SYNTHESIS OF NON-REGULAR ARCHITECTURAL FORMS

NG CHU MING

(B.Eng.(Computer Engineering)(Hons), NUS)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2006

Name: Ng Chu Ming

Degree: Master of Science

Dept: Department of Computer Science

Thesis Title: Synthesis of Non-Regular Architectural Forms

Abstract

This thesis presents an interactive system for synthesizing non-regular architectural forms that features a growth-based approach for rapidly constructing architectural models in the large scale, while complementing it with intelligent form completion for rapid design synthesis on the small scale. Our growth-based strategy for large scale synthesis allows the incorporation of different behavioral growth routines to generate building units meeting the requirements of view and shape variation. In maximizing views, we exploit the capability of the GPU to achieve good speedup of the computation. For designing in the smaller scale, our system supports fast generation of design plans, rapid form completion and synthesis. Besides optimizing these requirements, our approach exhibits emerging behavior with outputs having complex but useful inter-lockings. These emergent features suggest clever utilization of space as, for example, suspended open spaces, mid-air gardens and ventilation corridors that are of good architectural value. On the whole, our work advances interactive generative design with objective measures that are hard, if not impossible, to achieve manually. It contributes towards rapid prototyping and in taking design of forms beyond monotonous cookie-cutter architecture.

Keywords: modeling, building design, architecture, GPU

Acknowledgments

It took great effort earning this honor of being bestowed the liberty to have one page in my piece of scholarly work in which to dedicate to all the people important in my life. Thank you very much to everyone who has accompanied me in this wonderful journey of trials, tribulations and most importantly, one of great fun and fulfilment in pursuing my passion.

My deepest gratitude goes to thesis advisor and friend, A/P Tan Tiow Seng for having faith in me and taking me into the research group. I am deeply impressed by the professionalism and passion that you have for your work. Your belief in integrity in one's work and your meticulous eye for detail never fails to amaze me and will always be the shining example of what I shall always try to emulate. Thank you so much and I would not be where I am today without you.

My faithful friend Quek Boon Kiat, thanks so much for being my best confidante and being always there for me. You are the best!

To my friends in graduate school, Ng Wee Teck, Geoffrey Koh, Koh Chung Haur, Donny Soh and also Low Joo Kai for all the fun times that made graduate student life more bearable. Thanks for all the encouragement in times of self doubt!

To the Wushu Family - Goh Wah Ing, Thien Vui Khien, Tiew Ghim Chuan and Lau Yiehui for being such a wonderful bunch of people with whom wushu training will never be the same. And thanks for helping me out when I'm in trouble.

I would also like to thank Ms Joanna Koh without whom I will not be a better person that I am today. Thanks for your love in that four years.

My deepest gratitude also to Ms Koh Yihan. I absolutely wouldn't be able to make it without you. You are my guardian angel!

Finally, I would like to dedicate this work to my most beloved parents, whose care, concern, support and love I will always be grateful for. Hopefully, one day I will be able to honor them and write yet another page of acknowledgements again.

Till then...

Carpe Diem. And keep believing in your Rendezvous with Destiny...

Contents

List of Tables

List of Figures

Summary

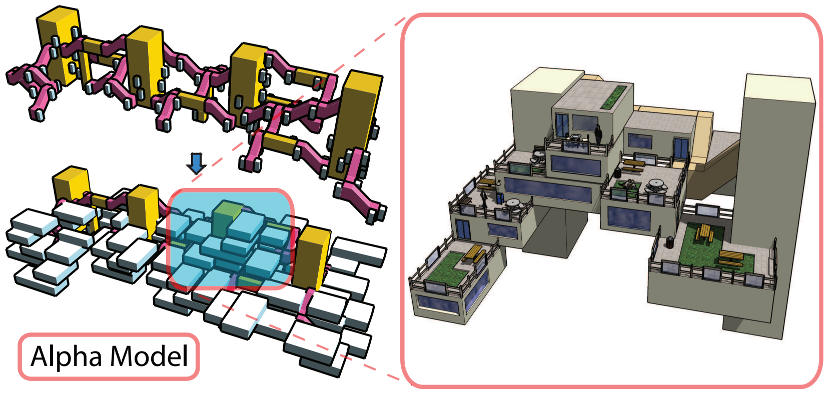
This thesis present an interactive system for synthesizing non-regular architectural forms that features a growth-based approach for rapidly constructing architectural models in the large scale, while complementing it with intelligent form completion for rapid design synthesis on the small scale. Our growth-based strategy for large scale synthesis allows the incorporation of different behavioral growth routines to generate building units meeting the requirements of view and shape variation. In maximizing views, we exploit the capability of the GPU to achieve good speedup of the computation. For designing in the smaller scale, our system supports fast generation of design plans, rapid form completion and synthesis. Besides optimizing these requirements, our approach exhibits emerging behavior with outputs having complex but useful inter-lockings. These emergent features suggest clever utilization of space as, for example, suspended open spaces, mid-air gardens and ventilation corridors that are of good architectural value. On the whole, our work advances interactive generative design with objective measures that are hard, if not impossible, to achieve manually. It contributes towards rapid prototyping and in taking design of forms beyond monotonous cookie-cutter architecture.

Chapter 1

Introduction

As civilization progresses, the notion of a Housing Unit or, simply *unit* in one aspect of the modern ideology was seen in utilitarian terms as a facility that could be capable of removing waste, providing light and space for use. But beyond such provisions which have become basic requirements prevalent in the contemporary context, there are more important issues to appeal to the occupant of a dwelling. What are other important necessities and functional qualities with respect to spatial arrangements of units, termed *architectural forms* or *forms*, that are appealing to their owners or residents? To this end, this thesis studies generative design of non-regular form that incorporates architectural qualities such as views and shape variations.

Often, due to standardization for economies of scale, units in a precinct such as in a multi-storey building do not vary in spatial configuration and have repetitive structures. Such cookie-cutter approach to housing architecture confine residents to their standardized units. Consequently, housing accommodation is viewed merely as satisfying quantitative parameters of housing a certain number of residents on a given land area. As civilization progresses, housing standards have transcended the mentioned provisions, and architects now deliberately vary the design and layout of units to provide residents with unique living environments beyond just a facility. Overall, there are efforts toward creating non-regular architectural forms such as the famous Moshe Safdie's Habitat'67 in Figure 1.1 [Safdie, 1967]. Such designs are, however, rare as they are complex to conceptualize and visualize and realize manually.



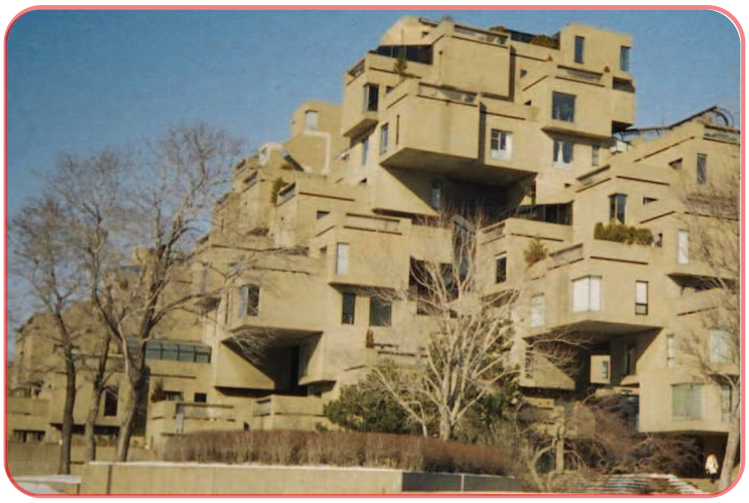


Figure 1.1: Moshe Safdie's Habitat '67, in Montreal, Canada.

In view of these changes, building architects and designers are constantly exploring new ways of using computational means to materialize novel designs. However, in this aspect, computational tools for design generation remain rudimentary and difficult to use. Commercial computer-aided design tools till present day remain synonymous with popular software packages, such as Autodesk's AutoCAD© and Revit© [Inc., 2005]. These are mainly drafting tools that are hardly generative in nature.

Existing research work in generating forms often produce outputs that are regular or ones that still require much post-processing by the architect before it is in a form in which is physically realizable. As such, the design process places tremendous load on the shoulders of architects. Thus, the task of creating unique forms and spatial arrangements is an arduous endeavor that quickly overwhelms the designer. At the same time, the architecture community is still fervently embracing new paradigm shifts in designing non-regular architectural forms, as attested to in [Schittich, 2004] and [Herr, 2003].

In this thesis, we introduce a novel growth-based approach to generative design, and develop a prototype interactive system. The approach begins with seeds of units planted in space to allow them to claim space as time progresses in a simulation manner to eventually end as units of the architectural form. Such approach is flexible as the control of units and their spatial configuration can be incorporated as programmable growth strategies. This growth-based synthesis method is complemented with our form completion engine which allows for rapid interactive synthesis of forms for localized regions in the building model. The final architectural form generated, firstly, consists of units in near-production form that optimize on view and shape variation within preset room proportions, and secondly, among units with emergent features that occupy space in an overall site volume.

The rest of the thesis is organized as follows. Chapter 2 presents related work in generative designs. Chapter 3 discusses architectural issues and states our goals on architectural forms generated. Chapter 4 describes our system components and the process to generate forms. Chapter 5 details the growth strategies of view and shape variation. Chapter 6 describes our data structure for representing floorplan. Chapter 7 describes the form completion engine and Chapter 7.3 details the completion procedure. Chapter 8 showcases our experimental results, and Chapter 9 concludes with future work.

Chapter 2

Related Work

The field of design computing is rapidly gaining relevance and in this chapter, we review a few existing works in generative design. They were considered but found unsuitable in dealing with our problem of optimizing attribute values while generating non-regular architectural forms.

2.1 Shape Grammars

The most prominent method of design generation is that of shape grammars [Chase, 1989]. The power of shape grammar lies in its simplicity and yet ability to produce relatively complicated but coarse designs. It operates via a rule-based reproduction system that is of some similarity in function to L-systems. Figure 2.1 shows an example of the workings of a shape grammar rule (middle diagram) applied to two simple rectangular shapes. Using the reference points (shown as red and grey dots) as indication of relative positioning, the grammar replicates the structure presented by the rule specified. This process is repeated down to rotation symmetries of the specified shapes in the diagram. The output of the shape grammar rule is the final configuration of shapes on the right of Figure2.1

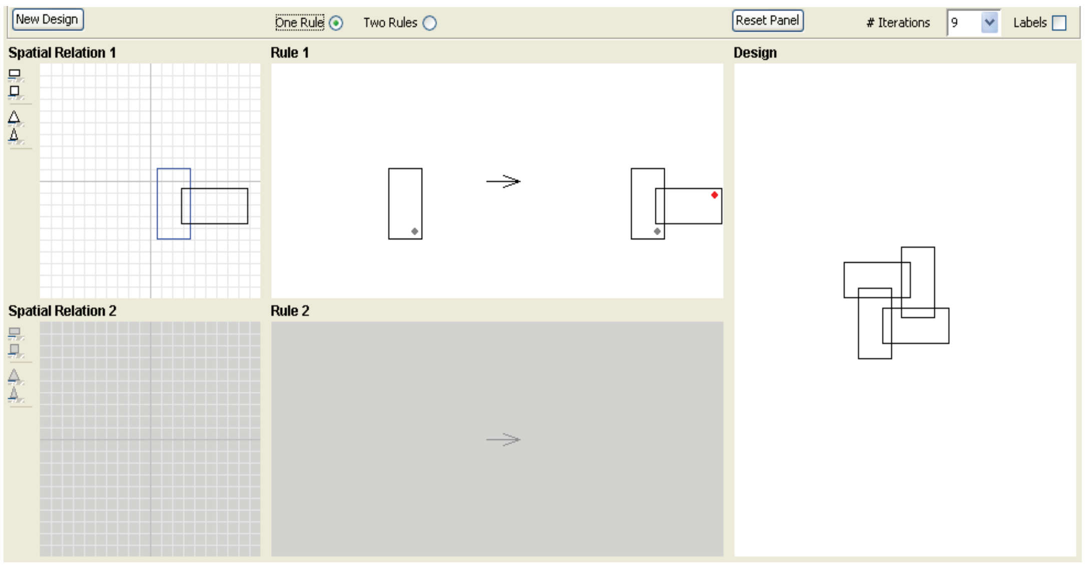


Figure 2.1: Illustration of production of shapes using Shape Grammar.

However, there is no concept of structural constraint abidance neither is there concept of optimization of form attributes. As yet there are no obvious ways to augment shape grammar to generate housing units that obey structural constraints and optimize on attribute values. Structural constraints embody requirements such as unit to corridor or unit to lift core connection. Attribute values denote unit assessment figures such as the number of units having multiple views of the environment.

Furthermore, it is difficult to produce meaningful 3D forms by extending the shape grammar concept to handle 3D reproduction of shapes. As such, shape grammar remains a design tool that only generates coarse spatial layouts of a building which is still far from the final form of an actual physical design that can be realized. It is often unclear as to which are the actual units, which are the corridors and circulation, as well as where the support structure for the building should be.

[Loomis, 2002] presents a synergy of shape grammar and genetic algorithms for improvement in design generation. It is driven by the insight that though shape grammars effectively define a design space, it offers no method of systematically exploring that space. By using a genetic algorithm to cross-breed two of the existing designs so that good design properties can be inherited by the new design, the shapes generated can be effectively explored by the iterations of the genetic algorithm. However, a metric for quantifying the goodness of forms is not available and thus the system requires user input at every generation to indicate the better designs from the rest of the population. Similarly, designs remain coarse due to the basis of using shape grammars as the underlying representation for the generation of coarse forms. To use the system for the design of an entire building will not be straightforward.

2.2 Split Grammars

In the work of [Wonka et al., 2003], split grammars are used to generate 3D building models. In their system, a large database of grammar rules is first created and designs are generated based on repeated application of compatible rules. The system intelligently sieves out and apply only those rules that maintain structure coherence and order like that observed in real buildings. Their approach is able to rapidly generate large cityscapes with complicated 3D building facades in a fully automatic manner. However, it is noted in their paper that the system only generates a description of the syntatic and spatial framework for the final building. It is not clear how the system could be extended to handle attribute values such as floor area requirements, multiple views and non-standard housing units. Furthermore, since split grammars work by splitting an initial basic shape into sub-shapes from its vocabulary, it is difficult to extend it for generating non-standard building forms such as that in Moshie Safdie's Habitat '67.

2.3 L-Systems

L-systems [Prusinkiewicz and Lindenmayer, 1990] is another notable generative design technique with production rules. Such modeling places its emphasis on generating the topological structure of objects. It is popular in modeling structures appearing in nature such as plants and trees. For example, [Měch and Prusinkiewicz, 1996] use L-systems to model plant growth with some form of environmental awareness. In the work of [Parish and Müller, 2001], an extension to L-systems is used for rapid procedural generation of large city scapes. However in building design, there are more imperative requirements such as form attributes that require attention during design generation. This resembles in some sense an optimization of certain quality criteria. Such requirements are however, irrelevant to those objects that L-systems are being used to model, and consequently, production and re-writing systems based on grammar are not well-suited for the synthesis of non-regular architectural forms. For our purpose of generating architectural forms, we utilize relatively more complex optimization criteria than that possible with L-systems. Also, geometric units under our consideration do not exhibit recurring substructure properties that might lend itself toward modeling via L-systems or its variants.

Our problem of generative design with optimization criteria requires fine grained decisions in generating units (that are competing with each other on gaining good attribute values) than that offered by existing grammar and production rules. In exploring possible solutions to the problem, we experimented with a naïve method of generating forms with a set of predefined units. The method iteratively computes plausible placement of units at various free spatial locations until a certain number of desirable units have been placed. However, our experience with this was not successful - it results in very low utilization of space due to the limited choices of units in claiming usable space that is highly fragmented, and it is expensive computationally in its search for usable space. With the above, we thus arrive at our step-by-step growing of units that can accommodate and compromise on the requirements of all concurrently.

