the article *Urban Vernacular Architecture in Nineteenth-Century Baltimore* by (Hayward, 1981), and the monograph *The Baltimore Rowhouse* by (Hayward and Belfoure, 2005). The former serves as the primary basis, providing the detailed information about the rowhouse morphology; the latter is auxiliary, providing a more detailed discussion of the cultural factors that have influenced the morphology. Limited by the information available, the focus is on the configuration of the first floor, while the mechanism applies the grammar development for other floors. Figure 6-2 shows a set of rowhouse samples from the Federal Hill district of Baltimore, and Figure 6-3 shows the photographs of some samples.

6.3.1 Abstract shape representation

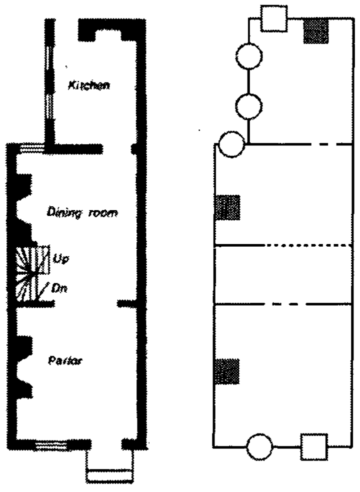
To identify the patterns of the rowhouses, a shape representation (Figure 6-2) is adopted. The shape representation of a plan is essentially an abstracted form of the actual plan. The shape representation emphasizes topological information about a plan, e.g., the relationship between spaces, rather than the details of the building

itself. Spaces are simplified for clarity while some building features, e.g., wall thickness, are more-or-less eliminated.

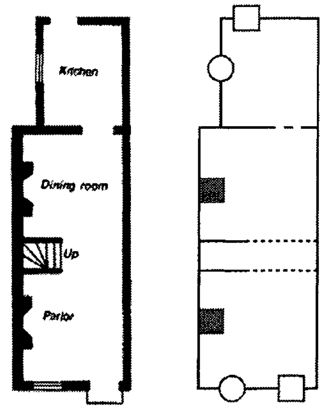
The difference between this representation and that used by other shape representations is the addition of icons to present features. Pertinent exterior features are represented through graphic conventions: a window is represented by a blue circle; a door, by a hollow green rectangle; a fireplace as a solid red rectangle; and a staircase as a solid grey area. Interior features, such as doorways between rooms, are shown in the shape representations as dashed lines. For the purpose of layout determination, other interior features are not incorporated here.



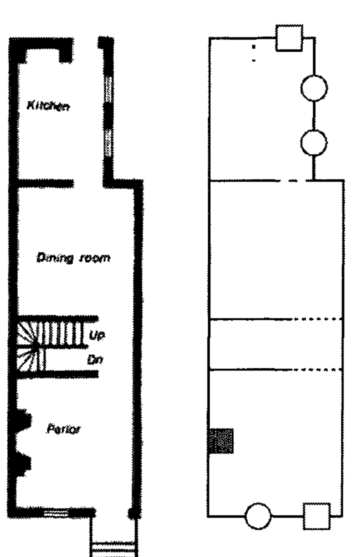
(a) 821 South Charles Street (b) 43 East Hamburg Street



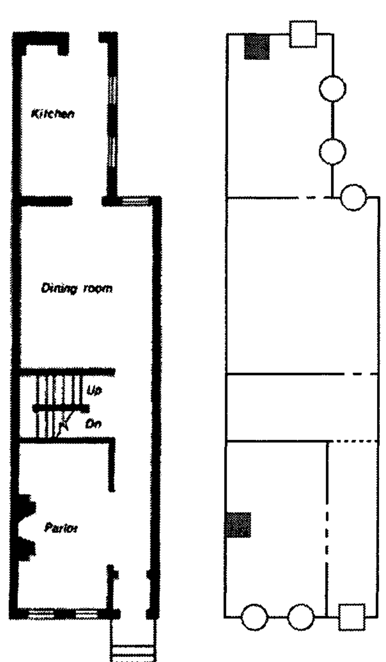
(c) 21 East Wheeling Street



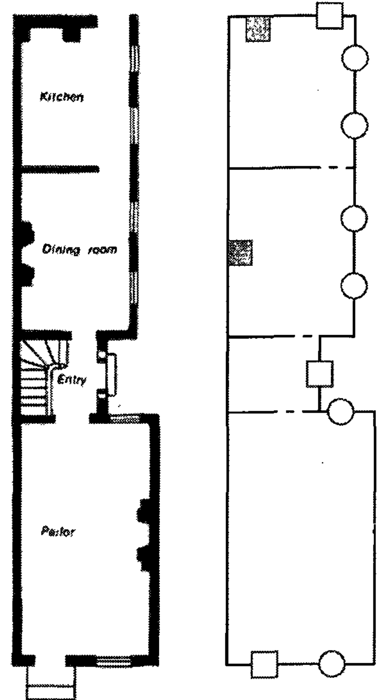
(d) 1028 Patapasco Street

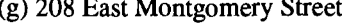


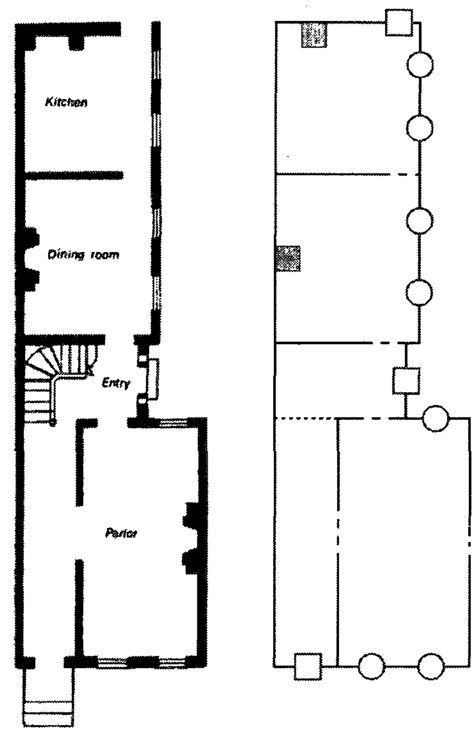
(e) 401 Grindall Street



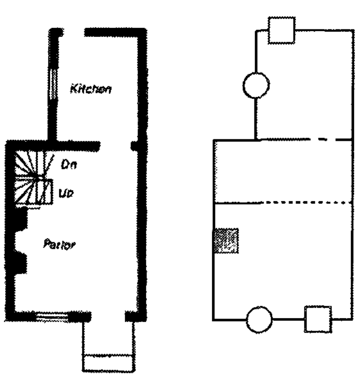
(f) 1029 South Hanover Street



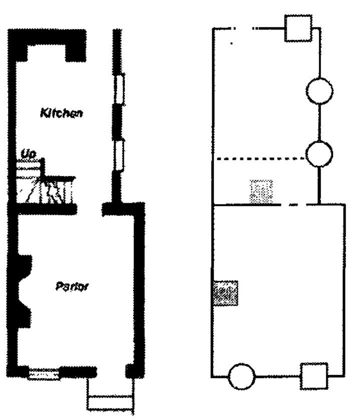

(g) 208 East Montgomery Street



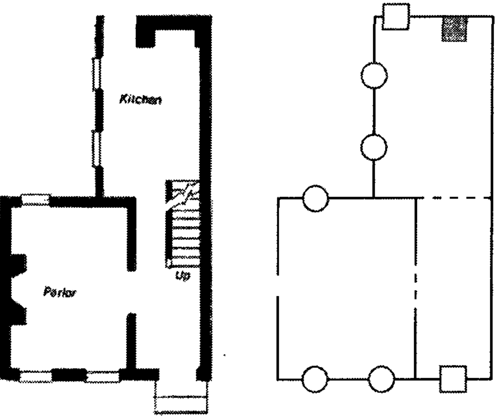
(h) 236 East Montgomery Street



(i) 14 West Cross Street



(j) 819 South Charles Street



(k) 3 East Montogomery Street

Figure 6-2: The shape representation of rowhouse samples

(a) 1-11 East Montgomery Street (1 is on the right)

(b) 202-208 East Montgomery Street

(c) 815-829 South Charles Street (815 is on the left)

(d) 3-25 East Wheeling Street

Figure 6-3: Photos of sample rowhouses
Adapted from (Hayward, 1981)

6.3.2 Variation in interior configuration

As presented in Hayward's article, the rowhouses of Baltimore show little morphological variation. The lack of significant variation is clearly visible when comparing three buildings of Figure 6-2a~c.

The building of Figure 6-2a, located at 821 South Charles Street, was constructed in 1818 and is of the 'two-and-a-half-story federal style'. The building of Figure 6-2b, located at 43 East Hamburg Street, was constructed in 1838 and is of a later variation of the federal style. The building of Figure 6-2c, located at 21 East Wheeling Street, was constructed in 1850 and is of the 'two-story-plus-attic Greek revival style'. Although these three buildings were constructed during decades apart from one another and are of nominally distinct styles, each follows the same basic plan.

This does not suggest that rowhouses show no variation whatsoever. Rather, the variation they show is fairly uniform and follows well-defined patterns. For example, we can identify at least five major variations across the entire corpus:

(i) For safety and convenience, rowhouses are divided into two blocks: a main block toward the street and a kitchen block toward the rear (Figure 6-4). The two blocks may be directly adjacent to one another, as diagrammed on the left, or they may connect to one another through a short corridor, as diagrammed on the right.

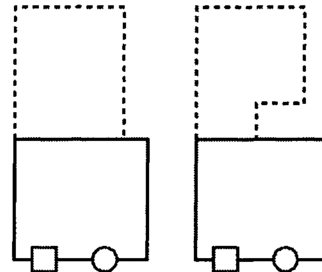


Figure 6-4: Block configurations

(ii) The main block of a rowhouse is two or three bays wide (Figure 6-5). A bay, in this context, is defined by a single window or door on the front façade. In a two-bay-wide house, as diagrammed on the left, the front door enters directly into a parlor. In a three-bay-wide house, as diagrammed on the right, the front door enters into a hallway, which is directly adjacent to a parlor.

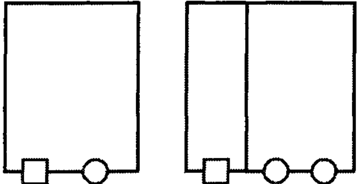


Figure 6-5: Width configurations

(iii) The main block of a rowhouse is one or two rooms deep (Figure 6-6). In a two-room-deep main block, as diagrammed on the left, the front room is a parlor and the back room is a dining room. In a one-room-deep main block, as diagrammed on the right, the parlor may serve as a dining room.

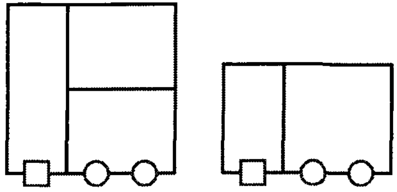


Figure 6-6: Depth configurations

(iv) The main staircase can exist in an assortment of locations within a rowhouse, diagrammed below (Figure 6-7): a) in the parlor, toward the back of the house; b) in the dining room, toward the front of the house; c) between the dining and parlor; d) in the hallway, occupying its entire width; e) in the hallway, toward the outer side of the house; and f) in the kitchen block, toward the front of the house.

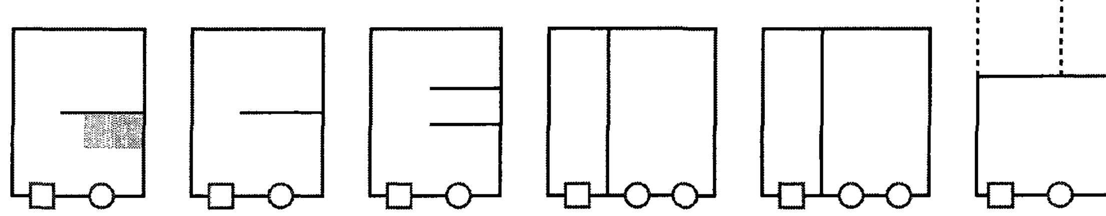


Figure 6-7: Stair configurations

(v) Rowhouses can follow an assortment of story and basement configurations: a) two full stories, but no attic or dormer story; b) two full stories and a dormer story; c) two full stories and an attic; d) with a full basement, partially underground; e) with a full basement, entirely underground; and f) with no basement.

6.3.3 Patterns identified

The following are patterns identified:

Division between main block and kitchen block

- Front and back portions connected by a mutual wall – more common (Figure 6-2a~f, i~k)
- Front and back portions connected by a corridor – less common (Figure 6-2g, h)

- Isolated front portion – relatively rare

Overall width

- Two bays width – more common (Figure 6-2a~e, g, i, j)
- Three bays width – less common (Figure 6-2f, h, k)

Entryway configuration

- Enter into the parlor – all two-bay-wide houses follow this pattern (Figure 6-2a~e, g, i, j)
- Enter into a dedicated hallway that runs full depth of the front block (Figure 6-2h, k)
- Enter into a dedicated hallway that runs partial depth of the front block (Figure 6-2f)

Location of dedicated dining room

- In main block – more common (Figure 6-2a~f)
- In kitchen block – less common (Figure 6-2g~h)

Depth of front portion

- One space deep – a parlor (Figure 6-2g~h), or a combined parlor and dining room (Figure 6-2i~k)
- Two spaces deep – a parlor and a dining room (Figure 6-2a~c)
- Three spaces deep – a parlor, a dedicated stair, and a dining room (Figure 6-2d~f)

Stair location

- In the front division
    - On the other side of the front entrance
        - Between the separate parlor and dining room (Figure 6-2d~f)
        - Within a combined parlor and dining room, toward the back (Figure 6-2i)
        - Within separate dining room, toward the front (Figure 6-2a~c)
    - On the same side of the front entrance

      - In the hall way (Figure 6-2k)
- In the back division
  - Within the kitchen, toward the front (Figure 6-2j)
- In the connection between front and back
  - On the same side as the front entrance (Figure 6-2g~h)

Stair shape

- U-shaped (Figure 6-2a~f, i)
- L-shaped (Figure 6-2g~h, j)
- Straight, bound by a wall on one side (Figure 6-2k)

Above-ground floor variations

- Two stories (Figure 6-2d, e)
- Two full stories and a 'half' dormer story (Figure 6-2a, b, j, k)
- Two full stories and an attic (Figure 6-2c, i)
- Three stories (Figure 6-2g, h)

Style

- Federal (Figure 6-2a, b, d, j, k)
- Greek Revival (Figure 6-2c, g)
- Italianate (Figure 6-2e, f, h)

Note that, of all the different patterns visible within the rowhouse, stairs present the most intriguing set of combinations. In general, stairs exist in a distinct space that can take one of two forms, a literal room, separated from other rooms by walls, or in a 'phenomenal' room, which exists within a literal room and is defined by the stair itself. Within the shape representation (Figure 6-2), the boundaries of phenomenal rooms are designed with dotted lines.

6.3.4 The rowhouse grammar

The Baltimore Rowhouse grammar consists of 52 shape rules that generate first floor configurations with features of stairs, fireplaces, windows, exterior doors and interior doors. Rules are organized into phases, progressing from the major configurations

that constrain the design process to minor configurations that follow logically from other configurations, namely: I) Block generation: rules 1~4; II) Space generation: rules 5~7; III) Stair generation: rules 8~17; IV) Fireplace generation: rules 18~22; V) Space modification: rules 23~24; VI) Front door and window generation: rules 25~29; VII) Middle and back door, and window generation: rules 30~39; and VIII) Interior door generation: rules 40~52.

Rules are marked as either required (*req*) or optional (*opt*). Required rules must be applied if applicable while optional rules may be applied at the interpreter's discretion. The decision whether to apply an optional rule directly impacts the overall design. In effect, the final design is determined by the set of optional rules that were applied. Whenever a rule is applied, it must be applied exhaustively; that is, the rule must be applied to every subshape that matches the rule's left-hand-shape. Finally, rules must be applied in sequence: after Rule x has been applied exhaustively, only Rules $x+1$ and greater may be applied.

Like other shape grammars, labels are used in two ways: to control where shape rules may apply, and to ensure that mutually exclusive rules cannot be applied to the same design. Spaces and stairs are labeled with two or three characters that indicate the general location of the space or stair within the house. For instance, *Rfb* indicates a room in the front block of the house that is oriented toward the back, a dining room. Wall labels are always of the form *x*(*y*) where x is a label for a space that the wall bounds (or P in the case of certain perimeter walls) and y is a one letter code indicating the side of the space the wall defines. For example, the front wall of the room labeled *Rfb* is labeled *Rfb*(*f*). Within some rules, variables are used to match more than one label: the character * matches any string of characters while the string$\{x \mid y\}$ matches the strings x or y. ***Boolean*** global labels are used to ensure that mutually exclusive rules are not applied with default value *false*. Figure 6-8 shows all the shape rules. A sample derivation is given in Figure 6-9.

Phase I: Block generation

1) Generate the front block
2) Mirror the front block

3) Generate the back block
4) Generate the middle block

Phase II: Space generation

5) Generate a hallway in the front block
6) Generate two spaces within the front block
7) Generate two spaces within the back block

Phase III: Stair generation

8) Generate stair at the back wall of a single-space front block
9) Generate stair between the two spaces of a double-space front block
10) Modify the stair generated by Rule 9 if it runs the entire house width
11) Generate partial-width stair in the front hallway
12) Generate full-width stair in the front hallway
13) Generate stair in the middle block
14) Generate stair at the front of a single-space back block
15) Generate partial-width stair between the two spaces of a double-space back block
16) Generate full-width stair between the two spaces of a double-space back block
17) Generate accessory stair on the back wall of the back room of a back block

Phase IV: Fireplace generation

18) Generate required front-block fireplaces
19) Generate optional front-block fireplaces
20) Generate back-block fireplaces
21) Generate back-block fireplaces on the back wall
22) Generate back-block fireplaces on a side wall

Phase V: Space modification

23) Modify the back room of a front block if the front hallway does not adjoin the middle or back block

24) Generate a service stair behind a partial-width stair in the front hallway

Phase VI: Door and window generation

25) Generate a hall way in the front of the back block, removing the fireplace
26) Generate the exterior door into the front hallway of a three-bay configuration
27) Generate an entry vestibule in the front hallway of a three-bay configuration
28) Generate the front windows of a three-bay configuration
29) Generate the front door and window for a two-bay configuration

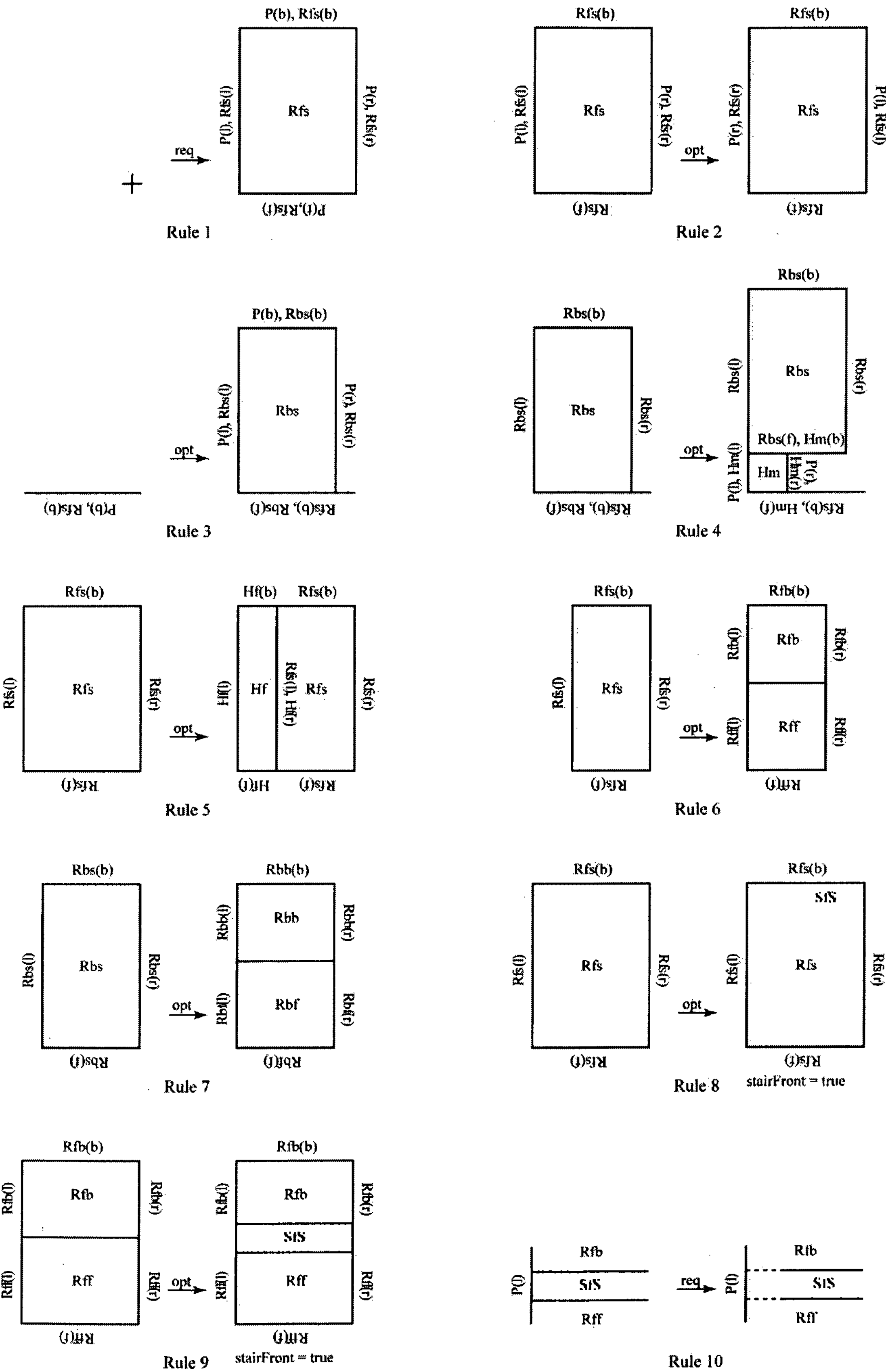
Phase VII: Middle and back door, and window generation

