

Mapping Boundaries of Generative Systems for Design Synthesis

By

Maher El-Khaldi

Bachelor of Architecture. 2004
American University Of Sharjah
United Arab Emirates

Submitted to the Department of Architecture
in Partial Fulfillment of the Requirements for the Degree of

Master Of Science In Architecture Studies
Massachusetts Institute Of Technology

June 2007

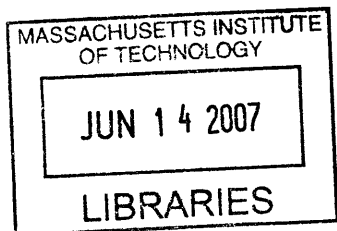
©2007 Maher El-Khaldi. All rights reserved.

The author hereby grants to M. I. T. permission to reproduce and distribute publicly
paper and electronic versions of this thesis document in whole or in part
in any medium now known or hereafter created.

Signature of Author: _____
Department of Architecture
May 24, 2007

Certified by: _____
George Stiny
Professor of Computation, M. I. T.
Thesis Supervisor

Accepted by: _____
Julian Beinart
Professor of Architecture, M. I. T.
Chairman, Department Committee on Graduate Students



ROTCH

Thesis Readers:

Terry Knight
Professor of Computation, M. I. T.

Ann Pendleton-Jullian
Associate Professor of Architecture, M. I. T.

Mapping Boundaries of Generative Systems for Design Synthesis
By
Maher El-Khaldi

Submitted to the Department of Architecture on May 24, 2007
in Partial Fulfillment of the Requirements for the Degree of
Master Of Science In Architecture Studies

Abstract:

Architects have been experimenting with generative systems for design without a clear reference or theory of what, why or how to deal with such systems. In this thesis I argue for three points. The first is that generative systems in architecture are implemented at a skin-deep level as they are only used to synthesize form within confined domains. The second is that such systems can be only implemented if a design formalism is defined. The third is that generative systems can be deeper integrated within a design process if they were coupled with performance-based evaluation methods.

These arguments are discussed in four chapters:

- 1- Introduction: a panoramic view of generative systems in architecture and in computing mapping their occurrences and implementations.
- 2- Generative Systems for Design: highlights on integrating generative systems in architecture design processes; and discussions on six generative systems including: Algorithmic, Parametrics, L-systems, Cellular Automata, Fractals and Shape Grammars.
- 3- Provisional taxonomy: A summery table of systems properties and a classification of generative systems properties as discussed in the previous chapter
- 4- Conclusion: comments and explanations on why such systems are simplicity implemented within design.

Thesis Supervisor: George Stiny
Title: Professor of Computation. M. I. T.

Acknowledgements:

I would like to start by thanking God for the blessings He bestowed upon me. Only He would know how thankful I am for everything, not least of which for all the genuine people I met. I envy myself for how blessed I am when I think of how many they are. The least I can do is thank them for making a difference.

I would like to thank my father Sami El-Khaldi and my mother (God bless her soul) Waheeda Abu Sharkh; my sisters Lina and Maha; and brothers Ahmed, Mohammad and my twin Munther. They have been always supportive of me in every form. They never said no to anything they could do to help me realize my dreams.

I also would like to thank some of the most patient and supportive friends I met in my undergraduate school, alphabetically: Abdulla, Alia, Ahmad, Azadi, Bashar, Hisham, Kareem, Khaider, Khaldoon, Lamees, Marriam, Mohammad, Mohannad, Mouiz, Noor, Nibras, Rawan, Tamer, Zaki. When my mother passed away I shattered into sharp pieces. Yet, some friends tried as hard as they could to put me back together. Amin, Bashar, Lamees and Nibras, I can't think of what made them do this! I also would like to thank Abdulla for being such a supportive friend since the day I knew him.

I also would like to thank the person who believed in me and sponsored my first semester at M.I.T with no strings attached! Nancy for helping me find a scholarship for my master's study, and Danielle and Dina Kan'aan for helping me obtain a visa.

I also would like to thank, from the American University of Sharjah, professors: Ahmad, George, Kevin, Nadia, Nawar, Tarik, Randle, and Rula for being such passionate teachers.

I also would like to thank some of the wonderful people I met at M. I. T. and Cambridge, who showed me how beautiful Islam is, Dalia and Elizabeth.

I would like to thank my friends Saud, Hassen, and Nadeem for being such great company; Kenfield and Kyle for debugging my Rvb. code; and Arjun, Josh and Meleena for their feedback.

I would like to thank Lamees for putting a great effort in reading and editing this whole thesis. She has always been a wonderfully patient and dedicated friend.

I would also like to thank Anas for being such a great mentor throughout my study at M.I.T. He taught me many things starting from day one. Things I might not have thought of.

I also would like to thank Terry and Ann for the interesting design discussions we had, and Axel for the interesting workshops he offered.

I also would like to thank Bill Mitchell for the deep and rich discussions he enticed my mind with.

I also would like to thank George Stiny for giving me the freedom, coupled with the right amount of pressure to explore and build my own path.

Having met all of these individuals, I believe I could not be any luckier. Thank you all for investing in me.

Content:

1.0 Introduction:

2.0 Generative Systems In Design:

2.1 Designing with Generative Systems:

2.2 Highlights On Selected Generative Systems:

2.2.1 Algorithms:

2.2.2 Parametric:

2.2.3 L-Systems:

2.2.4 Cellular Automata:

2.2.5 Fractals:

2.2.6 Shape Grammars:

3.0 A Provisional Taxonomy Of Generative Systems:

4.0 Conclusion:

5.0 Bibliography

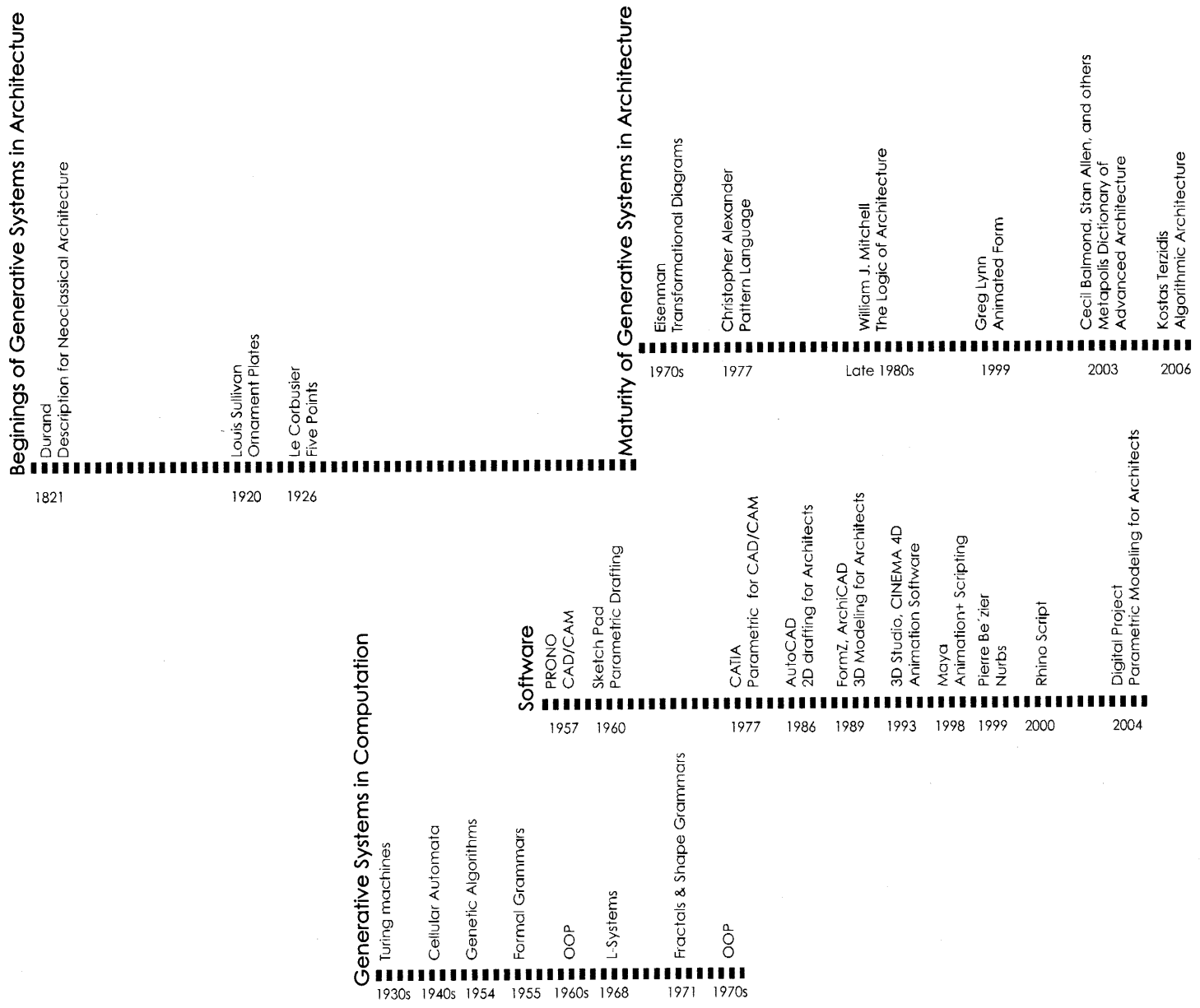
1.0 Introduction:

In order to effectively study a topic it is useful to first establish an understanding of key terms. Below are a series of keywords and meanings found in the Oxford Dictionary. These are loose suggestions to clarify main concepts from a design point of view. Let's start with the title: "Mapping Boundaries of Generative Systems for Design Synthesis"

- Mapping: "...a diagrammatic representation of an area of land..."
- Boundaries: "...a line that marks the limits of an area..."
- Generative: "...of or relating to reproduction..."
- System: "...a set of things working together as parts of a mechanism..."
- Design: "...an arrangement of lines or shapes..."
- Synthesis: "...combination or composition, in particular: the combination of ideas to form a theory or system..."

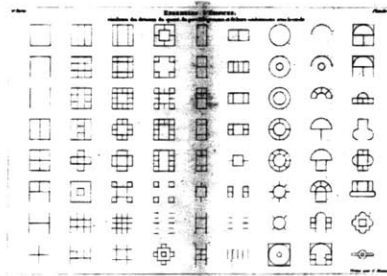
The maturity of Generative systems in architecture suffers from severe "lateness". Architects have been solely experimenting with them for form synthesis. These systems appeared "later" in the in computation for simulating various phenomena for the purposes of analysis. Both approaches seemed valid in their respective contexts, however, their level of success translate differently. In architecture, most of the generated outcome is skin-deep where in computation it is endless. The reasons behind such variance lie in the process, goals and context in which each is being implemented.

The following diagram maps some of the most important occurrences of generative systems in architecture, and in computation. It also maps the developments of digital tools (softwares) and the related concepts in architecture theory leading to current practices.



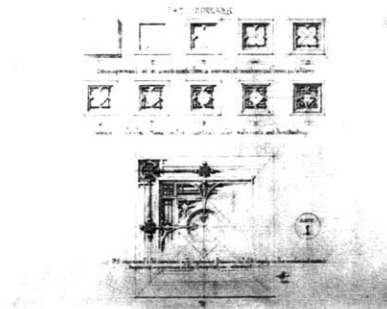
Time line showing various developments in architecture, computation and software.

The previous diagram is presented as a cascade of four time lines so that the readers may create their own connections. For now, let's start by the "beginnings of generative systems in architecture". In 1821, Durand wrote the "Partie graphique des cours d'architecture" describing two design stages to generate Neoclassical Architecture, a bottom-up and top-down approach. The first guides the creation of an architecture skeleton where the second directs the addition of details. Both processes complement each other.¹



One of Durand's Plate 20. Image Credits: Louis Durand, (trans.) David Britt, Precis of the Lectures on Architecture: With Graphic Portion of the Lectures on Architecture, (Getty Trust Publications: Getty Research Institute for the History of Art and the Humanities, 2000).

Along the same lines, but on a different scale and almost a hundred years later, *Louis Sullivan* produced a series of plates titled "A System of Architectural Ornament, According with a Philosophy of Man's Powers". These plates devised detailed processes for reproducing various types of floral ornamentation based on geometrical constructs.²



Louis Sullivan. Image Credits: Louis Sullivan, ed. Narciso G. Menocal, Robert Twombly, The Poetry of Architecture, 1st edition, (W. W. Norton & Company, 2000).

1 William J. Mitchell, The Logic of Architecture: Design, Computation, and Cognition, (Cambridge: The MIT Press, 1990), 145.

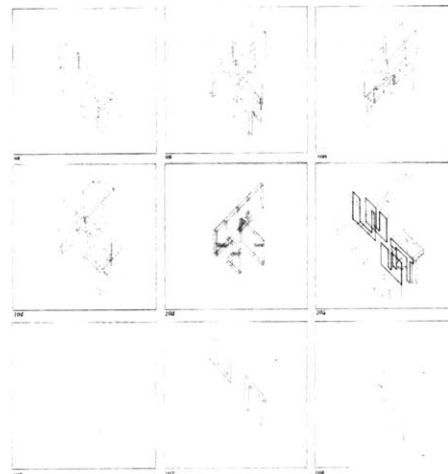
2 Louis Sullivan, ed. Narciso G. Menocal, Robert Twombly, The Poetry of Architecture, 1st edition, (W. W. Norton & Company, 2000).

Six years later, Le Corbusier formalized his style in Les 5 points d'une architecture nouvelle. These included: 1- The pilotis elevating the mass off the ground; 2- The free plan that is achieved through the separation of the load-bearing columns from walls; 3-The free façade as the corollary of the free plan; 4- The long horizontal sliding window; 5- The roof garden restoring the area of ground covered by the house.³ Although these points described only a style, they implicitly impose objectives and goals by which a designer may tailor their own generation methods.