Chapter 3

Architecture Primer

In this chapter, we provide a general overview of architectural concerns in building design as well as some architecture terminology for describing major core components of a building form. A discussion of the several objective criteria (termed *attribute values*) used by architects for assessing the quality of form shape is also provided.

3.1 Terminology

In the layout of apartment blocks, the main structural components of a building form are rationalized as follows:

- **Cores**. These are major shafts running through the entire height of the architectural form. They function as support structure of the form. The most common examples here are the lift cores.
- **Circulation**. These refer to stairs and corridors structures that serves to connect to the units in a building.
- **Units**. The actual habitable blocks of space.
- **Site Envelop**. This demarcates a 3D volume of space within which the building

can be build. Conventionally, this is a rectilinear box but arbitrary envelop shapes can also be handled.

There are other major components such as power and sanitation pipes in an architectural form. These are, however, less pertinent to our work here as our primary interest is to generate the main appearance parts of a form.

3.2 Architectural Criteria

With contemporary architecture's move towards non-regular forms of housing units, various criteria have surfaced in the architectural field as objective design goals in designing forms. (For more details, the reader can refer to [Schittich, 2004].) In technical terms, these are *attribute values* we seek to optimize. We provide a brief overview of some common goals below:

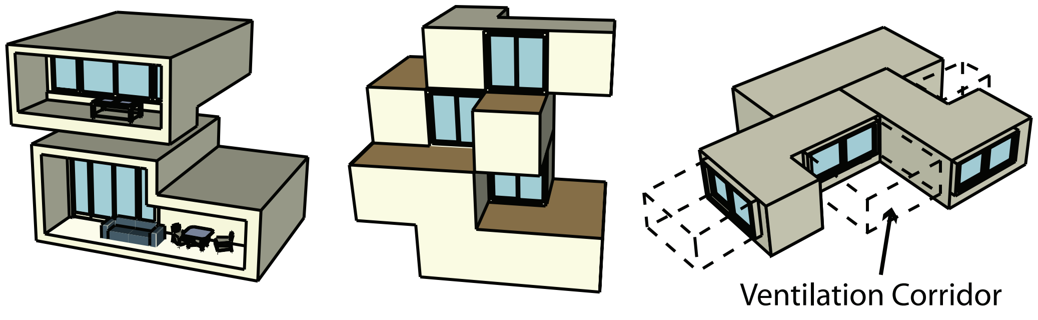


Figure 3.1: Left: Variation in height of unit. Center: Using other a unit's roof as another's outdoor space. Right: Multiple views available in units.

Shape Variation With cubic meter instead of square meter to quantify a unit, an architect describes more about the spatial possibilities of the unit. Variation in room sizes also allow for families to change its accommodation as family size increases or income increases but still stay within the vicinity. Similarly, variation in room heights can accommodate different needs of the residents (See left of Figure 3.1). We also introduce the concept of form complexity (chapter 7) on the floorplan level for characterizing different designs.

Multiple Views In any site context, the layout and orientation of a house relates key interior spaces to preferred views. This is perhaps more imperative a consideration than its response to sun paths and wind directions on its site. But standardized housing blocks have its views fixed by one or both of its two window walls. Nevertheless, units are now planned for multiple views as a reaction to the problem of obstructed view. Multiple views are sought in form to enhance spaciousness while also achieve good ventilation in the living environment. See the right picture of Figure 3.1.

Suspended Open Spaces In conventional apartments of multi-storey architectural form, units at high levels have little or no good relationship to ground floor amenities. A garden in the mid-air utilizing communal terrace or widened access deck is thus a valuable asset to units nearby. Small play areas within sight of most units are also very desirable for family type units. To achieve this, the concept of using one unit's roof as another unit's garden in the air is a useful concept. See middle of Figure 3.1.

Ventilation Good ventilation is crucial in multi-storey architectural form to provide good indoor air quality and thus healthy living environment to its residents. Conventional point block design compromises indoor air quality and eventually proved detrimental to the occupant's health. Non-regular architectural forms with suspended open spaces give rise to a new means of ventilation not normally available in conventional design, hence improving air quality. See right of Figure 3.1.

3.3 Specific Considerations

In this work, we focus primarily on using our system to address the issue of generating forms with the aim of optimizing attribute values. Specifically, we address the issues of generating units with the following requirements:

- **Floor Area** attribute. User specified requirement in m^2
- **Multiple View** attribute. Calculated as the total amount of unobstructed view out of the unit.
- **Shape Variation**. Create units with variation of single and double volume as well as units with non-rectilinear floorplans.

Our growth approach to generating design focuses primarily on the criteria of multiple views and variation of shape. Specifically, we adapt two simple shape operators (Chapter 5.1) to advance faces to grow view and to split faces to generate variation of shape. These operators also work in conjunction with our measure of form complexity defined for a group of units. On the other hand, the desires of suspended open spaces and good ventilation interestingly appear as emergent spatial features in our architectural forms. These are consequences of the use of non-regular units in our forms and are of value to architects attempting complex configurations which cannot be pre-visualized in their minds.

Chapter 4

The Growth-Based System

Our growth-based system works by utilizing a growth mechanism in a time based simulation framework. The initial setup for the simulation comprises cores and circulation with the placement of seed units at exit locations along the circulation. *Seed units* are units of small initial volumes and with possibly configurable behaviors. All these input can be interactively specified or automatically generated by the system within user controls. The major components of the proposed growth-based system are as shown in Figure 4.1.

4.1 System Components

To present an intuitive understanding of the working of our entire system, we describe the major components of our system and subsequently outline the interplay of each component in one single execution of a simulation cycle.

Growth Simulation Engine This is the brain of the whole system. It handles the running of the entire simulation framework and coordinates the interactions of the various components in the system during the simulation.

Arbitration Rule Handler Means of arbitration is important for ensuring that the

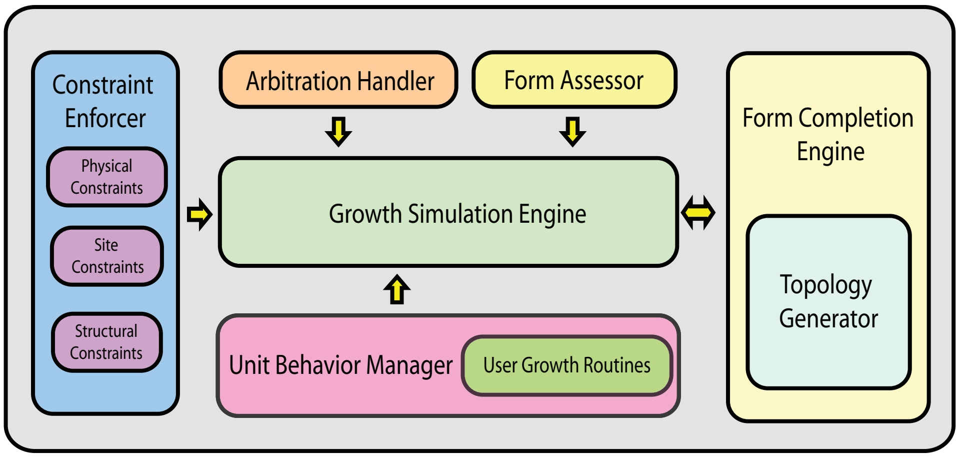


Figure 4.1: The growth-based system.

order of invocation of the growth routines is fair and random, since even fixed order invocation can present starvation of growth. Our framework provides for the incorporation of user-defined arbitration routines which can for example favor units with small current floor area, whose growth routine invocation will be more frequent than the rest. Other possibilities include favoring units of different quality classes or giving priority to units with more imminent need to improve their view quantifier. In our particular implementation we prioritize according to view since it indirectly influences the floor area.

Form Assessment Component There are numerous architectural criteria for assessing the quality of individual units and our framework supports handling of different criteria by registering callback functions which evaluate individual unit quality based any user desired computation. This component calculates the attribute values of each unit and assesses the implication of these as compared to their goal values. For example, the ratio of the current view to the goal view can influence the priority of the unit in its subsequent growth. Our current implementation calculates mainly the view values for all units by exploiting the capability of modern GPU (see Chapter 5.2).

Constraint Enforcer Constraints are needed to ensure the generated design is near production form. These constraints are, for example, physical where design has to be geometrically plausible, site related where design has to be constrained to grow within the site envelope, and structural where units has to have certain amount of clearance from the cores. Only valid moves of growth checked by the constraint enforcer will be committed.

Unit Behavior Manager This Manager realizes the grow behavior specified for each unit and among different units. It manages the registration of user defined callback functions that comprises routines which determine the "behavior" of units. These set of callback functions can then be assigned to different units in the model thereby changing their growth behavior during the simulation. The advantage of of such a callback framework is its flexibility to allow some user experimentation in controlling the assignment of different behavior functions to groups of different units in space. The user can then experiment with a variety of different ways to grow units. Typically, a unit is to firstly grow to a certain volume and then considers varying the shape into multiple storeys. In more advanced cases, two or more units can grow into an interlocking pattern with good attribute values.

Form Completion Engine The Form Completion Engine enables rapid synthesis of floorplan (adhering to user specified form complexity) within a small region of a form. It can serve to fill up voids left from the growth process, or can be used to generate intriguing interlocking pattern. The synthesized small form can possibly be a module to be replicated in other regions of the form.

4.1.1 The Simulation Process

The initial setup for the entire simulation process comprises the specification of cores and circulation (staircases) with the placement of seed units at exit locations along the

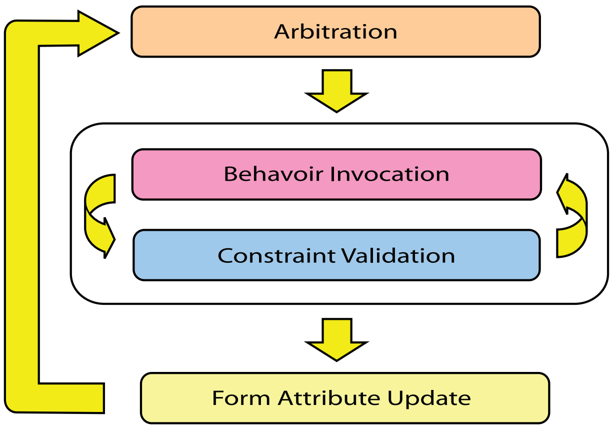


Figure 4.2: Simulation Process Flow

circulation. For an example of such a setup see left most diagram in Figure 1.1.

The growth process then commences with repeated execution of the simulation loop. The process flow diagram of a single simulation run is shown in Figure 4.2, and it comprises of the following phases:

1. **Arbitration Process** - In the beginning of every simulation time step, the system first performs an arbitration process that determines an ordering of the units based on their form quantifiers. On completion, the growth engine is notified on the sequence of execution of unit callback functions for the next simulation run.

2. **Behavior Invocation** - The growth engine invokes the callback functions of using the call sequence produced by the previous step. Each callback function then generates a candidate move list which will be passed to the constraint enforcer for validation. Details on the behavior routines will be given in Chapter 5.3.

3. **Constraint Violation Check** - With each move list generated by the unit callback function, the constraint checker verifies move validity by checking against the defined constraint set. Moves that violate constraint rules are discarded and the remaining list of valid moves is returned to the unit callback routine.

4. **Form Quantifier Update** - From the valid move list each unit determines the

quality of each move based on their own user-specified assessment criteria. Each unit then selects the best move which the grow engine will then commit. At the end of all units' move commitment, the growth engine updates the form quantifiers (floor area, view) of each unit and the simulation run then repeats.

The simulation runs will evolve the units over time and units that have satisfied their own termination criteria will then be removed from the callback list. The simulation stops when all units have reached termination condition.

Chapter 5

Growth Strategies

This chapter presents in detail our implemented strategies to grow units. In particular, Chapter 5.1 discusses two primitive operators in our growing process, and Sections 5.2 and Chapter 5.3 present our strategies to gain view and shape variation.

5.1 Shape Operators

The geometric representation of our architectural form is a 3D mesh in the CGAL library [CGAL, 2005]. We design the following two operators to manipulate the 3D mesh as needed during the simulation loop.

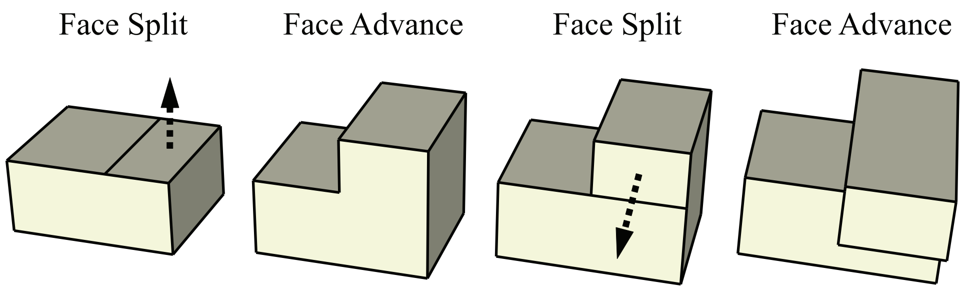


Figure 5.1: Form generation via a series of applications of the two primitive shape operators.

Face Advance. For a face F defined by a chain of vertices with face normal $\vec{n}$, the operator $Advance(F, \alpha)$ moves the vertices of F in the direction of $\vec{n}$ by a magnitude of

α. The value of α is a user parameter that controls how much each face is to be moved. A conservative value of between 0.2m to 0.5m is used in our implementation.

Face Split. For a face F defined by a chain of vertices $v_1, v_2, \ldots, v_i$, the operator *Split*(F) introduces two new vertices v' and v'' on edges $v_j v_{j+1}$ and $v_k v_{k+1}$ respectively, to result in two new smaller faces with the chain of vertices $v_1, v_2, \ldots, v_j, v', v'', v_{k+1}, \ldots, v_i$ and $v', v_{j+1}, \ldots, v_k, v''$.

Interestingly, a combination of these two operators are able to model virtually all commonly encountered shapes for our purposes; see Figure 5.1 for an illustration on their uses. In particular, face advance is used in growing view, while face split in shape variation. The same operations (for 2D) are used in the generation of forms meeting a desirable form complexity in Chapter 7.

5.2 View Guided Growth

For a vertical face of a unit, its *view* is defined as the total area of the portions of the face that are visible when viewed, along the normal direction of the face under orthographic projection, from outside the site envelope. The *view of a unit* is the sum of all the views of its vertical faces. In our current implementation, each unit is bounded by axis-aligned vertical faces (orthogonal to x-axis or z-axis). Thus, we only measure views of units along 4 directions, i.e. the positive and negative x-axis and z-axis.

View Map Computation For efficient view computation, we use an image-based approach to simultaneously compute views for all units and store them as a *view map*. To do so, we project all units (i.e. of the current version of form) orthogonally along each of the mentioned four axes. During rendering, we activate a vertex program to store alongside with each pixel the identity of the unit occupying the pixel. If there are no more than 256 units, a single byte is enough to store the identity. In such a case, the four passes of rendering can be packed into the four RGBA channels of a texture, i.e. as the view map. From the view map, the total view of a unit is obtained by summing up all the pixels in the four channels storing the identity of the unit.

Gain in View For an operation of $Advance(F, \alpha)$ on face F of a unit, each face of the unit sharing an edge $v_i v_j$ with F is enlarged by an area of $\alpha|v_i v_j|$. Each increase in area for a vertical face is potentially an increase in view of the unit owning the face. To facilitate the calculation of gain in view, we store a depth map of the scene along each of the four projection directions while generating the view map. Each pixel, due to one projection of the enlarged part of a face, is a gain in view if its depth is smaller than that in the corresponding pixel found in the depth map under the same projection. The gain in view of $Advance(F, \alpha)$ is then defined as the sum of gain in views of all faces incident to F. In our implementation, we regard the move with largest gain in view as the best.