30) Generate a window on the back wall of the front block
31) Modify the number of windows on the back wall of the front block from one to two
32) Generate an exterior door into the middle block
33) Generate a window in back-block spaces
34) Modify the number of windows in back-block spaces from one to two
35) Modify the number of windows in back-block spaces from two to three
36) Generate an exterior door on the side wall of the back-most space when a stair is present on the back wall
37) Generate an exterior door on the 'right' side of a back wall
38) Generate an exterior door on the 'left side of a back wall
39) Generate an exterior door in a back block with partial-width stair

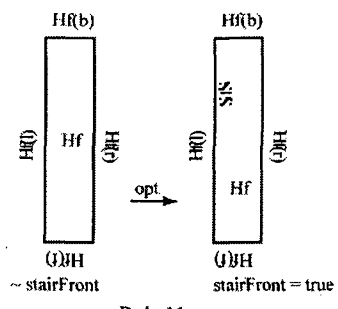
Phase VIII: Interior door generation

40) Generate interior doors connecting the front, middle and back blocks
41) Generate an interior door connecting front hallway and back block when there is no middle block
42) Generate an interior door connecting front and back blocks when a stair is present on the front wall of the back block

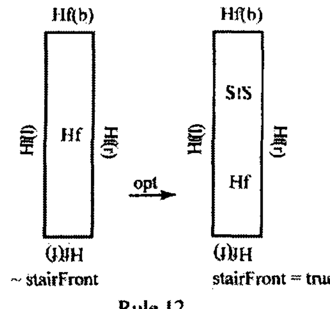
43) Generate a left-side interior door connecting the front and back blocks when there is no middle block or front hallway
44) Generate a right-side interior door connecting the front and back blocks when there is no middle block or front hallway
45) Generate an interior door between the front and back spaces in the front block
46) Generate interior doors between a front space and front hallways when the front block contains two divided hallways (This occurs when the front hallway contains a full-width stair and when the front block contains a separate service stair.)
47) Generate asymmetric interior doors between the hallway and spaces in the front block
48) Generate symmetric interior doors between the hall way and spaces in the front block
49) Generate interior doors when the back block has a hallway
50) Generate an interior door between the front and back spaces in the back block
51) Generate interior doors between front, middle and back spaces in the back block
52) Generate an interior door between adjacent front hallways (after Rule 46)



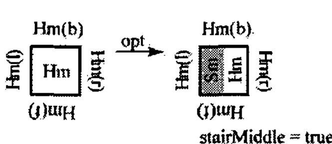
P(b), Rfs(b)
P(l), Rfs(l)
Rfs
P(r), Rfs(r)
req
P(f), Rfs(f)
Rule 1
Rfs(b)
P(l), Rfs(l)
Rfs
P(r), Rfs(r)
Rfs(f)
opt
Rfs(b)
P(r), Rfs(r)
Rfs
P(l), Rfs(l)
Rfs(f)
Rule 2
P(b), Rfs(b)
P(b), Rbs(b)
P(l), Rbs(l)
Rbs
P(r), Rbs(r)
opt
Rfs(b), Rbs(f)
Rule 3
Rbs(b)
Rbs(l)
Rbs
Rbs(r)
Rfs(b), Rbs(f)
opt
Rbs(b)
Rbs(l)
Rbs
Rbs(r)
Rbs(f), Hm(b)
Hm
P(l), Hm(l)
P(r), Hm(r)
Rfs(b), Hm(f)
Rule 4
Rfs(b)
Rfs(l)
Rfs
Rfs(r)
Rfs(f)
opt
Hf(b)
Rfs(b)
Hf(l)
Hf
Rfs(l), Hf(r)
Rfs
Rfs(r)
Hf(f)
Rfs(f)
Rule 5
Rfs(b)
Rfs(l)
Rfs
Rfs(r)
Rfs(f)
opt
Rfb(b)
Rfb(l)
Rfb
Rfb(r)
Rff(l)
Rff
Rff(r)
Rff(f)
Rule 6
Rbs(b)
Rbs(l)
Rbs
Rbs(r)
Rbs(f)
opt
Rbb(b)
Rbb(l)
Rbb
Rbb(r)
Rbf(l)
Rbf
Rbf(r)
Rbf(f)
Rule 7
Rfs(b)
Rfs(l)
Rfs
Rfs(r)
Rfs(f)
opt
Rfs(b)
SIS
Rfs(l)
Rfs
Rfs(r)
Rfs(f)
stairFront = true
Rule 8
Rfb(b)
Rfb(l)
Rfb
Rfb(r)
Rff(l)
Rff
Rff(r)
Rff(f)
opt
Rfb(b)
Rfb(l)
Rfb
Rfb(r)
SIS
Rff(l)
Rff
Rff(r)
Rff(f)
stairFront = true
Rule 9
P(l)
Rfb
SIS
Rff
req
P(l)
Rfb
SIS
Rff
Rule 10



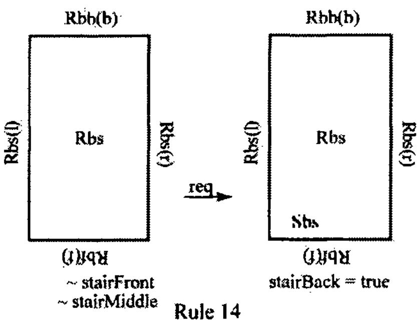
Rule 11



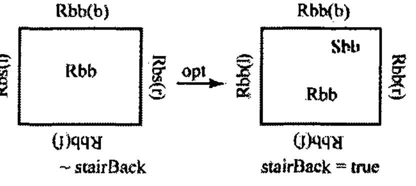
Rule 12



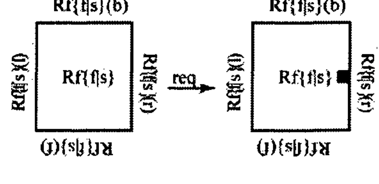
Rule 13



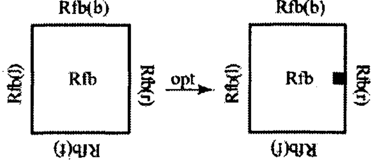
Rule 14



Rule 15

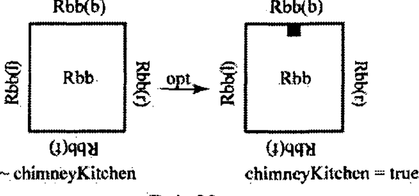


Rule 16



Rule 17

Rule 18

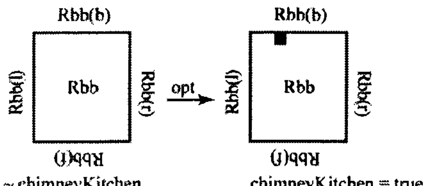


Rf{f|s}(b)
Rf{f|s}(l)
Rf{f|s}
Rf{f|s}(r)
Rf{f|s}(f)
req
Rf{f|s}(b)
Rf{f|s}(l)
Rf{f|s}
Rf{f|s}(r)
Rf{f|s}(f)

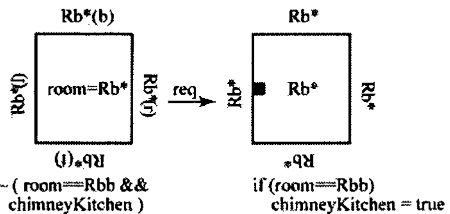
Rfb(b)
Rfb(l)
Rfb
Rfb(r)
Rfb(f)
opt
Rfb(b)
Rfb(l)
Rfb
Rfb(r)
Rfb(f)

Rule 19

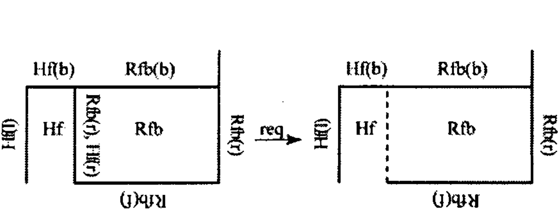
Rule 20



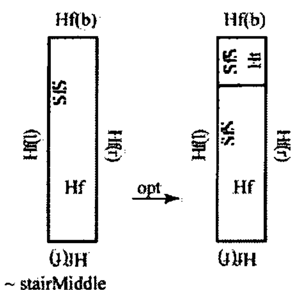
Rule 21



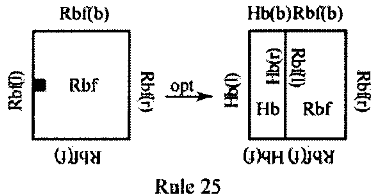
Rule 22



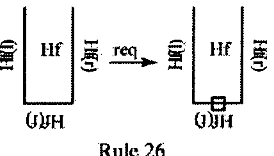
Rule 23



Rule 24



Rule 25



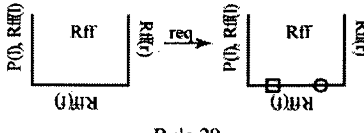
Rule 26

Hf(l) Hf Hf(r) opt Hf(l) Hf Hf(r)
Hf(f) Hf(f)

Rule 27

Hf(l) Hf Rff(l), Hf(r) Rff Rff(r) req Hf(l) Hf Rff(l), Hf(r) Rff Rff(r)
Hf(f) Rff(f) Hf(f) Rff(f)

Rule 28

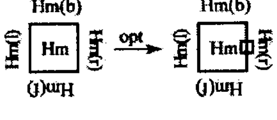


Rule 29

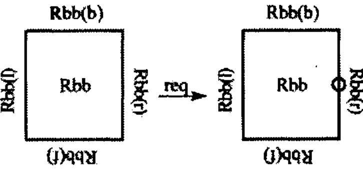
Rfb(b) Rfb(r) opt Rfb(b) Rfb(r)

Rule 30

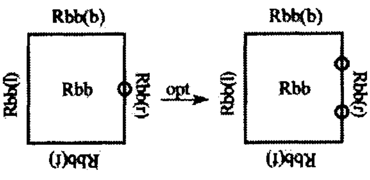
Rule 31



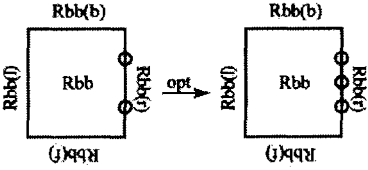
Rule 32



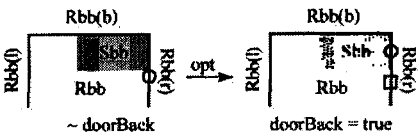
Rule 33



Rule 34



Rule 35

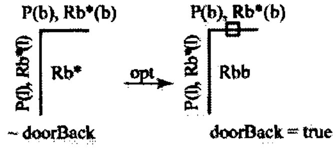


Rule 36

P(b), Rb*(b) Rb* Rb*(r) opt P(b), Rb*(b) Rb* Rb*(r)

~ doorBack doorBack = true

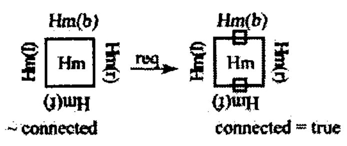
Rule 37



Rule 38

Rbm(b) Rbm Rbm(r) Rbm(f) req Rbm(b) Rbm Rbm(r) Rbm(f)

Rule 39



Rule 40

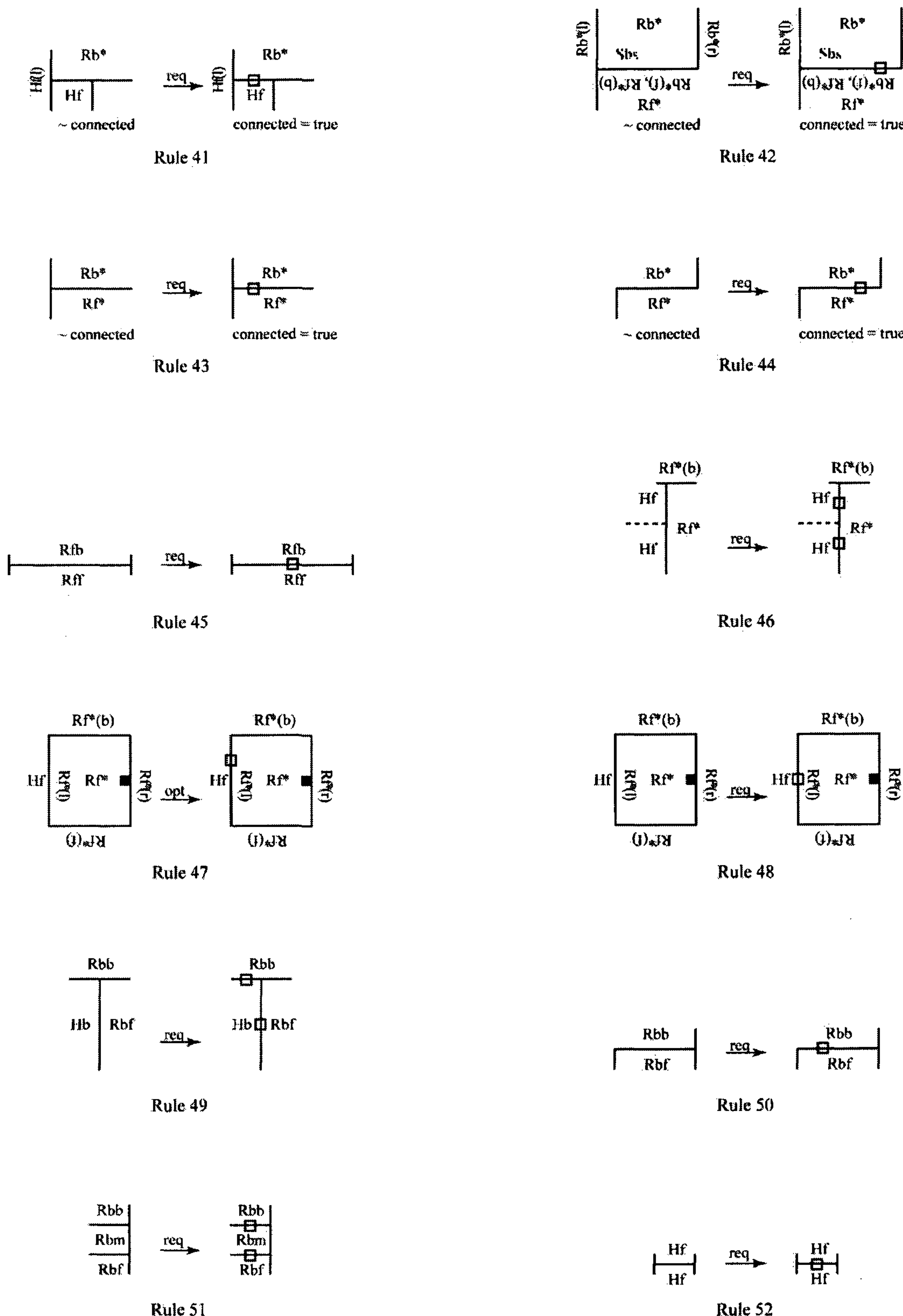


Figure 6-8: The traditional rowhouse grammar

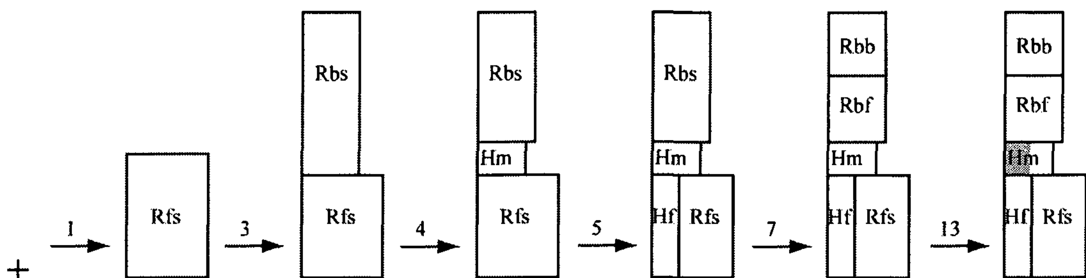


Figure 6-9: Derivation of 236 East Montgomery Street by the old rowhouse grammar

6.4 A new computation-friendly rowhouse grammar

In many aspects, the above grammar is not computation-friendly. As with many other traditional shape grammars, the focus is on generating all possible designs while leaving the conditions that apply to shape rules not fully specified. This becomes particularly evident when applying shape grammars to determine building interior layouts. A building feature input posits constraints over possible layouts, which further posits constraints on which shape rules apply, through comparing the conditions of the shape rules against the available constraints in the current configuration.

In order to apply the rowhouse grammar to determine the interior layout from the feature inputs of a rowhouse, a new computer-friendly version of the grammar has to be developed. For convenience, the traditional rowhouse grammar is named as *the old rowhouse grammar*, *the new rowhouse grammar* for the computer-friendly version. To focus on how to make a traditionally designed shape grammar computation-friendly, we consider only a subset of the corpus, namely, working-class rowhouse, excluding large, luxurious rowhouse, which are considered in the original grammar. Unlike their luxurious counterparts, a working-class rowhouse usually have a unique staircase on the first floor. Figure 6-2 actually shows all cases under consideration. Note that the mechanism of generating fireplace is essentially identical to generating interior doors or staircases. Therefore, we omit the shape rules for generating fireplaces. Moreover, for layout determination, as the feature inputs include windows and exterior doors, rules relating to generating windows and exterior doors are not considered here. A computation-friendly shape grammar needs

to specify the allowable transformation. In the new rowhouse grammar, the allowable transformation is translation, horizontal reflection, or combination. Under the context of interior layout determination, instead of a point on a two-dimensional Cartesian coordinate system, the initial shape is a shape from the pre-processing of the feature inputs, in particular the footprint. To be exact, the initial shape contains a list of rectangular blocks, as well as the 2D bounds of windows and doors. Such an initial shape helps avoid the complexity of pruning and fixing of the underlying layout tree (see Section 7.7.2 for a detailed explanation). As with the original grammar, rule application is sequential. Table 6-1 shows the details of the new shape rules; in particular, the rightmost column shows the description of the corresponding shape rule in the format of meta-language. Figure 6-10 shows a sample derivation of the new rowhouse grammar.