Le Corbusier's Five Points. Image Credits: <http://www.geocities.com/rr17bb/LeCorbusier5.html>

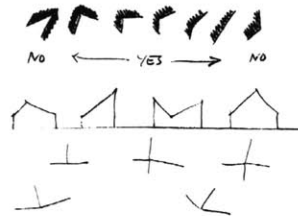
In 1971, Eisenman presented a series of “transformational diagrams” in the House X project series. Although they were deterministic and descriptive of an existing design artifact, they conveyed possible variations and clear representations of design steps.⁴



Transformational Diagrams. Image Credits: Peter Eisenman, “House X,” [Rizzoli](#) 15 April 1983.

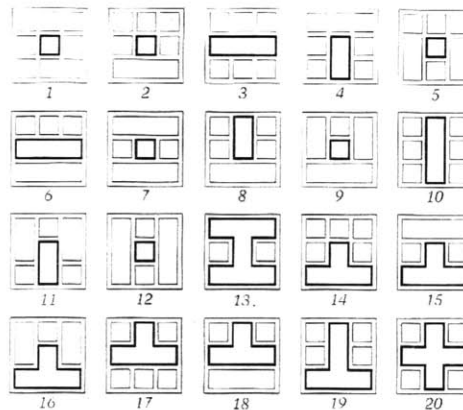
3 Leland M. Roth, Understanding Architecture: Its Elements, History, and Meaning. (HarperCollins Publishers 1993), 475-480.
 4 Peter Eisenman, “House X,” [Rizzoli](#) 15 April 1983.

In 1977, Christopher Alexander published "Pattern Language". His work is similar to that of Le Corbusier but at a bigger scale. The patterns he proposed serve as guidelines for composing an architectural context.⁵ He discussed cities, villages, districts, land parcels, streets, public spaces, private spaces, etc. It is important to note that Alexander's work had a major influence on the field of computer science, specifically, on building hierarchies in programming languages.



Visual description of good and bad street corners, roof tops, urban corners. Image credits: Christopher Alexander, *A Pattern Language: Towns, Buildings, Construction*, (USA: Oxford University Press, 1977).

The concept of *reproduction* was inherent to all of the above processes. However, none of them explicitly discussed the idea of "*reproduction for variation*". One of the first systems written for variation in architecture came right after Pattern language. It was based on the formalism of Shape Grammars.⁶ Below is an example of the work developed by William Mitchell and George Stiny for generating Palladian Villas⁷.



Examples on variation generated through Shape Grammars for Palladian Villas. Image credits: William J. Mitchell, *The Logic of Architecture: Design, Computation, and Cognition*, (Cambridge: The MIT Press, 1990), 173.

5 Christopher Alexander, *A Pattern Language: Towns, Buildings, Construction*, (USA: Oxford University Press, 1977).

6 Shape Grammars was developed by George Stiny and James Gips for visual calculation.

7 George, Stiny and W. J. Mitchell. "The Palladian Grammar." *Environment and Planning B: Planning and Design* 1978, 5: 5-18.

These examples were followed by a flow of various implementations by Greg Lynn, Ali Rahim, Cecil Balmond, Norman Forster, and many others who used systematic design methods to generate “new” architectural forms. Their processes were mainly influenced and informed by evolving digital tools.

In the previous pages I have tracked some of the major occurrences of generative systems in architecture showing when and how they were implemented for design. In the following pages, I will introduce another set of (more established) generative systems in computation and show how they reappeared in architecture as new systems.

As a start, I would like to clarify a certain point to avoid later confusion. Computation should not to be mixed with computerization. The former is a process of calculating where the later is a technique whereby computation is performed by computers. Humans can also perform computing to a certain degree, however, they are unable to equal computers ability to perform tasks repetitively and flawlessly, or to store large amounts of information and immediately retrieve it. This was one of the limitations that Alan Turing attempted to explore through his abstract machines (Turing machines) in the late 1930s.⁸

As an extension to the concept of automated calculation, John Von Neumann built an abstract model of self-reproduction in the late 1940s to mimic biological growth. He constructed a system of cells and states where each cell would react to its neighbors’ conditions.⁹ This system was known as “Cellular Automata”. This represents one of the early models of simulate to analyze.

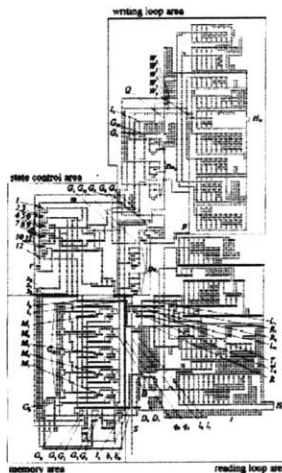


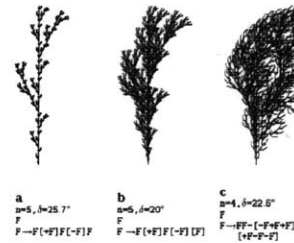
Diagram of Neumann's Universal Computer. Image Credits: Umberto Pesavento, “An Implementation of von Neumann's Self-Reproducing Machine,” *Artificial Life* 1995, Volume 2, Number 4: 337–354.

8 Turing, (1936) *Undecidable* p. 118; footnote Davis (2000), p.151.
9 Stephen Wolfram, *A New Kind of Science*, (Wolfram Media, 2002), 876.

Neumann's work inspired Nils Aall Barricelli who pioneered building genetic algorithms to simulate evolution, a more sophisticated model of reproduction. His publication "Solving multi-objective optimization problems using an artificial immune system," in 1957 is the earliest published record of an evolutionary simulation¹⁰.

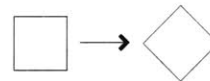
Around the same time period, Chomsky published his *Logical Structure of Linguistic Theory* offering a taxonomy of language structures based on a formalism of sequential string rewriting. In the early 1960s, programming methods made a major leap forward with the introduction of "Object Oriented Programming", a structure where "Objects" inherit properties of "Classes" within a "hierarchical" data structure. This served as the base component for major developments in software like: animation, parametric modeling, Building Information Models (BIM), etc.

In the late 1960s, Aristid Lindenmayer formulated L-Systems as a customized version of Chomsky's Grammars. These were created to simulate botanic growth behaviors and patterns. Unlike Chomsky's Grammars, L-systems were based on parallel string re-writing where all alphabets are rewritten at the same time.



Example showing L-system formalism and generated geometric interpretation.

In the early 1970s, George Stiny and James Gips¹¹ introduced *Shape Grammars* as a system to simulate visual calculation processes. It was the first design-oriented system. Shape Grammars were used in a many fields to reverse engineer existing design or devise methods for creating design languages, one of which was mentioned earlier in the Palladian Grammars.

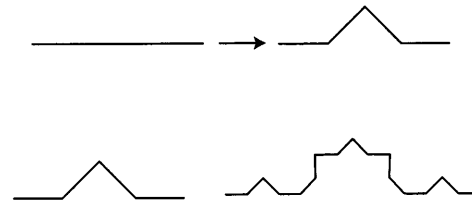


Example showing Shape Grammar Rules.

10 C.A. Coello Coello and N.C. Cortés, "Solving Multi-Objective Optimization Problems Using an Artificial Immune System," *Genetic Programming and Evolvable Machines* June 2005, vol. 6, no. 2: 163–190.

11 J. Gips and G. Stiny, ed., C. V. Freiman, "Shape Grammars and the Generative Specification of Painting and Sculpture," *Information Processing 71*. (Amsterdam, 1972), 1460–1465. -- Republished by O. R. Petrocelli (ed.) *The Best Computer Papers of 1971*, (Philadelphia: Auerbach, 1972), 125–135.

In 1975 Benoît B. Mandelbrot coined the term “Fractal” describing what was known since the early 1900s as Mathematical Monsters.¹² These were used to generate self-similar structures posing questions on dimensions and topology.



Example of visual representation of fractal formalism through replacement rules. This example shows the generation of a Koch Curve, (This will be explained later).

Most of these systems reappeared in architecture for form synthesis through the introduction of new architecture-oriented tools. Their appearance triggered a chain of new thoughts on theory, form, program, tectonics, culture, etc. Such conclusions can be drawn from the “Software development” and the “Maturity of Generative Systems” timelines. For example, it was after the introduction of computers when Eisenman said: “One can set up a series of rule structures for inputting into the computer not knowing a priori what the formal results will be. Then the process becomes one of testing algorithms against possible formal results. The writing and correcting of these algorithms becomes one of the tasks of design.”¹³ This was also the period when William J. Mitchell published *The Logic of Architecture* introducing the concepts of algorithms, evolution, grammars, and logic into architecture design process; when Greg Lynn published *Animate Form and theorized about Nurbs in relation to Deleuze*; when we started reading theories about forces carving programs, or scooping out forms, or deforming landscapes, etc.

At the beginning, architecture-oriented-sofwares served as documentation tools for they gave architects the ability to ensure accuracy and consistency in all drawings; expedite generation of “copies”; and help visualize possible options right in the monitor space without destroying drawing-sheets by erasing/drawing ink lines repetitively. These documentation tools witnessed a different type of implementation with the introduction of parametric features. Now, architects can “update” their drawings by changing parameters like: dimensions, relationships, formula, history, etc. This replaced the concept of “redraw” by that of “regenerate”.

Parallel to the development of parametric softwares was the creation of solid modelers. These added a third dimension to the architect’s digital drawing board. The development continued to include smooth modeling modules allowing architects to create complex forms never been conceived of before.

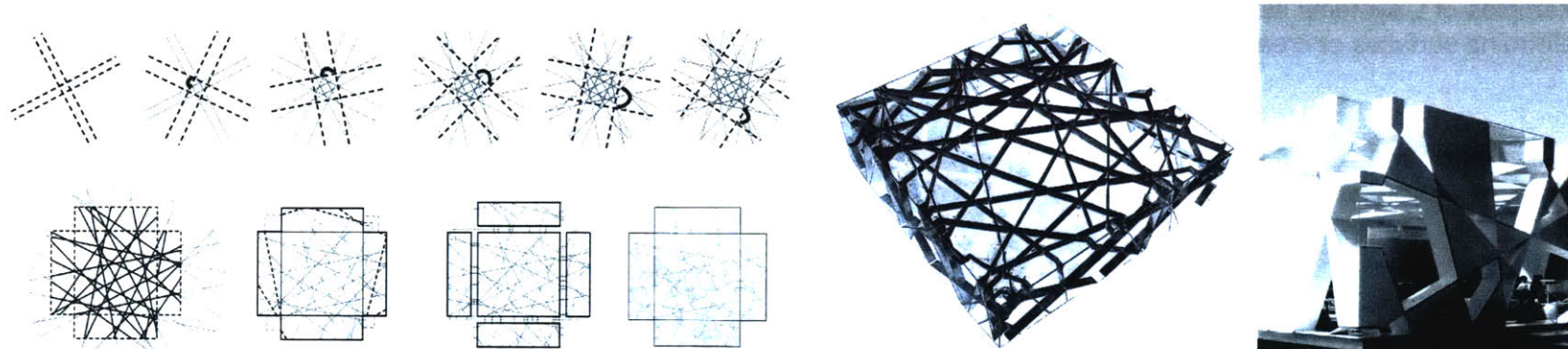
12 Mandelbrot, *Fractals and Chaos: The Mandelbrot Set and Beyond*, 1st edition, (Springer; 2004).

13 Selim Koder, “Interview with Peter Eisenman.” *Ars Electronica* 1992. / This can be found digitally at: http://www.jhfc.duke.edu/jenkins/Publications/Lenoir_FlowProcessFold.pdf

Later, a new type of parametric softwares came to break boundaries of static geometry. In the mid 1990s, Animation packages were offered by major software companies like: Discrete, Alias, and Maxon. These were channeled to serve the media and film industry; however, architects found them to be rich environments for making “provocative” forms. They were able to create morphing blobs, extract moments of deforming surfaces, or capture instances of moving geometries, etc. Parametrics softwares (modelers and animators) helped revive the concept of building systems to generate variations for design.

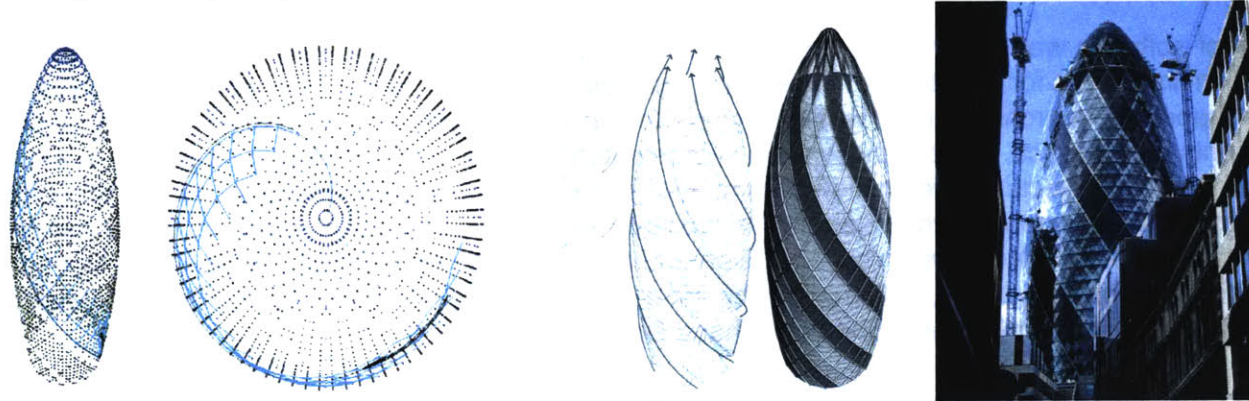
The development of software was continuous. New programming and scripting languages were introduced in the late 1990s. They offered enhanced environments for architects to write their own tools (to a certain level). This was the period when generative systems from computing reappeared in architecture. It was finally possible to write algorithms on top of an existing platform instead of building specific environments from scratch. Most of these were implemented to solve confined design problems, or slightly described scopes. In both cases, the resulting objects were shallow if compared to the complexity of architecture. In the following pages, I will present six different implementations of generative systems in architecture addressing one design problem, that is buildings’ envelopes. These implementations include: Serpentine Pavilion by Cecil Balmond and Toyo Itto (Algorithmic), Swiss Re by Norman Foster (Parametric), Some academic Projects on L-systems, Automaton by Mike Silvers’ (Cellular Automata), Federation-Square by Lab Architecture (Fractals), and EMP skin solution by Dennis Sheldon (Shape Grammars).