View Conflict An operation of $Advance(F, \alpha)$ on face F can increase the view of its unit, but can also decrease the view of other units. Our experience shows that it would be too restrictive in the growth process if we do not allow one unit to intrude the views of the other. Therefore, we introduce a threshold τ as a user parameter to control the amount of views a unit allows the other to intrude into. Consequently, $Advance(F, \alpha)$ on face F of a unit can be a valid move only if the enlarged area of each face incident to F does not introduce a violation of τ of all the other units. Also, we can penalize a move of a unit that intrudes views of other units.

5.3 Shape Considerations

With selection of the growing face based entirely on view, we have experienced biased growth of units in one particular projection direction, leading to elongated ones which are not usable space in practice. This problem is resolved when we consider shape variation during the growth of a unit.

Shape Variation in General At the beginning of the simulation, each seed unit is set to either automatically or manually a target minimum and/or maximum volume (or floor area) among others. To achieve shape variation, one possible way is to build randomness into the various stages of the growth as follows. We may start by dividing the needed target volume into a few, say 2 to 3, usable sub-units (of having at least the standard of $3m \times 3m$) to grow, one after the other. While growing one sub-unit till at least a usable volume, we may again split a face at random locations to grow the next sub-unit. With randomness, we can sometimes favor a move that maintains a good aspect ratio of the unit to one that improves the largest view.

Choice of Splitting Face Extending for a unit u from one of its sub-unit u_i to include another of its sub-unit u_j, we choose an arbitrary face of u_i to split and subsequently advance (either one of the two new faces) to create u_j. Generally there are many faces available for splitting. To maximize on the chances of growing u_j, we select the face F of u that has the largest clearance from all existing faces. In other words, F is such that, among all faces of u, its closest distance (measured parallel to the x-z plane) to all the other faces (inclusive of the faces of the site envelope) is the largest.

Choice of Locked Faces To better control the generation of some specific class of units, we incorporate a mechanism to disable the splitting and advancing of one

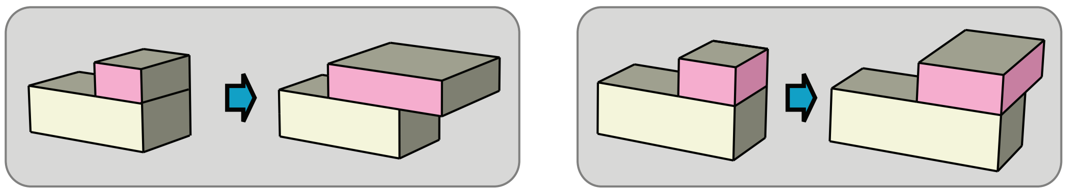


Figure 5.2: Locking of different faces to result in the variation of shape.

or more faces of a unit. In other words, a face that is locked will maintain its shape throughout the rest of the simulation or until it is unlocked, while its area may still grow due to the advances of faces incident to it. See Figure 5.2.

Choice of Unit Height Units of non-standard heights further enhance the variation in the architectural form. To support this, we allow splitting and advancing of floor or ceiling faces of a unit into a volume with height of up to twice (i.e., *double volume*) the standard height. We use a similar clearance consideration as in the last paragraph (but now measured parallel to the y axis) to pick a face to split and then advance. Note that height larger than double volume is also possible to implement.

Collide-and-Extrude As units are allowed to grow autonomous to each other, collision can occur between units and between units and cores. A straightforward way to handle collision is to lock the faces involved to prevent overlapping volume. Alternatively, we introduce the collide-and-extrude feature to generate more interesting spatial arrangements. On this, we exploit the Boolean operations (polygon difference) from the CGAL library to create extrusions that are novel in the usage of space. For example, Figure 5.3 shows a way to generate intriguing interlocking more intuitively and rapidly, either by the simulation steps or with a few mouse motions. Also, Figure 5.4 shows an example use of the top faces of an external staircase as the base faces of an internal staircase of a unit.

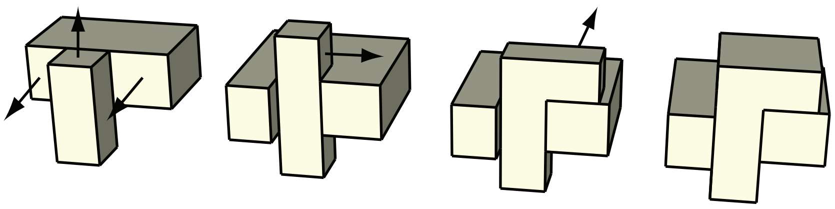


Figure 5.3: Collide-and-extrude feature to achieve intriguing interlocking more intuitively and rapidly.

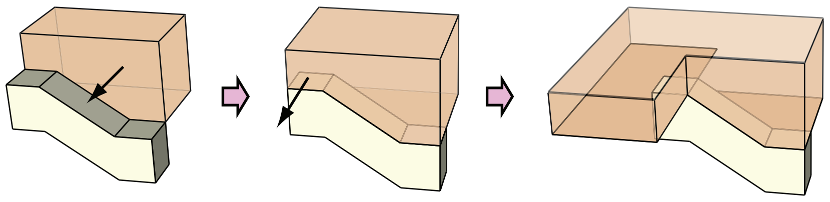


Figure 5.4: Collide-and-extrude feature used to create a unique double storey unit that utilizes an existing staircase's roof as its stair base to the second level.

Look-ahead The left of Figure 5.4 illustrates a scenario where a unit has to decide whether to occupy either whole of the top of the staircase or none at all. A partial occupancy of the staircase is not useful to the unit. In some cases, a step-by-step growth may result in a space too small to be of any practical uses. We implement the support of a query by a unit to the Growth Simulation Engine of whether there is a clearance for advancing a face till the end of a certain number of simulation loops. This is computed as a conservative estimate derived from potential growths of other units in the proximity. With this, as in our example, the unit can better decide whether to venture into occupying the whole of the top of the staircase or to lock the colliding face.

Chapter 6

The ShapeTree Data Structure

In this chapter, we digress for a brief introduction of our novel representation that forms the core data structure crucial for supporting the implementation of the form completion engine described in the next Chapter.

6.1 A Tree-based Representation

The ***ShapeTree*** is a tree-based representation that provides an implicit encoding of the shape of a given rectilinear unit in 2D. Our new representation features some interesting properties such as component decomposition and history (useful for user interaction) which will be presented later in the Chapter.

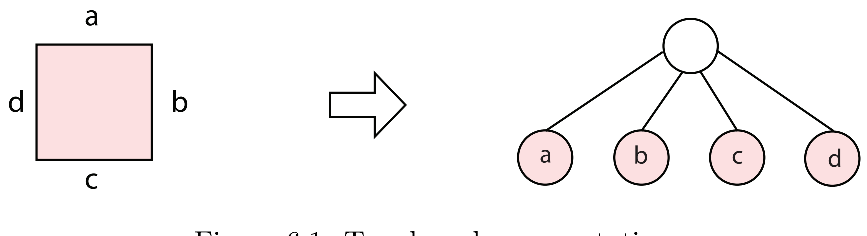


Figure 6.1: Tree based representation.

For any shape, its representation is a tree with a set of nodes $V = (v_1, v_2, \ldots, v_i)$ and a set of corresponding weights $W = (w(v_1), w(v_2), \ldots, w(v_i))$ to keep for example, the lengths of edges. The leaf nodes of the tree at any point in time correspond to edges of the current 2D geometric shape, while internal nodes implicitly record a history of unit shape variation, as well as providing an immediate decomposition into constituent boxes. Subtrees also represent a decomposition into sub-shapes. The face advance and face split operations in Chapter 5.1 translate straightforwardly to the 2D case to work on the shapetree (See Figure 6.6).

Figure 6.1 shows an example of a unit square with its corresponding tree. For any node n of the ShapeTree, there is an associated weight value *w(n)* that represents the length of the edge represented by that node. Each node is labeled with a, b, c or d which corresponds to the directions north, east, south, west. For example, a_{ij} is used to denote some *j-th* north facing edge of Shapetree *i*. Figure 6.2 shows a more complicated example of how the shapetree changes with the evolution of unit shape from (i) to (iv).

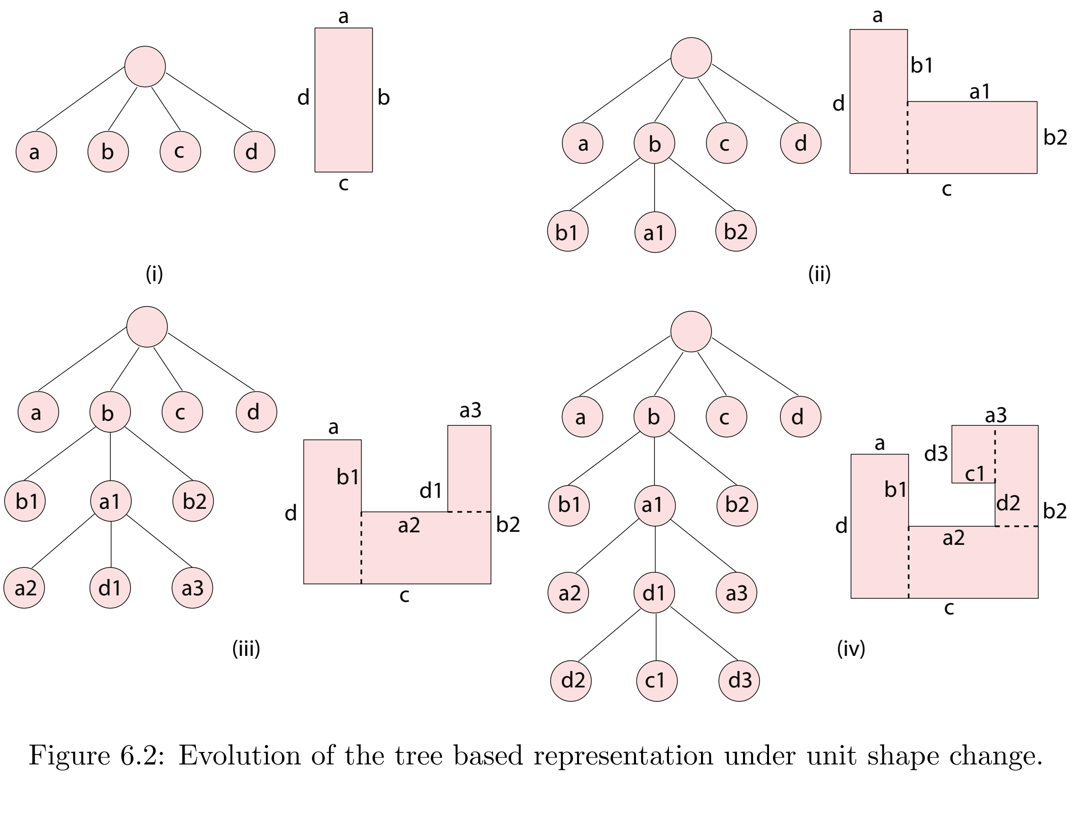


Figure 6.2: Evolution of the tree based representation under unit shape change.

A Shapetree is used to represent the 2D footprint of the unit and the actual 3D polyhedron is obtained by extruding the 2D footprint of the unit by a height displacement h. Figure 6.3 shows a unit with an L-shaped footprint extruded vertically.

6.2 Face Advancement

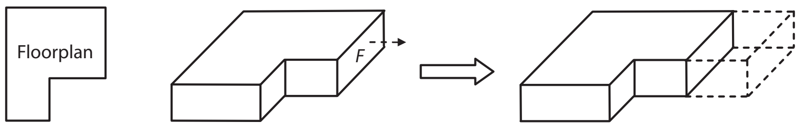


Figure 6.3: Illustration of floorplan and the Face Advance operation.

Each unit floorplan is represented by nodes in the Shapetree, which correspond to edges of the 2D geometric shape. Specifically, each edge E_i is defined by a pair of vertices (u_0, u_1) and a face F_i of the unit is defined by a chain of vertices, $F_i = \{u_0, u_1, u_0 + \bar{h}, u_1 + \bar{h}\}$ where $\bar{h} = (0, h, 0)^T$. The face advancement operator $Adv(F, \alpha)$ perturbs the vertices of F_i in the direction of its face normal N_i by a magnitude of α. This face advancement operation corresponds an increase of the node weights in the implicit tree representation. Let v be the node in the tree corresponding to the face that is being advanced. Let v_n and v_p represent the next and previous nodes corresponding to the next and previous edges of the moved face in the floor plan. Then the face advancement operation is conceptually as follows,

$$Adv(F, \alpha) \quad => \quad v_i = v_i + \alpha N_i$$

and this maps to the tree operations,

$$w(v_n) = w(v_n) + \alpha \qquad \text{and} \qquad w(v_p) = w(v_p) + \alpha$$

6.3 Face Split

Conceptually, the face split operation in 3D introduces two new vertices and creates two new faces F_1 and F_2 from an existing face F. Let $F = \{v_1, \cdots, v_n\}$ be some face of the polyhedron, the face split operation $Facesplit(F)$ then introduce two new vertices

v' and v'' with the following restrictions,

$$v' = \alpha v_i + (1-\alpha)v_{i+1}, \qquad \text{such that } e_1(v_i, v_{i+1}) \text{ is an edge .}$$

$$v'' = \beta v_j + (1-\beta)v_{j+1}, \qquad \text{such that } e_2(v_j, v_{j+1}) \text{ is an edge .}$$

$$\angle(v_i, v', v'') = \frac{\pi}{2} \quad \text{and } \angle(v', v'', v_{j+1}) = \frac{\pi}{2} \text{ and } e_1||e_2$$

The above formulation enforce that the face split is done along a cut that is orthogonal to the edges of the face. Figure 6.4 shows how an L-shaped unit can be created via a face split operation followed by a face advance operation. The reverse application of these operations will facilitate history and undo GUI features required of most interactive design tools. The geometric behavior of units will be defined in terms of a combination of primitive operations coupled with simple decision procedures.

6.4 Primitive operations and Corresponding Tree Manipulation

The 2D analogue of primitive shape operators maps elegantly to tree operations as described below. For the *FaceSplit* operation, child nodes are introduced into the node of the tree representing the current face being split. Two types of nodes are created, normal tree nodes representing the newly created faces and virtual nodes that will be

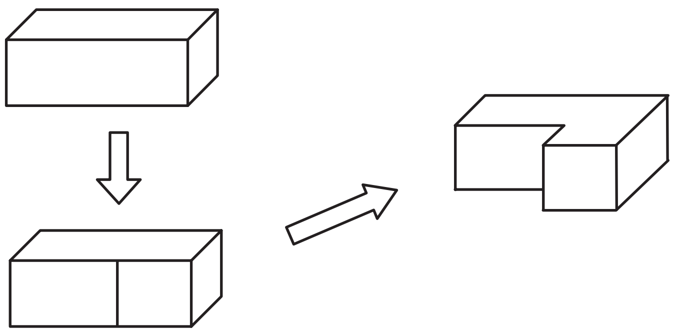


Figure 6.4: Illustration of the Face Split operation.

used to capture shape changes induced by subsequent *FaceAdvance* operations. An illlustration of primitive operations and their corresponding modifications to the tree structure is illustrated in the following figures.

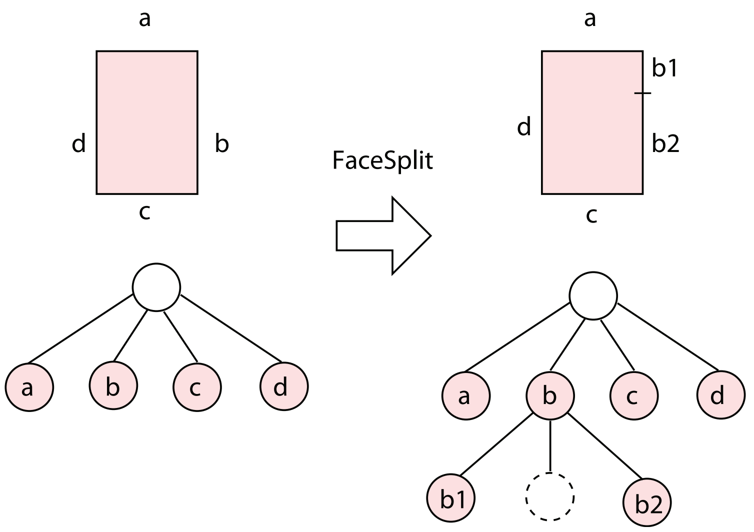


Figure 6.5: Modifications to the tree structure due to *FaceSplit*.