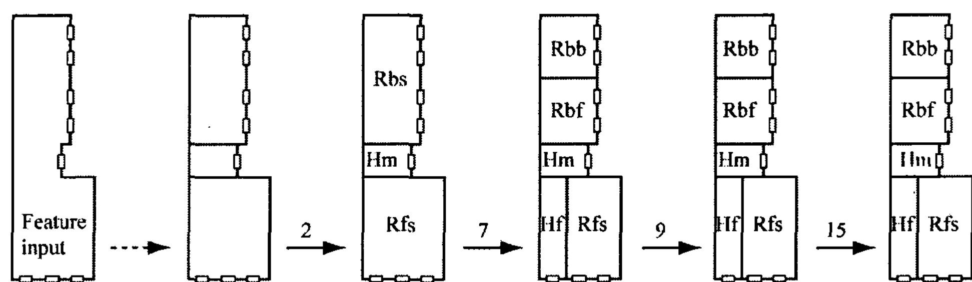
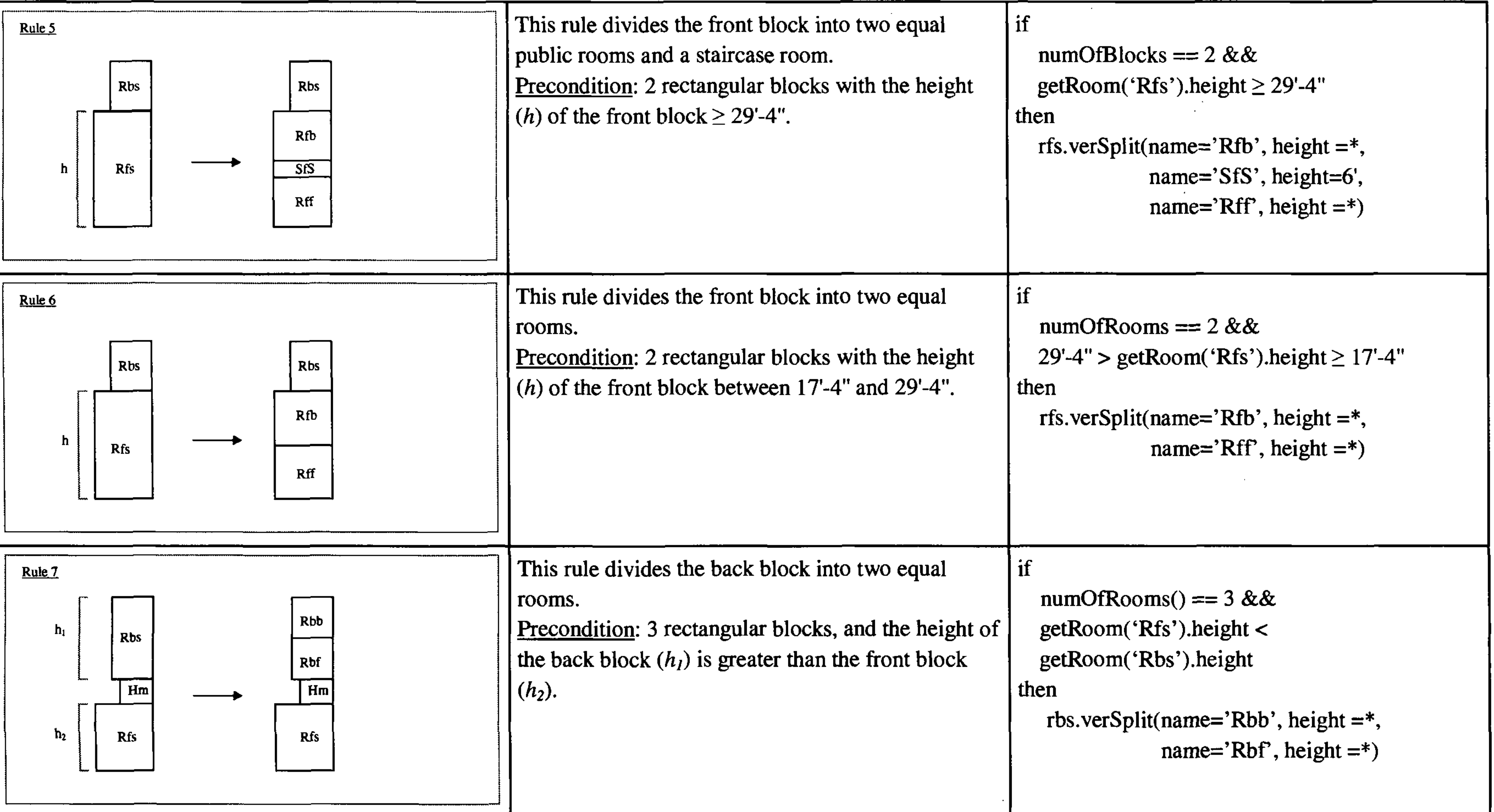


Figure 6-10: Derivation of 236 East Montgomery Street by the new rowhouse grammar

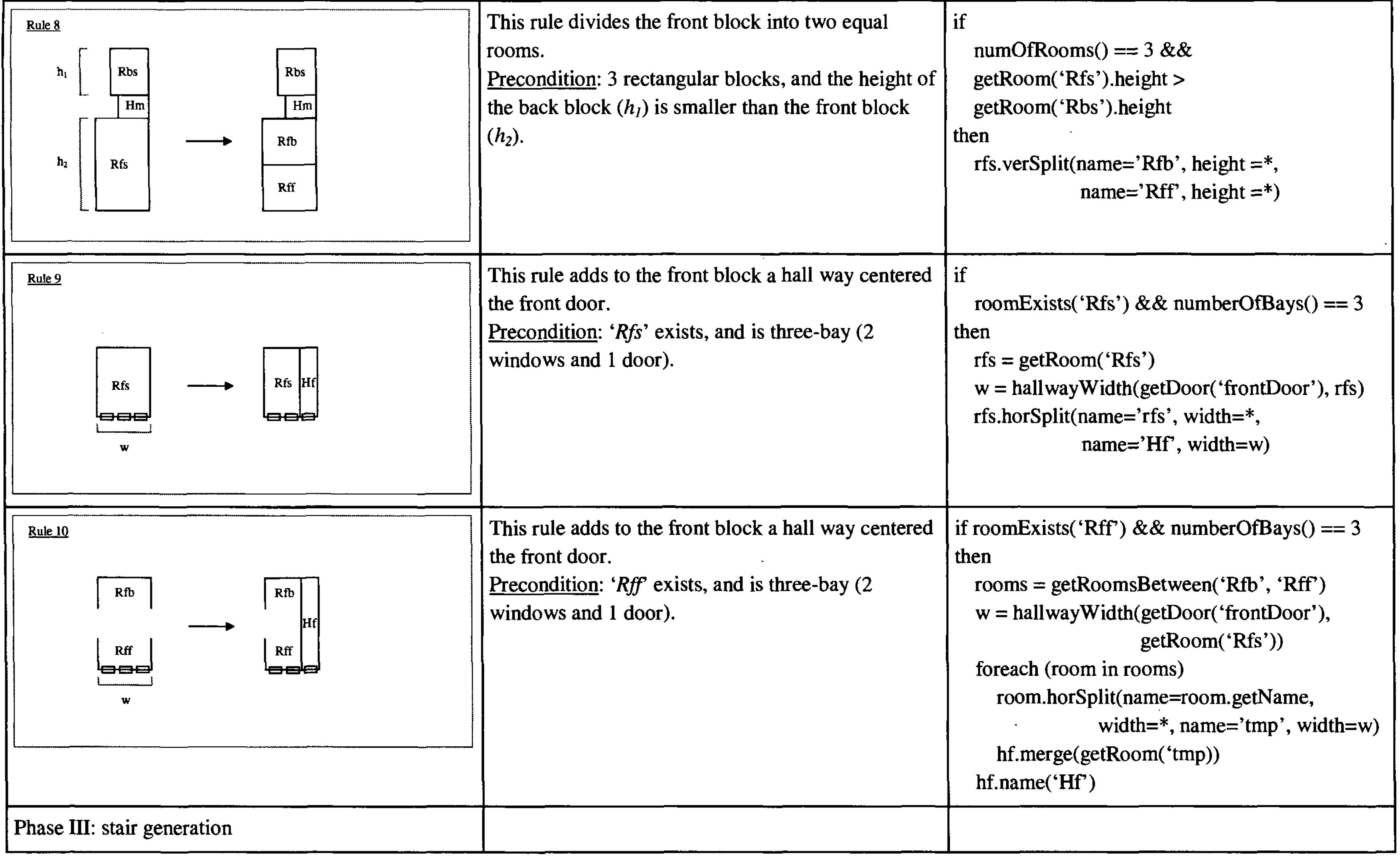
Table 6-1: Shape rules of the new computation-friendly rowhouse grammar

Initial Y X	On pre-processing the building feature inputs, the initial shape is a list of rectangular blocks, which is a decomposition of the footprint, as well as 2D bounds of windows and doors, which have been assigned to the rectangular blocks. All lines are either X- or Y-Axis aligned. The line at the bottom corresponds to the front of the building. The column on the left shows two typical examples.	
Phase I: block (mass) generation		
Rule 1 Rbs Rfs	This rule assigns names to the front and back blocks. <u>Precondition</u>: 2 rectangular blocks	if numOfBlocks == 2 then createRoom(rect=getBackBlock(), name='Rbs') createRoom(rect=getFrontBlock(), name='Rfs')

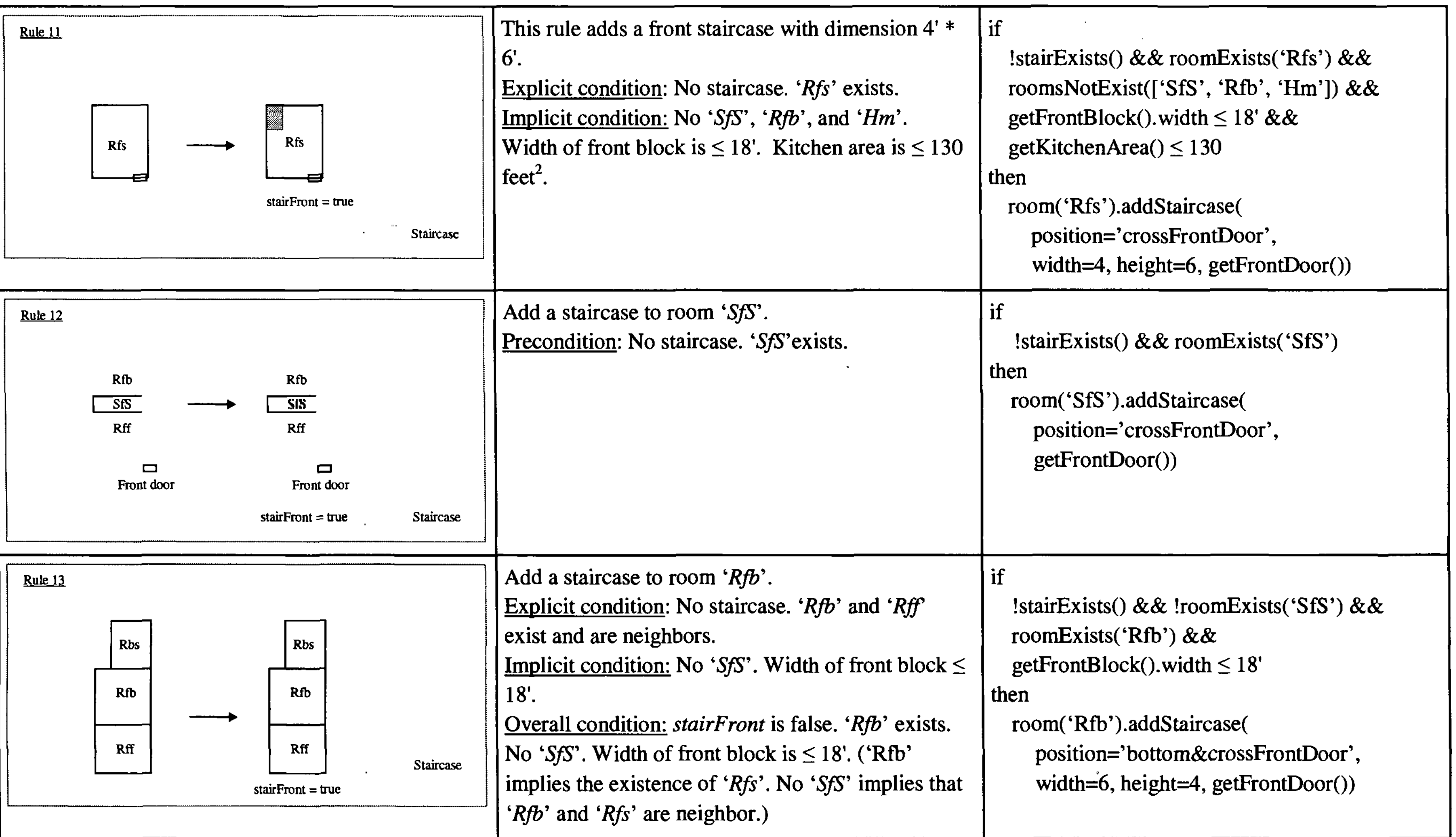
Rule 2 Rbs Hm Rfs	This rule assigns names to the front, middle, and back blocks. Precondition: 3 rectangular blocks	if numOfBlocks == 3 then createRoom(rect= getBackBlock(), name='Rbs') createRoom(rect= getMidBlock(), name='Hm') createRoom(rect= getFrontBlock(), name='Rfs')
Rule 3 Rbs Rfs isRightSideStraight = true	Blocks of rowhouses are either left- or right- aligned with a straight line. This information is captured by the attribute, *isRightSideStriaght*. This rule sets the boolean attribute, *isRightSideStriaght*.	rbs = getRoom('Rbs') rfs = getRoom('Rfs') if rbs.cornerLR.X == rfs.cornerLR.X then isRightSideStriaght = true
Rule 4 Rfs isFrontDoorRight = true	Description: This rule sets the boolean attributes, *isFrontDoorRight*.	rfs = getRoom('Rfs') front = getDoor('frontDoor') if |rfs.cornerLL.X - front.cornerLL.X| > |rfs.cornerLR.X - front.cornerLR.X| then isFrontDoorRight = true
Phase II: space generation		



Rule	Description	Code
Rule 5 (diagram: Rbs, Rfs, h → Rbs, Rfb, SfS, Rff)	This rule divides the front block into two equal public rooms and a staircase room. Precondition: 2 rectangular blocks with the height (h) of the front block ≥ 29'-4".	if numOfBlocks == 2 && getRoom('Rfs').height ≥ 29'-4" then rfs.verSplit(name='Rfb', height =*, name='SfS', height=6', name='Rff', height =*)
Rule 6 (diagram: Rbs, Rfs, h → Rbs, Rfb, Rff)	This rule divides the front block into two equal rooms. Precondition: 2 rectangular blocks with the height (h) of the front block between 17'-4" and 29'-4".	if numOfRooms == 2 && 29'-4" > getRoom('Rfs').height ≥ 17'-4" then rfs.verSplit(name='Rfb', height =*, name='Rff', height =*)
Rule 7 (diagram: Rbs, Hm, Rfs, h_1, h_2 → Rbb, Rbf, Hm, Rfs)	This rule divides the back block into two equal rooms. Precondition: 3 rectangular blocks, and the height of the back block (h_1) is greater than the front block (h_2).	if numOfRooms() == 3 && getRoom('Rfs').height < getRoom('Rbs').height then rbs.verSplit(name='Rbb', height =*, name='Rbf', height =*)



Rule 8	This rule divides the front block into two equal rooms. Precondition: 3 rectangular blocks, and the height of the back block (h_1) is smaller than the front block (h_2).	if numOfRooms() == 3 && getRoom('Rfs').height > getRoom('Rbs').height then rfs.verSplit(name='Rfb', height =*, name='Rff', height =*)
Rule 9	This rule adds to the front block a hall way centered the front door. Precondition: '*Rfs*' exists, and is three-bay (2 windows and 1 door).	if roomExists('Rfs') && numberOfBays() == 3 then rfs = getRoom('Rfs') w = hallwayWidth(getDoor('frontDoor'), rfs) rfs.horSplit(name='rfs', width=*, name='Hf', width=w)
Rule 10	This rule adds to the front block a hall way centered the front door. Precondition: '*Rff*' exists, and is three-bay (2 windows and 1 door).	if roomExists('Rff') && numberOfBays() == 3 then rooms = getRoomsBetween('Rfb', 'Rff') w = hallwayWidth(getDoor('frontDoor'), getRoom('Rfs')) foreach (room in rooms) room.horSplit(name=room.getName, width=*, name='tmp', width=w) hf.merge(getRoom('tmp')) hf.name('Hf')
Phase III: stair generation		

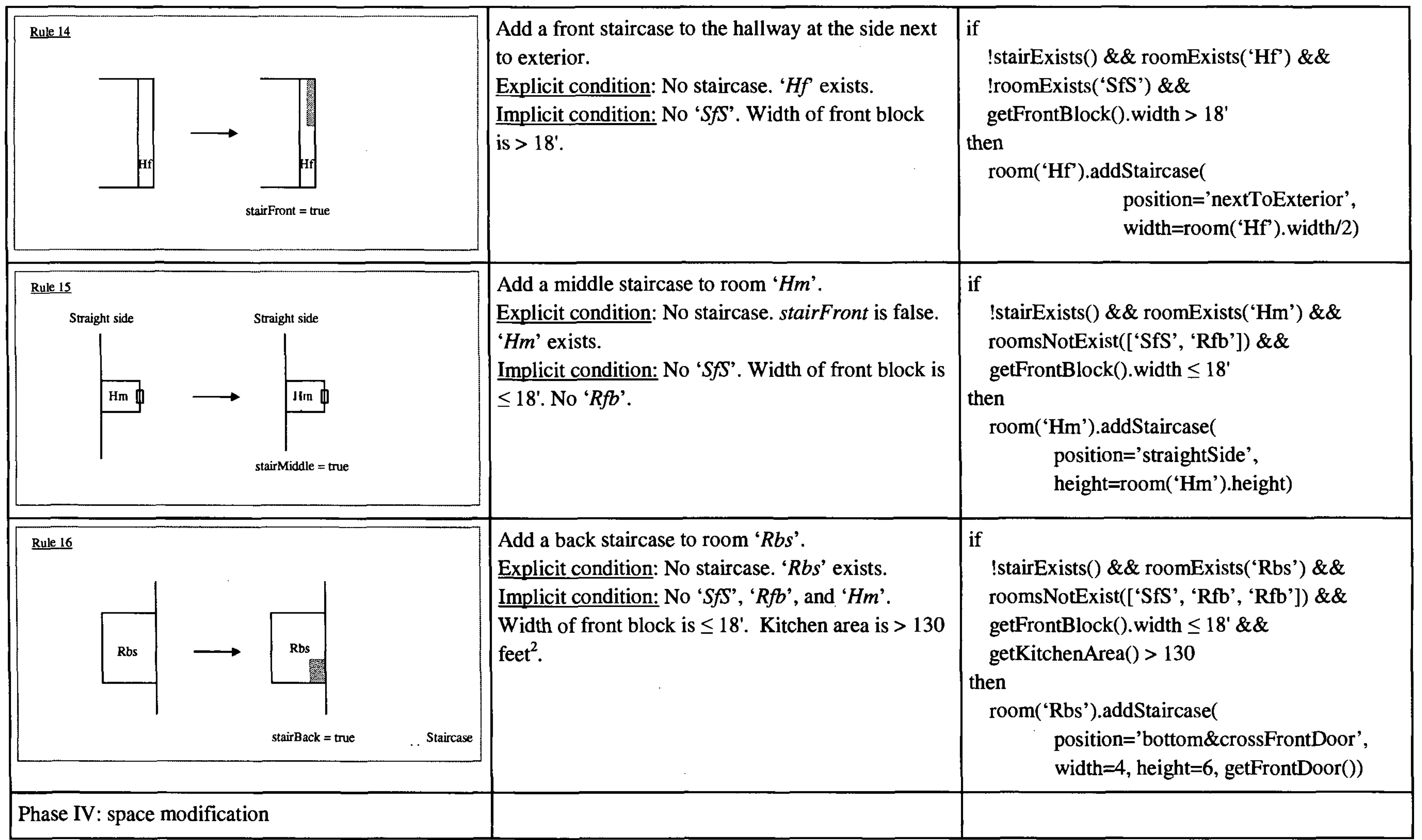


Description	Rule

This rule adds a front staircase with dimension 4' * 6'.
Explicit condition: No staircase. '*Rfs*' exists.
Implicit condition: No '*SfS*', '*Rfb*', and '*Hm*'. Width of front block is ≤ 18'. Kitchen area is ≤ 130 feet^2.
	if
	  !stairExists() && roomExists('Rfs') &&
	  roomsNotExist(['SfS', 'Rfb', 'Hm']) &&
	  getFrontBlock().width ≤ 18' &&
	  getKitchenArea() ≤ 130
	then
	  room('Rfs').addStaircase(
	    position='crossFrontDoor',
	    width=4, height=6, getFrontDoor())

Add a staircase to room '*SfS*'.
Precondition: No staircase. '*SfS*'exists.
	if
	  !stairExists() && roomExists('SfS')
	then
	  room('SfS').addStaircase(
	    position='crossFrontDoor',
	    getFrontDoor())