Cecil Balmond and Toyo Itto designed the Serpentine Pavilion by implementing an algorithmic system. In a square shape, it draws a line from the mid point of a segment to the first third of its adjacent iteratively. After a certain number of iterations it will extend all the lines sufficiently to cover the surface area of a cube (roof and walls). Then, it breaks, bends and trims the extended lines at the edges of the cube. It finally adds links between the edges and corners. In the construction phase, the architects applied a checkerboard-coloring scheme and custom built connections.



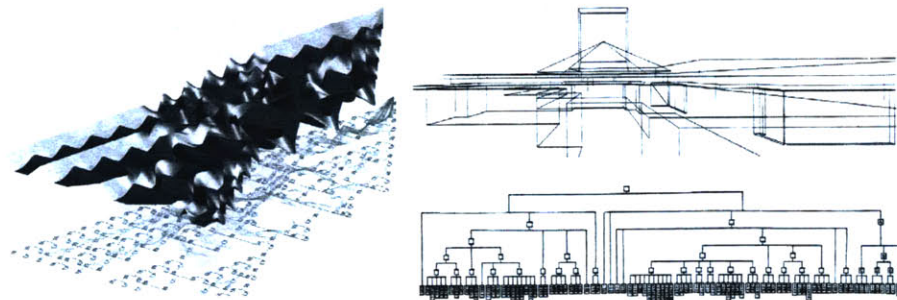
Diagrams showing algorithmic procedure of generating pattern. Left: Michael Kubo and Farshid Moussavi, “The Function Ornament,” [Actar](#) 2006. Middle and right: Image from Jaime Salazar et al., “Verb Matters”, [Actar](#) 2004.

Forster partners took that algorithmic design approach to a second level in the Swiss re project. They used parametric systems where a constant (live) link exists between hierarchies of geometries and numbers. Their system was determined by parametric modeling software that interpreted numerical data generated by spreadsheet software. The separate representations (numerical and geometric) delivered an easier, and unusually accurate design process. As such, the parametric model becomes constantly responsive to change, offering a degree of design flexibility not previously available.



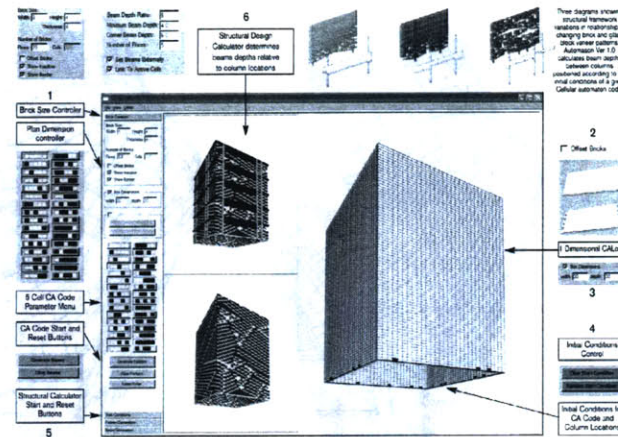
Showing driving geometry (points) generated by Spread sheet. Right: Image from Jaime Salazar et al., "Verb Matters", Actar 2004. Left: Image from Michael Kubo and Farshid Moussavi, "The Function Ornament," Actar 2006.

Examples of L-systems in architecture are only present in Academia. Most of the work published does not go beyond explorations in deforming surfaces or creating branching geometries.



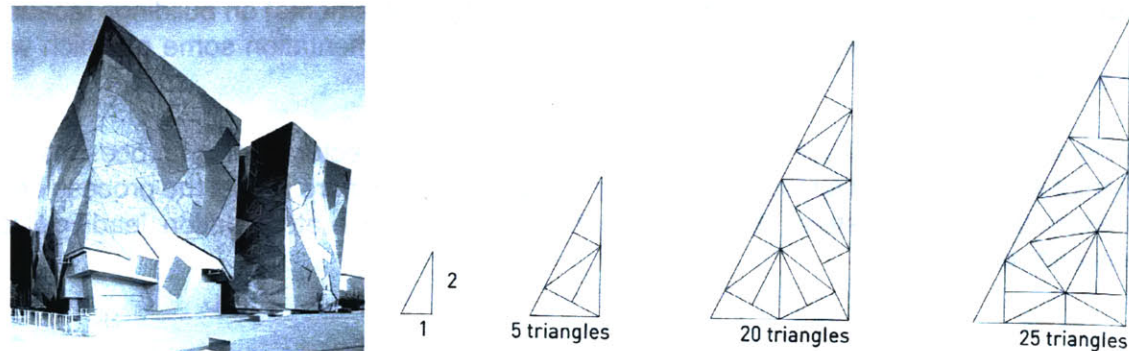
Left: Example on deforming Surface by displacing control points based on an L-system generation, Image Credits: Mike Silver, Programming Cultures: Architecture, Art and Science in the Age of Software Development, Academy Press, 2006.

One of the developed examples on using cellular automata for design is a software created by Mike Silver. In an *automasonry*, Cellular automata patterns become translated as brick formations using a push-pull algorithm. Again, the implementation of generative systems did not go beyond applying patterns to buildings' facades.



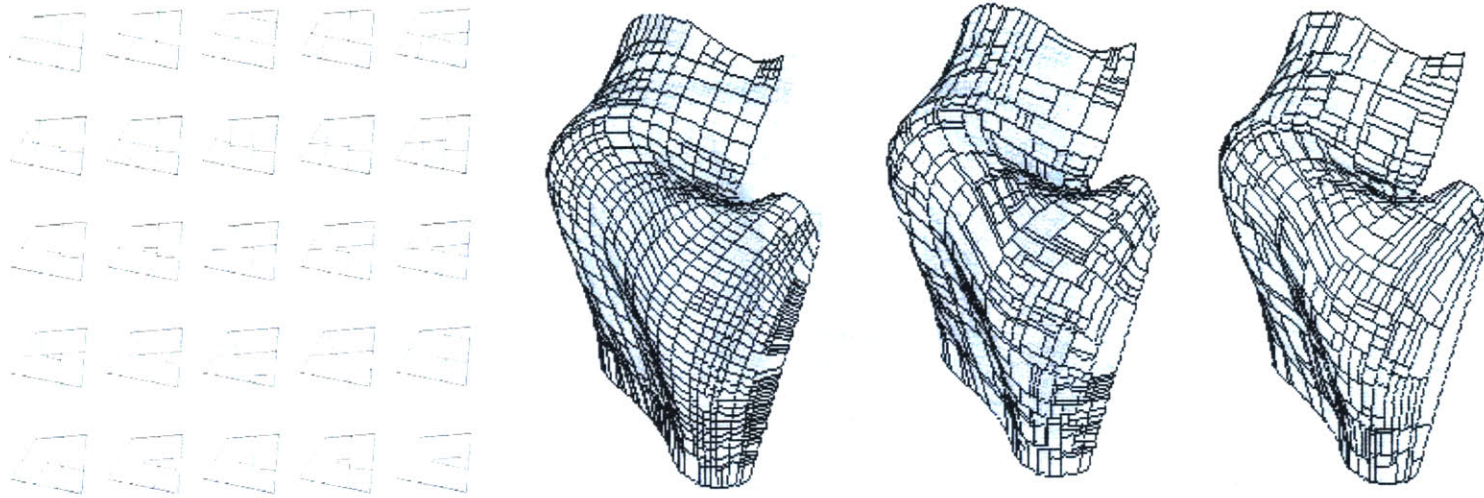
Screen shot of Automason software showing various parameters for creating variations of brick patterns. Image Credits: Mike Silver, Programming Cultures: Architecture, Art and Science in the Age of Software Development, Academy Press, 2006.

Designed by lab Architecture through a fractal system, the façade of Federation-square building complex included a triangular pinwheel grid hosting a five-tile pattern arranged differently in five panels. In addition, a collection of materials was applied to certain types of panels suggesting a visual complexity.



Left & Middle: Image of Federation Square-Façade. Right: Panels of five triangles. Image Credits: left, <http://www.labarchitecture.com>. Right: Michael Kubo and Farshid Moussavi, "The Function Ornament," Actar 2006.

The skin of Gehry's EMP building in Seattle was designed by Dennis Sheldon who implemented a grammar to transform, and break regular rectangular metal sheets in several shapes. The intent of this implementation was to arrive at a variety of panel shapes to hide the regularity of the usual grid systems used for constructing facades.



Experience Music Project Skin design with shape grammars rules. Image Credits: Mike Silver, Programming Cultures: Architecture, Art and Science in the Age of Software Development, Academy Press, 2006.

In the previous examples, design was more of an outcome than a process. Architects took what generative systems gave them at face value. They only mapped systems behaviors as geometric patterns (or rather ornaments) on buildings facades, what Adolf Loos might have called a "crime". There are many reasons behind such a skin-deep implementation some of which will unfold through out the following chapters of this thesis.

One might ask, what is the relevance or value for implementing generative in architectural design processes? Akin described the design process as a combination of knowledge and strategy. The richer a strategy is, the more robust the process becomes¹⁴. There are many benefits in every fold of every generative system. These include the ability to A) generate variation, leading to emergence of new forms, B) unpack dependencies and relationships.

14 Krishnamurti, Ramesh. Artificial Intelligence for Engineering Design, Analysis and Manufacturing, 20, (Cambridge University Press, 2006), 95ñ103.

In architecture studios (both in academia and practice), designers usually decide and select by comparison. For example, an architect may go with solution A as it satisfies program requirements, or solution B as it expresses “design relationships” more clearly, or solution C because it satisfies the building code, etc. Comparison can be built on A) heuristics for how things should work; B) numerical evaluation methods for quantities like area, cost, loads, etc; C) aesthetic judgment. Neither do these three evaluation methods present themselves in a hierarchical order, nor do we follow one way to prioritize them. Such uncertainty in synthesis and analysis of design solutions depends dramatically on the amount of available design candidates from which to compare and select.

One of the main incentives in exploring new design processes is “emergence” of new forms. We have seen that architects embraced softwares as they continued to offer new tools, methods, and scripting platforms, etc. Architects sought emerging shapes in automating transformations, embedding randomization functions, layering various parameters, etc.

The benefit of implementing generative systems in design reaches beyond variation. In his diagrams, Eisenman demonstrated consistency in decomposing relationships, clarity of design thinking and mastery of unpacking dependencies of design elements. Systematic design approaches help in devising decision-making strategies, prioritizing requirements and clarifying distinctions between qualities and quantities. If implemented correctly, they could push the design process further.

In this introduction I surveyed the current status quo of implementing generative systems in architecture showing their early origins, current and evolving approaches in design, and their relevance for architects. The following chapter includes discussions and highlights on six generative systems: Algorithmic, Parametric, L-systems, Cellular automata, Fractals and Shape grammar

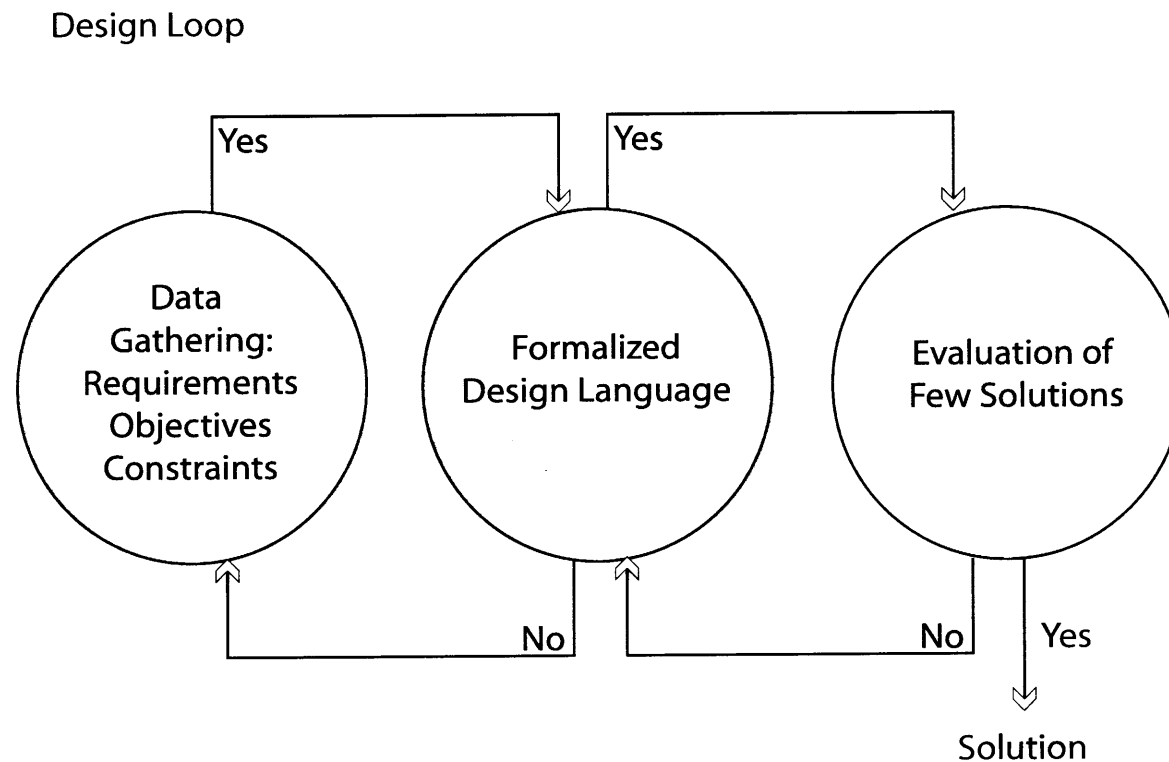
2.0 Generative Systems In Design:

This chapter is broken to two sections. The first introduces general ideas about integrating generative systems in design processes. The second offers highlights on intrinsic and shared properties between six systems. These include: Algorithmic, Parametrics, L-systems, Cellular Automata, Fractals and Shape grammars.