As illustrated in Figure 6.5, the *FaceSplit* operation creates two new child nodes *b1* and *b2* as well as a virtual node (shown in dotted lines) that is to be used later to capture shape modifications due to *FaceAdvance* operations. Figure 6.6 shows how the *FaceAdvance* operation modifies the virtual node previously created into an actual child node of the parent *b*.

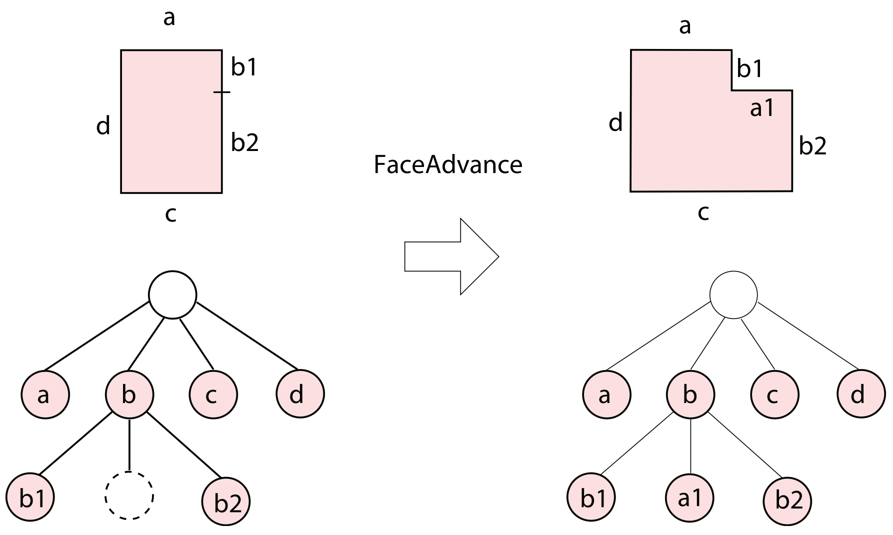


Figure 6.6: Modifications to the tree structure due to *FaceAdvance*.

A particular shape is represented by a set of *splits* on the node set V. To constrain the shape to a particular class, a set of constraints is placed on the weights of the nodes as well as the number of splits for each subtree of the ShapeTree. To control the number of face splits, we can use the following constraint

$$\sum_i SplitCount(v_i) \leq SplitCount(v_x) \tag{6.1}$$

where *SplitCount(v_x)* denotes the number of split operations allowed at the subtree rooted at x and v_i denotes the i-th child node of v_x.

6.5 Properties of the tree-based representation

Representing the unit shapes using a tree provides a number of useful properties that both facilitate shape understanding as well as enable easy implementation of certain user interaction commands such as history and undo. In the following, we highlight some of the properties of the tree structure as well as its uses.

History and Undo The tree based representation extends naturally to capturing user interaction history since modifications to the tree encodes exactly the steps taken by the user to achieve the final state of the shape that is currently represented. Any intermediate state of the tree represents an intermediate shape that is encountered during user manipulation or software guided modification. Although such a state might represented a group of out-of-order undo operations, the tree-based structure still facilitates history based undo with just minimum modifications and book-keeping.

Shape Decomposition into Components Using the tree-based representation, any subtree of the Shape-Tree represents a sub-component of the entire shape. Figure 6.7 shows an illustration of a shape and its component. We note though that

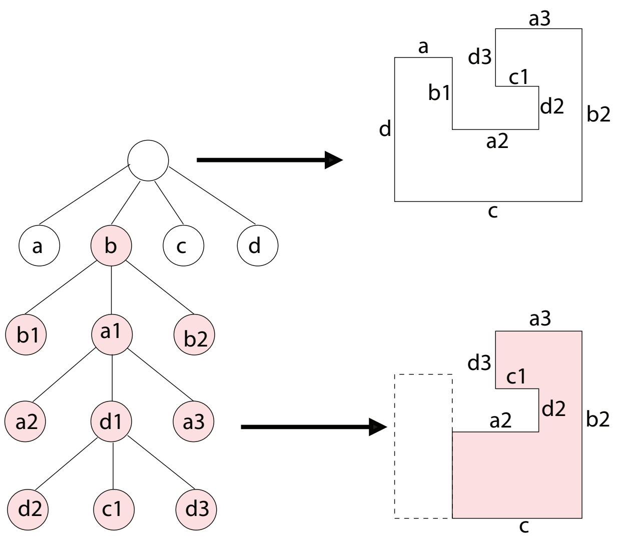


Figure 6.7: Shape-tree and components decomposition.

the subtree in itself does not sufficiently describe it shape entirely. To complete its shape description, it requires node *c*, which is the next leaf node from the left to right traversal of the tree. The node *c*, together with node *b1*, completes the full description of the component shape.

The ability to decompose a shape into components and the possibility of representing it as a subtree of it Shape-tree opens up the possibility of shape modification and mutation. Synthesis of new shapes by crossing mutating their shape tree is then possible and such mutation may possibly be guided by some quality metric. Generation of 'interesting' shapes are then possible if some heuristic can be conceived to control the cross mutation of shape trees.

Shape Decomposition into Boxes A decomposition of the shapetree into constituent boxes is immediately obtained from the shapetree data structure without the need for any additional computation. This is simply achieved by keeping track of the face-split operations performed. Such a decomposition is potentially useful in the architecture context in which the boxes can conceptually be a partition of the unit into rooms. The ability to obtain such a decomposition is also crucial to our form completion approach described in Chapter 7. See Figure 6.8 for an illustration of the resultant rectangular decomposition for a given floorplan

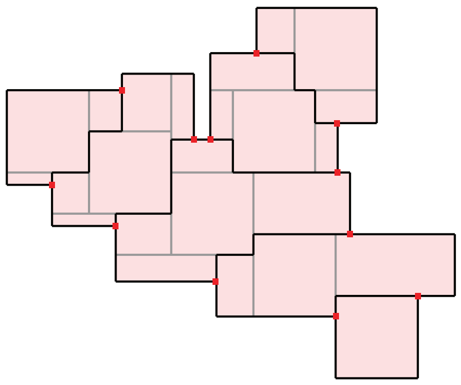


Figure 6.8: A decomposition of each shapetree in the floorplan into rectangular subunits.

Chapter 7

The Form Completion Engine

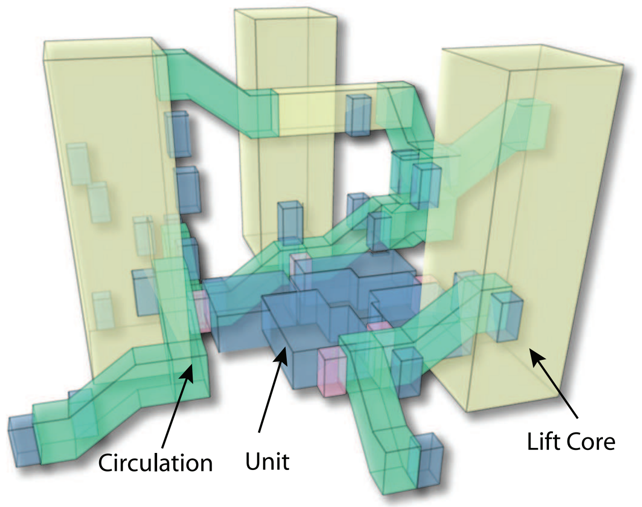


Figure 7.1: An illustration of five completed units with entrances colored in green.

The form completion engine provides an interactive tool that aids the user in synthesizing various completed forms corresponding to a selected number of partially grown units or seed units. It can be used as a stand-alone tool for rapid design synthesis, or can also serve to complement the growth process to better claim gaps (voids) in space for units; see Figure 7.1 for a simple illustration. Before presenting the working of the completion procedure in Chapter 7.3, we first discuss our design representation in Chapter 7.1 to support the completion procedure, and a form complexity measure in Chapter 7.2 to categorize designs.

7.1 Design Representation

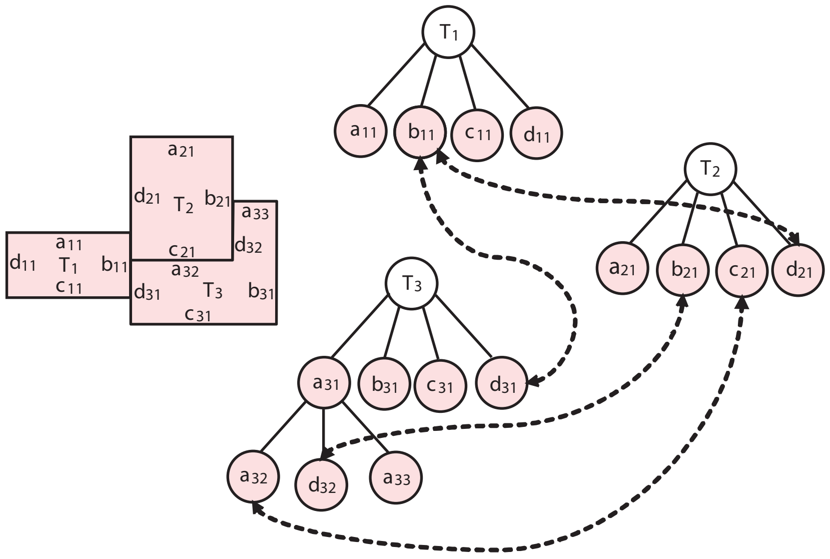


Figure 7.2: A configuration of shapes and their tree based representation.

For the purposes of intuitive manipulation and ease of conceptually visualizing designs mentally by the user, we adopt a representation based on 2D floorplans, which will eventually be synthesized as 3D volumes. This is entirely adequate as a form design work spans at most 2 to 3 levels at a time.

To represent a particular configuration of floorplan with n units, we use a set of n shapetrees $S = \{T_1, T_2, \cdots, T_n\}$ with additional graph edges linking some node of T_i to another node of T_j for any incidence relation between the two faces in the floorplan. The general setup is a set of trees connected at their leave nodes via graph edges, see Figure 7.2.

As the units in the floorplan evolve, the data structure for the floorplan is updated by introducing new links joining the appropriate new nodes created to their incident edges. These operations are efficient and also support backtracking during design search. In Figure 7.3, we show how the data structure is updated when a face of the unit T_2 is being split and subsequently advanced. When the node d_{21} in Figure 7.2 is split, three new nodes d_{23}, c_{22} and d_{22} are created. The graph edge (shown in gray in Figure 7.3) linking nodes b_{11} and d_{21} is removed and new graph edges are created linking the pairs (b_{11}, d_{23}) and (a_{11}, c_{22}). The weights of c_{22} and a_{21} are also updated following the face advance operation on node d_{22}.

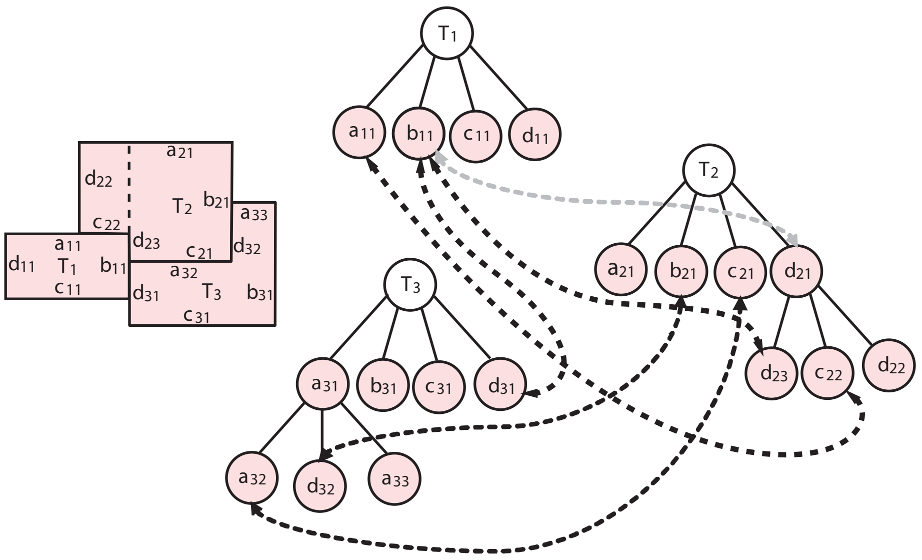


Figure 7.3: The evolution of the floorplan and the creation of new nodes and new links in the corresponding data structure.

We have considered other planar graph representations for our floorplan designs, but found this simple representation by a set of shapetrees sufficient (and efficient) in manipulating topological designs in our process. Our representation is not merely *ad hoc* but carefully designed to accommodate the algorithmic operations crucial to our form completion approach.

7.2 A Measure of Form Complexity

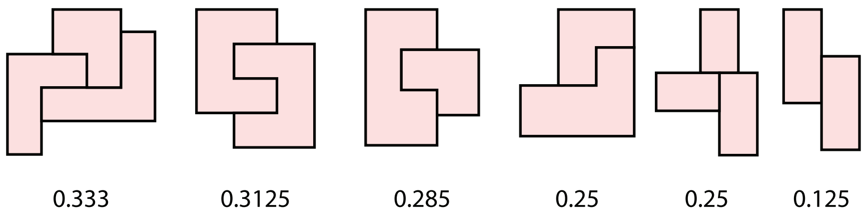
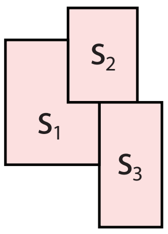


Figure 7.4: Various set of shapes and their form complexity measure.

For purposes of generating a spectrum of shape configurations from simple to complex for a variety of applications, we introduce a form complexity measure that characterizes intriguing interlockings of two or more units. Our measure captures the proportion of the total number of faces that are interlock (incident) to other units. Specifically, for a set of shapes $S = \{s_1, s_2, \ldots s_k\}$, with $|s_i|$ denoting the number of faces of s_i, the complexity measure F_q of S is determined as

$$F_q(S) = \frac{1}{\sum |s_i|} \sum_{i=1}^{k} \sum_{j=i+1}^{k} C_{ij}$$

where C_{ij} is the total number of incident faces of s_i and s_j. We have $F_q(S)$ being trivially 0 for a set of shapes that do not interlock.



$$\begin{aligned} F_q(S) &= \frac{1}{14}[C_{12} + C_{13} + C_{23}] \\ &= \frac{1}{14}[2 + 1 + 1] = 0.285 \end{aligned}$$

Figure 7.5: An illustrated example of the computation of quality measure for an arrangement of three interlocking units.

Figure 7.5 shows the computation of the complexity measure for an arrangement of three interlocking units. In Figure 7.4 we show further examples of various sets of shapes and their corresponding form complexity measure. Its is apparent from the examples in Figure 7.4 that our complexity measure provides an intuitive sense of the complexity of interlocking in a set of units in an arrangement. This formalism is inspired by the work of [Gero and Kazakov, 2004] on qualitative symbolic modeling. Configurations of shapes are distinguished as a class rather than as instances with quantitative descriptions. This is to say that actual dimensions of the shape are not taken into account during classification via our complexity measure.

7.2.1 Monotonicity of Quality Measure

In this section, we examine an interesting property of the quality measure under shape enumeration using the set of moves described in the previous section. We show that our measure will always increase monotonically when the shape configurations are subject to the previously introduced moves. We consider the behavior of the quality measure under the application of the two operators of face split and face advance applied only to the following two complexity split arrangements in a set of shapetrees in a design configuration. See Figure7.6.