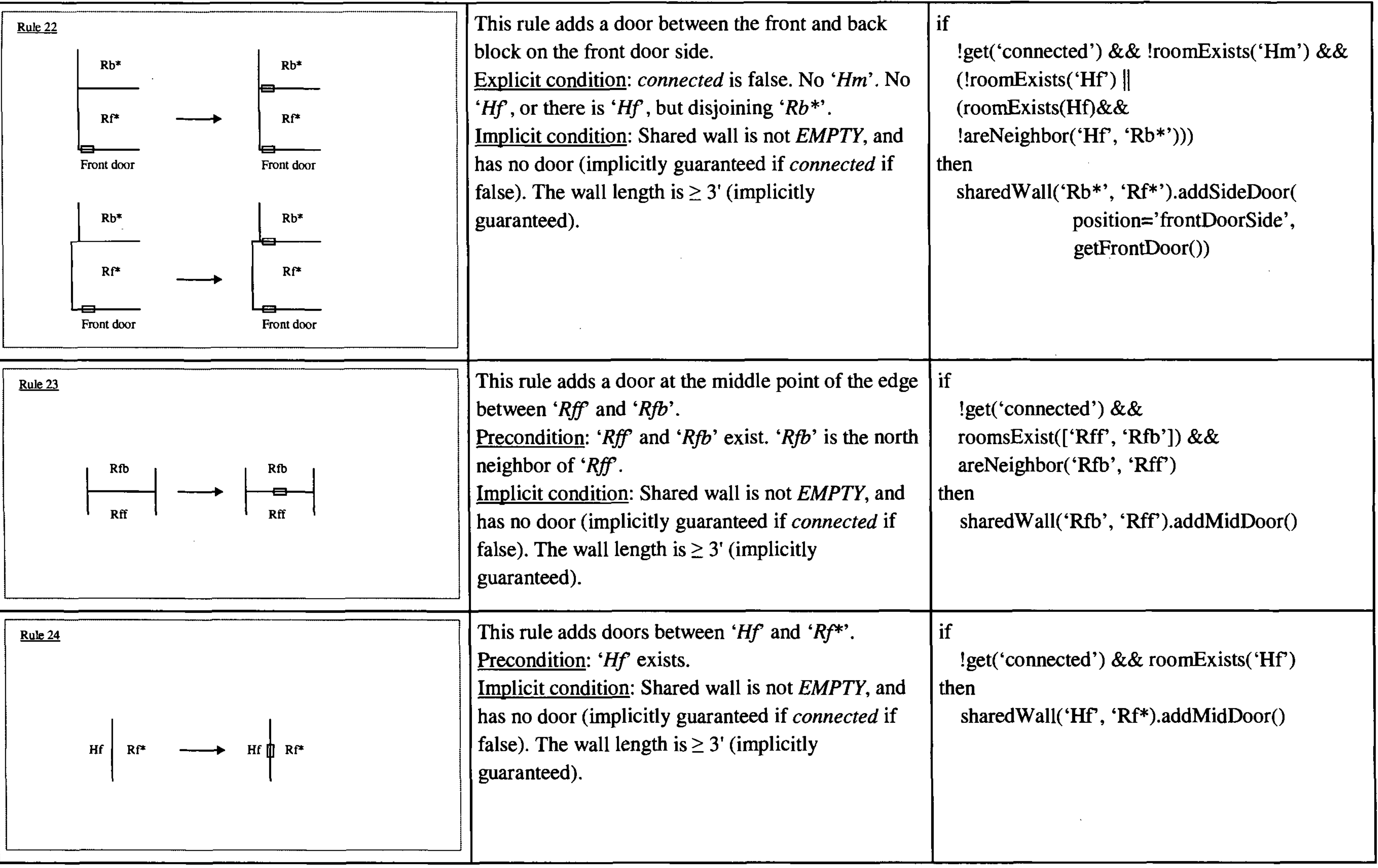
Add a staircase to room '*Rfb*'.
Explicit condition: No staircase. '*Rfb*' and '*Rff*' exist and are neighbors.
Implicit condition: No '*SfS*'. Width of front block ≤ 18'.
Overall condition: *stairFront* is false. '*Rfb*' exists. No '*SfS*'. Width of front block is ≤ 18'. ('Rfb' implies the existence of '*Rfs*'. No '*SfS*' implies that '*Rfb*' and '*Rfs*' are neighbor.)
	if
	  !stairExists() && !roomExists('SfS') &&
	  roomExists('Rfb') &&
	  getFrontBlock().width ≤ 18'
	then
	  room('Rfb').addStaircase(
	    position='bottom&crossFrontDoor',
	    width=6, height=4, getFrontDoor())



Rule	Description	Code
Rule 14	Add a front staircase to the hallway at the side next to exterior. Explicit condition: No staircase. '*Hf*' exists. Implicit condition: No '*SfS*'. Width of front block is > 18'.	if !stairExists() && roomExists('Hf') && !roomExists('SfS') && getFrontBlock().width > 18' then room('Hf').addStaircase(position='nextToExterior', width=room('Hf').width/2)
Rule 15	Add a middle staircase to room '*Hm*'. Explicit condition: No staircase. *stairFront* is false. '*Hm*' exists. Implicit condition: No '*SfS*'. Width of front block is ≤ 18'. No '*Rfb*'.	if !stairExists() && roomExists('Hm') && roomsNotExist(['SfS', 'Rfb']) && getFrontBlock().width ≤ 18' then room('Hm').addStaircase(position='straightSide', height=room('Hm').height)
Rule 16	Add a back staircase to room '*Rbs*'. Explicit condition: No staircase. '*Rbs*' exists. Implicit condition: No '*SfS*', '*Rfb*', and '*Hm*'. Width of front block is ≤ 18'. Kitchen area is > 130 feet^2.	if !stairExists() && roomExists('Rbs') && roomsNotExist(['SfS', 'Rfb', 'Rfb']) && getFrontBlock().width ≤ 18' && getKitchenArea() > 130 then room('Rbs').addStaircase(position='bottom&crossFrontDoor', width=4, height=6, getFrontDoor())
Phase IV: space modification		

Rule 17 Rfb SfS Rff Hf → Rfb SfS Rff Hf	This rule opens the shared wall of '*Hf*' and '*SfS*'. Precondition: '*Hf*' and '*SfS*' exist, and are neighbors.	if roomsExist(['Hf', 'SfS']) && areNeighbor('Hf', 'SfS') then openSharedWall('Hf', 'SfS')
Rule 18 Rfb SfS Rff Front door → Rfb SfS Rff Front door	This rule adds openings on the shared wall of '*Rfb*' and '*SfS*', as well as the shared wall of '*Rfs*' and '*SfS*', on the front door side. Precondition: No '*Hf*'. '*Rfb*', '*Rff*', and '*SfS*' exist, and are neighbors (implicitly guaranteed).	if roomsExist(['Hfb', 'Hfs', 'SfS']) && !roomExists('Hf') then room('SfS').createOpening(position='northWall&crossFrontDoor', getFrontDoor()) room('SfS').createOpening(position='southWall&crossFrontDoor', getFrontDoor())
Rule 19 Hm Hf → Hm Hf	This rule opens the shared wall of '*Hf*' and '*Hm*'. Precondition: '*Hf*' and '*Hm*' exist, and are neighbors (implicitly guaranteed).	if roomsExist(['Hf', 'Hm']) then openSharedWall('Hf', 'Hm')

Rule 20 Rfb → Rfb Hf Hf	This rule opens the shared wall of *'Hf'* and *'Rfb'*. Precondition: *'Hf'* and *'Rfb'* exist, and are neighbors (implicitly guaranteed). *'Hf'* disjoins the middle and back block.	if roomsExist(['Hf', 'Rfb']) && !areNeighbor('Hf', 'Hm') && !areNeighbor('Hf', 'Rb*') then openSharedWall('Hf', 'Rfb')
Phase V: Interior door generation		
Rule 21 Hm → Hm	This rule adds two doors: one on the north edge of *'Hm'*, the other on the south. Explicit condition: *connected* is false. *'Hm'* exists. Implicit condition: Shared wall is not *EMPTY*, and has no door (implicitly guaranteed if *connected* is false). The wall length is ≥ 3'.	if !get('connected') && roomExists('Hm') then a = sharedWall('Hm', 'Rf*) b = sharedWall('Hm', 'Rb*) if a.type() != EMPTY && a.length() ≥ 3' then a.addMidDoor() if b.type() != EMPTY && b.length() ≥ 3' then b.addMidDoor()



Rule	Description	Code
Rule 22 Rb* / Rf* / Front door → Rb* / Rf* / Front door Rb* / Rf* / Front door → Rb* / Rf* / Front door	This rule adds a door between the front and back block on the front door side. <u>Explicit condition</u>: *connected* is false. No '*Hm*'. No '*Hf*', or there is '*Hf*', but disjoining '*Rb**'. <u>Implicit condition</u>: Shared wall is not *EMPTY*, and has no door (implicitly guaranteed if *connected* if false). The wall length is ≥ 3' (implicitly guaranteed).	if !get('connected') && !roomExists('Hm') && (!roomExists('Hf') || (roomExists(Hf)&& !areNeighbor('Hf', 'Rb*'))) then sharedWall('Rb*', 'Rf*').addSideDoor(position='frontDoorSide', getFrontDoor())
Rule 23 Rfb / Rff → Rfb / Rff	This rule adds a door at the middle point of the edge between '*Rff*' and '*Rfb*'. <u>Precondition</u>: '*Rff*' and '*Rfb*' exist. '*Rfb*' is the north neighbor of '*Rff*'. <u>Implicit condition</u>: Shared wall is not *EMPTY*, and has no door (implicitly guaranteed if *connected* if false). The wall length is ≥ 3' (implicitly guaranteed).	if !get('connected') && roomsExist(['Rff', 'Rfb']) && areNeighbor('Rfb', 'Rff') then sharedWall('Rfb', 'Rff').addMidDoor()
Rule 24 Hf | Rf* → Hf | Rf*	This rule adds doors between '*Hf*' and '*Rf**'. <u>Precondition</u>: '*Hf*' exists. <u>Implicit condition</u>: Shared wall is not *EMPTY*, and has no door (implicitly guaranteed if *connected* if false). The wall length is ≥ 3' (implicitly guaranteed).	if !get('connected') && roomExists('Hf') then sharedWall('Hf', 'Rf*).addMidDoor()

Rule 25	This rule adds a door between '*Rbf*' and '*Rbb*'. Precondition: '*Rbf*' and '*Rbb*' exist. Implicit condition: Shared edge is not *EMPTY*, and has no door (implicitly guaranteed if *connected* if false). The wall length is ≥ 3' (implicitly guaranteed).	if !get('connected') && roomsExist(['Rbf', 'Rbb']) then sharedWall('Rbf', 'Rbb').addMidDoor()
Rule 26	This rule adds a door between '*Rb**' and '*Hf*'. Precondition: '*Rb**' and '*Hf*' exist, and are neighbor. Implicit condition: Shared edge is not *EMPTY*, and has no door (implicitly guaranteed if *connected* if false). The wall length is ≥ 3' (implicitly guaranteed).	if !get('connected') && roomsExist(['Rb*', 'Hf']) then sharedWall('Rb*', 'Hf').addDoor()

The new shape grammar comprises five phases: block (mass) generation (rule 1~4), space generation (rule 5~10), stair generation (rule 11~16), space modification (rule 17~20), and interior door generation (rule 21~26).

A significant difference between the new and original shape grammars is that every shape rule of the new shape grammar quantitatively specifies the conditions that apply. For example, this condition can be the number of spaces in terms of blocks (rules 1 and 2), a value in a specific range (rules 5 and 6), and a relationship of two or more values (rules 7 and 8). Some conditions are straightforward. Others require not only reasoning based on common design knowledge, but also certain threshold values, statistically determined. The following illustrates the complexity, using as exemplar, of the rules for generating staircases.

Firstly, rules (rule 11~16), in the form of the original shape grammars, are not necessarily exclusive to one another. For example, to those layouts with room *Rfs* and *Rbs*, where no exclusive condition has been specified as to when to apply each rule, both rules 11 and 16 can apply. As stated previously, we currently only consider working-class rowhouses, each with a unique staircase on its first floor. Therefore, for each layout, only one of the shape rules for generating staircases applies.

Secondly, if there is a staircase room *SfS*, then rule 12 has to apply. As a result, an implicit condition for Rule 11, 13, 14, 15, and 16 is that the current layout has no staircase room *SfS*.

Rule 14 adds a staircase to a hallway. Obviously, the hallway needs to be wide enough to hold the staircase, hence the width of the front block. From the samples (Figure 6-11), 18 feet is a good threshold value to distinguish whether or not rule 14 can apply. To ensure the exclusive application of rule 14, an implicit condition for rules 11, 13, 15 and 16 is that the width of the front block is smaller or equal to 18′.

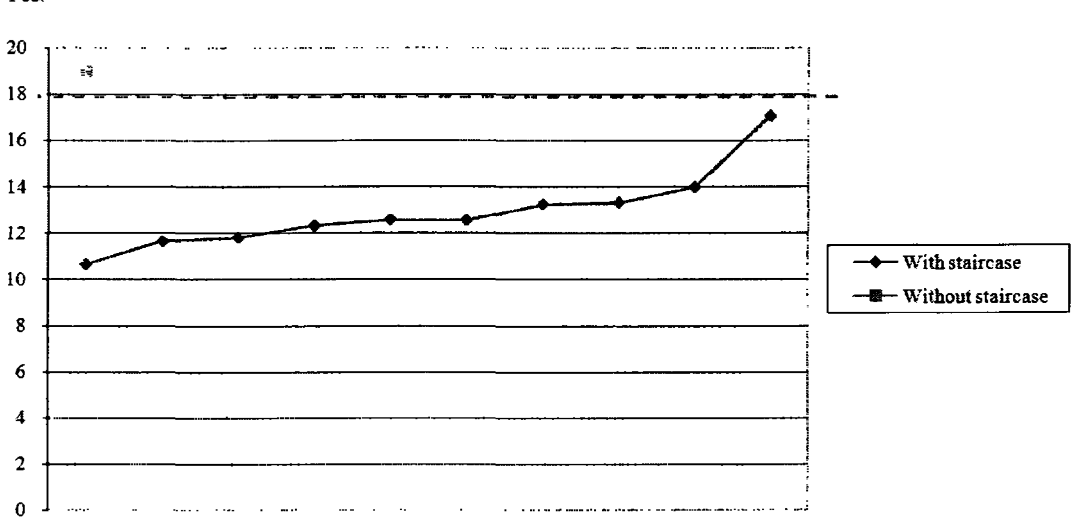


Figure 6-11: Quantifying the shape rules generating staircases

For rules 11, 13, 15, and 16, if there is an *Rfb* room in the layout, then rule 13 should be applied to add a staircase there. Accordingly, an implicit condition for rule 11, 15 and 16 is that there is no *Rfb* room. For rules 11, 15, and 16, if there is a middle block *Hm*, rule 15 should be applied to add a staircase in the middle block. Thus, an implicit condition for rules 11 and 16 is that there is no *Hm* room.

It remains to distinguish between rules 11 and 16. The implicit conditions added by rules 12, 13, 14, and 15 can be summarized as: if there are only a *Rfs* room (the front block) and a *Rbs* room (the black block) in the current layout, then possibly rules 11 and 16 can be applied. Rule 16 adds a staircase to an *Rbs* room, which is actually a kitchen. Therefore, the kitchen space has to be large enough to hold a staircase as well as function as a kitchen. In the sample available, only one uses rule 11 and one uses rule 16. Because of this, the related statistical data for all samples is computed as a reference: the average area of kitchens without a staircase is 127.7 feet^2, the minimum is 92.8 feet^2, and the maximum is 185.4 feet^2. The area of a staircase is about 26~30 feet^2. The kitchen area of the case that uses rule 11 is 94.4 feet^2, and the kitchen area of the case that uses rule 16 is 165.5 feet^2. The average of these two cases is about 130 feet^2, which is close to the average of kitchens without

staircases. So, 130 $feet^2$ is used as the threshold value. As a result, an added condition for rule 16 is that the area of kitchen is greater than 130 $feet^2$. An additional condition for rule 11 is that the area of the kitchen is smaller or equal to 130 $feet^2$. Table 6-2 gives a summary of implicit conditions to make rules for generating staircases exclusive.

Table 6-2: Implicit conditions to make staircase rules exclusive

	Rule 12	**Rule 14**	**Rule 13**	**Rule 15**	**Rule 11**	**Rule 16**
Rule 12	With 'SfS'	No 'SfS'	No 'SfS'	No 'SfS'	No 'SfS'	No 'SfS'
Rule 14		Front block width > 18'	Front block width ≤ 18'	Front block width ≤ 18'	Front block width ≤ 18'	Front block width ≤ 18'
Rule 13			With 'Rfb'	No 'Rfb'	No 'Rfb'	No 'Rfb'
Rule 15				With 'Hm'	No 'Hm'	No 'Hm'
Rule 16					Kitchen ≤ 130 ft^2	Kitchen > 130 ft^2

Chapter 7 Layout tree pruning and initial layout estimation

The basic research task of the AutoPILOT project is to employ a shape grammar to determine the interior layout of a building from its feature input. Although a computer implementation of the shape grammar is capable of enumerating all possible layouts within the language space (in this context, the layout space) of the shape grammar, it still remains to make connections between the layout space and the feature input so that those layouts consistent with the feature input can be 'picked out'. In this chapter, related techniques, in particular, layout tree pruning and initial layout estimation will be discussed through two test cases: the Baltimore Rowhouse and the Queen Anne House. Note that building input features used in this chapter are simply taken from existing drawings from either (Flemming et al., 1985), for Queen Anne houses, or (Hayward, 1981) for Baltimore rowhouses.

7.1 Layout tree pruning

A computer implementation of a shape grammar that captures the corpora of a building style is essentially an enumeration of all possible layouts. The enumeration procedure, that is, the derivation of a shape grammar, can be viewed as a tree structure, namely, a layout tree. Valid layouts correspond to certain nodes of the tree.

These nodes are mostly leaf nodes, though certain internal ones are possible; an internal one is a layout of smaller size, while a leaf one is a layout of larger size and typically with more rooms.

Accordingly, layout determination becomes a process of 'picking up' those nodes consistent with feature input from the layout tree. As it is generally difficult to do so directly, the pickup is typically achieved by tree pruning — eliminating those nodes inconsistent with certain constraints with the remnants being the desired results.

Shape grammars capturing building corpora are typically parametric; that is, only 'topological' relationships are described, for example, which room is next to which other room under a certain condition, leaving the extract dimensions largely unspecified. As a result, many constraints for pruning a layout tree are necessarily of a topological nature, especially for those used to prune branches near the root of the layout tree. This makes it difficult to conduct tree pruning, as a feature input is for objects with real dimensions. A procedure to obtain topological constraints becomes necessary. As discussed later in this chapter, such topological constraints are mainly obtained through a process termed, *initial layout estimation.* This process makes use of building knowledge in the form of constraints on building features.

What is more, the topological constraints cannot prune the tree in a way that the desired final layouts can be directly determined. Before reaching the final layouts, the variables (aka. parameters) in the intermediate configurations have to be 'fixed' to match the feature input at a certain stage, and the fixing process can be progressive or done in a single shot. Progressive fixing relies on the constraints, directly or inferred from the feature input, or from a priori knowledge, and parameters are partially fixed at each step until all are fixed. Parameter fixing of the Queen Anne houses is an example of progressive fixing. For parameter fixing in a single shot, control parameters are fixed in one step using a certain procedure. Parameter fixing of the Baltimore rowhouses is such an example. Note that, in this dissertation, parameter fixing of the Baltimore rowhouses has been implemented while parameter fixing of Queen Anne houses not.

7.2 Building feature constraints

Building features constrain one another. For example, a window normally belongs to a unique room (space), no wall falls within a window; to be appropriate for use, the ratio of a room dimensions is usually ½; and to be practically functional, the minimum width of a space is larger or equal than 2'. Other constraints require specific knowledge. For example, the height of each story is determined by window and door geometries (Figure 7-1). Moreover, once the height of a story is estimated, the dimension of various forms of staircase can be estimated through common dimension of treads and risers *(24" ≤ tread + 2 * riser ≤ 25")*, and the width of staircase is another constant (for example, usually 3' for houses) in a given context. Another example is the interaction between window and staircase. As shown in Figure 7-2, for windows with known positions, possible positions for the staircase are greatly reduced (the impossible region for a staircase is shaded based on the principle that no staircase will 'cross' a window in a sectional view). Together with further constraints from the surrounding rooms, the exact position of the staircase can be narrowed down to a small range.

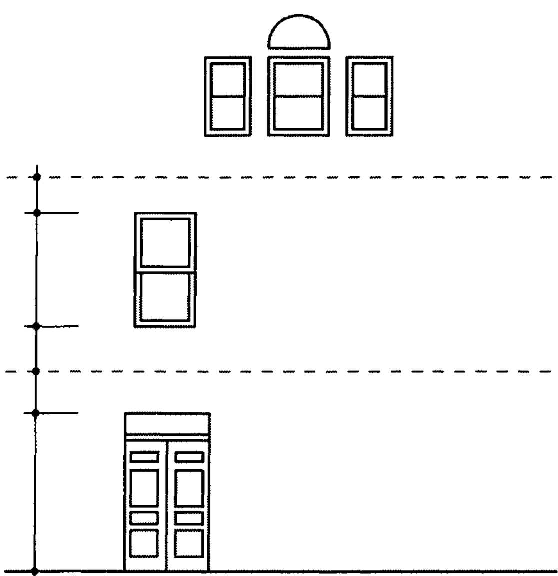


Figure 7-1: Windows and doors constrain the height of each story

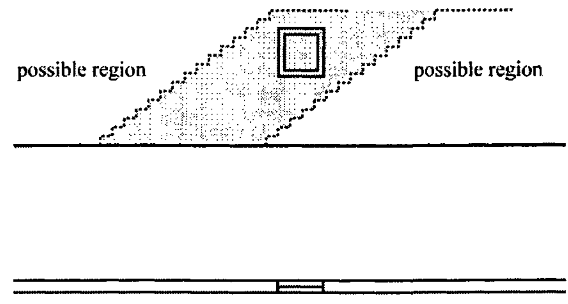


Figure 7-2: Window position constrains possible arrangements of a staircase

7.3 Constraint satisfaction

Determination of building interior layout is to identify all of the rooms within a building as well as their dimensions. A subset of such rooms is adjacent to the building's exterior. Accordingly, they correspond to a set of exterior features. For instance, most rooms in Queen Anne houses (Flemming, 1987) have centrally-located fireplaces that correspond to chimneys visible over buildings' roof. The abundance of such constraints among exterior building features suggests that a preliminary layout could be found by treating it as a problem of constraint satisfaction (Russell and Norvig, 2002).