2.1 Designing with Generative Systems:

Herbert, Lionel March, Yahuda E. Kalay, and many others, discussed the concept of (generate-test) design loops. They defined design as a result composed by two engines, one is involved with generation and the other is involved with evaluation.

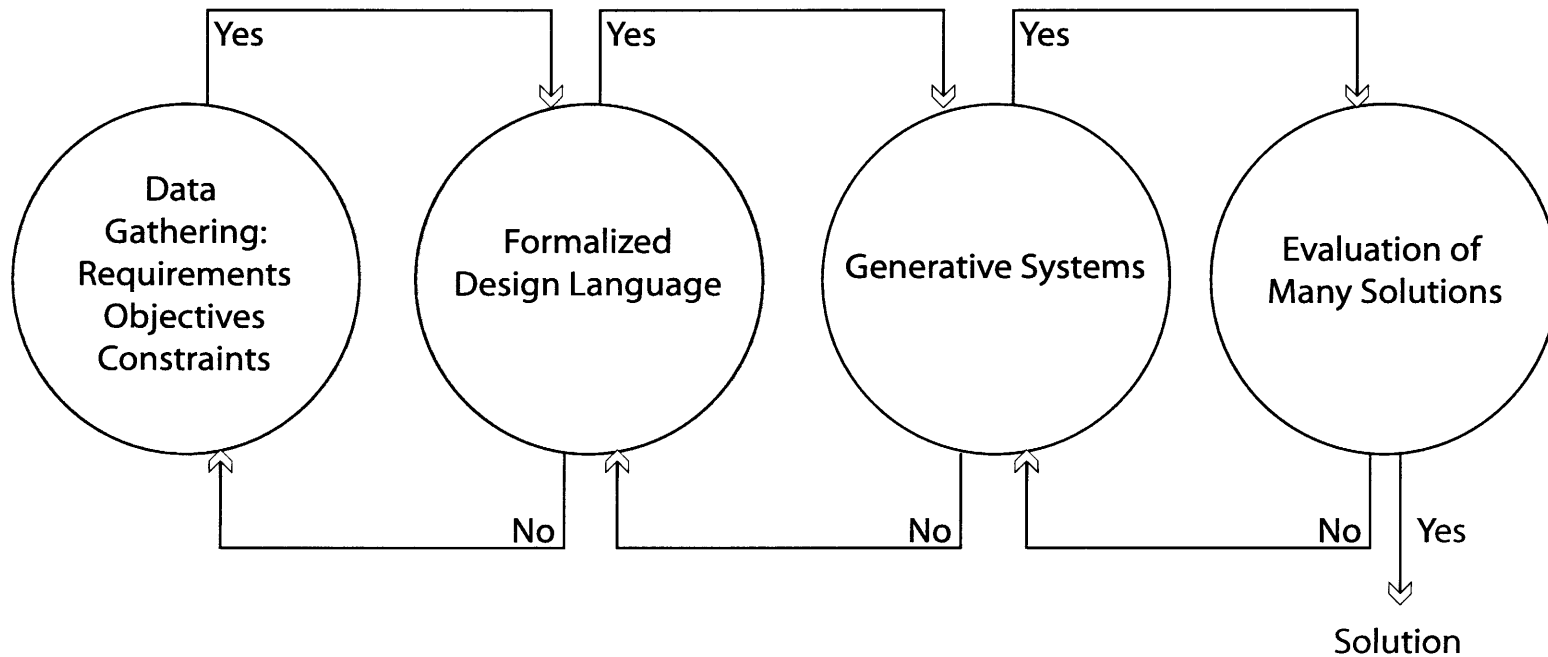
In architecture studios, we typically collect data and investigate sites, build a concept around which we structure a design, and then analyze possibilities based on an array of criteria we define or receive from clients. This process can be loosely illustrated as shown below in a three-node diagram. One of the limitations in this process appears in the number of solutions that the “design language” node can generate. It is usually very few, if not one.



History is a great resource. You can understand how phenomena mature or decay, last or end, continue or break. In the first chapter of this thesis, we were able to identify the points in time when architects like Durans, Sullivan, Le Corbusier and others prescribed their processes. If you noticed, all of them externalized their design processes after building a body of work, defining a certain style, working by a set of architectural elements for certain goals. As for computing, we also found how L-systems or Cellular automata and others were created for simulation. It was after phenomena were broken down to units, relationships and behaviors. Generative systems can only be built after defining design objectives, processes and relationships.

The following illustration shows a revised possible diagram for integrating generative systems in design. A fourth node, “generation”, is placed between concept and evaluation. The main gain behind integrating such systems within a process is the ability to test “many” generated solutions and be able to compare between them. This allows for capturing more possible design solutions for every conceptual design language. It is important to realize generative systems are context specific for they come after a formalized (defined) design language.

Design Loop



2.2 Highlights On Selected Generative Systems:

Just like in the introduction of the first chapter, I would like to list a number of keywords to help define the scope of the following sections. In Oxford dictionary:

- Structure: "...the arrangement of and relations between the parts or elements of..."
- Algorithms: "...a process or set of rules to be followed in calculations or other problem-solving operations..."
- Component: "...a part or element of a larger whole..."
- Rules: "...one of a set of explicit or understood regulations or principles governing conduct within a particular activity..."
- Formalize: "...give (something) a definite structure or shape..."
- Formalism: "...concern or excessive concern with form and technique..."
- Representation: "...the description or portrayal of someone or something in a particular way or as being of a certain nature..."
- Associate: "...connect (someone or something) with something else..."
- Hierarchy: "...a system or organization in which people or groups are ranked one above the other according to status or authority..."
- Inherit: "...derive (a quality, characteristic, or predisposition) genetically from one's parents or ancestors ..."
- Inheritance: "...A thing that is inherited ..."
- Parallel: "...Computing involving the simultaneous performance of operations..."
- Sequential: "...Computing performed or used in sequence..."
- Random: "...made, done, happening, or chosen without method or conscious decision..."

Let's define Generative systems as structures capable of producing many results. These systems are composed of Algorithms that describe parallel or sequential or random processes. Let's also define formalisms as a special type of a generative system formalized by rules. These systems and formalisms may display a specific representation.

A generative system might include one or bi-directional associations (relationships). One-directional associations generate hierarchical structures where elements are placed within ranks. Usually known as a Parent-Child relationship. Within such a structure, inheritance becomes possible for properties reappear in many generations within a family of elements.

In the following six sections, I will try to highlight major concepts that I find relevant to the scope of this thesis. Each section will introduce one generative system including background information, experimental implementations within design, and a short summary.

It is important to not that the experiments I am showing for each system were also built to address similar design problems to those shown in the first chapter (building envelopes). Each experiment is structured to work for synthesis with almost no evaluation. The reason I chose such a defined context to design for was to strip each system down to its bones and focus the discussion on its structure.

In the following sections I will introduce six generative systems: Algorithmic, Parametric, L-systems, Cellular Automata, Fractals, and Shape grammars. Each section is composed of 11 segments. These include:

- 1.0 History and Background: offering information on when, how and why each system was created.
- 2.0 System: introducing main ideas and concepts.
- 3.0 Components: discussing how each system might be structured/ utilized within a design context.
- 4.0 Units: showing the units' types that a system can recognize and deal with.
- 5.0 Inheritance, Constraints and associativity: commenting on some prosperities of each system.
- 6.0 Mechanics: showing how each system works, its behavior or rules.
- 7.0 Representation: discussing how each design component and units are represented.
- 8.0 Solution space: showing a way to evaluate how generative a system is.
- 9.0 Experiments: showing experiments utilizing each system within a confined context (building envelope).
- 10.0 Summary: offering a brief description of each system.
- 11.0 References

Algorithmic Generative Systems

Introduction:

Algorithmic systems are the basic components in all generative system. They are the most malleable when it comes to customization because they don't impose a specific structure, or relationship, or representation, or units' type or context. They only provide a working environment as opposed to recipes to implement. For this reason, these systems are the most popular of all systems among architects. In fact, designing with algorithms is not a totally new concept in architecture. As you have seen in the introduction, many architects repackaged their design languages in algorithmic descriptions for others to implement. Thinking in terms of algorithms is a mapping process of design objectives onto step-by-step descriptions. Such a process helps designers decompose context, understand relationships and devise methods to judge the utility of the outcome.

1.0 History and Background

2.0 System

3.0 Components

4.0 Units

5.0 Inheritance, Constraints and associativity

6.0 Mechanics

7.0 Representation

8.0 Solution space

9.0 Experiments

10.0 Summary

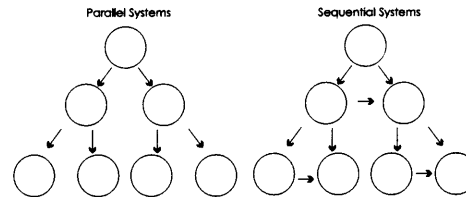
11.0 References

1.0 History and Background:

The word Algorithm is an evolved transliteration of the Muslim scientist's family name: Al-Khwarizmi. Algorithms were known as descriptions for arithmetic manipulations of Arabic number.¹ The modern view of algorithms extends this definition to include process. "Algorithms are prescriptions of computational processes that lead to desired results", says Andrei Markov.² Descriptions of processes require decomposing them to steps and relationships. They need to be definite and precise not to leave place to arbitrariness. Algorithms vary in their structure, logic, representation, etc. Literature on algorithms is deep enough for any scientist to drown in. For the purpose of this thesis, I will go with the most basic definition, which is: a set of instructions to achieve a certain goal.

2.0 System:

Boolos and Jeffrey beautifully framed the concept of using algorithms in the *Computability and Logic* (1974, 1999): "No human being can write fast enough, or long enough, or small enough to list all members of an enumerable infinite set by writing out their names, one after another, in some notation. But humans can do something equally useful, in the case of certain enumerable infinite sets: They can give explicit instructions for determining the *n*th member of the set, for arbitrary finite *n*. Such instructions are to be given quite explicitly, in a form in which they could be followed by a computing machine, or by a human who is capable of carrying out only very elementary operations on symbols" (boldface added).³ This concept lends itself smoothly to our context: architectural design. Constructing such a dual environment of human and machine can only work through the creation of a common language that both understand. One that translates, or rather simulates, the human thinking process into a language that machines can work with. Such a system can spark unmatched synergy where the creation of ideas exceeds the limitations of human immediate perceptions.⁴ Algorithmic systems can be constructed or networked in different ways. They can run in parallel where they process different sets of information at the same time without affecting one another. They may also run sequentially such that a predecessor affects its successor. They may also run in random without following any specific structure.



1 <http://www.islamonline.com/cgi-bin/news_service/profile_story.asp?service_id=755> , <http://www.muslimheritage.com/day_life/default.cfm?ArticleID=317&Oldpage=1>

2 A. A Markov, *Theory of Algorithms*, (Moscow: Academy of Sciences of the USSR, 1954). Original title: Teoriya Algorifmov. Referenced from: /, <http://en.wikipedia.org/wiki/Algorithms/>..

3 George Boolos and Richard Jeffrey, *Computability and Logic*, First Edition, (London: Cambridge University Press, 1999) 19.

4 Kostas Terzidis, *Algorithmic Architecture* (Architectural Press, June 21, 2006), Chapter 3.

3.0 Components:

Building an algorithmic system for design will require mapping of design intentions onto a series of discrete steps and units. They may include descriptions of quantitative properties like required area, or number of units, geometric descriptions of objects, etc. For example, designers may be able to evaluate how “well-lit” a space is based on percentage of openings in a wall, or based on the relative location of these openings to a center of interest, or materials reflections, etc. Designers may also choose to formalize qualitative properties like: “open”, “wide”, or “stylish”, etc. This type of mapping is usually harder to describe for it requires more details, and possibly different types of evaluations. One might encode a quality by defining a certain geometric language, along with some materials attributes to ensure the generation of a certain style. For example, a designer who appreciates open modern plan with organic layout of walls would write an algorithm to A) push structural elements to the building envelop, B) place free form splines, C) fillet walls edges, D) create thick slabs to eliminate beams, E) apply glass materials to certain façade elements, etc. In any case, an algorithmic system for design will require components to generate data, others to interpret them as output of various types: geometric descriptions, material properties, etc. Usually another third component is needed to evaluate existing output based on performance-based or aesthetic-based criteria. Evaluating architecture is no easy task for design is usually ill defined, proprieties change by the minute based on clients, site conditions, structural requirements, etc. A fully automated algorithm is very complex to achieve and the results are always questionable. A human controlled process is not as fast, accurate or robust. The complexity of mapping processes and translations of goals, intentions, performance criteria between human and machine is very new to architectural profession. All of the above imposes many questions on the implementation of algorithms in architecture design.