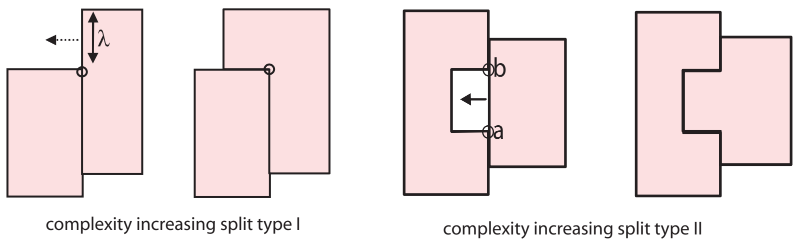

λ
b
a
complexity increasing split type I
complexity increasing split type II

Figure 7.6: Various set of shapes and their form complexity measure.

Consider the initial condition of two cubes with touching each other on one of their faces, the complexity measure $F_q(S)$ is 1/8. With each application of the complexity increasing move (Type I) described above, the quality measure forms a series:

$$R = \{\frac{1}{8}, \frac{2}{10}, \frac{3}{12}, \cdots\}.$$

In general, the $(k+1)th$ term is related to the kth term by

$$r_{k+1} = \frac{x_k + 1}{y_k + 2}, \text{ with } r_k = \frac{x_k}{y_k}, x_k = x_{k-1} + 1, y_k = y_{k-1} + 2.$$

The nominator of the fraction is the summation term in the formula for F_q while the denominator is the N_s term. The kth term can be written as

$$r_k = \frac{x_0 + \sum_1^k 1}{y_0 + \sum_1^k 2} = \frac{x_0 + k}{y_0 + 2k}$$

Consider

$$r_{k+1} - r_k = \frac{x_0 + k + 1}{y_0 + 2k + 2} - \frac{x_0 + k}{y_0 + 2k}$$
$$= \frac{y_0 - 2x_0}{(y_0 + 2k + 2)(y_0 + 2k)}$$

From the above we can see that the moves will improve the complexity so long as $y_0 - 2x_0$ is positive. Going back to the case of two cubes with one of their faces incident to each other, we have the initial condition of $y_0 = 8$ and $x_0 = 1$. Hence the moves we introduced previously will always improve the complexity measure. For the case of three cubes in the initial configuration, the worse case values are $y_0 = 12$ and $x_0 = 3$, which still gives a positive value for $y_0 - 2x_0$. (*Note : In general, the above is not true for any configuration of N cubes, see counter example in Figure 7.7).

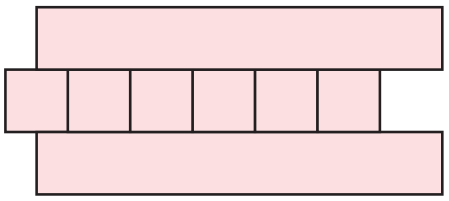


Figure 7.7: A counter example for the initial quality measure that is greater than 1/2.

Using a similar analysis as described previously, it is found that for complexity increasing move (Type II), the initial conditions are $2y_0 - 4x_0 > 0$ and $3y_0 - 4x_0 > 0$ for both cases of Figure 7.6 respectively. Thus, combining all these restrictions, we end up with the criteria that initial configurations must satisfy $x_0/y_0 < 1/2$.

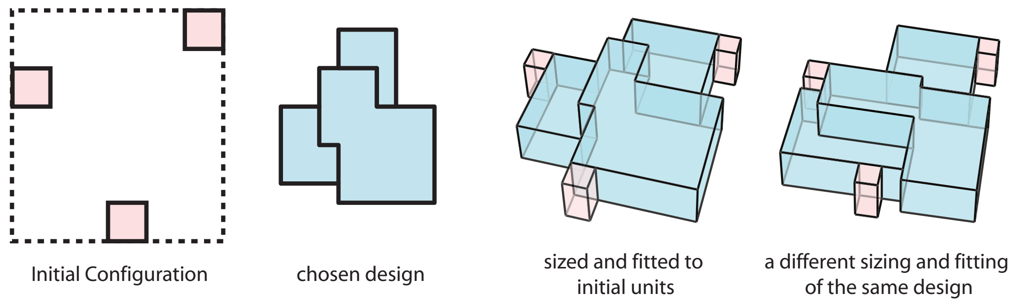

Initial Configuration
chosen design
sized and fitted to initial units
a different sizing and fitting of the same design

Figure 7.8: An illustration of a chosen design and its different completed forms. Pink squares represent entry points to units.

7.3 The Completion Procedure

For simplicity of discussion and without loss of generality, we model the problem setup for partially completed units or initial seed units alike and assume there are N canonical units which we accept as input to the form completion engine (Refer to Figure 7.8). These can be thought of as entrances to the units which has to belong to some particular unit in the floorplan after the completion process.

There are four steps to complete a form. First, user selects a required form complexity to generate forms (Chapter 7.3.1). Second, a processing step converts the selected design into a rectangular cartogram (Chapter 7.3.2). Third, a form is selected to link up with the initial seeds (Chapter 7.3.3). Finally, the form is sized into useful units (Chapter 7.3.4). The resulting form of these steps can further be modified (such as adding some balcony features) by users depending on their needs.

7.3.1 Topology Generation

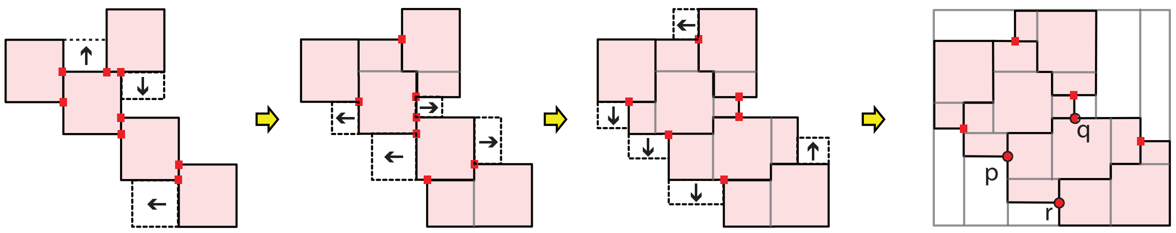


Figure 7.9: The starting configuration of 5 canonical units, and the steps in generating form with complexity in the range of 0.36 to 0.38. Red points represent possible split points to increase form complexity.

With n input seeds and an input form complexity, the system (by default) starts with n canonical units to start enumerating forms of the required form complexity; see Figure 7.9. To do this, the system first identifies split positions (shown as red points in Figure 7.9) that can increase form complexity, then select one or more among them to perform the actual split and face advance operations. The selection takes into account considerations such as equal distribution of unit shape complexity. With different initial arrangement of canonical units or different choices in the split and face advance operations, the system generates a pre-defined number of forms for selection to proceed onto the next step.

Spanning of Configuration Space. With an initial setup of trees representing an initial configuration, new configurations can be obtained via the application of the two primitive operations of *face advance* and *face split*. It can be shown that using the previous two operations, all possible configurations of units are derivable.

To prove the spanning of configuration space, we first make the following important observations,

- Each unit is a rectilinear shape representable by a shapetree.
- There exist a shapetree representation for all possible rectilinear unit shapes.
- For every shape tree representation above, there exist a sequence (not neccessarily unique) of face split and face advance operations leading from an initial canonical unit to the final unit shape.

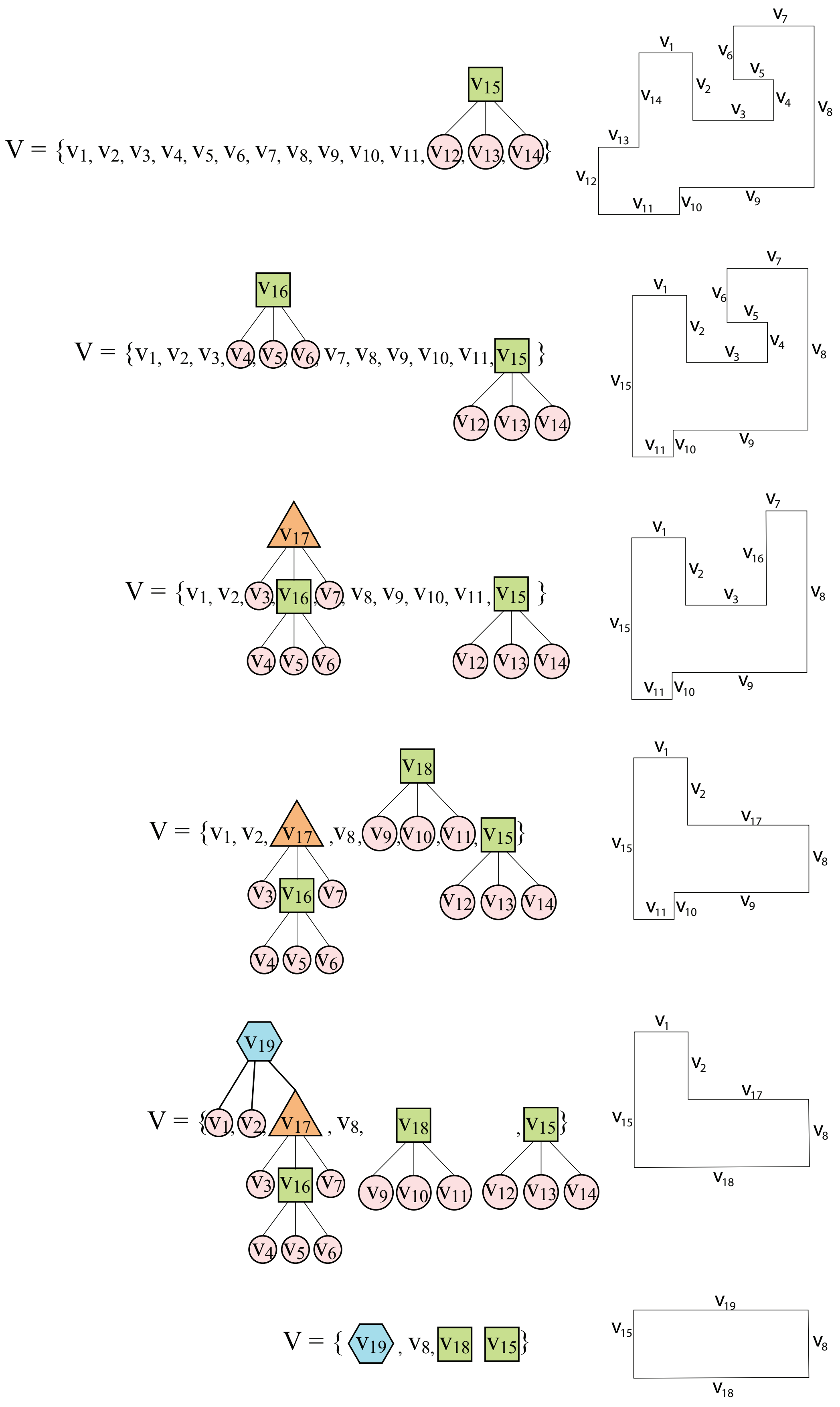
It suffices to prove that for each unit in the final floorplan, there exists some sequence of operations leading to the final unit shape. We will show that using the 'collapsing steps' operation described in Figure 7.14, it is possible to obtain a constructive proof of the above claim. The essence of the proof is similar to that of ear cutting for polygon triangulation.

For any existing unit in the floorplan, we begin by creating a set of tree nodes $V = \{v_1, \cdots, v_n\}$, with each v_i corresponding to some edge e_i in the geometric shape of the current unit. The procedure of the constructive proof then proceeds as follows,

- While $\exists$ nodes (v_i, v_j, v_k) such that they form a 'staircase' arrangement.

    1. 'Collapse' the staircase arrangement, merging (v_i, v_j, v_k) into a new node v'
    2. Insert v' into the set V

Clearly the above algorithm terminates and at the end of the above procedure, we are left with the four top level nodes representing the initial canonical unit. The crucial thing to observe is that the stair collapsing operation and merging procedure is exactly the reverse operation of the spliting and advancing a face. Hence, we have shown by a constructive proof that there exists a sequence of face split and advance operations that allows the evolution of any initial shape to a final rectilinear shape. This is by virtue of the fact that we have shown that a procedure exists for the collapsing any existing shape systematically to the initial square, and that the collapsing and face splits are reverse operations of each other. Figure 7.10 illustrates an examples of the constructive proof procedure on a given unit, which iteratively performs merging of nodes by randomly selecting stair arrangements to collapse. It should be noted that the order of collapsing is unimportant and the procedure produces a canonical unit no matter the choice of merging. Multiple ways of evolving a unit exists and each permutation of the order of merging represents a different valid sequence producing the same shape.



V = {v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14}
V15
V = {v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, V15 }
V16
V12 V13 V14
V = {v1, v2, V3, V16, V7, v8, v9, v10, v11, V15 }
V17
V4 V5 V6
V = {v1, v2, V17, v8, V9, V10, V11, V15}
V18
V = {V1, V2, V17, v8, V18, V15}
V19
V = { V19, v8, V18 V15}
v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 v13 v14 v15 v16 v17 v18 v19

Figure 7.10: Illustration of the constructive proof procedure of collapsing staircases.

The diagram in Figure 7.11 shows the fully reconstructed shapetree from the merging procedure performed in Figure 7.10. The shapetree depicted in the figure is one possible way of the creating the shape, among the many other possible ways that might be found depending the order of merging performed.

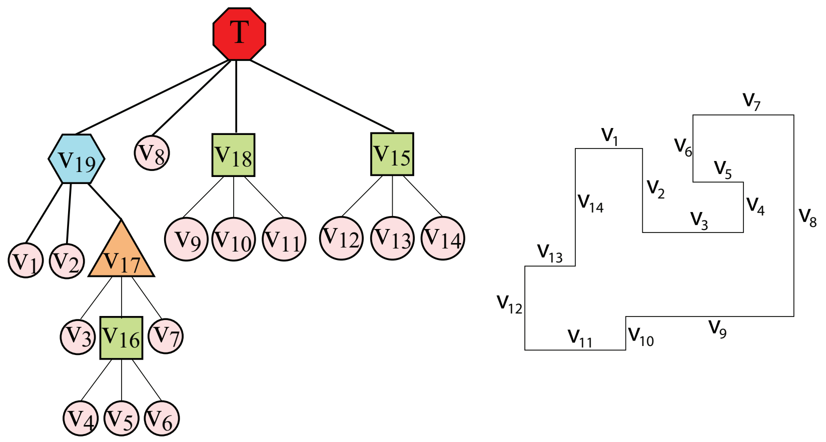


Figure 7.11: Illustration of the constructive proof procedure of collapsing staircases.

7.3.2 Rectangular Cartogram Construction

For the next two steps, the system needs to maintain the form chosen by the user. To this end, we define the *topology* of a form in the following discussion. A form (as shown in Figure 7.9) is lying within a smallest bounding box. The form of each unit (from the starting n canonical units) consists of sub-units (that are boxes) that resulted from split and face advance operations as captured in the shapetree. The empty space outside of all sub-units within the bounding box is considered as the *sea region.*

Processing the sea region

For the purpose of facilitating the sizing algorithm described in the next chapter, there is a need to decompose the "sea" region. These are the regions within the bounding box of the floor plan but are not part of any one of the units. We can conceptually partition the sea region by sweeping a vertical plane from left to right to generate a unique set of rectangular boxes (as shown in Figure 7.9(bottom left)). Analogously, we can do a partitioning with horizontal plane sweeping from top to bottom. However, for efficiency purposes, we describe a more efficient and elegant way of decomposing the region as follows. Firstly, the extremal edges (shown in thick rectangles in Figure 7.12) are computed. We let e_t, e_b, e_l and e_r denote the top, bottom, left and right extremal edges respectively.