A constraint satisfaction problem (CSP) has three components: i) a set of variables $X = \{x_1, \dots, x_n\}$, ii) a set of possible domain values, D_i, for each x_i, and iii) a set of constraints to restrict the values that variables can simultaneously take. Different acceleration techniques, for example, forward checking and constraints propagation are developed to eliminate impossible values efficiently, thereby speeding up solving the CSP.

There are important benefits to treating a problem as a CSP: i) representation as a CSP is typically much closer to the original problem: variables directly correspond to problem entities and constraints can be expressed more explicitly without awkward translation; ii) representation as a CSP conforms to a standard pattern so that many algorithms can be written in a generic way; iii) effective, generic heuristics can be developed without requirements of additional, domain-specific

expertise; and iv) the structure of the constraint graph can be used to simplify the solution process, potentially leading to an exponential reduction in complexity.

The CSP algorithm for initial layout estimation starts by generating rooms with conservative dimensions to accord with the given features. The algorithm then manipulates rooms as variables according to constraints established by common building properties. Two kinds of manipulations are considered: expanding room dimensions and merging two rooms into one. Note that merging two rooms eliminate a variable initialized at the beginning; this stands in contrast to typical CSP algorithms where variables persist throughout the life of the algorithm.

7.4 CSP and Queen Anne houses

Figure 7-3 shows the results of applying the CSP algorithm to the first floor of a Queen Anne house—5816 Walnut Street in Pittsburgh, Pennsylvania. Input features are locations in plan of the windows, doors, and chimneys, together with possible symmetry axes inferred from the facades. Note that in Queen Anne houses chimneys vertically correspond to fireplaces on the first floor. Step 1 extends the axes of exterior wall inward (assuming a wall thickness of 1') to form wall hotspots; this enforces a tendency of interior rooms to be aligned with one another. Step 2 uses the fact that larger public rooms on the first floor have fireplaces, which corresponds to the chimneys. By projection, if the chimney falls within the interior of the footprint, then there are possible two rooms that share the chimney, with a fireplace each. If the chimney is on an exterior wall, only one room can use the chimney. Such rooms are assigned with an initial dimension of 8' × 8'. Step 3 adjusts rooms that are close (based on 1' threshold) so that they should align with the nearest axis. Step 4 uses the fact that rooms can contain, but do not intersect with other rooms and doors. Rooms are extended to include such features to resolve any conflict. Step 5 uses the fact that the minimum distance between two walls has to be large enough to be a useful space (usually > 3'). In step 6, rooms generated from the chimneys are stable. Step 7 specifies rooms (5' × 5') to unassigned windows and doors. Note that a room may be left-, center-, or right aligned with a window or door. Two largely overlaying rooms are merged as one. Also, the narrow space remaining between *RM1* and *RM5* is

assigned to *RM5* according to the symmetry axis, *SYM-4*. Step 8 shows the final result; though incomplete, but it is close to the actual condition. At this stage, there are ambiguities that cannot be resolved without prior knowledge, which is left for the shape rules to handle.

In the above example, the estimated rooms are restricted to rectangles. This can pose certain difficulties when evaluating buildings partially with non-rectangular rooms as demonstrated by 719 Amberson Avenue (Figure 7-4h). The strategy is to approximate the original non-rectangular rooms by rectangles, and project the related windows correspondingly. Such simplified approximations retain the necessary constraints introduced from the original window features, making constraint satisfaction applicable for a wider range, and greatly reduce the complexity in implementing geometric manipulation. Figure 7-4 shows both the original inputs (solid lines), and the simplified version (dashed line), as well as the manual derivation procedure based on these approximations.

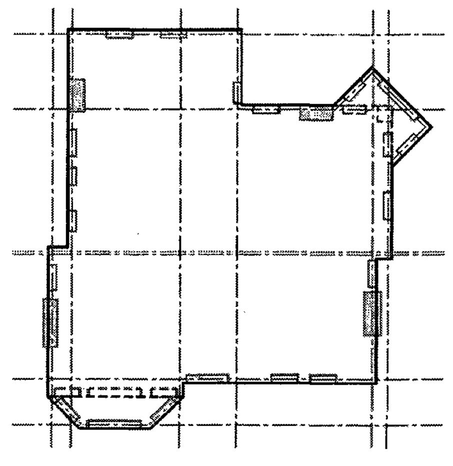
Figure 7-5 shows the results of the computer implementation of the above CSP algorithm. Besides the complexity of geometry manipulation, the implementation has to deal with potential numerical round-off and tolerance issues. For example, an intersection test between a rectangle (chimney) and a line segment (wall axis) is used to determine whether a chimney is on the boundary of a footprint or not. However, the chimney data is derived from the parts over the roof, whose dimension potentially shrinks. This may result that a chimney may be close enough to the line representing a wall, without actually intersecting or being coincident. Imperfect threshold value also causes issues. For example, as shown in Figure 7-4a, the assumption of wall thickness of 1' results two very close hotspot wall axes, which can break the algorithm if not handled specially. Moreover, different from the manual derivation, the rooms in the up-right corner of Figure 7-5b do not merge as desired due to the difficulty to set a 'universal' threshold value—a threshold value working for a build or a part of a building may fail for another building of the same style or another part of the same building.

(a) Step 1 (b) Step 2 (c) Step 3

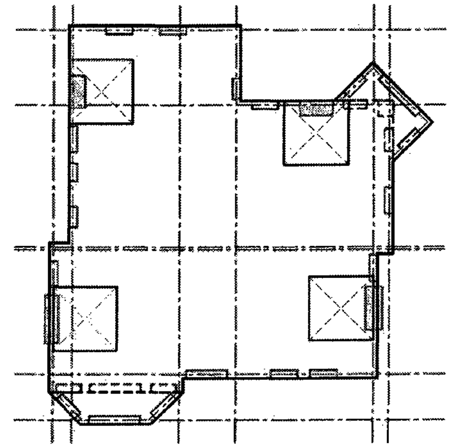
(d) Step 4 (e) Step 5 (f) Step 6

(g) Step 7 (h) Step 8 (i) Ground truth

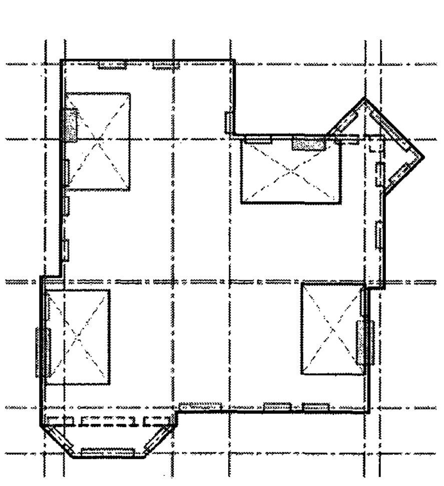
Figure 7-3: Initial layout estimation of Queen Anne houses by CSP



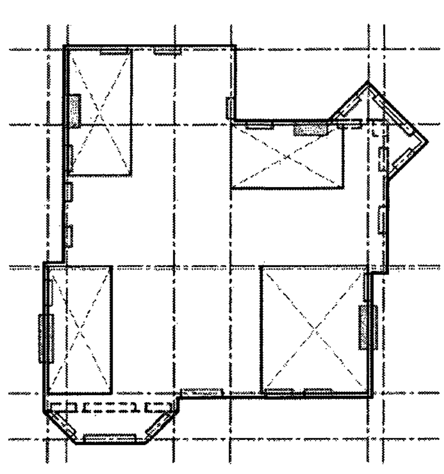
(a) Step 1



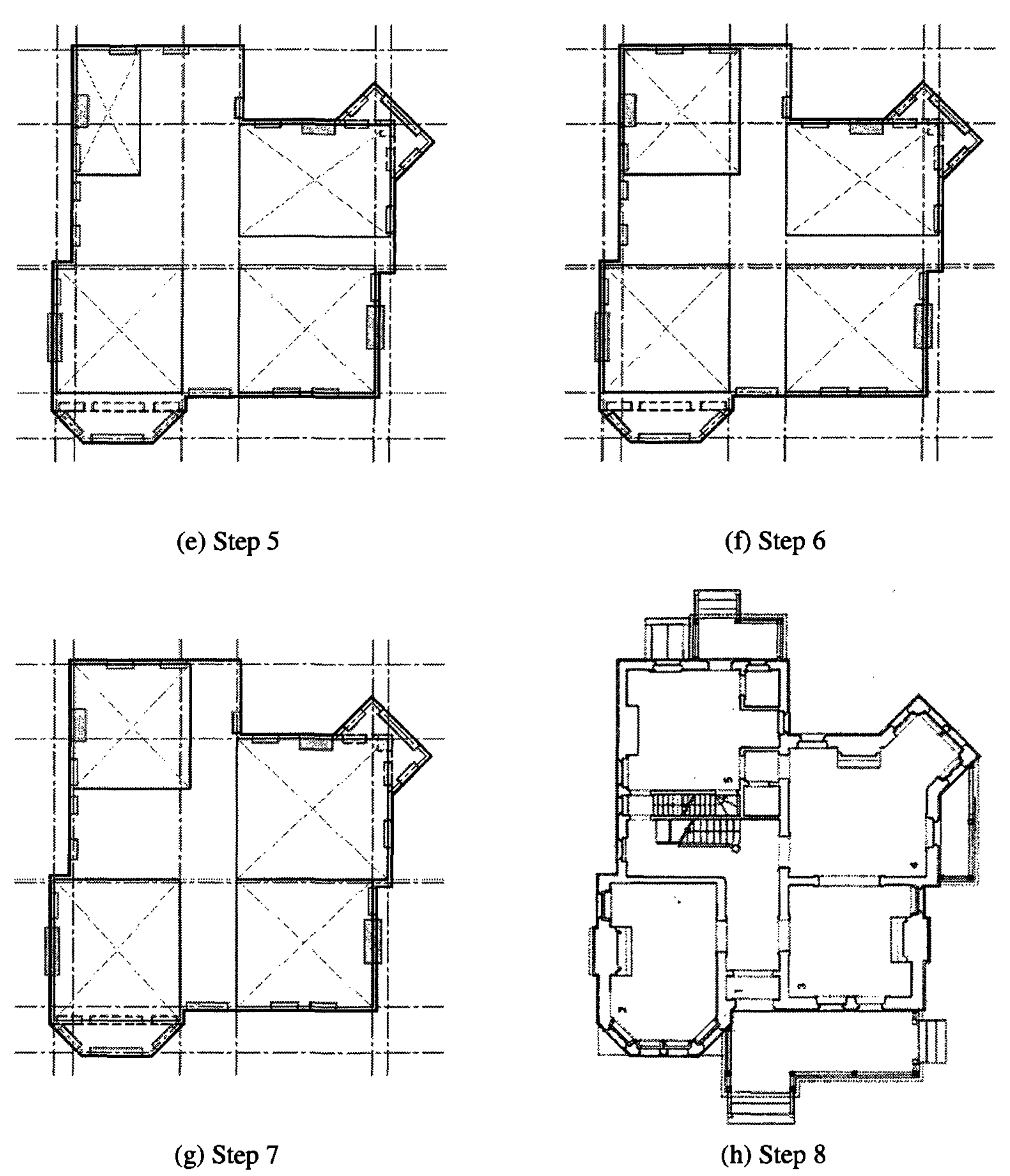
(b) Step 2



(b) Step 3



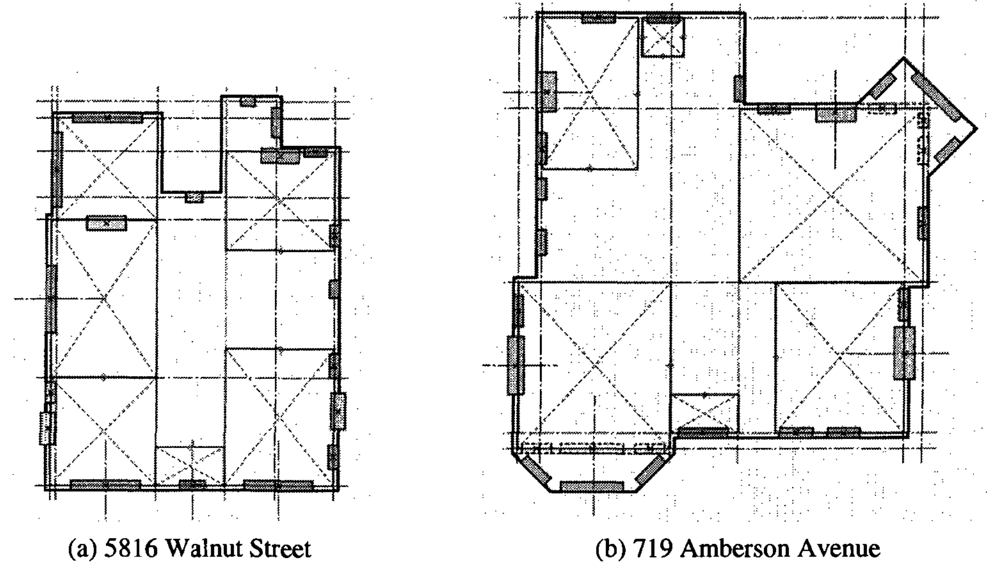
(c) Step 4



(e) Step 5 (f) Step 6

(g) Step 7 (h) Step 8

Figure 7-4: Manual layout derivation of 719 Amberson Avenue



(a) 5816 Walnut Street

(b) 719 Amberson Avenue

Figure 7-5: Results from the computer implementation of CSP

The partial layout results from the initial layout estimation by CSP do provide useful topological constraints for pruning the layout tree. For example, in Figure 7-5a, there are three rooms at the bottom side and their widths are determined. This converts to a topological constraint of three rooms at the bottom side. Moreover, the width parameters of these rooms can be fixed to those values, respectively. The three rooms on the left side can be used in a similar way. And the two partial rooms on the right side can convert to a slightly weaker topological constraint that there are at least two rooms on the right side with a determined width value. In Figure 7-5b, the two rooms on the bottom side have fixed width dimensions. For the left side, the dimension of the up-left room is fixed; otherwise, it is only sure that there are at least three rooms on that side. Similarly, there are at least two rooms on the upside.

7.5 Other constraints on Queen Anne houses

The number of topological constraints from the CSP algorithm may be insufficient to prune the layout tree so that it is necessary to introduce other types of constraints. As the hallway plays an important role in the Queen Anne grammar (Flemming, 1987), the determination of the hallway will help pruning by deciding the number of rooms on the front side of the house.

According to (Flemming, 1987), Queen Anne houses can be classified into three categories based on the position and the shape of the hallway, namely, *side hall*, *center hall*, and *corner hall*. Table 7-1 is a summary of these three types.

Table 7-1: Three types of hallways of Queen Anne houses

Type	**Front width**	**Feature**
Side hall	1 ½ bays wide	½ bay hall and 1 bay room
Corner hall	2 bays wide	1 bay hall and 1 bay room
Center hall	2 ½ bays wide	½ bay hall, 1 bay room on the left and right, respectively

Moreover, from a footprint, the hallway type can be distinguished by using the minimum distance from the center of the front door to the leftmost and rightmost side of the front part of the footprint (Figure 7-6), assuming the orientation of the footprint is so adjusted that the front door is at the bottom side.

- For a *side hall*, the minimum distance d is equal to the half width of the hallway. Such a hallway is at most 8' width, thus d is at most 4'.
- For a *center hall*, the minimum distance (d) is equal to the width of a front public room plus the half width of the hallway. A front public room is at least 10' wide, and the center hallway is at least 4' wide. Therefore d is at least 12'.
- For a *corner hall*, the hallway becomes a hall room, and the minimum distance (d) is equal to the width of the hall room (one-bay wide) minus d_1, which is the distance from the center of the front door to a wall of the hall room. It is trivial that the minimum distance d of corner hall is greater than side hall. However, it is much more subtle to distinguish a corner hall from a center hall. Let assume that d is at least 12', which is the same as the center hall. Since the d_1 is at least 2', the hall room is at least 14' wide. Since the corner hall is usually a small public room, therefore such a hall room is highly unlikely in general. Therefore, we are pretty safe to conclude that the minimum distance d of corner hall is less than 12'. Overall, the minimum

distance *d* of corner hall is less than 12' and greater than 4'. As a result, if a footprint is neither a side hall nor a center hall, it should be a corner hall.

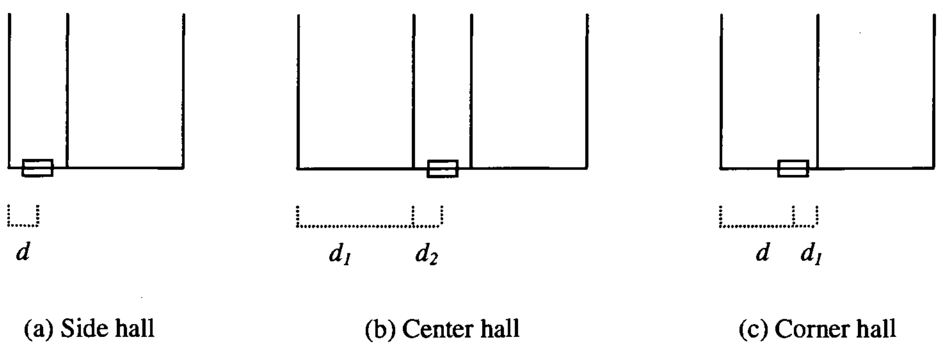


Figure 7-6: Determination of hallway types

Likewise, other types of constraints can be further introduced if the constraints obtained so far are still insufficient to prune the layout tree and fix the configuration.

7.6 Layout determination of Queen Anne houses

The layout determination of Queen Anne houses is carried out in two steps: first implementing the Queen Anne grammar to generate a layout tree, and then pruning the layout tree by using the constraints from the initial layout estimation.

Figure 7-7 shows the screenshot of the computer implementation. On the left, there are four small windows: i) the Truth window shows the true layout for comparison; ii) the Feature input window shows the feature input used for initial layout estimation; iii) the Derivation window shows the result of initial layout estimation by the CSP algorithm and is animated step by step; and iv) the Constraints window shows the constraints manually exacted from the partial layout results from initial layout estimation. On the right is the Grammar tree window. The top-left panel shows the layout tree generated by applying all the shape rules, in which those that are crossed out correspond to layouts inconsistent with the constraints extracted. The top-right panel shows the layouts remaining after pruning the layout tree by the extracted constraints. The central panel on the top is the drawing panel; when

clicking on entries of the top-right or top-left panel, the corresponding layout is displayed. The bottom is the status bar, displaying the summary of layout tree generation and pruning. Above the status bar is the rule panel, displaying all the shape rules for the Queen Anne grammar; when clicking on an entry of the layout tree, the current applicable shape rules are highlighted.