4.0 Units:

Algorithms can deal with any types of units, ones with fixed, or flexible definitions. In Computers, an algorithm can only deal with discrete unit, one with fixed descriptions or identity, or boundaries. These units can be numbers, alphabets, geometric elements, etc. An Algorithm deals with units through functions. A function is an expression of dependencies (equations) between units through operators. Operators include: Additions, subtractions, etc. For example: $A = 3 + X$. Collections of functions lead to developing techniques for achieving certain tasks. These techniques are known as methods. Related methods can be grouped as a class responsible for generating certain types units. In this context, an algorithm is a collection of methods.⁵ In algorithms, units might be fixed, or changeable. They also must be stored in placeholders. A placeholder for fixed units is called a constant, for changeable data is called a variable. A placeholder for variables or constants is a Parameter. Here is an example of an algorithm described verbally: Create a cube that has a red color, move a number N of copies distance X towards the North direction, rotate each cube around its Y axis A degrees, change its color randomly. “Create”, “Move a copy”, “Rotate” and “Change colors” are functions. Distance X , Angle A , and number of copies N are

5

Kostas Terzidis, Algorithmic Architecture (Architectural Press, June 21, 2006), Chapter 3.

variables that can be changed any time. “Color”, “North Direction”, and “Y-axis” are parameters that channel variable units “Random Color”, “Number of Copies” & “Distance X”, and “Angle A” to the functions “Move a copy”, “Rotate” and “Change Color”.

5.0 Inheritance, Constraints and associativity:

The term “Inheritance” in algorithms describes one directional relationship that is possible through hierarchies, usually known as Parents and children where the latter inherits properties of the former. Inheritance appears when properties flow from one parent to its children. Such propagation happens through updating variable data among children sharing the same set of parameters. Constraints are viewed as limitations, or ranges by which a value is allowed to vary. If were implemented incorrectly, constraints may eliminate potential solutions or hinder the generation process. Associativity is a set of relationships between elements. It is usually bi directional like: $X = 2 Y$. In this case, changing the value of X will update Y and Vice Versa. Associations allow for triggering a series of changes by only changing a few parameters. This is true for any number of elements involved in a relationship with the same changing set of parameters. For example, given a set of relationships: $Z = 2 A$, $X = 3Z$, $Y = X+Z$, and $C= 2X-Z$. If you change the value of A, a change of changes will ripple through the system in the following order: Z, then X, then Y and C at the same time. This is because the values of Y and C rely on X and Z, and the value of X relies on Z. An Algorithm cannot update any element before identifying the values of all of its parameters.

6.0 Mechanics:

An algorithm can be thought of as a description for actions within a certain range of possibilities. These actions can be context-free or context-sensitive. The latter is usually known as a rule-based algorithm. A rule is best explained as a condition that should be matched before task is triggered. A condition is a relationship between two entities, or an object property, etc. This might be expressed as: If “condition” is “true”, then trigger function “do this”. Rules types are usually inferred from the algorithms they trigger. For example, you can write a “replacement” rule, or a “transformation” rule, or a “deformation” rule, etc. Replacement algorithms perform replacement operations on objects or parts of them (This will be explained in the 2.2.6). A simplistic descriptions for this process can be: Pick object A, and replace it by object B, or Pick a part of object A and replace it by a part of object B, or pick Object C and replace it by nothing, etc. Rule based systems tackled in this thesis include: L-systems, Cellular automata, Fractals, Shape Grammars.

7.0 Representation:

Algorithms can be described or represented in many forms like: Pseudo code, Graphics and symbols, verbal descriptions or in any combination. The way an algorithm is built will dictate the representation method. Generation and Interpretation may be combined within a representation or be dealt with separately. For example, two algorithms may be used to create a bar chart. One generates a

series of numbers. The other interprets these numbers as graphics. In this case, each algorithm will require a separate representation since interpreting (building) graphics cannot start before generating numbers. The opposite example is the generation of a geometric pattern using a shape grammar. In this case, an algorithm can be written to pick an object and replace it by another at once. In shape grammars, representations combine both generation and interpretation. This will be explained in more detail later on. Algorithms concerned with evaluation are always separate. They cannot be merged within representations of other algorithms for they can be only applied “after” a result is being generated or interpreted.

Representations dictate the type of information being expressed. For example, a cellular automata representation is very powerful when it comes to describing neighborhoods conditions, but not geometry. On the other side, Shape grammars are capable of handling complex transformations and geometric descriptions, but not describing neighborhoods' conditions.

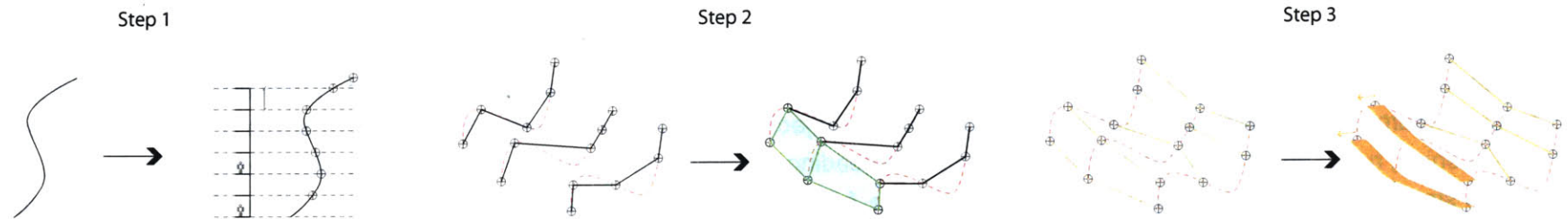
8.0 Solution space:

Solution space is a term used to gage the quantity of possible solutions within a certain design system. It indicates how generative a design system is. The ability to generate in an algorithm is subject to the number of parameters, allowed range of variables, types of relationships, rules, and number of iterations. It will be misleading to pose a more specific recipe behind algorithms' ability to generate variety since there is no strict structure for them. In the following sections on Parametrics, L-systems, CAs, Fractals and shape grammars, I will explain in more detail how each can generate such variety.

9.0 Experiments:

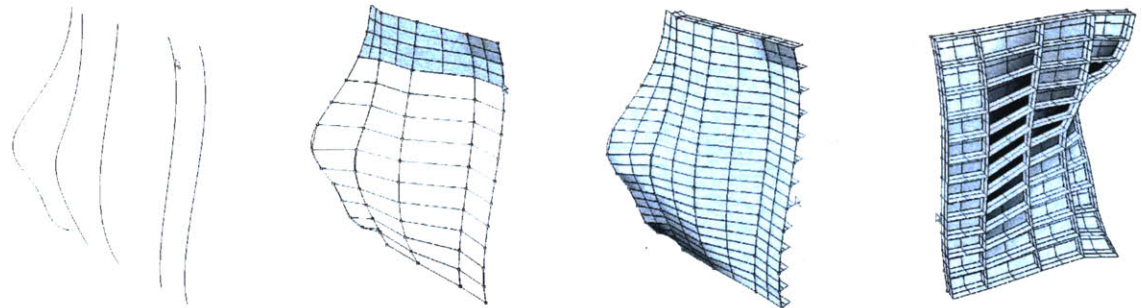
9.1 Skin and Bones:

This experiment included a generator engine only. It is meant as an example to demonstrate how a basic algorithm can run deterministically without rules. It also illustrates a simple algorithm by which certain tasks in architectural design processes can be automated. The algorithm constructs a surface with ribs from a series of curves in three steps. It starts by dividing selected curves into segments of various sizes such that the projection of these divisions reflects a user-defined slab's height. Then it draws straight-line segments between every two points on each curve by which it creates patches. Then it draws a poly-line passing through point that are at the same level in all curves and extrude that as a rib. Below are images showing the steps and results.



Make Skins + Bones:

- ◇ Divide curves in relation to a user-defined slab-height
- ◇ Generate straight-line segments between every two points on each curve (to rationalize the geometry) and construct a patch
- ◇ Generate a poly-line segments passing through all points in all curves at each level and create a rib.



9.2 Vertical Village Concept:

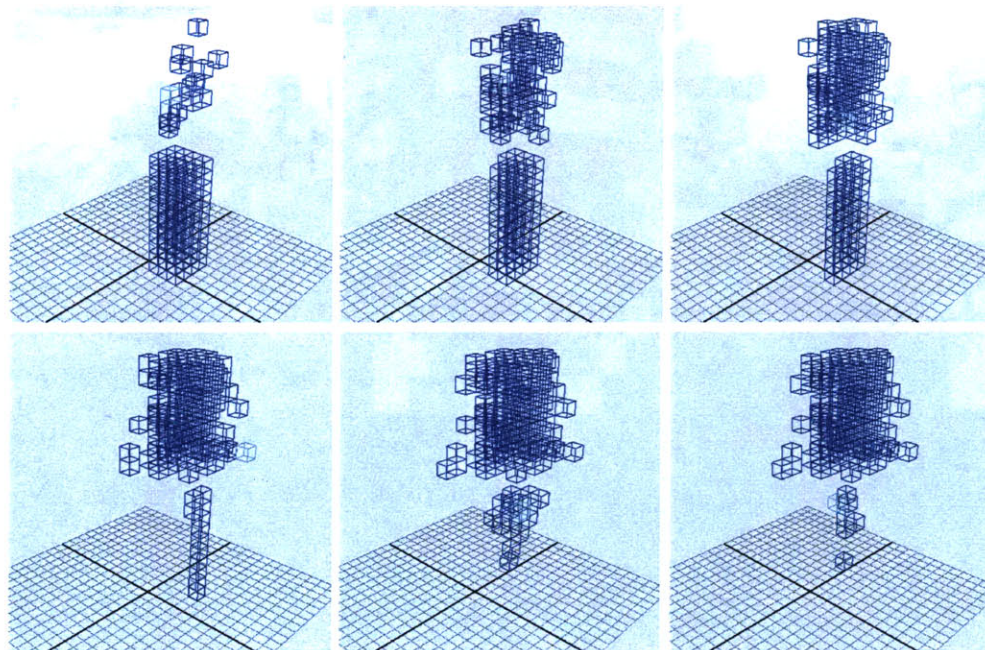
In this experiment design was implemented by utilizing two engines: a generator and an evaluator. The first is responsible for creating units where the other deals with evaluating solutions based on number of basic real-estate concepts.

The goal was to revisit the concept of a high-rise building in two folds, spatial zoning and exterior envelop. The intention was to drive the latter by the former. The concept for spatial zoning revolved around mixing a typical architectural program by a) dislocating units horizontally and vertically starting from a typical floor plate and b) shuffling various programs in different configurations. This generated an interesting dialogue between inner void spaces and solid units, subtracted corners and stepped facades, public and private, commercial and residential spaces, etc.

The algorithm was able to generate various dialects of such a design language depending on parameters like: number of units, number of iterations, and percentage of public commercial Vs residential spaces. Selection of design candidates was guided by evaluating the level of exposure, types of space as suggestion to integrating real-estate values within the generative system. Below is a description of the algorithm and screen shots of the generation process.

Make Vertical Village:

- ◇ Define a number of units and iterations.
- ◇ Move each unit in 3D space once per iteration then:
 - ‡ Register units' heights and numbers of exposed faces for all units.
 - ‡ Store values
- ◇ Repeat B based on the number of iterations defined in A.
- ◇ Select Best Result



| System Control | Generation | Evaluators

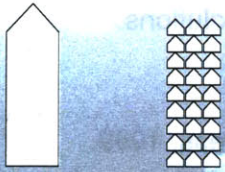
Concept For Vertical Village

What if urban fabric was weaved vertically? What if program interlaced within a confined area and generated mixed public/ private; living/ shopping; and work/ leisure environments? In this experiment, we developed an algorithm to shift, rearrange and orient solids in relation to view, number of shared edges, and amount of voids. The generated data informed various formations of solid-void conditions allowing for a variety of inhabitable environments



Parameters

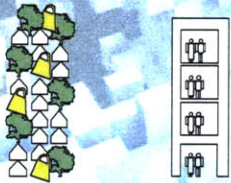
Neighborhood



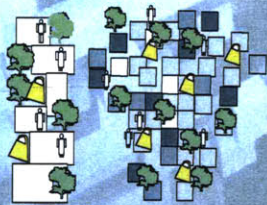
Variety



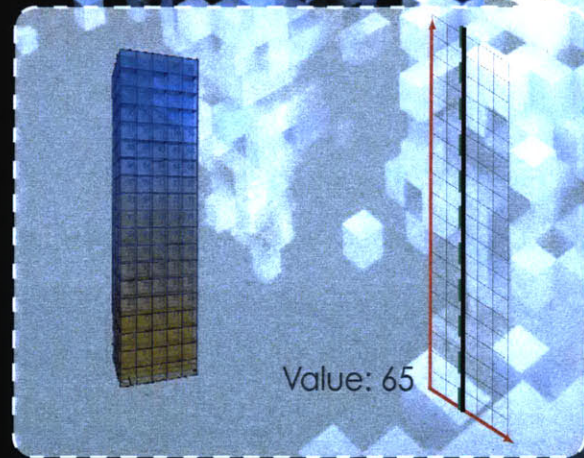
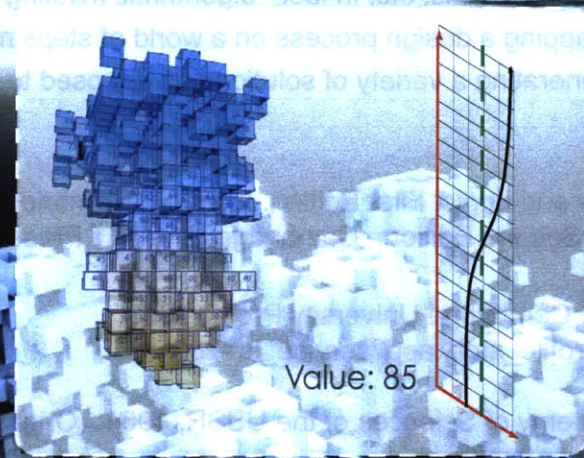
Variation