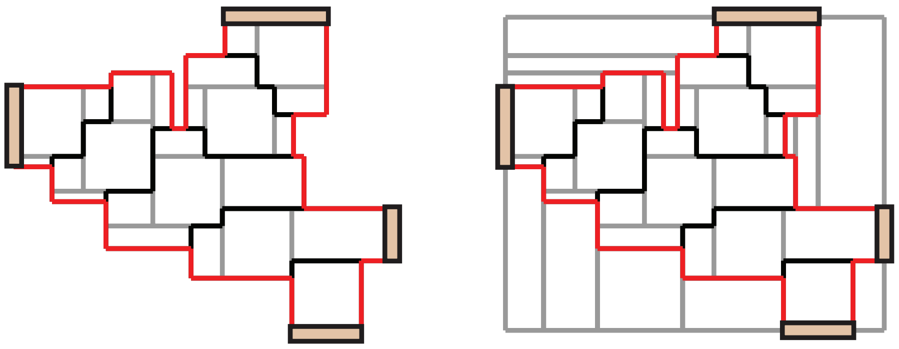


Figure 7.12: Computing the boundary path and decomposing the sea into rectangles.

Computing the boundary path. The computation of the boundary path begins by tracing the route from one extremal segment to the next, i.e. from $e_t \to e_r$, $e_r \to e_b$, $e_b \to e_l$ and $e_l \to e_t$. Since the units are always connected, we are certain that such a path always exists. The data structure for representing the floor plan allows the traversal of the edges of the units as well as jumping from unit-to-unit using their adjacency links on faces whereby the units contact. Each edge e is associated with two vertices v_1 and v_2 and is directed from v_1 to v_2. All edges are oriented in ***clockwise*** manner.

The computation of the boundary path uses two edge pointers *prevedge* and *curedge* (denoted p and c resp.) to traverse the shapetree data structure. Down to symmetry, there are 4 cases to consider during the traversal of the data structure. The 4 cases are illustrated in Figure 7.13 and the algorithm for tracing the boundary path is described in Algorithm 1. Figure 7.12 shows the result (in red colored paths) of applying the above algorithm to compute the four boundary paths of the given configuration.

Algorithm 1: Computing the boundary path.
Input: Pointers to start and end edges.
Output: A list of edges forming the boundary.

```
COMPUTEPATH(edge $e\_start$, edge $e\_end$)
$prevedge \leftarrow e\_start$
$curedge \leftarrow e\_start.nextedge$
while ($curedge! = e\_end$)
    if curedge's incidence list is empty
        $path.add(new\ edge(prevedge.v2, curedge.v2))$
        $prevedge \leftarrow curedge$
        $curedge \leftarrow curedge.nextedge$
    else
        if (curedge.v1 is in the interval of some edge $e$ in the incidence list)
            $curedge \leftarrow e$
        else if (there exist some edge $e$ to the right of prevnode.v2)
            $path.add(new\ edge(prevedge.v2, e.v2))$
            $prevedge \leftarrow e$
            $curedge \leftarrow prevnode.nextedge$
        else
            $path.add(new\ edge(prevedge.v2, curedge.v2))$
            $prevedge \leftarrow curedge$
            $curedge \leftarrow curedge.nextedge$
```

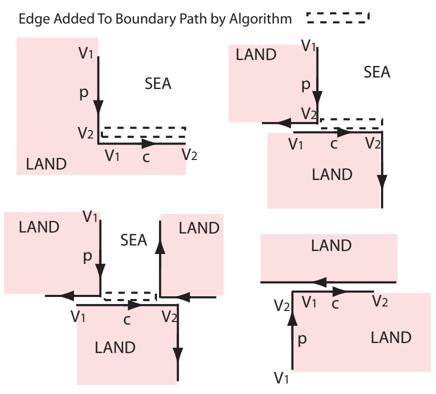


Figure 7.13: Cases to consider for the boundary path tracing algorithm.

Rectangular decomposition of sea region

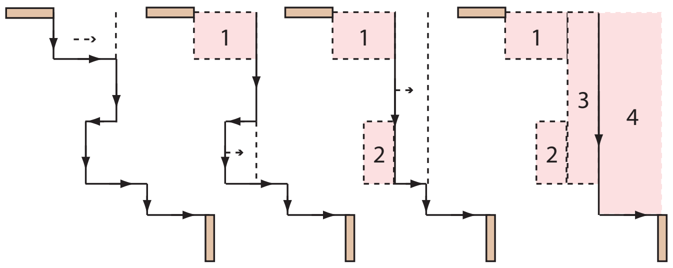


Figure 7.14: Illustration of creating a decomposition of the "sea" region into rectangles using the method of collapsing steps.

With the boundary path computed, a rectangular decomposition of the sea is obtained from the simple method of "collapsing steps" (See Figure 7.14 for an illustration). The algorithm for decomposition is described below:

```
**Algorithm 2:** Computing the rectangular decomposition.
**Input:** A pointer to the starting edge e_start.
**Output:** A list of rectangles comprising the decomposition

COMPUTERECTDECOMP(edge e_start)
curedge ← e_start
while (Number of edges < 2)
    if curedge is a step region
        Collapse the step.
        Add the collapsed rectangle to the list.
        curedge ← curedge.nextedge
    else
        curedge ← curedge.nextedge
Add final rectangle resulting from last 2 edges.
```

Special cases. The algorithm above needs some additional checks to handle the following special cases.

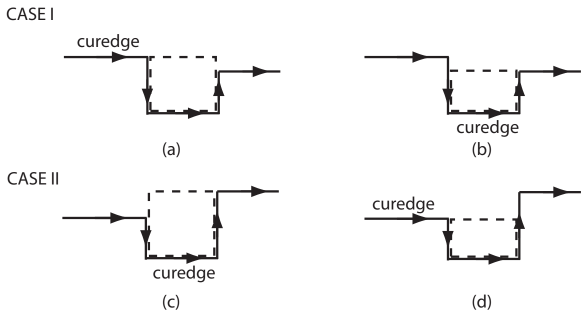


Figure 7.15: Special cases to be handled by the algorithm for rectangular decomposition.

For the case in Figure 7.15(a), the curedge is advanced two links forward as shown in Figure 7.15(b). For the other case shown in Figure 7.15(c), the curedge is reversed two links backward as shown in Figure 7.15(d). This is so that no change is required to the orientation and coordinates of the edges to handle these cases.

7.3.3 Incidence Assignment and Realization

In this step, we are required to manipulate the chosen form in such a way that preserves the topology of the generated floorplan. Firstly, we provide a formal explanation of what maintaining of topology means.

Let E be the set of all the edges of the sub-units and rectangular boxes of the sea region, resulted from both the vertical and horizontal sweep planes. Edges in E are used to construct paths from some corner (vertex) of a sub-unit to another corner (vertex). In the following, for simplicity, we ignore the case where corners p and q are endpoints of an edge or are connected by a chain of vertical edges or a chain of horizontal edges - for this special case, we just need to modify the consideration accordingly. Two corners p and q of possibly different sub-units are such that $p \prec_x q$ if, and only if, there exists a path in E from p to q with no horizontal edge oriented towards the negative x axis. Similarly, $q \prec_x p$ if, and only if, there exists a path in E from p to q with no horizontal edge oriented towards positive x axis. If both of the above do not exist, then p and q are not comparable under $\prec_x$. Analogously, we can define the relationship $\prec_z$ in the z direction between two corners. See the example in lower left of Figure 7.9 where we have $p \prec_x q$, and r is not comparable to q under $\prec_x$. To maintain the topology of a form in the next two steps, we maintain the relationships $\prec_x$ and $\prec_z$ of all the corners.

With the definition of topology taken care of, the main function of this step is to take a chosen form to assign correspondences of some of its edges (or corners) on the boundary to some edges (or corners) in the initial configuration. It stretches the form to fit into the initial configuration obeying the mentioned correspondences of edges or corners, while maintaining the topology. The correspondences can be interactively assigned or autonomously performed by the system. Naïve methods of automatically assignment may not result in a physically realizable form. On the other hand, it can also be the case that multiple choices of realizable correspondences can exist; see Figure 7.8.

For a boundary path starting at corner p and ending at corner q with $p \prec_x q$ and $p \prec_z q$, a realizable correspondence of p to $p' = (p'_x, p'_z)$ and q to $q' = (q'_x, q'_z)$ is one such that $p'_x < q'_x$ and $p'_z < q'_z$. Analogously, we can define realizable correspondences for all the other configuration of $\prec_x$ and $\prec_z$. When p and q are not comparable under $\prec_x$ ($\prec_z$, respectively), then there is no constraint on the relationship of p'_x and q'_x (p'_z and q'_z, respectively). With this, it is easy to validate manual correspondences specified by the user, or to generate valid correspondences automatically.

Following realizable correspondences, the system computes a physical realization iteratively by moving one by one edges of sub-units to their constrained locations. The edge moving process uses two operations to preserve the topology of the form. In *cascading edge move* (Figure 7.17), the movement of an edge can result in movement of other edges so as to maintain non-zero area for each sub-unit and rectangle in the sea region. In *staircase edge move* (Figure 7.18), the movement of an edge can bring along a movement of another parallel edge (separated by just one orthogonal edge) when the separation between the edges violates some minimum separation requirement. These edge moves can propagate to other affected sub-units.

Cascading Edge Move. Refer to Figure 7.16. Conventionally, moving edge $e1$ with the aim of constraining p to p' will decrease the edge lengths of the affected adjacent rectangles (see first two diagrams of Figure 7.16). However, beyond a specified threshold d, we can invoke the operation of moving the previous edge of d (denoted $e2$) by the same incremental amount in the same direction. This effectively 'shifts' the entire affected

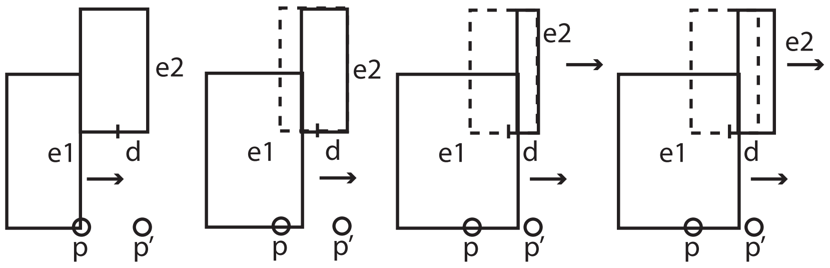


Figure 7.16: An illustration of effect of an cascading edge move on the bold edge while maintaining the relative positioning of p and q.

geometry to the right, with the same operation propagated to other affected parts of the floorplan.

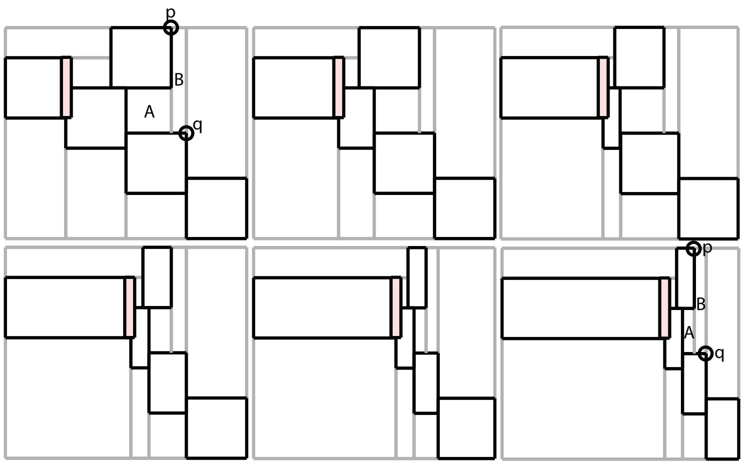


Figure 7.17: Illustration of effect of the cascading move operation.

Figure 7.17 shows the effects of the cascading edge move used to move the bold edge in the diagrams. We note here the special property of cascading edge move operation coupled with the rectangular decomposition of the sea region is such that the topological structure of the layout is always preserved. For example, the relative positions point p and q are always preserved. This is a property of the way the sea region is being decomposed. In particular, it is because of the way the special cases are handled that rectangles A and B are created in their particular arrangement. In the special cases, the “lowest level” are “filled” first and hence this construction, coupled with the cascading move operation ensures that the relative positions of p and q will always be preserved due to the existence of A and B.

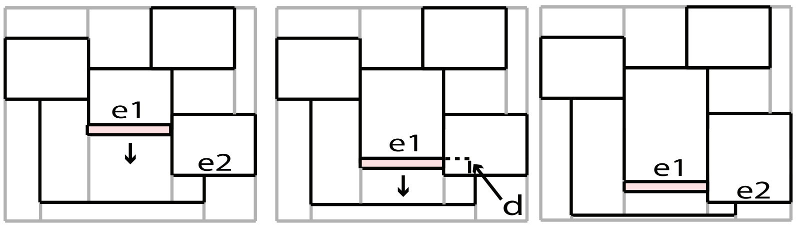


Figure 7.18: An illustration of effect of a staircase edge move on *e1* that induces a move of *e2* and other part of the design.

Staircase Edge Move. Consider the case shown in Figure 7.18. As the edge *e1* moves such that its vertical separation from *e2* violates some specified threshold *d*, a move is also induced in the edge *e2*, which will be in turn propagated to other affected parts of the floor plan. This ensures that the topology of the incidences between units are always preserved.

7.3.4 Geometry Sizing

With a realized topology, the next step of the completion process constitute the sizing of the topology, the aim of which is to size each sub-unit to one that is sufficiently large enough to be a room of some architectural use. Since our system is to provide a rapid prototyping tool for previewing a particular instance of floorplan, we require a fast method for computing good sizing of the floorplan and subsequently synthesize the geometry for visualization.

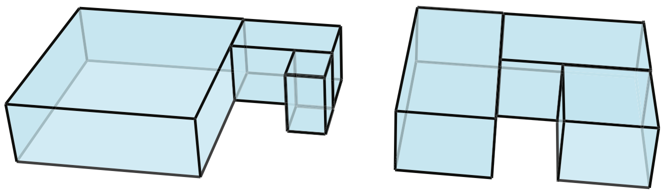


Figure 7.19: Comparison of good and bad unit sizings.

Defining Good Sizings. To arrive at some measure for discerning architecturally useful designs from the rest, we employ the concept of equally-sized subunits (See Figure 7.19). From the diagram, it is easy to see that units with evenly distributed subunit areas are more desirable than those with large differences in their subunit areas.

With the mentioned edge moving operations, the user can manually size each sub-unit. As for automatic sizing, we aim for equally-size sub-units (see Figure 7.19). The algorithmic problem is as follows. For an input set of rectangles $R = \{r_1, r_2, \ldots, r_k\}$ with initial areas $A_0, A_1, \ldots, A_k$, we want to change the areas of each r_i into a targeted value A'_i. For our problem, we assign to those rectangles belonging to sub-units a large target value M (which can be the total area of the bounding box of the design over the number of sub-units in the design) whereas those to the sea region a small positive value m.

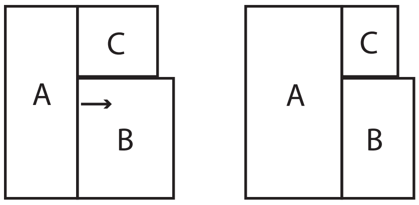


Figure 7.20: Illustration of moving segment heuristic.