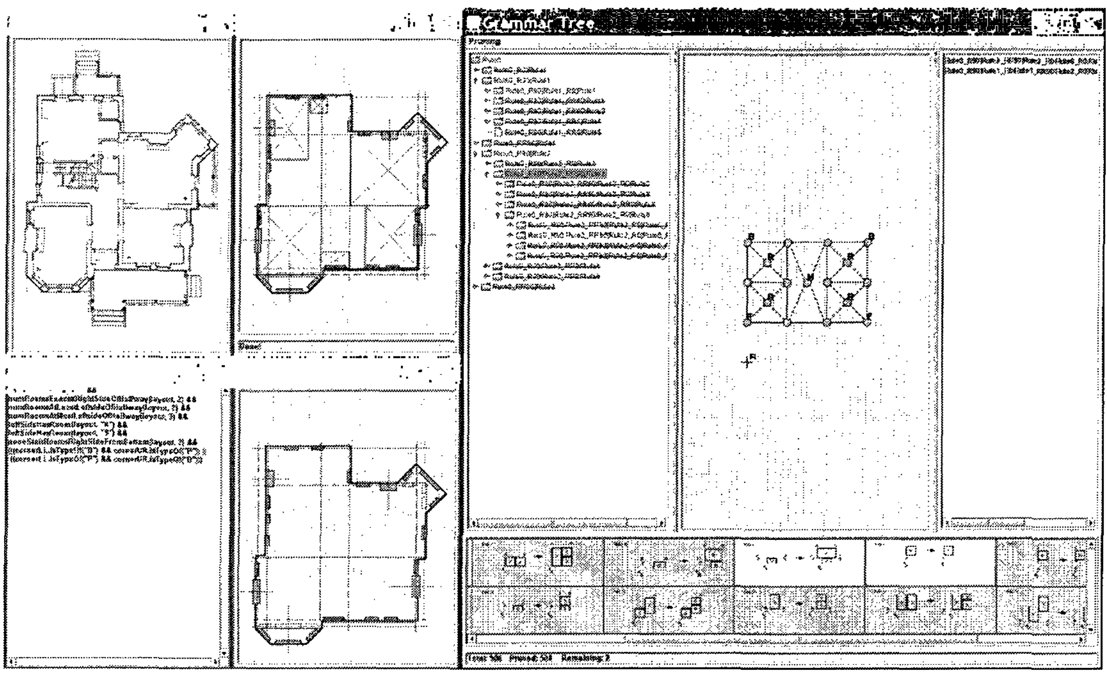


Figure 7-7: Screenshot of layout determination of Queen Anne houses

7.6.1 Implementation of the Queen Anne grammar

To obtain the layout tree of Queen Anne houses, it is essential to implement the Queen Anne grammar. The implementation is based on the rectangular sub-framework. That is, a layout is represented by a graph-like data structure. In essence, the Queen Anne grammar just captures the topology of Queen Anne houses; that is, the grammar simply specifies the possible neighborhood of different room spaces, with little concern of room dimensions.

The Queen Anne grammar described in (Flemming, 1987) is not really computation-friendly. In order to be converted to a piece of code, certain shape rules require additional constraints; others need to be *de-compacted* so that different

possibilities are clarified. For example, according to the diagram describing Rule 2 (Figure 7-8a), it is applicable to both Figure 7-8b and Figure 7-8c. While the application on Figure 7-8b produces a reasonable layout, the application to Figure 7-8c will produce a too small room. Note that, in general, dimension is not important in the implementation of the Queen Anne grammar. Yet, in order to eliminate such inappropriate cases, a certain sense of dimension has to be employed.

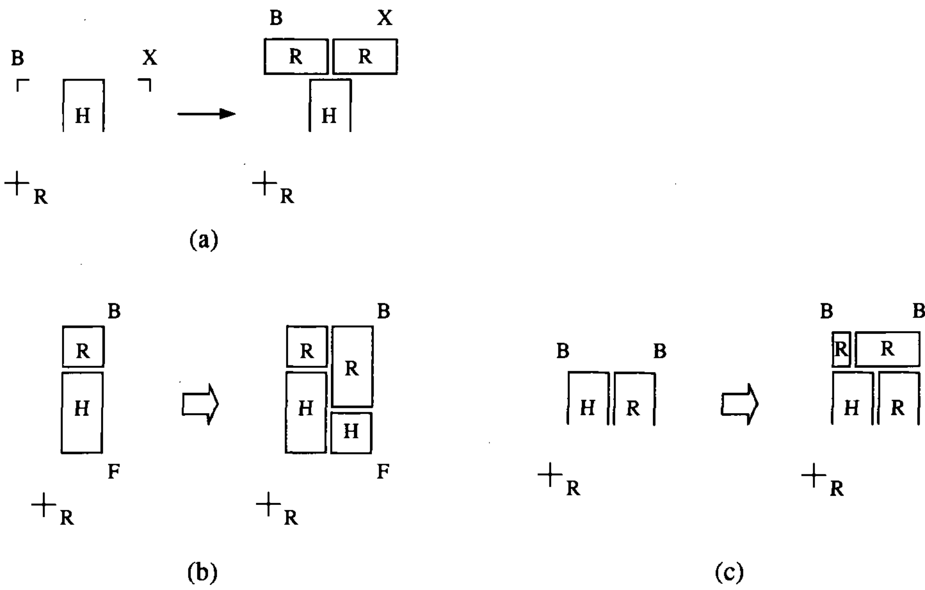


Figure 7-8: Application of Rule 2

Another example is the interpretation of Rule 8 (Figure 7-9a). From the diagram, it seems that the two rooms have to partially overlap in order for this rule to apply. However, from the sample layouts shown in (Flemming, 1987), the case of Figure 7-9b is applicable, as well. In a computation-friendly shape grammar, these two cases need to be described in the way shown in Figure 7-9 so that the implementation stage simply focuses on the task of coding.

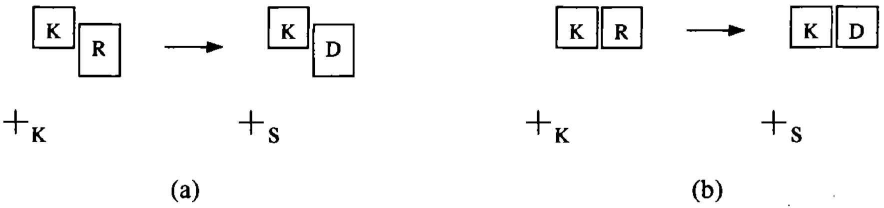
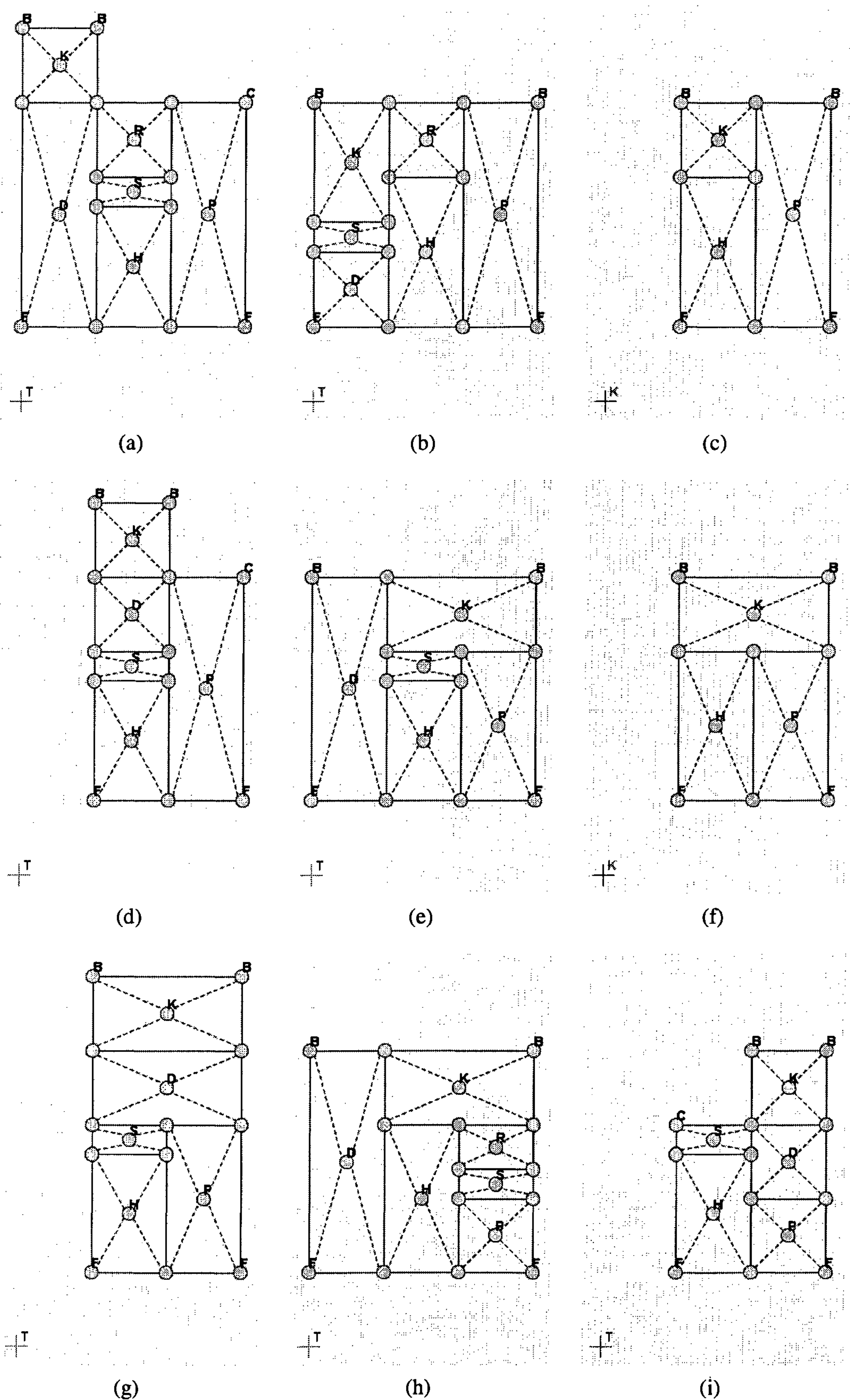


Figure 7-9: Interpretation of Rule 8

Figure 7-10 shows sample results generated by my implementation. Note that the layouts of Figure 7-10k and l are not *valid* in any practical sense as the top rooms are much too wide. However, there is no simple solution to eliminating such cases by putting constraints on shape rules themselves (that is, making them computation-friendly). However, during the fixing step, unreasonable dimensions will be obtained for such rooms so that these layouts can be removed. As the fixing step is not implemented in the demo of this dissertation, instead, a post-processing step is implemented to remove those invalid layouts. In this implementation, there are 506 unique layouts generated in total.



(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)

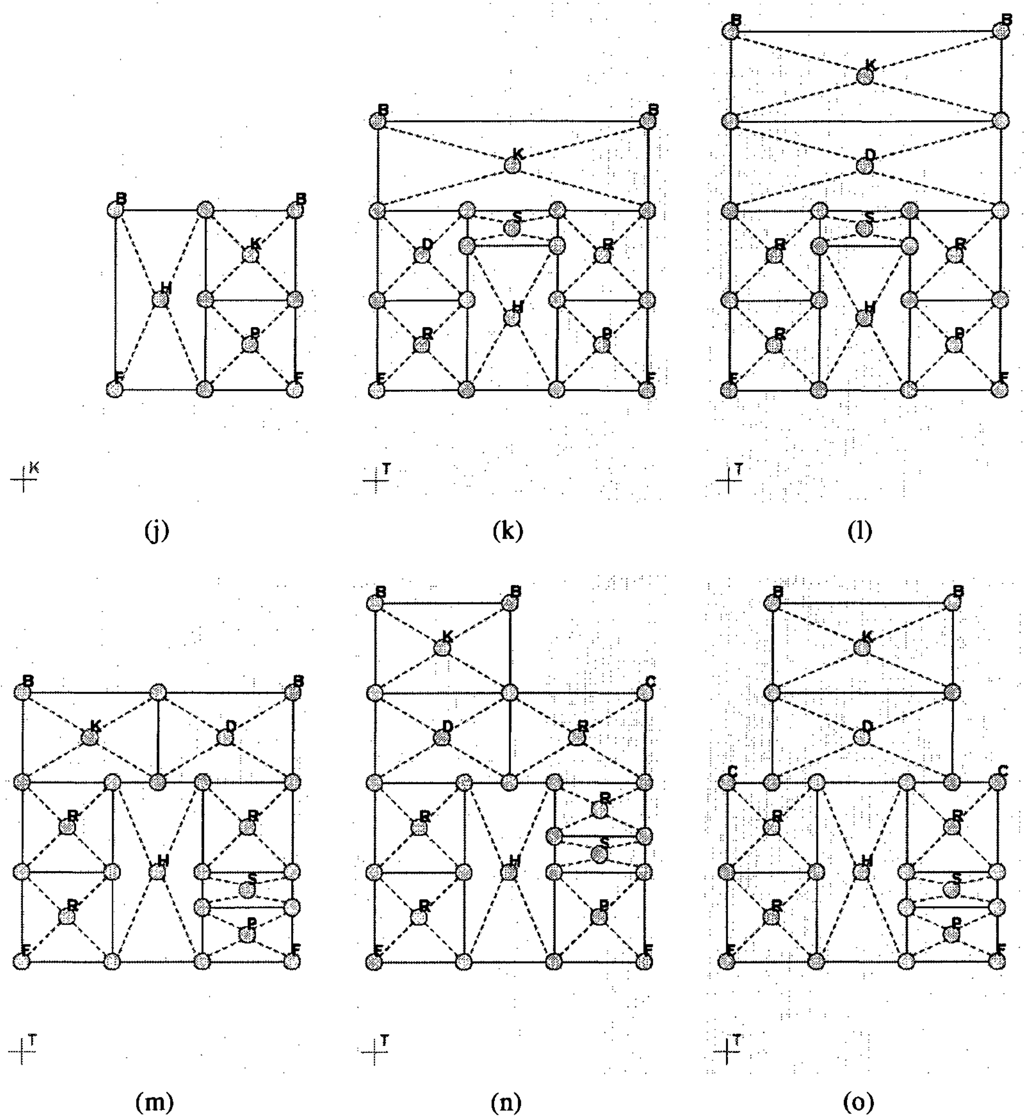


Figure 7-10: Screenshots of sample layouts generated by the Queen Anne grammar

7.6.2 Layout tree pruning of Queen Anne houses

As explained in Section 7.4, some simple topological constraints can be extracted from the partial layout results of the initial layout estimation. For the building of Figure 7-5a, the exacted constraints includes: i) the hallway is central, ii) the number of rooms at the left side of the hallway is at least 2, ii) the number of rooms at the right side of the hallway is at least 2 and at most 3, iii) there must be kitchen and staircase rooms at the right side of the hallway, iv) there is no dining room in the rooms which are right behind the hallway, and v) rooms behind the hallway are

aligned with the hallway vertically. Figure 7-11 shows the results by pruning the layout tree with these constraints. Layouts of Figure 7-11a, b and d are two layout results closest to truth; the deviation is due to the defects of Queen Anne grammars; the leftmost room on the top is a sun porch room, which is not so common in Queen Anne houses that the grammar does consider this special cases. Layouts of Figure 7-11c and e are not quite right; these layouts can be removed when fixing the dimensions.

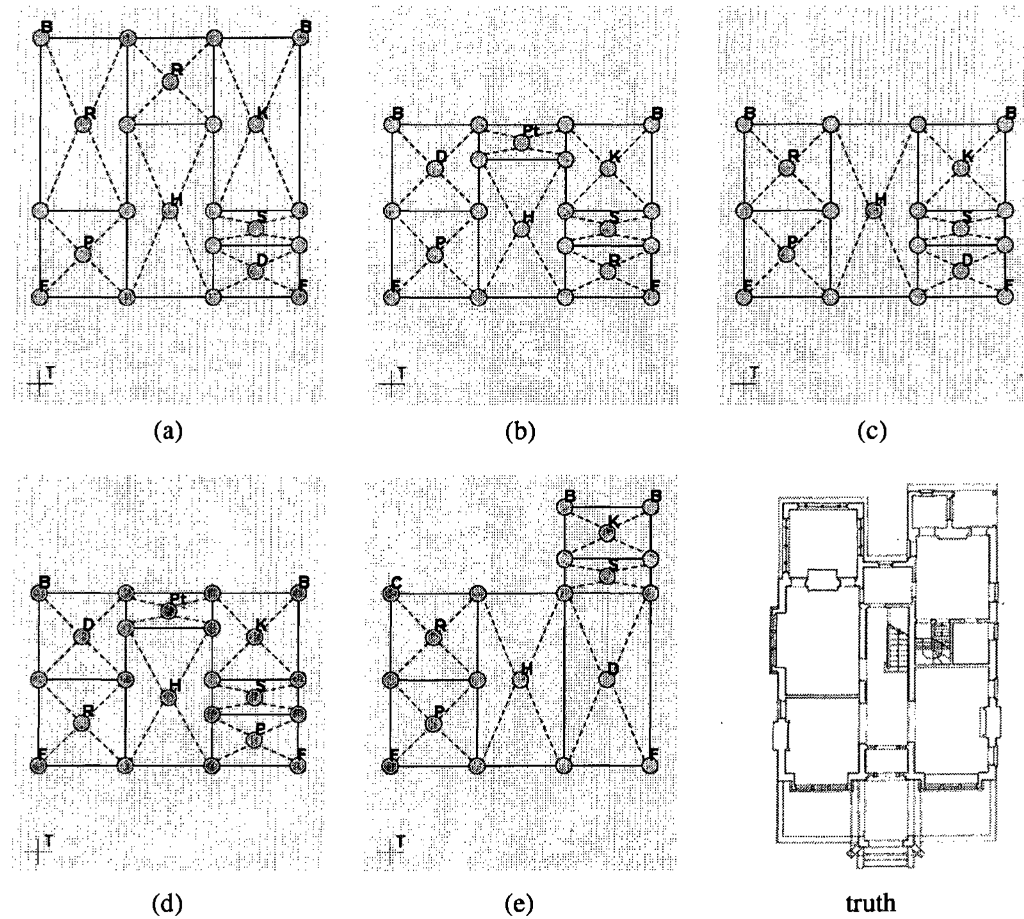


Figure 7-11: Layout results of 5816 Walnut Street

For the building of Figure 7-5b, the exacted constraints includes: i) the hallway is central, ii) the number of rooms at the right side of the hallway is 2, ii) the number of rooms at the left side of the hallway is at least 2 and at most 3, iii) there must be kitchen and staircase rooms at the left side of the hallway, iv) from the bottom, the

first two rooms are not staircase rooms, and v) when the lower left corner room is a dining room, the upper right corner room must be a parlor room; vice versa. Figure 7-12 shows the results by pruning the layout tree with these constraints. Layout of Figure 7-12b is closer to the truth than layout of Figure 7-12a. Again, layout of Figure 7-12a can be removed when fixing the dimensions.

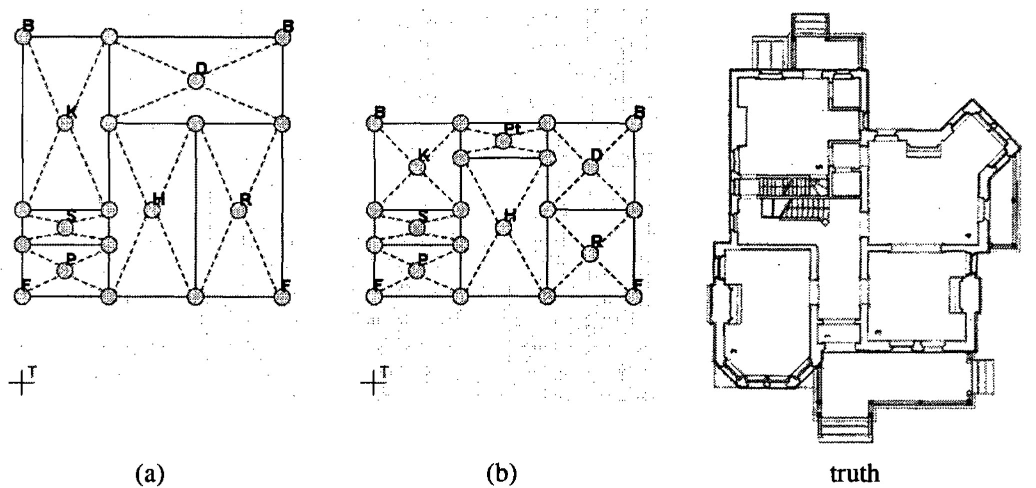


Figure 7-12: Layout results of 719 Amberson Avenue

7.7 Layout determination of Baltimore rowhouses

The layout determination of Baltimore rowhouses is carried in only one step. It is an example in which the initial layout estimation happens to also prune all inconsistent results from the layout tree (Figure 7-15 and Figure 7-16). Layout determination then becomes, simply, rule application on the layout result from the initial layout estimation.

Figure 7-13 shows the screenshot of the computer implementation. On top, the left window shows all the building samples of Baltimore rowhouses from a database, and the right window shows the shape rules. At the bottom, from left to right, the first window shows the tree structure of shape rule application. There is always one path in this window. Clicking on an entry in the tree structure, the corresponding shape rule applied will be highlighted in the shape rule window. The second window shows the true layout. The third window shows the layout generated. The fourth

window shows the feature input. The rightmost window shows the three-dimensional view by extruding the two-dimensional layout with certain default values.