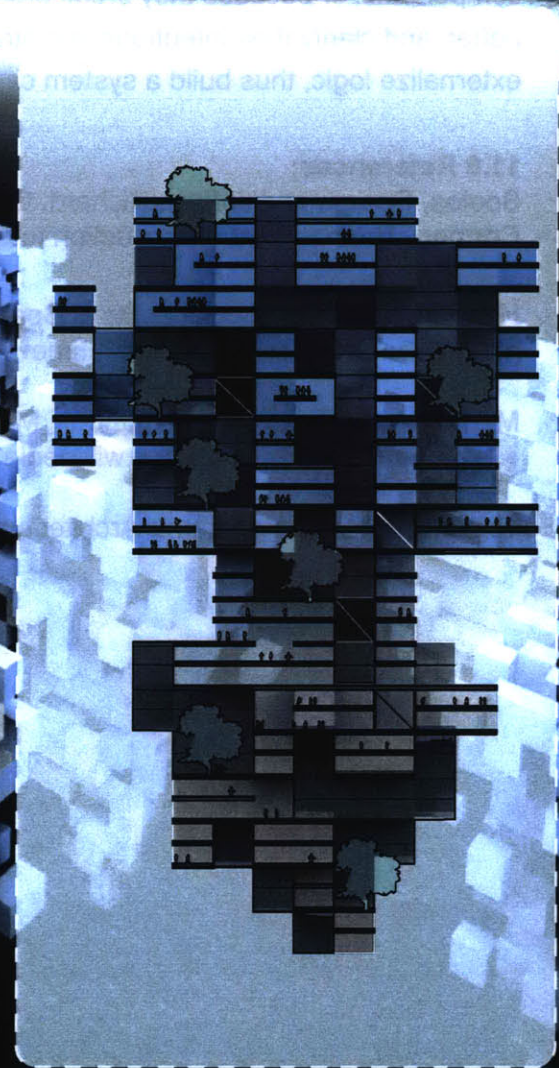
Program Possibilities



Evaluation based on number of voids



Possible Zoning



10.0 Summary:

Algorithmic systems resemble the core of all generative systems. They package a series of tasks in various structures, representations or rules. The use of algorithms in design processes might be mistaken as a limitation for they can only deal with explicitly defined components, or because they break process into discrete tasks, etc. In fact, “algorithmic thinking” can help designers define their goals better, and clarify their intentions and strategies. Mapping a design process on a world of steps might not be easy or direct, but it helps externalize logic, thus build a system capable of generating a variety of solutions as opposed to one or a few solutions.

11.0 References:

Boolos, George and Jeffrey, Richard. *Computability and Logic*, First Edition. London: Cambridge University Press, 1999.
Cormen, Thomas et al. *Introduction to Algorithms*, Second Edition. Cambridge: The MIT Press, 2001.

Hillier William. *The Social Logic of Space*. London: Cambridge University Press, 1990.
http://www.islamonline.com/cgi-bin/news_service/profile_story.asp?service_id=755

Markov, A. A. *Theory of Algorithms*. Moscow: Academy of Sciences of the USSR, 1954. (Original title: Teoriya Algorifmov).
Referenced from: <<http://en.wikipedia.org/wiki/Algorithms>>.

Terzidis, Kostas. *Algorithmic Architecture*. Architectural Press, 2006.

Parametrics Systems

Introduction:

In the previous section I tried to highlight a number of concepts about Algorithmic systems, mainly is the fact they don't follow a certain structure. Algorithmic systems are the basic component on top of which other systems are build. In this section, I will introduce a specific case of an algorithmic system: Parametrics. These are built around two concepts: *associativity* and/or *Inheritance by hierarchy*. Parametric systems in architecture are usually understood in the context of: A) Geometric Modelers B) Animation packagers. Modelers like CATIA, Generative Components, or Solid works can create geometry, structure data within hierarchies and create dependencies through relationships. Animation packages like 3D Max, Cinema-4D and Maya offer a different type of parametric systems where designers can relate elements to each other through dynamics and inverse kinematics solvers. The fact is any system capable of associating elements with one another is a parametric one.

1.0 History and Background

2.0 System

3.0 Components

4.0 Units

5.0 Inheritance, Constraints and associativity

6.0 Mechanics

7.0 Representation

8.0 Solution space

9.0 Experiments

10.0 Summary

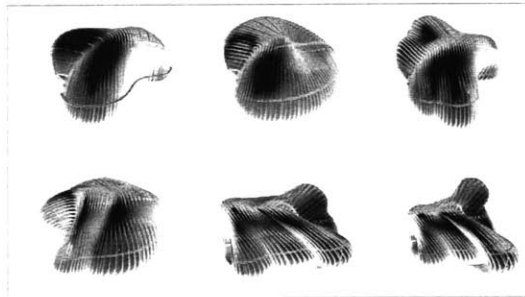
11.0 References

1.0 Brief History and background:

Parametric systems were created to accommodate for variations in the manufacturing industry. Mechanical engineers were always after optimizing parts based on performance analyses. The available processes required them to redraw their designs from scratch every time a design changes no matter how minor the changes are. Hence was born the concept of “regenerating” as opposed to “redrawing”.

W. Orchard-Hays, S.I. Gass, and G.B. Manne, among others, introduced the concept of Parametrics to the field of computer science in early 1950. They investigated the case when the right hand side (result) changes in a linear program.¹ In 1957, Dr. Patrick J. Hanratty created *PRONTO*² as the first commercial software to provide parametric algorithms for translating data from computers to manufacturing machines. In the early 1960s, Sutherland created the first graphical representation of Parametrics for the design profession in *Sketchpad*. In 1978, the term “feature” was presented in a bachelor degree thesis, “Part Representation Based on Feature in CAD (Computer Aided Design) System”. A feature is an object attribute achieved by applying Boolean operations and Euclidean transformations. This attribute is driven by relationships instead of numbers. Thus a change in object dimensions will not destroy or require rebuilding the same features. The introduction of feature’s laid the foundations for current parametric design modelers.³

In the late 1990s was the introduction of a different type of Parametrics, animation softwares. These softwares could move, deform, and change objects properties based through time by manipulating relationships, constraints, dimensions, etc. This was a period when architects started to use software for more than documenting their work, but rather, to explore forms with. Some of the most famous examples were a series of projects by Greg Lynn who proposed theories on mutating programs, deforming shapes, and capturing transformations.



Examples from Greg Lynn work with animation softwares.

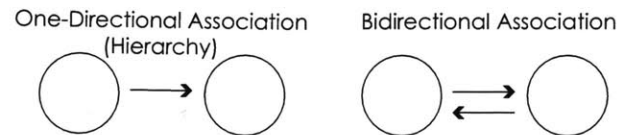
1 T. Gal, “A Note on the History of Parametric Programming,” *The Journal of the Operational Research Society* Feb, 1983, Vol. 34, No. 2: 162-163.

2 <http://mbinfo.mbdesign.net/CAD1960.htm>

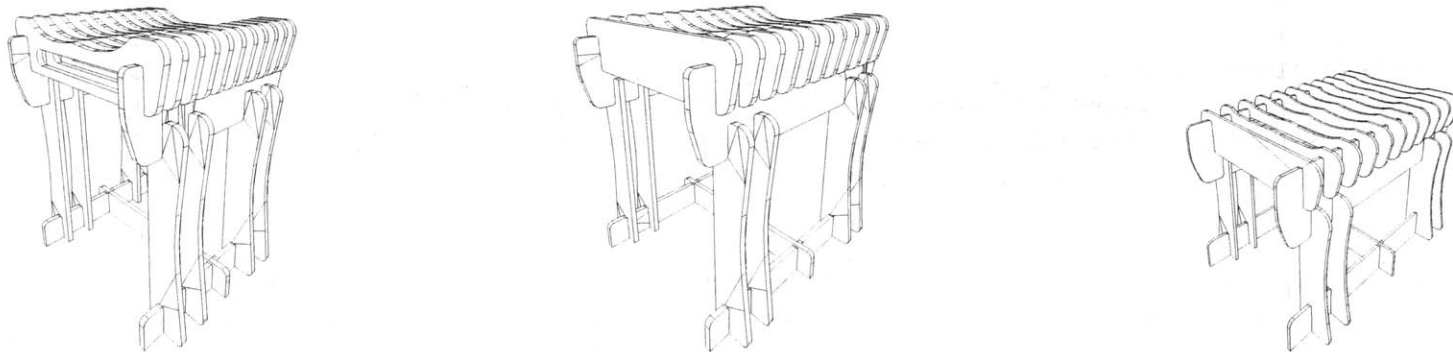
3 Robin Saitz, “Electronic Design Automation,” *MCAD Magazine* 15 December 2005.

2.0 System:

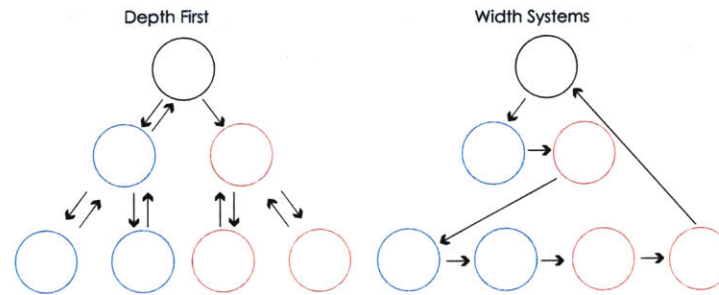
The very essence of Parametrics is the concept of associativity where objects' properties stem from relationships and/or inheritance. There are two types of associations, one directional and bi-directional. The first allows for data to flow from top to bottom only, what is known as Parent-Child relationship. The second allows for data to flow both ways, which is similar to the mathematical view of associations where the change of one side will update the other. Parametric systems with one-directional (Parent-child) relationships are known as Hierarchical systems for they force a certain ranking into the organization of elements within the system. Hierarchies allows for inheritance for every child receives properties of its parents by definition. This allows designers to create families of elements where change in parents (values) will ripple through the children.



Below is an example of a stool design created in a parametric environment (CATIA). The geometry is driven by a series of parameters within hierarchies. For example, wood thickness will drive the tolerance value, spacing of the seating-ribs, and depth of all wooden members; the height of the stool legs will drive their width and locations of the nudges. In this case, the system is controlled by only two main parameters: Thickness and height. Changing one of the other parameters will only affect their subordinates. For example, changing tolerance will update spacing for the seating ribs, but not the thickness of wooden sheets.



In hierarchical-parametric systems, a change in values will propagate through parameters in a vertical or a horizontal fashion. Vertical propagation updates values of the branches by reaching to the last element in the last generation then to its sister elements, then back track to the generations higher in the hierarchy. This method is known in as (depth first). Horizontal propagation updates values for branches at the same level of hierarchy (Parents) at time, then the lower and lower. In either way, the propagation happens in a sequential fashion since every child depends on data passed from parents. Below are two visualizations of propagations:



Left: Vertical Propagation of values. Right: Horizontal Propagation of values. Both start by the left direction then the right one

3.0 Components:

Parametric systems, like Algorithmic systems, include three types of components. These include Generators, Interpreters and evaluators. Within design, generators are those components including concerned with constructing data (objects) like solids, surfaces, and wires, excel sheets, graphs, number, etc. Interpreters are those concerned with solving, updating and controlling flow of information through associations (relationships). For examples: a designer might generate a solid as an extrusion of a curved surface. This surface degree curvature is associated with the extrusion parameter. The higher the extrusion is, the flatter the curve becomes. Interpreters translate these changes in height as degrees of curvature. In Parametrics, interpreters are usually implicitly defined within generators. A specific type of interpreters can be found in animation systems in the form of inverse-kinematics solvers. These interpret changes in objects properties based on embedded relationships and changing data. Evaluators are functions capable of analyzing, registering, and processing objects attributes. For example, an evaluation component can analyze surface curvature (Gaussian or Mean), register values, and express them in various forms like colors, numbers, etc.

4.0 Units:

In abstract, parametric systems can handle any type of units because they don't have specific representations or rules. They only provide a platform rich in associations to build other systems. In fact, they can be classified as a certain type of Algorithms systems. However, there are two limitations that force parametric systems to deal with units as discrete. The first is the fact that we implemented them

through digital computer environments, ones that can only understand discrete units. The second limitation is imposed by parametric systems themselves through one-directional associations. For example, if B is a child of A, then it can only be found through that relationship only.

5.0 Inheritance, Constraints and associativity:

Inheritance appears in parametric systems through hierarchy. Constraints manifest themselves in the form of internal relationships or externally exerted limitations. Associations in parametric systems are one-directional. They allow for dependencies between elements in through hierarchies where children inherit parent's properties. Associations can only exist between families, or elements within a family of objects. Associating two elements from different families is only possible if these families share a similar global parameter than can be passed to the parents first, then to the children.