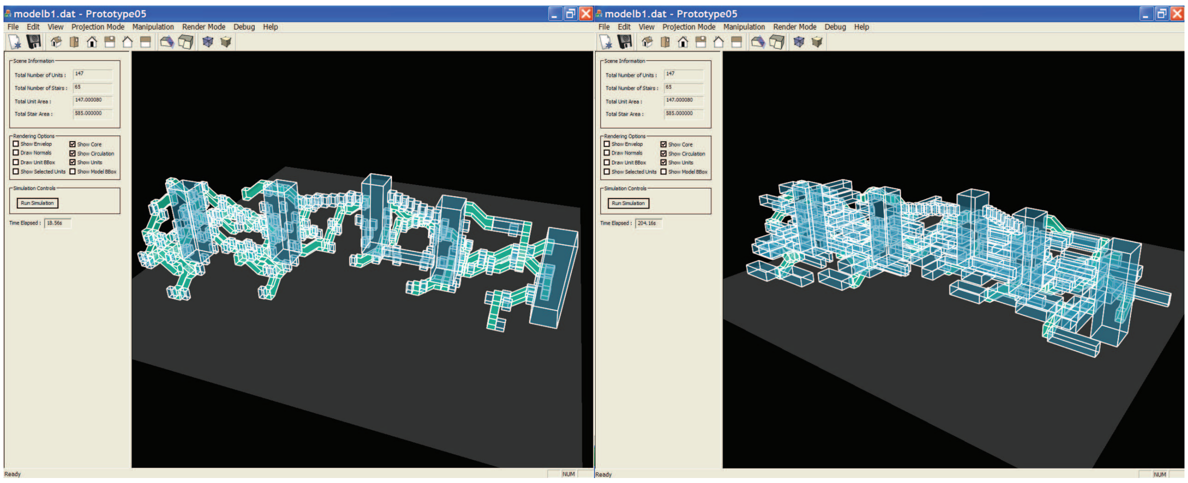
To size our floorplan, the algorithm in [Kreveld and Speckmann, 2004], which is based on the moving segment heuristic is used with appropriate modifications. Refer to Figure 7.20, the vertical segment of A shared between B and C can be moved to the right to increase the area of A. It can also be moved to the left achieving the opposite effect. The sizing algorithm iteratively loops over all such segments in the floorplan and move them in direction which decreases the maximum errors of the adjacent regions. Intuitively, after a number of iterations, all the segments would have moved to a locally optimum position. Such an iterative process, adapted from [Kreveld and Speckmann, 2004], with some bound on the number of iterations, is performed to size each sub-unit while maintaining the topology of the form. The process gives priority to size sub-unit with the largest difference to its target size.

With the four-step process described in the last few sections, we are able to rapidly generate completed forms for units in small regions in the building model. Figure 7.1

shows an example of an actual model of the form completed level generated by our program.

Chapter 8

Experimental Results


modelb1.dat - Prototype05
File Edit View Projection Mode Manipulation Render Mode Debug Help
Run Simulation
Ready
NUM
modelb1.dat - Prototype05
File Edit View Projection Mode Manipulation Render Mode Debug Help
Run Simulation
Ready
NUM

Scene Information
Total Number of Units : 147
Total Number of Stairs : 65
Total Unit Area : 147.000080
Total Stair Area : 585.000000

Rendering Options
☐ Show Envelop ☑ Show Core
☐ Draw Normals ☑ Show Circulation
☐ Draw Unit BBox ☑ Show Units
☐ Show Selected Units ☐ Show Model BBox

Simulation Controls
Time Elapsed : 18.56s

Scene Information
Total Number of Units : 147
Total Number of Stairs : 65
Total Unit Area : 147.000080
Total Stair Area : 585.000000

Rendering Options
☐ Show Envelop ☑ Show Core
☐ Draw Normals ☑ Show Circulation
☐ Draw Unit BBox ☑ Show Units
☐ Show Selected Units ☐ Show Model BBox

Simulation Controls
Time Elapsed : 204.16s

Figure 8.1: Screenshot of the initial and final stages of largescale model generation using our system.

Our system is implemented with the CGAL library on a Pentium IV 3.0GHz, 1GB DDR2 RAM and nVidia GeForce 6600 GT with 128M DDR3 video memory. Figure 8.1 shows the screenshot of our system during the initial and final stages of the generation of our large scale models.

Alpha and Beta (Figure 1.1(top) and Figure 8.2) are two large scale models, generated using our system with their input cores as shown too. These models are generated in less than 5 minutes in an interactive session. See the accompanying video for an animation of the generation process of the Beta model. A partial example of the form

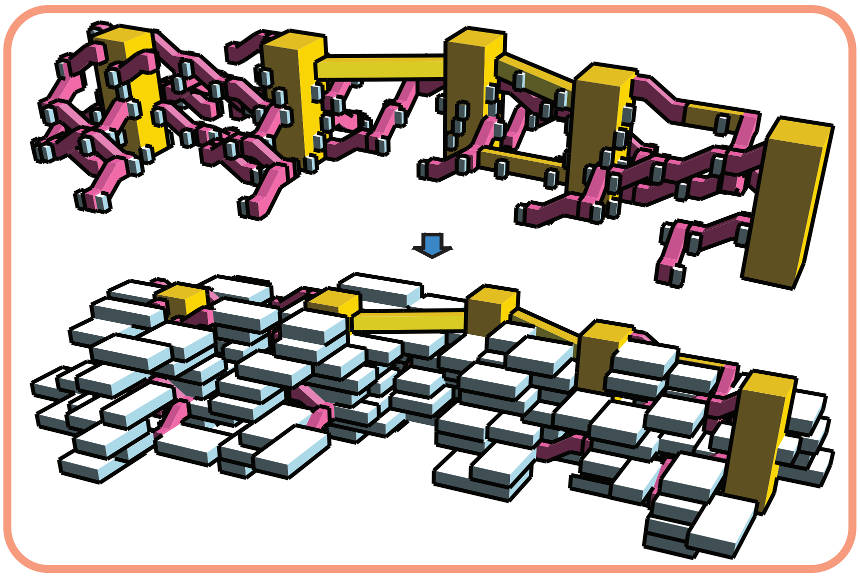


Figure 8.2: Initial and final generated form of our Beta model.

Model	Site Area	No. of Units	Total No. of Occupants	Occupancy Ratio
Alpha	90m x 50 m	71	213	0.0095
Beta	100m x 60m	134	364	0.0101
	Type A Units		Type B Units	Type C Units
Alpha	28		15	28
Beta	42		31	61
	Average View		Multiple View Units	
			Two Views	Three Views
Alpha	$28m^2$		17	29
Beta	$25m^2$		34	54
	Average Floor Area		Units with Double Volume	Units with Outdoor Space
Alpha	$62.2m^2$		25	27
Beta	$60.5m^2$		47	57

Table 8.1: Table of statistics on space use for generated models.

completion method is shown as Figure 7.1. See the accompanying video for a more comprehensive illustration of the process.

Table 8.1 summarizes the statistics on the use of space by the Alpha and Beta models. We note that Swedish housing standards [Swedish Regulation] specify that the minimum comfortable dwelling area is at 30 to 50m^2 for one person (Type C), at 50 to 85m^2 for three (Type B), and at greater than 85m^2 for a family of five (Type A). With these, we can calculate the number of total occupants for Alpha and Beta. To appreciate the occupancy ratio, we note a typical cookie-cutter building of about $32\text{m}\times 27\text{m}$ has

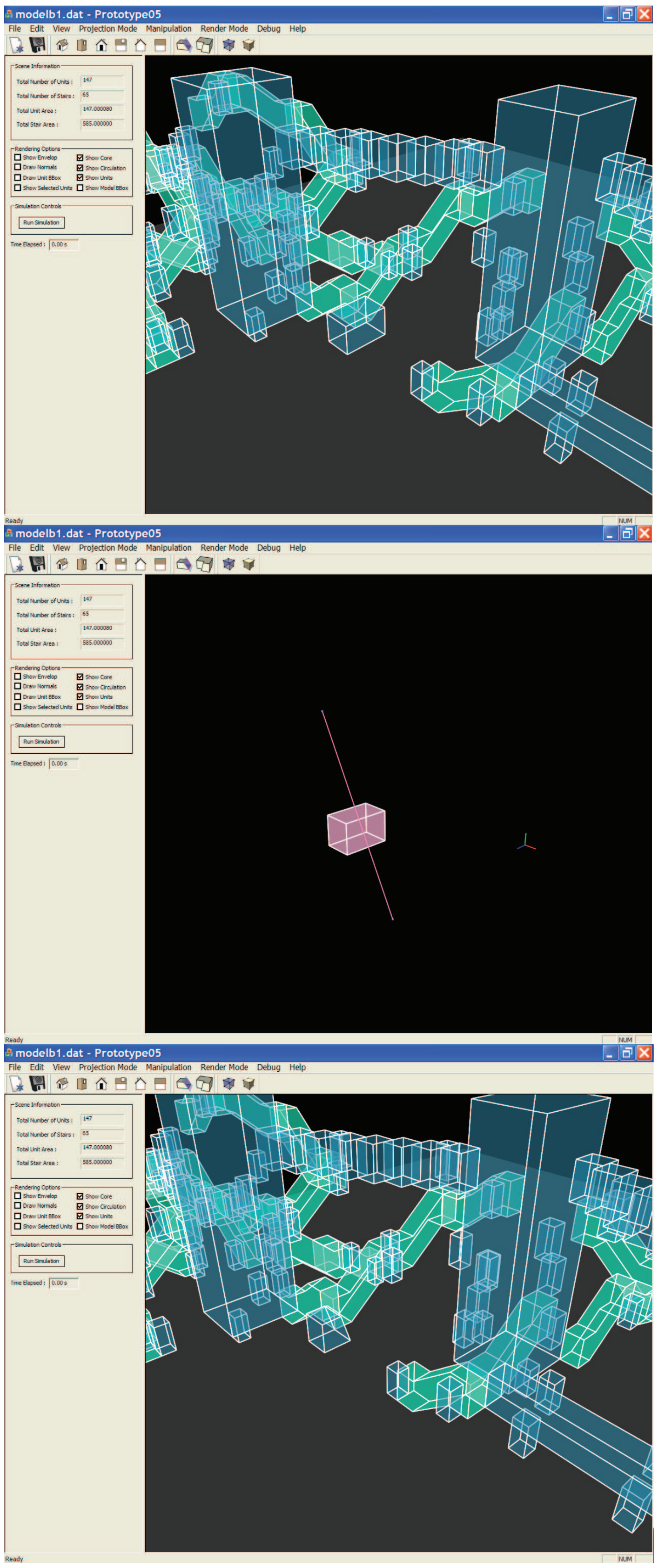
4 units per storey. Assuming each unit houses 4 persons, we then have 16 persons in one storey, which gives an occupancy ratio of 0.0185 person per square meter. Our (5 storey) Alpha and (6 storey) Beta forms can achieve about half the efficiency in usage of space compared to a typical conventional cookie-cutter building.

Apparently, the efficiency in space usage in the conventional way is traded to create outdoor space of good attribute values in Alpha and Beta; see for example the insert to Figure 1.1 on the possible usage of suspended open spaces. Table 8.1 also summaries the statistics on the view and shape variation of Alpha and Beta. Indeed, we observe that about half the number of units have good suspended open spaces. In addition, the average view of all the units are considered high as it is close to half the size of the average floor area. As for shape variation, we also observe good percentage of units having attractive double volume and multiple views. Due to the non-regular units, there are many ventilation corridors in both Alpha and Beta from our visual inspection.

The system also features some other shape manipulation features such as object slicing (See Figure 8.3), shadows and shadow placement control (See Figure 8.4) as well as the collision and extrude feature (See Figure 8.5).

Other than large scale model generation, the other major component of the system is the form completion engine for rapid prototyping of local regions of the model. Refer to Figure 8.7 for screenshots of the system used to generate the floorplans from which the user can choose from and subsequently integrated (semi)automatically into the existing model being built.

On the whole, we demonstrate the possibility (see Figure 8.7 for an artist's illustration of of creating forms that look beyond the usual occupancy ratio but a premium type of housing units with good views and variation of shapes to provide for quality living in modern housing.



Scene Information
Total Number of Units : 147
Total Number of Stairs : 65
Total Unit Area : 147.000080
Total Stair Area : 585.000000

Rendering Options
☐ Show Envelop ☑ Show Core
☐ Draw Normals ☑ Show Circulation
☐ Draw Unit BBox ☑ Show Units
☐ Show Selected Units ☐ Show Model BBox

Simulation Controls

Time Elapsed : 0.00 s

Scene Information
Total Number of Units : 147
Total Number of Stairs : 65
Total Unit Area : 147.000080
Total Stair Area : 585.000000

Rendering Options
☐ Show Envelop ☑ Show Core
☐ Draw Normals ☑ Show Circulation
☐ Draw Unit BBox ☑ Show Units
☐ Show Selected Units ☐ Show Model BBox

Simulation Controls

Time Elapsed : 0.00 s

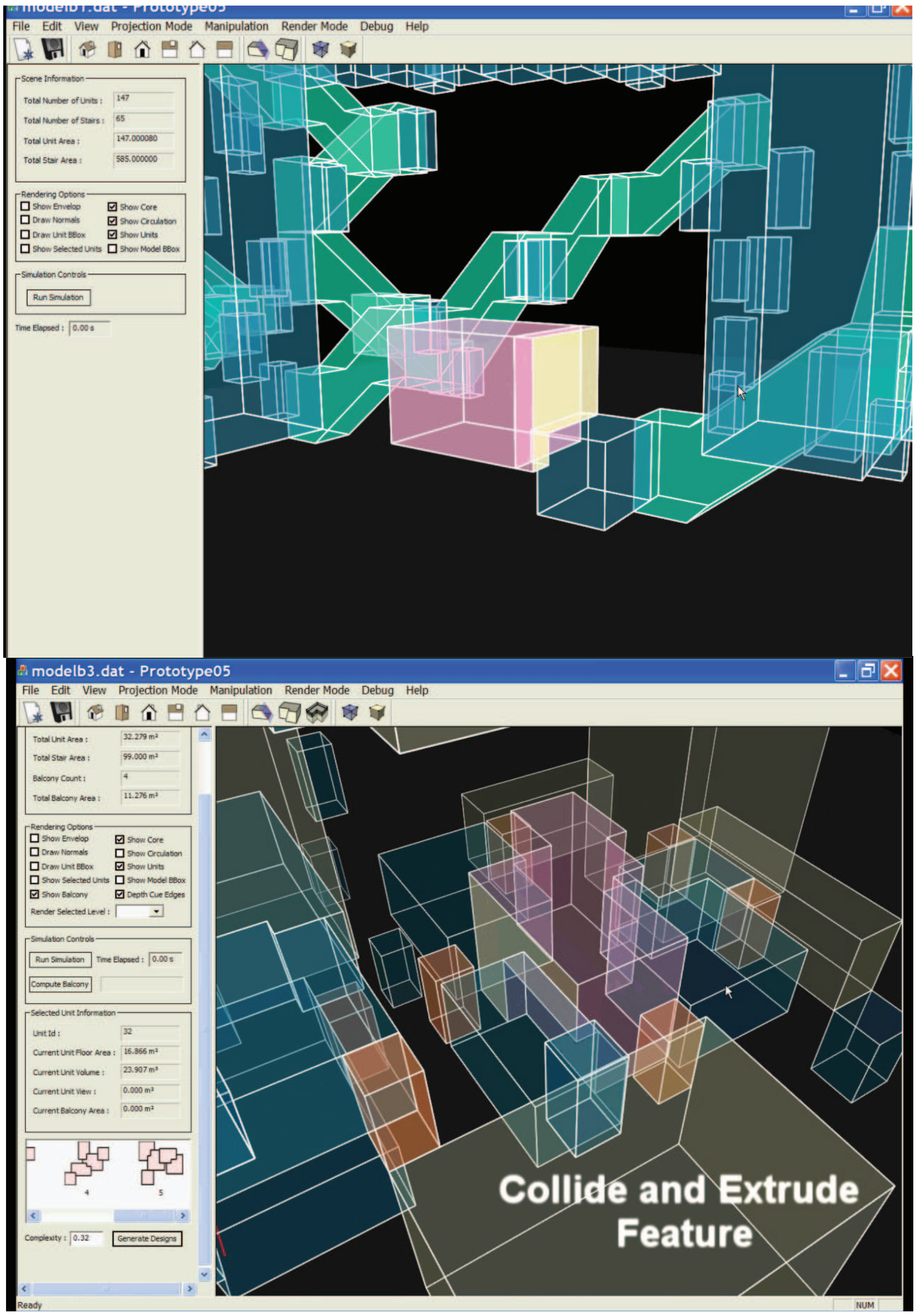
Scene Information
Total Number of Units : 147
Total Number of Stairs : 65
Total Unit Area : 147.000080
Total Stair Area : 585.000000

Rendering Options
☐ Show Envelop ☑ Show Core
☐ Draw Normals ☑ Show Circulation
☐ Draw Unit BBox ☑ Show Units
☐ Show Selected Units ☐ Show Model BBox