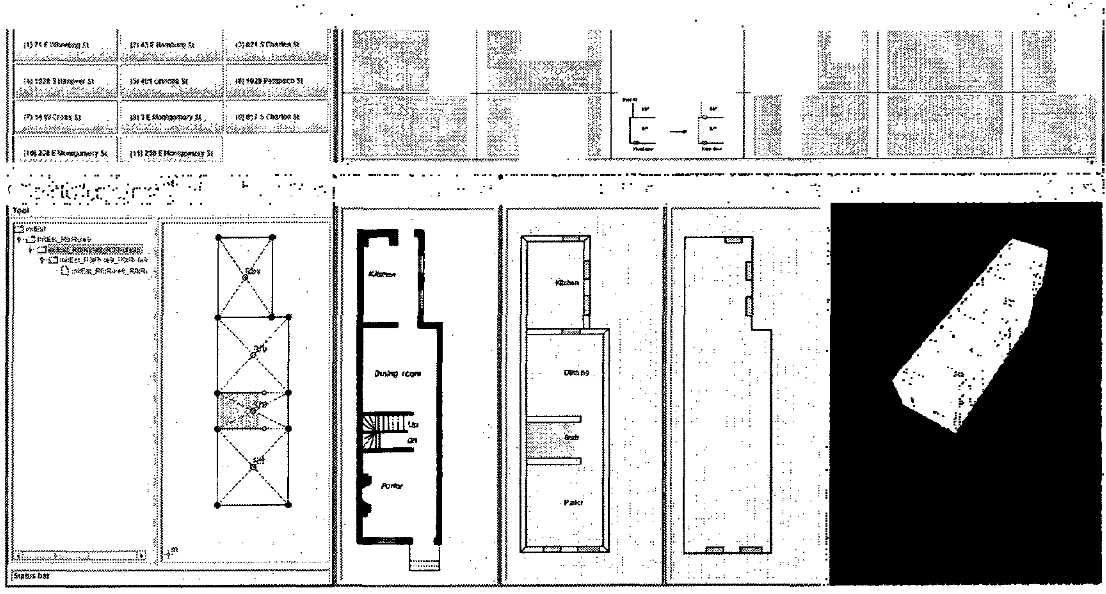


Figure 7-13: Screenshot of layout determination of Baltimore rowhouses

7.7.1 Space subdivision tree and Baltimore rowhouses

Theoretically, the implementation of the CSP algorithm should work for a variety of building types. However, it does not perform well on Baltimore rowhouses (Figure 7-14) even though there is relatively little morphological variation when compared to the Queen Anne houses. The reason for this is that several assumptions that apply to Queen Anne houses do not apply to the rowhouses, for instance: i) spaces within rowhouses tend to be narrower and deeper than in Queen Anne houses; ii) spaces within rowhouses that contain fireplaces are not always symmetric about that fireplace. This is especially true for rowhouse kitchens; iii) fireplaces in rowhouses do not necessarily correspond to a chimney visible from the building's exterior; and iv) the simplicity of the typical rowhouse footprint makes it more difficult to infer interior wall axes from the footprint alone.

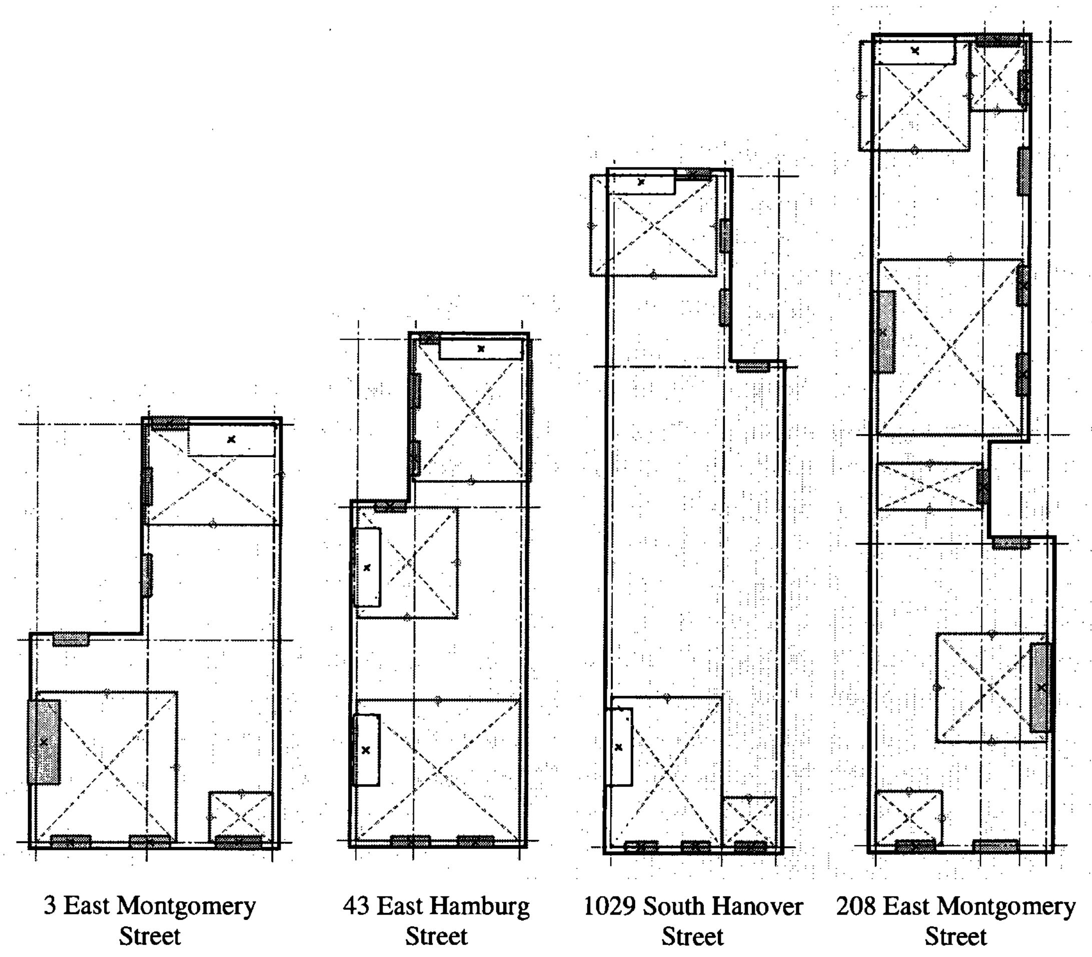


Figure 7-14: Results of the CSP algorithm on Baltimore rowhouses

As a consequence of the first two reasons, initial dimensions assigned to rooms with chimneys may fall outside the building's footprint. Moreover, the chimneys do not correspond directly to the fireplaces. Although it might be possible to modify the CSP algorithm to work with the Baltimore rowhouse by revising existing constraints and adding new types of constraints, another much simpler alternative was found, and is adopted.

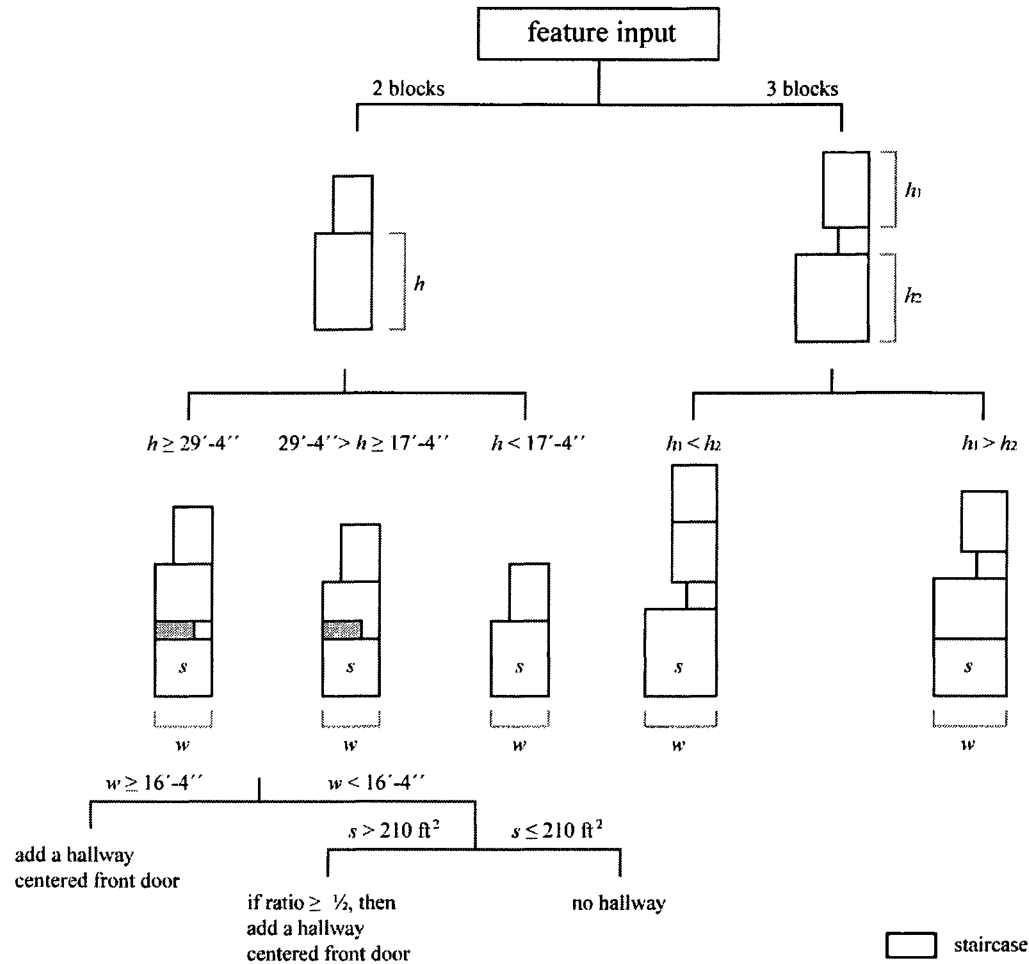


Figure 7-15: Space subdivision tree of Baltimore rowhouses

Procedurally, the first floor of the rowhouse can be determined by a decision tree (Figure 7-15)—essentially, as a subdivisive process. The first floor is typically decomposed into two or three rectangular blocks: a block containing a parlor towards the front, a block containing a kitchen towards the rear, and an optional, smaller central block that connects the two. In a three-block rowhouse, the central block contains a pantry or a stair, while the front and rear blocks are divided into one or two rooms. The kitchen is always the rear-most space while the parlor is the front-most space. The dining room usually appears in the front block behind the parlor or in the rear block forward of the kitchen. The two cases can be distinguished by comparing the depths of the front (h_2) and rear (h_1) blocks.

Two-block rowhouses are more involved. Depending on the depth (h) of the front block, it can contain a single room, or be divided into a parlor and dining room possibly separated by a staircase. If the front block comprises two rooms, the staircase can occupy an enclosed space or it can be open to one or both rooms. If the front block comprises a single room, the staircase may have multiple possible arrangements. These configurations are too complicated to be handled by the decision tree, which needs further refinement by using shape rules.

Regardless of whether the layout has two or three blocks, the front door enters into the front-most room or a dedicated hallway. This is determined from the width (w) and area (s) of the front-most room.

7.7.2 Layout tree pruning of rowhouses

Figure 7-16 shows the layout tree of the old Baltimore rowhouse grammar. Noticeably, after applying the shape rules for several steps from the initial shape, the layout must be one of two shaded nodes or a horizontal reflection of the two. On the other hand, the initial layout estimation of Baltimore rowhouses by space subdivision reaches the same results after decomposing the footprint input into rectangles. Thus, all parameters can be fixed at this step, and the desired layouts are obtained simply by continually application of shape rules. For the new computation-friendly version (Section 6.4), the grammar is designed to start from the rectangular decomposition of the footprint input so that the parameter-fixing step is automatically handled. Figure 7-17 shows sample results from the layout determination of Baltimore rowhouses; in each figure, the left is the truth, the middle is the layout determined and the right is the feature input.

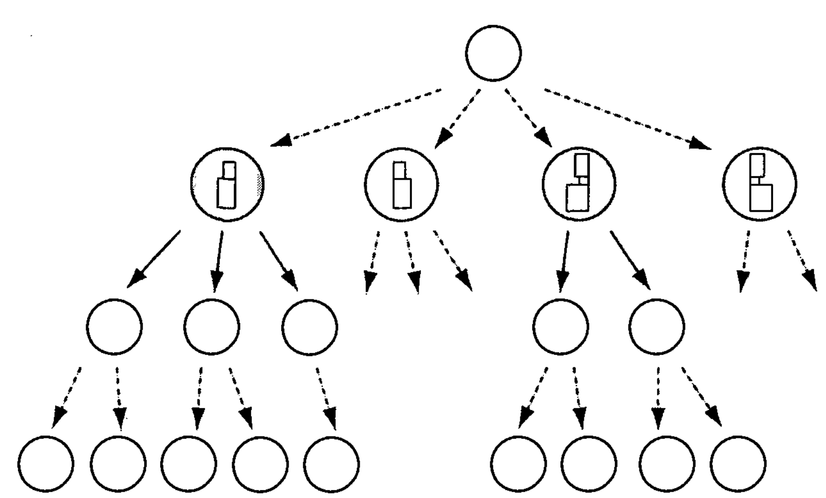


Figure 7-16: The layout tree of the traditional Baltimore Rowhouse grammar

Figure 7-17: Layout results of Baltimore rowhouses

Chapter 8 Conclusion

This thesis starts out with the problem of determining the interior layout of a building from its exterior features and tackles it with the help of a shape grammar that describes its underlying building style. The general approach adopted is based on the fact that the derivation of the entire language space of a shape grammar can be described by a tree structure. The general approach is supported and detailed by two test cases: the Baltimore Rowhouse and the Queen Anne House.

Conversely, the research question serves as a vehicle to investigate shape grammars, in particular, the implementation of a parametric shape grammar interpreter. The difficulty involved leads to an analysis of the computational complexity of interpreting shape grammars, particularly, by taking advantage of formal languages and identifying factors influencing tractability. The conclusion is that the implementation of shape grammars can become, in principle, intractable.

In practice, however, many shape grammars are special in a way in that they are tractable. That is, a practical, 'general' paradigm for implementing such parametric shape grammars can be and is considered. To ensure that shape grammars designed are tractable, the concept of computation-friendly shape grammars is introduced, for which there is a data structure, underlying manipulation algorithms and a meta-language for specifying shape rules. These three components together form a sub-

framework and many such sub-frameworks constitute the 'general' paradigm. For such a 'general' paradigm, it is desirable to classify tractable shape grammars in way so that the total number of sub-frameworks is minimal. And in this way, a classification based on the underlying data structures is argued to be optimal.

In this chapter, I conclude with an overview of the main contributions of this dissertation and an agenda for future research.

8.1 Contributions

The main contributions can be divided into two categories: those relating to the implementation of a parametric shape grammar interpreter and those relating to determining the interior layout of a building from its exterior image.

The following are the contributions with regard to implementing a parametric shape grammar interpreter:

- As corollaries from the fact that shape grammars are capable of simulating a Turing machine, three theoretical results are derived, which are significant for implementing a shape grammar interpreter, namely: a shape grammar may not halt; the language space of a shape grammar can be exponentially large; and the problem of parsing a configuration against a shape grammar (that is, the membership problem) is, in general, unsolvable.
- A practical, 'general' paradigm for implementing tractable parametric shape grammars comprising a set of sub-frameworks one for each subclass of shape grammars. Classifying shape grammars by the underlying data structures is argued to be optimal.
- Three exemplar sub-frameworks are described. These include the rectangular sub-framework based on space partitioning and space aggregation; the polygonal sub-framework based on polygonal subdivision using cutting polylines; and a graph sub-framework based on graph transformation. For each there is a description of the data structure, basic manipulation

algorithms, and meta-language used in describing shape rules so that further code translation can be easily carried out.

The following are the contributions with respect to determining the interior layout of a building from exterior image data:

- A general approach for determining the interior layouts of buildings describable by shape grammars by using their exterior features as input.
- A realistic pipeline for semi-automatically extracting building features from image data together with the details of the necessary algorithms.

8.2 Future research

To fully apply to practice the results of the research in this dissertation, the following additional research are essential:

- *Implementing a common platform for the 'general' paradigm*

  The practical, 'general' paradigm for implementing parametric shape grammars should rely on a common platform so that all sub-interpreters serve as Add-ons, which can be developed independently (Section 4.3). However, the implementation of such a common platform is a significant endeavor, which would need the efforts of a research team.

- *Implementing a realistic pipeline for building feature extraction*

  A realistic pipeline which is capable of semi-automatically extracting building features from image data is described in Section 3.4 and left unimplemented. An implementation of such a pipeline will benefit other similar projects, which require the features of existing buildings as input. It would be preferable for such an implementation to be independent and transparent so that it can be easily revised to fit the special needs of a target project.

The research in this dissertation covers topics from a wide spectrum, which can also leads to other avenues of further research:

- *Ontology-backed shape grammars*

  In the manner by which they are developed, shape grammars tend to capture knowledge common to a set of buildings based on a subset of building samples. In this respect there are similarities to machine learning techniques (Mitchell, 1997). However, the development of a shape grammar is essentially manual, and as such, lacks a mechanism of learning on the fly. Moreover, as described early, during conversion from a traditional shape grammar to one that is computation-friendly, a number of values are manually computed from building samples. The central notion is that certain aspects of shape grammars could be dynamically updated when updating ontology related to subject domain of the grammars. This is similar to learning on the fly.

It is clear that more research is required, and research into building ontology provides a new opportunity. Building ontology (Grobler et al., 2008) captures knowledge about the design of particular building types. Features of a building that relate to design include size and dimensions, surrounding area, façade treatment, circulation and movement patterns, form, structure, materials, etc. The ontology can be used to describe the relationships between these elements, as well as the social, cultural, and environmental factors.

Such ontology can be used to back up both the development and quantification of shape grammars. By building a link between shape grammars and a corresponding ontology, we may be able to dynamically update the shape grammar developed. For example, the threshold value of Figure 6-11 can be just a compound query to a building ontology, and such a value could be updated automatically whenever new building samples are added into the building ontology. This idea points to a new research direction: of particular interest is the re-development of existing shape grammars based on building ontology and with it establish comparisons with the existing grammars; another is to examine mechanisms for creating a dynamic link between shape grammars and a building ontology.

- *Parsing a configuration against a shape grammar*

  It has been shown that the membership problem, that is, parsing a configuration against a shape grammar, is in general unsolvable. This is analogous to the membership problem for the unrestricted grammar in the field of generative grammars. However, there are also special classes of generative grammars, for example, context-free grammars, which can be parsed. The question of whether shape grammars can be similarly restricted so that a particular subclass of shape grammars can be parsed remains open.

- *Equality of pattern books and shape grammars*

  In this dissertation, the focus is on buildings describable by shape grammars (Section 1.2). However, the exact scope that this focus covers is still an open question. That is, can all buildings describable by pattern books also be describable by shape grammars? Moreover, if we restrict the class of buildings to those that have been constructed, and further, if we were to reverse engineer the assembly of their construction processes, we might be able to provide a pattern book description for their designs. The question then becomes whether or not, most typical buildings can be described by certain kinds of pattern books? These questions require substantive further research.

- *Extending the computation-friendly concept to sorts*

  Sorts is a concept based on the recognition that there is always a need for different representations of the same entity, albeit a shape or some other complex attribute (Stouffs and Krishnamurti, 1997). Conceptually, a sort specifies a set of similar models; sorts combine to form new sorts under operations closed within an algebra based on a part relationship. Common arithmetic operations between sorts including subsort ($\leq$), sum (+), product ($\cdot$), difference (–) and Cartesian product ($\times$) (Stouffs and Krishnamurti, 1998). The definition of a sort includes a specification of the operational behaviour of its individuals for common arithmetic operations. The foundation for sorts is an algebraic model for shapes (Stouffs, 1994) that offers a uniform and consistent approach for dealing with geometries of

mixed-dimensionality and is extended to non-spatial attributes using common arithmetic operations of sum, difference and product. In a curious twist, sorts originated from weighted shape grammars (Stiny, 1992), and then extended to shaped weights and other weighted generalizations. The basic mechanisms for shape grammars and sorts are 'isomorphic' — that is, both achieve a goal by a set of basic polymorphic operators. As a result, the analysis of factors influencing tractability of shape grammars (Section 4.2) applies equally to sorts, in particular, to sorts dealing with geometries. Moreover, the concept of computation-friendly is equally applicable to sorts. Completing the details of the above argument is left as future research — indeed, another dissertation in its own right.

References

AGARWAL, M. & CAGAN, J. (1998) A blend of different tastes: the language of coffee makers. *Environment and Planning B: Planning and Design,* 25, 205-226.

AGARWAL, M., CAGAN, J. & STINY, G. (2000) A micro language: generating MEMS resonators by using a coupled form-function shape grammar. *Environment and Planning B: Planning and Design,* 27, 615-626.

AKIN, O., DAVE, B. & PITHAVADIAN, S. (1992) Heuristic generation of layouts (HeGeL): based on a paradigm for problem structuring. *Environment and Planning B: Planning and Design,* 19, 33-59.