6.0 Mechanics:

There two type of structures in any parametric system. One deals with a family of elements (parts), and the other deals with families or elements (assemblies). The first, deals with the topological descriptions and internal relationships of an elements. For example, the way a solid is build, its components, attributes and parameters. All these are only manipulated at the element level. At this level, Part A is built by associations between children (A1, A2, A3). The second structure deals with associating families. For example: Associating family A with family B creates an assembly AB. There two types of relationships nested within assemblies. The first is possible between A (as a whole) and B (as a whole) just like associating A1 with A2. At the second is possible through associating elements from different families, say A1 and B1. In this case, it is only possible to associate elements if they share the same set of parameters.

7.0 Representation

Representation in parametric design system for architecture is similar to algorithmic systems. It does have a specific form. But in terms of parametric software, generators and evaluators are usually represented visually, or numerically where interpreters hidden in the background. Parametric systems only offer a platform, or an environment to build more specialized design systems. They provide for one-directional associations (hierarchies) where designers can build families of elements triggering chains of actions by updating a few parameters. In the following sections on L-systems, Cellular automata, Fractals and shape grammars you will notice that each has a specific representation like: strings, Symbols, numerical or geometrical.

8.0 Solution space:

In parametric systems, solution space is identified as a function of:
(Number of allowed discrete variables) X (Number of parameters) X (Number of relationships)

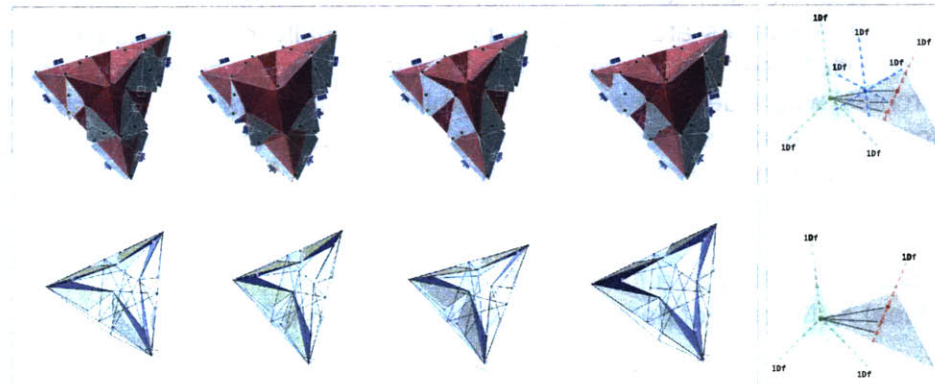
9.0 Experiments:

9.1 Parametric Skin Component:

The design intention in this experiment was to create a wall with various degrees of opacity based on the percentage of closed or open components that are driven by surface curvature. This was actualized by developing a parametric component capable of adapting to an array of surface deformations without bending or shearing. The component avoids such problems by rotations around hinges and pivots.

Such a components becomes easy to construct and assemble for it relies on flat panels, and basic hinge and pivot connections. This project can be perceived as a static wall or as a dynamically moving one for components are capable of adapting to changing surface conditions. However, they can only adapt to a certain range because panels are not design to expand beyond a certain bounding volume.

There were two designed components developed for this project. Both had the same number of joints and panels, however configured differently. Each component was design as a composite of three arms connected alternatively with each other. The first component design had more degrees of freedoms per joint compared to the second. In other words, it was harder to manipulate. Thus, the second component was selected as a solution.



The upper strip shows various deformation of the first configuration. The lower strip shows the second component, and the degrees of freedom each component has.

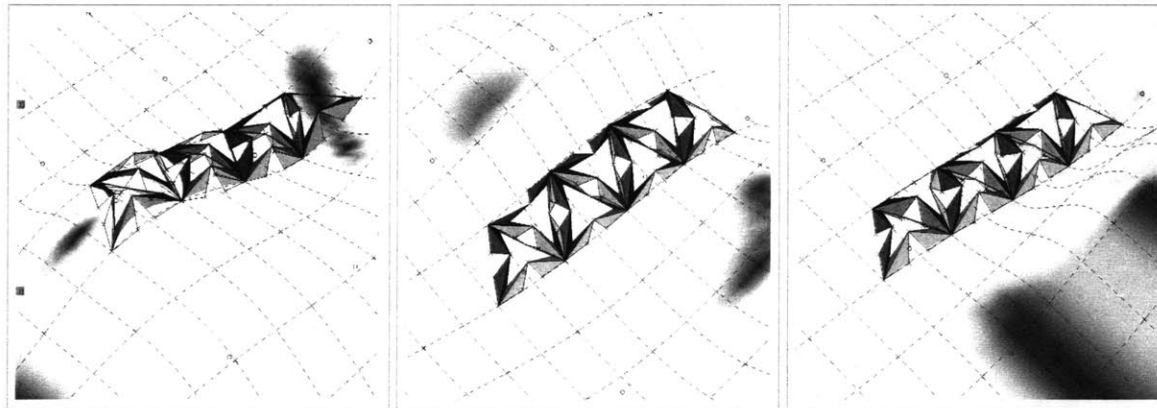
The project was realized by four systems. Controlling Geometry, Responsive Geometry, Mediating Geometry and Adaptive Geometry. The first three systems were built to simulate a changing context (deforming skin) where the fourth system handles the parametric component.

The first system is made of a point grid arranged in a hexagonal fashion. These points are used to generate a series of Nurbs curves (Non-Uniform Rational Bezier Spline) Curves, by which a Nurbs surface is built.

The second system builds a secondary layer of curves on top of the Nurbs surface, finds intersection points and centroid-locations based on these points. These points will be used to drive the mediating geometry in the third system.

The third system uses the intersection points and the centroid points to build pyramid-like surfaces that will control elements generated in the last system.

The fourth system will include a series of surfaces (panels) joint together at their vertices. This type of connection allows for rotation only.



The above shows components adapting to different skin deformations.

Every level is a parent to its lower one. One of the advantages of cascading systems is the ability to package many actions into a few control triggers. Adaptation of behavior becomes possible by propagating changes through hierarchies of elements. In this experiment, generation, interpretation and evaluation were all handled by CATIA internally.



Adaptive Component Design

In this project I explored a parametric design environment, CATIA, for expressing translucency through a discrete transition from "opened" to "closed" configurations of an adaptive component. Each component's behavior is driven and adjusted by different layers of mediating and responsive geometries.

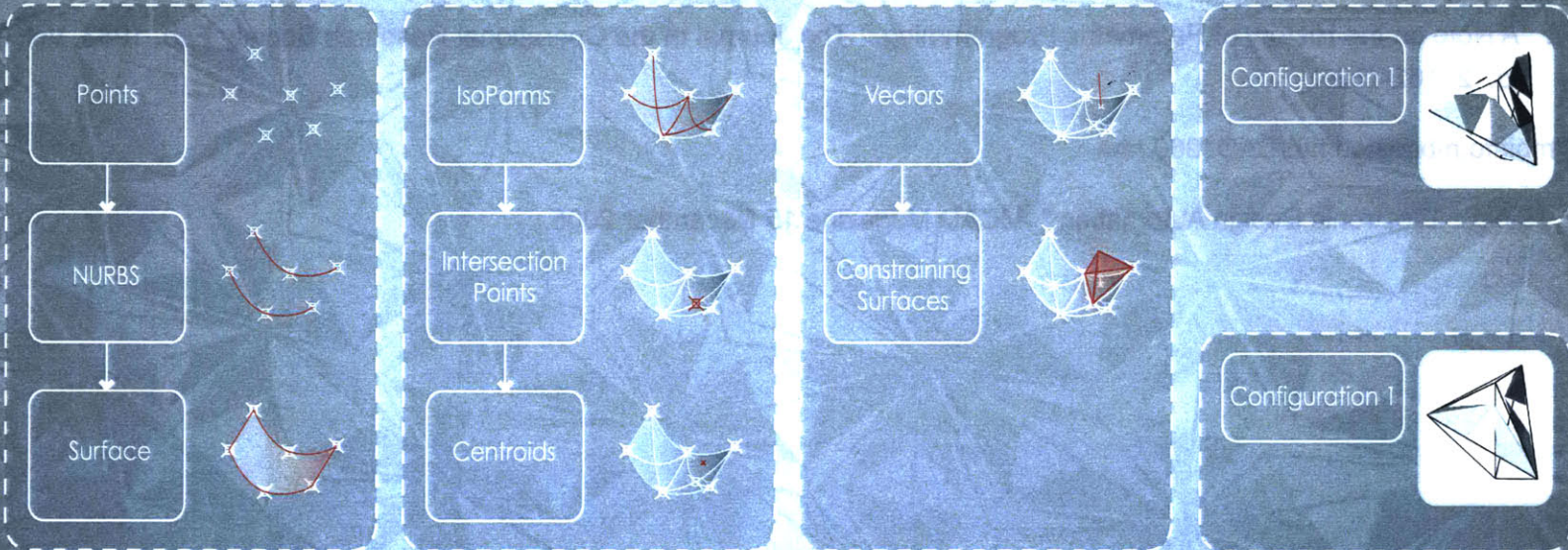
Design System

Controlling Geometry

Responsive Geometry

Restraining Geometry

Adaptive Geometry



10.0 Summary:

Parametric systems are hierarchical-algorithmic system controlled by one-directional associations. These systems allow for propagation of values in shared sets of parameters among elements within a family or different families. One of the limitations in parametric systems is that they generate results within a limited space for they are bound to relationships. In the introduction, I mentioned that Parametrics introduced the concept of regeneration instead of recreation. Creation requires reconstructing elements from the ground up every time a new result is needed. No matter how flexible such an environment is, it becomes unfeasible to implement for problems were solutions are constantly evaluated and recreated.

11.0 References:

Gal, T. "A Note on the History of Parametric Programming." The Journal of the Operational Research Society Feb, 1983, Vol. 34, No. 2: 162-163.

<http://mbinfo.mbdesign.net/CAD1960.htm>

Saitz, Robin. "Electronic Design Automation." MCAD Magazine 15 December 2005.

L-Systems

Introduction:

In the previous section, I introduced Parametrics as a specific case of algorithmic systems, (ones with associations). In this section, and the following three, I will introduce a series of more specific algorithmic systems. These are rule-based systems, what I defined earlier as formalisms. Rules are usually presented as a left side, arrow and a right side. For example, $X \rightarrow Y$. This means find X and replace it by Y.

It is important to note that these formalisms were created to simulate very specific phenomena as opposed to providing a working platform like Algorithmic or Parametrics systems. For example: L-systems were used to simulate botanic growth, Cellular automata were created to simulate reproduction, Fractals were created to simulate self-similarity in nature, and shape grammars were created to simulate human ability to see, or compute visually.

1.0 History and Background

2.0 System

3.0 Components

4.0 Units

5.0 Inheritance, Constraints and associativity

6.0 Mechanics

7.0 Representation

8.0 Solution space

9.0 Experiments

10.0 Summary

11.0 References

1.0 History & background:

L-systems have been always introduced in the light of Chomsky's work on formal grammars. Following this tradition, I will also start the discussion on L-systems by highlighting some concepts in Chomsky's grammars.

A formal grammar defines (or generates) a *formal language*, which is a (possibly infinite) set of sequences of symbols. It consists of: A) a finite set of *terminal symbols*; B) a finite set of *nonterminal symbols*; C) a finite set of *production rules* with a left- and a right-hand side, D) and a *start symbol*.

Terminal symbols form the parts of strings generated by the grammar, *nonterminal* symbols are containers of terminals. For example $A = \{2,3,4\}$. 2, 3,4 are terminal symbols and A is a nonterminal symbol. In other words, Terminal symbols act like (variables) that are stored in nonterminal symbols (parameters). Symbol replacement is sequential. It is implemented by matching the left side of a rule with a sequence of symbols. If a match exists, symbols in the right side of a rule replace those matched by the left side. This is called a *derivation*. Chomsky grammars are categorized in four types. These include: Type-0, Type-1, Type-2 and, Type-3.

Type-0 grammars (unrestricted grammars) include all formal grammars. They generate all languages that can be recognized by a Turing machine ¹. That is defined by an infinite set of elements following direct replacement rules such as: $a \rightarrow b$. (In this context, all generative systems are computed by Turing machines, except shape grammars because it is not limited to discrete units.) This type of grammar has unlimited set of nonterminal and terminal elements.

Type-1 grammars (context-sensitive grammars) are similar to linear bounded automata, i.e. 1-dimensional Cellular Automata (will be covered in the next chapter). In addition to replacement rules, they include methods to evaluate context in a string prior to rule application. These have a fixed set of nonterminal elements and an unlimited set of terminal elements.

Type-2 grammars (context-free grammars) are similar to the previous type except that they are not limited to a certain context to apply rules.

Type-3 grammars, known as (regular grammars), which are a context free grammar with a fixed set of nonterminal and terminal elements.²

On top of these grammars, Lindenmayer built L-systems in the late 1950s to simulate botanic growth patterns (i.e. how plants grow) through string replacement (will be explained in the following sections). In addition to the generation process, L-systems include components to interpret the generated alphabets as geometric objects to help visualize growth.

1 Marvin Minsky, Computation: Finite and Infinite Machines, (New Jersey: Prentice-Hall, Inc., 1967). See Chapter 8, Section 8.2 "Unsolvability of the Halting Problem."

2 Noam Chomsky, The Logical Structure of Linguistic Theory, (Univ. of Chicago Press, 1985).

2.0 System:

The essential difference between Chomsky grammars and L-systems lies in the method of derivation. In Chomsky grammars, rules are applied sequentially, whereas in L-systems they are applied in parallel³ replacing all letters simultaneously (this was explained in the section on Algorithmic systems). The parallel replacement process of L-systems mimics the production of cells where divisions occur at the same time.⁴

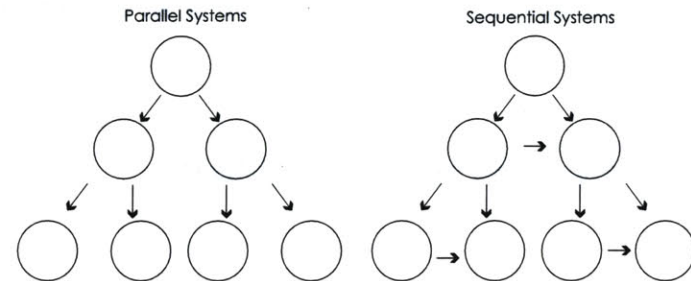


Diagram showing parallel and sequential replacement rules.