Simulation Controls

Time Elapsed : 0.00 s

Figure 8.3: Demonstration of user interface capabilities for object slicing.


modelb1.dat - Prototype05
File Edit View Projection Mode Manipulation Render Mode Debug Help
modelb3.dat - Prototype05
File Edit View Projection Mode Manipulation Render Mode Debug Help
Run Simulation
Compute Balcony
Generate Designs
4
5
Ready
NUM
Collide and Extrude Feature

Scene Information

Total Number of Units : 147

Total Number of Stairs : 65

Total Unit Area : 147.000080

Total Stair Area : 585.000000

Rendering Options

☐ Show Envelop ✓ Show Core

☐ Draw Normals ✓ Show Circulation

☐ Draw Unit BBox ✓ Show Units

☐ Show Selected Units ☐ Show Model BBox

Simulation Controls

Time Elapsed : 0.00 s

Total Unit Area : 32.279 m²

Total Stair Area : 99.000 m²

Balcony Count : 4

Total Balcony Area : 11.276 m²

Rendering Options

☐ Show Envelop ✓ Show Core

☐ Draw Normals ☐ Show Circulation

☐ Draw Unit BBox ✓ Show Units

☐ Show Selected Units ☐ Show Model BBox

✓ Show Balcony ✓ Depth Cue Edges

Render Selected Level :

Simulation Controls

Time Elapsed : 0.00 s

Selected Unit Information

Unit Id : 32

Current Unit Floor Area : 16.866 m²

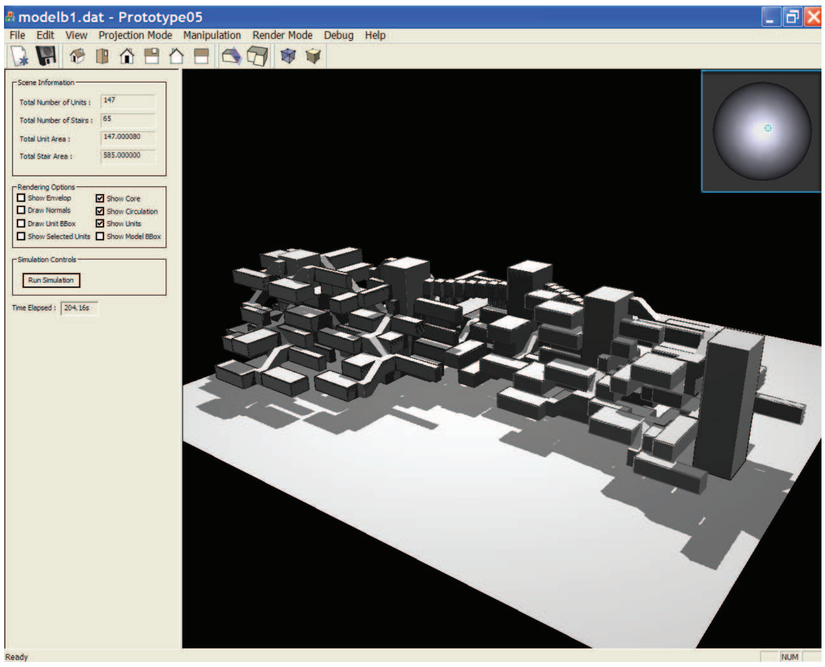
Current Unit Volume : 23.907 m³

Current Unit View : 0.000 m²

Current Balcony Area : 0.000 m²

Complexity : 0.32

Figure 8.4: Demonstration of user interface capabilities of collide and extrude. The highlighted blocks in the screenshots are the one being extruded.



Scene Information

Total Number of Units :	147
Total Number of Stairs :	65
Total Unit Area :	147.000080
Total Stair Area :	585.000000

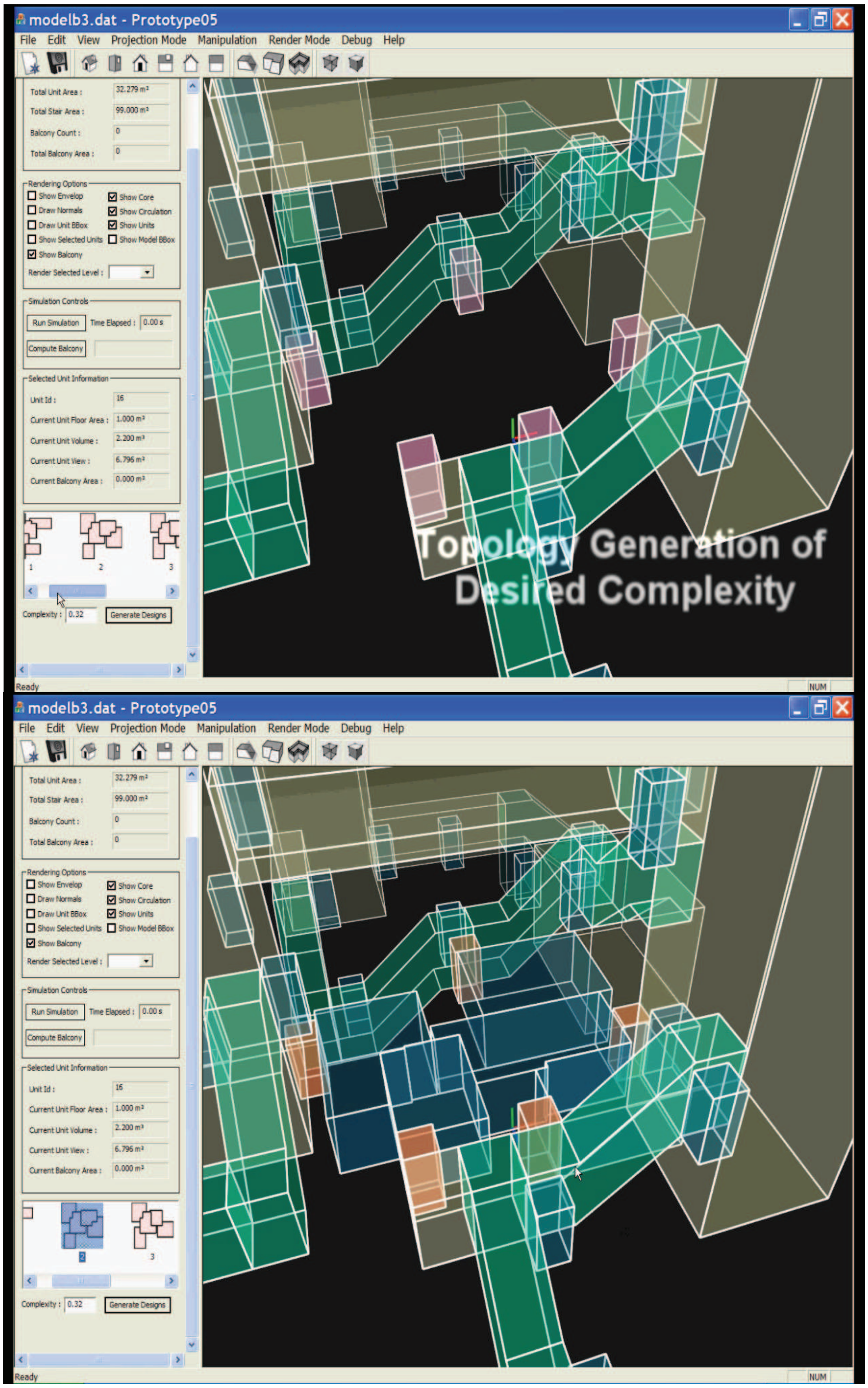
Rendering Options

- ☐ Show Envelop
- ☐ Draw Normals
- ☐ Draw Unit BBox
- ☐ Show Selected Units
- ☑ Show Core
- ☑ Show Circulation
- ☑ Show Units
- ☐ Show Model BBox

Simulation Controls

Time Elapsed : 204.16s

Figure 8.5: Demonstration of shadowing capabilities of our system implemented using the techniques of shadow mapping. The top right widget allows the user to interactively alter the placement of the light source allowing architects to envision how a generated model respond to sun paths.


modelb3.dat - Prototype05
File Edit View Projection Mode Manipulation Render Mode Debug Help
Run Simulation
Compute Balcony
Generate Designs
Topology Generation of Desired Complexity
Ready
NUM

Total Unit Area : 32.279 m²
Total Stair Area : 99.000 m²
Balcony Count : 0
Total Balcony Area : 0

Rendering Options
- ☐ Show Envelop
- ☐ Draw Normals
- ☐ Draw Unit BBox
- ☐ Show Selected Units
- ☑ Show Balcony
- ☑ Show Core
- ☑ Show Circulation
- ☑ Show Units
- ☐ Show Model BBox

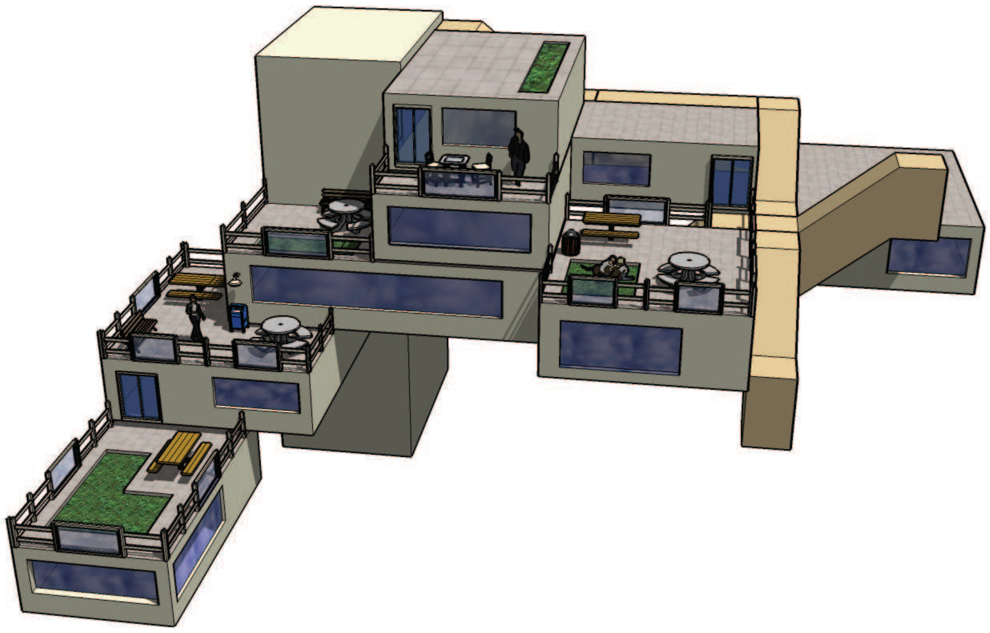
Render Selected Level :

Simulation Controls
Time Elapsed : 0.00 s

Selected Unit Information
Unit Id : 16
Current Unit Floor Area : 1.000 m²
Current Unit Volume : 2.200 m³
Current Unit View : 6.796 m²
Current Balcony Area : 0.000 m²

Complexity : 0.32

Figure 8.6: Demonstration of system and its use in generating floorplan and the automatic integration of generated floorplan into the existing model.



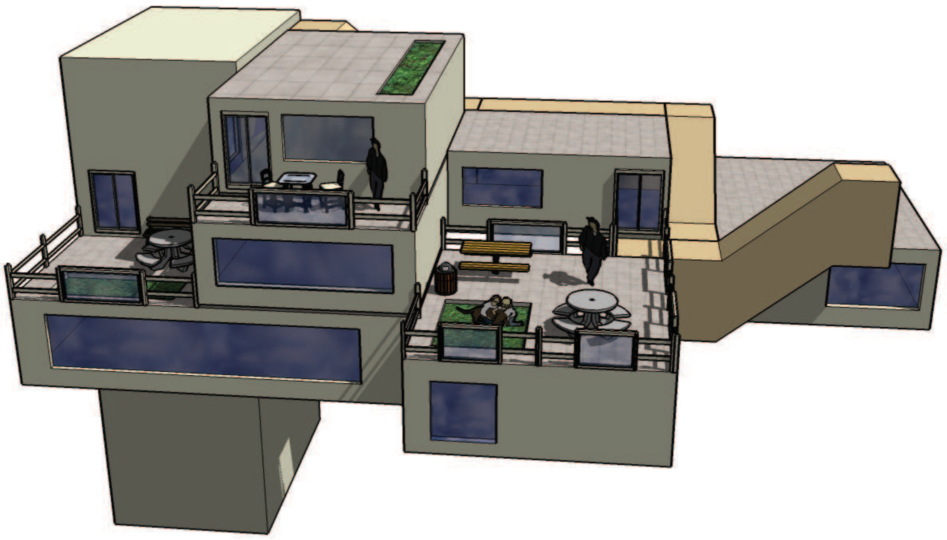


Figure 8.7: Artist's impression of possible realization of generated model.

Chapter 9

Concluding Remarks

This thesis proposes a growth-based approach for generative design in an interactive environment. The approach incorporates consideration of views and variation of shapes in generating units. It also has emergent spatial features to create units with suspended open spaces and good ventilation. Interestingly, the system relies on only two primitive shape operators of face splitting and face advancing to synthesize a variety of forms. The outputs of the system are physically valid and meeting building constraints. Our system also demonstrates the exciting possibility of rapidly synthesizing non-regular forms on the small scale using our completion engine.

There remain possible improvements to our current prototype system. Firstly, we can consider the possibility of fine tuning or defining alternative architectural criteria and growth strategies. For example, view may be modified to define with view volume truncated at some finite distance rather than the stringent, naïve infinite orthogonal projection. This has implication on how GPU can be used to speed up the view computation. Another example of possible work is to examine growth strategy involving a coupling of two or more units to form specific patterns of form. Secondly, on the architectural aspect, it would be interesting to look into a quantitative case study of comparing and contrasting the attribute values of our outputs with existing non-regular forms (should detailed architectural information be available) such as Habitat'67.

Bibliography

CGAL. The CGAL Consortium. Computational Geometry Algorithms Library., 2005. URL `http://www.cgal.org`.

S C Chase. Shapes and shape grammars : from mathematical model to computer implementation. In *Environment and Planning B : Planning and Design B*, volume 16, pages 215–242, 1989.

John Gero and Vladimir Kazakov. On measuring the visual complexity of 3d solid objects. In *International Journal of Design Sciences and Technology.*, 2004.

Christiane M. Herr. Using cellular automata to challenge cookie-cutter architecture. In *The Proceedings of the 5th Conference on Generative Art 2003*, pages 72–81, 2003.

AutoDesk Inc. http://www.autodesk.com, 2005.

Marc Kreveld and Bettina Speckmann. On rectangular cartograms. In *LNCS*, volume 3321, pages 724–735, 2004.

Benjamin A. Loomis. A user driven genetic algorithm for evolving non-deterministic shape grammars. In *MIT Department of Architecture Technical Report*, 2002.

Radomér Měch and Przemyslaw Prusinkiewicz. Visual models of plants interacting with their environment. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer Graphics and Interactive Techniques*, pages 397–410, New York, USA, 1996. ACM Press. ISBN 0-89791-746-4.

Yoav I. H. Parish and Pascal Müller. Procedural modeling of cities. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 301–308, New York, USA, 2001. ACM Press. ISBN 1-58113-374-X. doi: http://doi.acm.org/10.1145/383259.383292.

P. Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants.* Springer-Verlag New York, Inc., New York, USA, 1990. ISBN 0-387-97297-8.

Moshe Safdie. Habitat '67, 1967. URL `http://www.habitat67.com/`.

Christian Schittich. *High-density housing : concepts, planning, construction.* Birkhäuser, 2004.

Swedish Regulation. Swedish housing standards. URL `http://www.johngilbert.co.uk/resources/swedish.html`.

Peter Wonka, Michael Wimmer, Francois Sillion, and William Ribarsky. Instant architecture. *ACM Transactions on Graphics*, Vol. 22(3):669–677, 2003. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/882262.882324.