ANDRIES, M., ENGELS, G., HABEL, A., HOFFMANN, B., KREOWSKI, H.-J., KUSKE, S., PLUMP, D., SCHURR, A. & TAENTZER, G. (1999) Graph transformation for specification and programming. *Sci. Comput. Program.,* 34, 1-54.

BARNHILL, R. E., FARIN, G., JORDAN, M. & PIPER, B. R. (1987) Surface/surface intersection. *Comput. Aided Geom. Des.,* 4, 3-16.

BECKER, S. C. (1997) Vision assisted modeling from model based video representation. PhD dissertation. *Department of Architecture.* MIT, Cambridge, MA.

BROUNO, C. (1990) Graph rewriting: an algebraic and logic approach. *Handbook of theoretical computer science (vol. B): formal models and semantics.* MIT Press.

CANNY, J. (1986) A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.,* 8, 679-698.

CARLSON, C. (1993) Grammatical programming: An algebraic approach to the description of design spaces. PhD dissertation. *School of Architecture.* Carnegie Mellon University, Pittsburgh, PA.

CARLSON, C., MCKELVEY, R. & WOODBURY, R. (1991) An introduction to structure and structure grammars. *Environment and Planning B: Planning and Design,* 18, 417-426.

CHASE, S. C. (1989) Shapes and shape grammars: from mathematical model to computer implementation. *Environment and Planning B: Planning and Design,* 16, 215-242.

CHASE, S. C. (1997) Emergence, creativity and computational tractability in shape grammars. IN EDMONDS, E. & MORAN, T. (Eds.) *Interactive Systems for Supporting the Emergence of Concepts and Ideas, CHI'97.* Atlanta, GA.

CHAU, H.-H. (2002) Preserving brand identity in engineering design using a grammatical approach. PhD dissertation. *School of Mechanical Engineering and Keyworth Institute of Manufacturing and Information Systems.* The University of Leeds,

CHAU, H. H., CHEN, X., MCKAY, A. & PENNINGTON, A. (2004) Evaluation of a 3D shape grammar implementation. IN GERO, J. S. (Ed.) *Design Computing and Cognition '04.* Boston.

CHIEN, S.-F., DONIA, M., SNYDER, J. D. & TSAI, W.-J. (1998) SG-CLIPS: A System to Support the Automatic Generation of Designs from Grammars. *CAADRIA '98.* Osaka, Japan.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L. & STEIN, C. (2004) *Introduction to Algorithms, Second Edition,* The MIT Press.

CSURKA, G., DANCE, C. R., FAN, L., WILLAMOWSKI, J. & BRAY, C. (2004) Visual categorization with bags of keypoints. *Workshop on Statistical Learning in Computer Vision, ECCV.*

DAMSKI, J. C. & GERO, J. S. (1997) An evolutionary approach to generating constraint-based space layout topologies. IN JUNGE, R. (Ed.) *CAAD Futures 1997.* Kluwer, Dordrecht.

DONATH, D. & BÖHME, L. F. G. (2007) Constraint-Based Design in Participatory Housing Planning. *25th eCAADe Conference Proceedings* Frankfurt, Germany.

DOWNING, A. J. (1981) *Victorian cottage residences,* New York, Dover Publications.

DREWES, F., KREOWSKI, H.-J. & SCHWABE, N. (1996) COLLAGE-ONE: A system for evaluation and visualisation of collage grammars. *Machine Graphics & Vision,* 5, 393-402.

DREWES, F. & KREOWSKI, H. J. (1999) Picture generation by collage grammars. *Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools.* World Scientific Publishing Co., Inc.

FABER, P. & FISHER, B. (2002) How can we exploit typical architectural structures to improve model recovery? *3D Data Processing Visualization and Transmission.* Padova, Italy.

FISCHLER, M. A. & BOLLES, R. C. (1981) Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM,* 24, 381-395.

FLEMMING, U. (1987) More than the sum of parts: the grammar of Queen Anne houses. *Environment and Planning B: Planning and Design,* 14, 323-350.

FLEMMING, U., GINDROZ, R., COYNE, R. & PITHAVADIAN, S. (1985) A pattern book for Shadyside. *Technical report.* Pittsburgh, PA, Deparment of Architecture, Carnegie Mellon University.

FRUEH, C., SAMMON, R. & ZAKHOR, A. (2004) Automated texture mapping of 3D city models with oblique aerial imagery. *2nd International Symposium on 3D Data Processing, Visualization and Transmission.*

FULKERSON, B., VEDALDI, A. & SOATTO, S. (2008) Localizing Objects with Smart Dictionaries. *the European Conference on Computer Vision (ECCV 2008).*

GIPS, J. (1974) Shape Grammars and Their Uses. PhD dissertation. *Computer Science Department.* Stanford University, Stanford, California.

GIPS, J. (1975) *Shape Grammars and Their Uses: Artificial Perception, Shape Generation and Computer Aesthetics,* Birkhaüser, Basel, Switzerland.

GIPS, J. (1999) Computer implementation of shape grammars. Cambridge, MA: NSF/MIT Workshop on Shape Computation.

GRADY, B., ROBERT, M., MICHAEL, E., BOBBI, Y., JIM, C. & KELLI, H. (2007) *Object-oriented analysis and design with applications, third edition*, Addison-Wesley Professional.

GREINER, G. & HORMANN, K. (1998) Efficient clipping of arbitrary polygons. *ACM Trans. Graph.*, 17, 71-83.

GROBLER, F., AKSAMIJA, A., KIM, H., KRISHNAMURTI, R., YUE, K. & HICKERSON, C. (2008) Ontologies and shape grammars: communication between knowledge-based and generative systems. *To appear in Third International Conference on Design Computing and Cognition (DCC'08), Atlanta, GA, June 2008.*

GROSS, M. D. (1986) Design as exploring constraints. PhD dissertation. *Department of Architecture*. Massachusetts Institute of Technology, Cambridge, MA.

HAN, F., TU, Z. & ZHU, S.-C. (2004) Range Image Segmentation by an Effective Jump-Diffusion Method. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26.

HARRY, R. L. & CHRISTOS, H. P. (1997) *Elements of the theory of computation*, Prentice Hall PTR.

HAYWARD, M. E. (1981) Urban Vernacular Architecture in Nineteenth-Century Baltimore. *Winterthur Porfolio*, 16, 33-63.

HAYWARD, M. E. & BELFOURE, C. (2005) *The Baltimore Rowhouse*, New York, Princeton Architectural Press.

HECKEL, R. (2006) Graph transformation in a nutshell. *Electronic Notes in Theoretical Computer Science*, 148, 187-198.

HEISSERMAN, J. (1991) Generative geometric design and boundary solid grammars. PhD dissertation. *Department of Architecture*. Carnegie Mellon University, Pittsburgh, PA.

HEISSERMAN, J. (1994) Generative geometric design. *IEEE Computer Graphics and Applications*, 14, 37-45.

HEISSERMAN, J. & WOODBURY, R. (1993) Generating languages of solid models. *the second ACM symposium on solid modeling and applications*. Montreal, Quebec, Canada, ACM.

HERBERT, M. & KROTKOV, E. (1992) 3D measurements from imaging laser radars: how good are they? *Image Vision Comput.*, 10, 170-178.

HOOVER, A., JEAN-BAPTISTE, G., JIANG, X., FLYNN, P. J., BUNKE, H., GOLDGOF, D. B., BOWYER, K., EGGERT, D. W., FITZGIBBON, A. & FISHER, R. B. (1996) An experimental comparison of range image segmentation algorithms. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 18, 673-689.

HUBER, D. & HEBERT, M. (2001) Fully automatic registration of multiple 3D data sets. *IEEE Computer Society Workshop on Computer Vision Beyond the Visible Spectrum (CVBVS 2001).*

HUBER, D. & VANDAPEL, N. (2006) Automatic three-dimensional underground mine mapping *The International Journal of Robotics Research*, 25, 7-17.

HUR, S., OH, M.-J. & KIM, T.-W. (2009) Approximation of surface-to-surface intersection curves within a prescribed error bound satisfying G2 continuity. *Computer-Aided Design,* 41, 37-46.

IKEUCHI, K., NAKAZAWA, A., HASEGAWA, K. & OHISHI, T. (2003) The great buddha project: modeling cultural heritage for VR systems through observation *The Second IEEE and ACM International Symposium on Mixed and Augmented Reality.* Tokyo, Japan.

JOWERS, I. (2006) Computation with Curved Shapes: Towards Freeform Shape Generation in Design. PhD dissertation. *Department of Design and Innovation.* The Open University, Milton Keynes.

JOWERS, I., PRATS, M., EARL, C. & GARNER, S. (2004) On curves and computation with shapes. *Generative CAD systems symposium: G-CAD 2004.* Carnegie Mellon University, Pittsburgh, PA.

KATZ, S. & SEDERBERG, T. W. (1988) Genus of the intersection curve of two rational surface patches. *Computer Aided Geometric Design,* 5, 253-258.

KNIGHT, T. W. (1980) The generation of hepplewhite-style chair back designs *Environment and Planning B: Planning and Design,* 7, 227-238.

KNIGHT, T. W. (1981a) The forty-one steps: the languages of Japanese tea-room designs. *Environment and Planning B: Planning and Design,* 8, 97-114.

KNIGHT, T. W. (1981b) Languages of design: from known to new. *Environment and Planning B: Planning and Design,* 8, 213-238.

KNIGHT, T. W. (1983) Transformations of languages of designs. *Environment and Planning B: Planning and Design,* 10, 125-177.

KNIGHT, T. W. (1989) Transformations of the De Stijl art: the paintings of Georges Vantangerloo and Fritz Glarner. *Environment and Planning B: Planning and Design,* 16, 51-98.

KNIGHT, T. W. (1999) Shape grammars: six types. *Environment and Planning B: Planning and Design,* 26, 15-31.

KNIGHT, T. W. & STINY, G. (2001) Classical and non-classical computation. *Information technology,* 5, 355-372.

KONING, H. & EIZENBERG, J. (1981) The language of the prairie: Frank Lloyd Wright's prairie houses. *Environment and Planning B: Planning and Design,* 8, 295-323.

KRISHNAMURTI, R. (1981) The construction of shapes. *Environment and Planning B: Planning and Design,* 8, 5-40.

KRISHNAMURTI, R. (1982) SGI: A shape grammar interpreter. *Technical report, Design Discipline.* The Open University.

KRISHNAMURTI, R. (1992a) The arithmetic of maximal planes. *Environment and Planning B: Planning and Design,* 19, 431-464.

KRISHNAMURTI, R. (1992b) The maximal representation of a shape. *Environment and Planning B: Planning and Design,* 19, 267-288.

KRISHNAMURTI, R. & EARL, C. F. (1992) Shape recognition in three dimensions. *Environment and Planning B: Planning and Design,* 19, 585-603.

KRISHNAMURTI, R. & GIRAUD, C. (1986) Towards a shape editor: the implementation of a shape generation system. *Environment and Planning B: Planning and Design,* 13, 391-403.

KRISHNAMURTI, R. & STOUFFS, R. (1997) Spatial change: continuity, reversibility and emergent shapes. *Environment and Planning B: Planning and Design,* 24, 359-384.

KRISHNAMURTI, R. & STOUFFS, R. (2004) The boundary of a shape and its classification. *The Journal of Design Research,* 4.

KRISHNAN, V. (1990) Constraint reasoning and planning in concurrent design. Pittsburgh, Carnegie Mellon University, Robotics Institute, tech report no. CMU-RI-TR-90-03.

LEE, M.-X. & LEE, J.-H. (2006) Form, style and function - A constraint-based generative system for spartment façade design. *24th eCAADe Conference Proceedings*

LEVOY, M., PULLI, K., CURLESS, B., RUSINKIEWICZ, S., KOLLER, D., PEREIRA, L., GINZTON, M., ANDERSON, S., DAVIS, J., GINSBERG, J., SHADE, J. & FULK, D. (2000) The digital Michelangelo project: 3D scanning of large statues IN AKELEY, K. (Ed.) *Siggraph 2000, Computer Graphics Proceedings.* ACM Press / ACM SIGGRAPH / Addison Wesley Longman.

LOWE, D. G. (1987) Three-dimensional object recognition from single two-dimensional images. *Artificial Intelligence,* 31, 355-395.

LUMELSKY, V. J. (1985) On fast computation of distance between line segments. *Information Processing Letters,* 21, 55-61.

LUND, E. & YOST, P. (1997) Deconstruction - Building Disassembly and Material Salvage: The Riverdale Case Study. Upper Marlboro, MD, NAHB Research Center, Inc.

MCCORMACK, J. P. & CAGAN, J. (2003) Increasing the scope of implemented shape grammars: a shape grammar interpreter for curved shapes. *ASME 2003 International Design Engineering Technical Conferences and Information in Engineering Conference.* Chicago, Illinois, USA.

MCCORMACK, J. P., CAGAN, J. & VOGEL, C. M. (2004) Speaking the Buick language: capturing, understanding, and exploring brand identity with shape grammars *Design Studies,* 25, 1-29.

MCGILL, M. C. (2001) A visual approach for exploring computational design. The Department of Architecture, Massachusetts Institute of Technology,

MIKHAIL, E. M., BETHEL, J. & MCGLONE, J. C. (2001) *Introduction to modern photogrammetry,* John Wiley and Sons, Inc.

MITCHELL, T. M. (1997) *Machine learning,* The McGraw-Hill Companies, Inc.

NARAHARA, T. & TERZIDIS, K. (2006) Multiple-constraint genetic algorithm in housing design. *the 25th Annual Conference of the Association for Computer-Aided Design in Architecture.*

NIKLAUS, W. (1978) *Algorithms + Data Structures = Programs,* Prentice Hall PTR.

O'ROURKE, J. (1998) *Computational geometry in C (2nd ed.),* Cambridge University Press.

PATRIKALAKIS, N. M., MAEKAWA, T., KO, K. H. & MUKUNDAN, H. (2004) Surface to surface intersections. *Computer-Aided Design and Applications,* 1, 449-458.

PAUL, J. B. (1988) Active, optical range imaging sensors. *Mach. Vision Appl.*, **1**, 127-152.
PIAZZALUNGA, U. & FITZHORN, P. (1998) Note on a three-dimensional shape grammar interpreter. *Environment and Planning B: Planning and Design,* **25**, 11-30.
PRATS, M., JOWERS, I., EARL, C. & GARNER, S. (2004) Generative curves in product design. *Design Computing and Cognition DCC'04.* MIT, Cambridge, MA.
PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T. & FLANNERY, B. P. (2007) *Numerical Recipes: The Art of Scientific Computing. Third Edition,* Cambridge University Press.
ROBERTS, D., WARNAAR, D., MENOZZI, A., SAADAT, S. & SNARSKI, S. (2007) Rapid high-confidence intelligent interior mapping. *Unmanned Systems Technology IX.* 1 ed. Orlando, FL, USA, SPIE.
ROTHER, C. (2002) A new approach to vanishing point detection in architectural environments. *Image and Vision Computing,* **20**, 647-655.
ROZENBERG, G. (Ed.) (1997) *Handbook of graph grammars and computing by graph transformation: volume I. foundations,* World Scientific Publishing Co., Inc.
RUSSELL, S. & NORVIG, P. (2002) *Artificial Intelligence: A Modern Approach,* Prentice Hall.
SHELDEN, D. R. (1996) Shape grammar editor, beta version SG1.0B. *DSOF.* Massachusetts Institute Technology.
SHIH, N.-J. & WANG, P.-H. (2004) Point-Cloud-Based Comparison between Construction Schedule and As-Built Progress: Long-Range Three-Dimensional Laser Scanner's Approach. *Journal of Architectural Engineering,* **10**, 98-102.
SHUFELT, J. A. (1999) Performance evaluation and analysis of vanishing point detection techniques. *Pattern Analysis and Machine Intelligence, IEEE Transactions on,* **21**, 282-288.
SICILIANO, B. & KHATIB, O. (Eds.) (2008) *Springer Handbook of Robotics,* Springer.
SIMONDETTI, A. (1997) Rapid prototyping in early stages of architectural design. *Department of Architecture.* Massachusetts Institute of Technology, Cambridge, Massachusetts.
SIVIC, J., RUSSELL, B. C., EFROS, A. A., ZISSERMAN, A. & FREEMAN, W. T. (2005) Discovering objects and their location in images. *Computer Vision, 2005. ICCV 2005. Tenth IEEE International Conference on.*
STAMOS, I. (2001) Geometry and Texture Recovery of Scenes of Large Scale: Integration of Range and Intensity Sensing. PhD thesis. *Computer Science Department.* Columbia University, New York, NY.
STAMOS, I. & ALLEN, P. K. (2000) 3-D model construction using range and image data. *IEEE Conference on Computer Vision and Pattern Recognition.*
STAMOS, I. & ALLEN, P. K. (2002) Geometry and texture recovery of scenes of large scale. *Comput. Vis. Image Underst.,* **88**, 94-118.

STINY, G. (1975) Pictorial and formal aspects of shape and shape grammars and aesthetic systems. PhD dissertation. *System Science.* University of California, Los Angeles, CA.

STINY, G. (1977) Ice-ray: a note on Chinese lattice designs. *Environment and Planning B: Planning and Design,* 4, 89-98.

STINY, G. (1980a) Introduction to shape and shape grammars. *Environment and Planning B: Planning and Design,* 7, 343-351.

STINY, G. (1980b) Kindergarten grammars: designing with Froebel's building gifts. *Environment and Planning B: Planning and Design,* 7, 409-462.

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

STINY, G. (1991) The algebras of design. *Research in Engineering Design,* 2, 171-181.

STINY, G. (1992) Weights. *Environment and Planning B: Planning and Design,* 19, 413-430.

STINY, G. (1994) Shape rules: closure, continuity, and emergence. *Environment and Planning B: Planning and Design,* 21, s49-s78.

STINY, G. (2006) *Shape: Talking about seeing and doing*, MIT Press, Cambridge.

STINY, G. & GIPS, J. (1971) Shape grammars and the generative specification of painting and sculpture. IN FREIMAN, C. V. (Ed.) *Information Processing 71, North Holland, Amsterdam.*

STOUFFS, R. (1994) The algebra of shapes. PhD dissertation. *Department of Architecture.* Carnegie Mellon University, Pittsburgh, PA.

STOUFFS, R. & KRISHNAMURTI, R. (1993) The complexity of the maximal representations of shapes. *The IFIP WG 5.2 Workshop on Formal Design Methods for CAD.* Talinn, Estonia.

STOUFFS, R. & KRISHNAMURTI, R. (1997) Sorts: a concept for representational flexibility. IN JUNGE, R. (Ed.) *CAAD Futures 1997.* The Netherlands, Kluwer Academic, Dordrecht.

STOUFFS, R. & KRISHNAMURTI, R. (1998) An algebraic approach to comparing representations. IN BARALLO, J. (Ed.) *Mathematics & Design.* The University of the Basque Country, San Sebastian, Spain.

STOUFFS, R. & KRISHNAMURTI, R. (2006) Algorithms for the classification and construction of the boundary of a shapes. *Journal of Design Research,* 5, 54-95.

TANG, P., HUBER, D. & AKINCI, B. (2007) A Comparative Analysis of Depth Discontinuity and Mixed Pixel Detection Algorithms. *The 6th International Conference on 3-D Digital Imaging and Modeling.* Montréal, Québec, Canada, IEEE.

TAPIA, M. (1996) From shape to style. Shape grammars: issues in representation and computation, presentation and selection. PhD dissertation. *Department of Computer Science.* University of Toronto, Toronto, Canada.

TAPIA, M. (1999) A visual implementation of a shape grammar system. *Environment and Planning B: Planning and Design,* 26, 59-73.

TRESCAK, T., RODRIGUEZ, I. & ESTEVA, M. (2009) General shape grammar interpreter for intelligent designs generations. *CGIV09.* Tianjin University, Tianjin, China.

TULEY, J., VANDAPEL, N. & HEBERT, M. (2005) Analysis and Removal of Artifacts in 3-D LADAR Data. *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on.*

VOSSELMAN, G., GORTE, B. G. H., SITHOLE, G. & RABBANI, T. (2004) Recognizing structure in laser scanner point clouds. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences.* Freiburg, Germany.

WANG, Y. (1998) 3D architecture form synthesizer. Master thesis. *Department of Architecture.* Massachusetts Institute of Technology, Cambridge, Massachusetts.

WINOGRAD, T. & FLORES, F. (1986) *Understanding Computers and Cognition: A New Foundation for Design,* Intellect Books.

YOON, K. B. (1992) *A constraint model of space planning,* Southampton, UK.

ZHANG, J., MARSZA, M., EK, LAZEBNIK, S. & SCHMID, C. (2007) Local Features and Kernels for Classification of Texture and Object Categories: A Comprehensive Study. *Int. J. Comput. Vision,* 73, 213-238.