3.0 Components:

An L-system is composed of Axioms, Initial Strings, rules, and depth. Axioms are the set of all alphabets used in the system. Initial strings are the first seed for an L-system. Rules are presented in the form of: $R \rightarrow L$ where R is the initial Alphabet, L is the replacing Alphabet. For every new generation, the previous result becomes an initial string. Depth is the number of generations (iterations) of replacement.

Within a design context, the generation process of strings is always followed by an interpretation step where letters are used for constructing new objects, properties, geometric relationships, etc. Some architects experimented with L-systems to generate creases to fold paper patterns. Others used them to create branching architectural forms. All of these were implemented as experiments in academia, or as art installations.

The following example will clarify the difference between parallel and sequential replacement processes. Say you had a string of letters RRL, and a set of rules to replace R by UR. Using a parallel process will generate: (UR) (UR) (L) where each letter is being replaced at the same time. Using a sequential process you will generate: URL, then UURL, then, UUURL, etc.

Before looking at the components of an L-system in a design context, Let's look at another example. Given an initial string {RRLR}, and rules set $\{R \rightarrow L, L \rightarrow RR\}$, let's generate a solution of three generations.

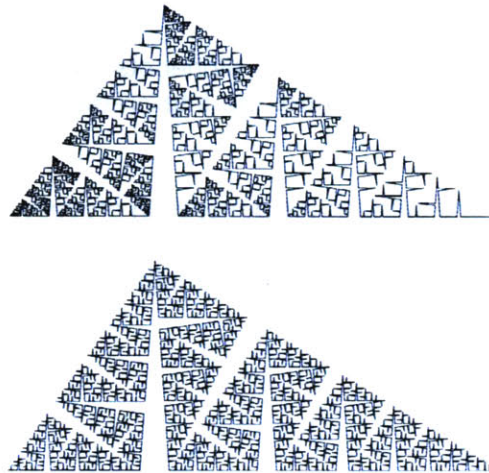
3 Aristid Lindenmayer and Przemyslaw Prusinkiewicz, The Algorithmic Beauty of Plants, (New York: Springer-Verlag, 1990).

4 <http://www.biologie.uni-hamburg.de/b-online/e28_3/l/sys.html>

Some L-systems may include parameterizations⁵. This is usually used for interpretation purposes. Below is an example of a parameterized L-system:

Initial String	RRRL
Alphabet Replacement Rule	$R \rightarrow RL^c$
Parameter Replacement Rule	$C \rightarrow C+1$

Below is an example on parametric L-systems. Parameters included in this example were segments rotations and lengths only.



The above example shows variations in the rotation angles and lengths of segments where the lower one shows a consistent angle. Image Credits: Aristid Lindenmayer and Przemyslaw Prusinkiewicz, *The Algorithmic Beauty of Plants*, (New York: Springer-Verlag, 1990).

The components mentioned above resemble the generator engine within an L-system. Generators are concerned with producing strings. After a generation process, strings are given to interpretation engines. An L-system interpretation process can happen incrementally per generation or for the whole solution. Interpreters are composed of two parts. The first is involved with parsing (iterating) through the generated strings of data and sets of parameters. The second is involved in constructing an interpretation (such as the graphical representations shown earlier). Evaluation engine can also be implemented through the generation or interpretation step in the case of context sensitive L-systems (more explanation is offered in section 6).

⁵ Aristid Lindenmayer and Przemyslaw Prusinkiewicz, *The Algorithmic Beauty of Plants*, (New York: Springer-Verlag, 1990).

4.0 Units:

As you may have guessed already, the smallest units in L-systems are alphabets. In fact, they are the only units. Unlike the general cases of algorithmic or parametric systems, L-systems can only deal with discrete symbols (units) known as alphabets. Their discrete identity makes them computationally (and programming wise) possible to integrate within any design process. However, their rigid structure makes them less robust than other generative systems in terms of variations because a letter means includes only one value. If you compare these units to the ones in the following system, cellular automata, you will find that CA symbols can carry more than one value (various states), thus, capable of giving more variations.

5.0 Inheritance, Constraints and associativity:

In L-systems, inheritance is not possible because of replacement rules. These will always replace existing information by new ones in every generation step.

Parameterization of alphabets allows for embedding properties to alphabets thus breaking the limitation of dealing with a set of fixed symbols. It also allows for applying local constraints to control the interpretation process later on. Constrains can also be applied externally as global conditions.

Associativity appears between alphabets within one string in the generation process, which will be explained in the following section in the context-sensitive L-systems. It is also possible to associate interpretation rules with embedded parameters or with each other. Below is an example of an L-system with a parameter n that gets updated through generations based on contextual conditions (constrains).

Given a string {LRRRL}

Rules: $R \rightarrow RRL$,
 $L \rightarrow L_n$: n = number of alphabets R,
 $L_n \rightarrow L_n L$

The re-writing process of this string for two generations will look like:

Initial String: LRRRL

Generation1: $L_6 RRL RRL RRL L_6$

Generation2: $L_{12} L RRL RRL L_{12} RRL RRL L_{12} RRL RRL L_{12} L_{12} L$

Applying external constraints to the above system within the replacement rules can be expressed in the following rule:

$L \rightarrow L_n$: n = number of alphabets R , and $n < \text{Limit}$.

6.0 Mechanics:

In this section, we will look at the mechanics (behavior types) of L-systems. The most famous and well-documented types of L-system behavior include: Deterministic, Non-deterministic, Context free, Context sensitive.⁶

6.1 Deterministic L-Systems:

This type of L-systems applies rules in a deterministic manner. There is only one replacement rule for a given set of alphabets. For example:

Initial string: {RRL},

Given rule: $R \rightarrow L$

R can be only replaced by an alphabet L.

6.2 Non-Deterministic L-Systems:

This type of L-systems includes more than one replacement rule. For example:

Initial string: {RRL},

Given rule: $R \rightarrow L$

$R \rightarrow M$

R can be replaced by an alphabet L or M.

6.3 Context-Free L-Systems (Sometimes known as 0L):

Such an L-system applies rules to alphabets regardless of any context within a given string.

6.4 Context-Sensitive L-system (Sometimes known as 1L):

This type of L-systems applies rules to an alphabet based on type of its adjacent alphabets. It is usually presented with "<" for right, ">" for left adjacent. For example:

Initial string: {RRL},

Given Rule: $R<R \rightarrow L$

6 M. Alfonseca and A. Ortega, "A Study of the Representation of Fractal Curves by L Systems and Their Equivalences," IBM Journal of Research and Development 1997, Volume 41, Number 6.

This means that the above rule only will replace alphabets R if its right-side neighbor is R. If not, the rule will not be triggered. In this example, there can be only one generation: LRL for the above rule is not applicable to any letter in the LRL (R's right neighbor is not R).

The above types may be combined as: Context-Free Deterministic, Context-Free Non-deterministic, Context-Sensitive Deterministic, or Context-Sensitive Non-Deterministic

7.0 Representation:

A representation introduces a methodology, an environment where the user interacts with the system. It highlights and eliminates certain systems properties. An L-system is usually broken into three components as briefly discussed earlier. The generative component is typically represented in the form of strings of alphabets, parameters, rules, and a set of symbols {<, >, →}. Since L-systems deal with discrete units (alphabets), a solution is a composite of constituent units. Thus, there not need be a sophisticated recognition engine especially that an alphabet' topology remains continuous.

Interpretation takes place after a solution is generated. Representation within the interpretation engine is typically: a) graphical, usually known as Turtle Graphics, or b) mathematical, known as Vectors graphics (Matrices).

Turtle graphics representation is widely used as it presents an intuitive representation of L-systems growth patterns. Seymour Papert developed a robot (Turtle) that would move forward, backward, turn left or right based on a given set of symbols.⁷ This representation method was intended to aid in teaching children geometric concepts like rotations, displacements.

In a design context, "Turtle geometry"⁸ is constructed via direct or indirect interpretive rules. Direct application of rules would translate alphabets directly to actions like: R = move a point distance d in direction c or draw a line with length so and so. An indirect application would translate letters to data, then to a representation like: R = value of X / height. The result would be used later as data for other interpretation processes

8.0 Solution space:

The concept of solution space was new to the design profession. It is mainly borrowed form the engineering discipline, mainly the optimization field. A solution space can be explained as all the possible solutions given by the strategy for solving a given problem. Within generative design system, the capacity of generating variation is mirrored in the number of parameters, variables, (relationships if implemented) and rules. In L-systems a solution space can be expressed as follows:

Number of generations X String length X Number replacing symbols sets

7 Seymour Papert, Mindstorms: Children, Computers, and Powerful Ideas, (New York: Basic Books, 1980).

8 Turtle Geometry is the pattern produced by the path of an imaginary turtle moving around on a plane.

9.0 Experiments:

9.1 Make A Tree:

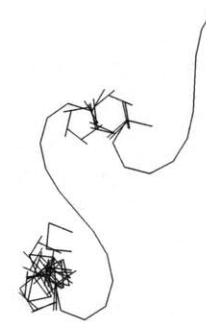
The goal behind this basic experiment was to present the concept of packaging certain representations (geometric in this case) into a series of letter. The system was built by one component only (interpreter) that deals with creating a geometric representation for a user-input of alphabets. There are no string-rewriting processes (generator), nor context-sensitive rules. The algorithm prompts the user asking for an input string of L for go left, M for go right, and R or S for reverse direction. Then it will interpret each letter as a moving point that leaves a trace (line) behind.

Make Tree-Branch:

- ◊ Insert a string of alphabets.
- ◊ Parse through the string
 - ‡ Translate each alphabets to a vector (point and a direction)
 - ‡ Place a line segment on each vector

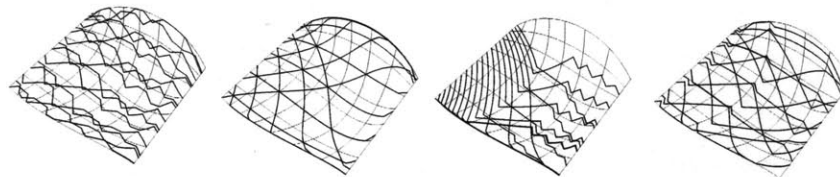


LLLLLLLLLLLLRMMMMMMMMLLLLLLLLLLLLRMMMMMM-
MMLLLLLLLLLLRMMMMMMMMLLRMMMRMMMMMMR
MMMMMLLLRLLLLSLLLLLLLLSMMMMMRLLL



9.2 Basket Structure:

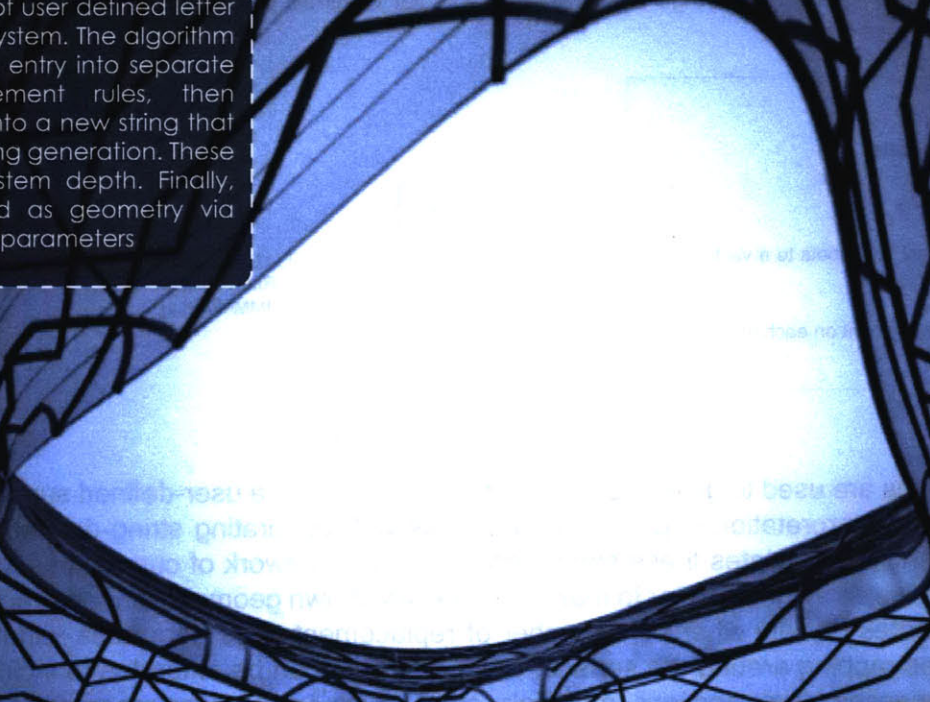
In the second experiment L-systems are used to generate a membrane structure on a user-defined surface. In this context, the system includes both a generation and an interpretation engine. The first deals with generating string-derivations by applying deterministic replacement rules. The second engine translates these string derivations to a network of curves on a surface. There was an implicit evaluation component embedded inside the interpreter to make sure the last drawn geometric shape reaches to opposite surface edge. The number of derivations generates density where the number of replacement rules generates variations. This created a basket-like network of intersecting curves rapping around the surface. Such a system can be extended to include structural parameters like maximum spans, members' thicknesses, cross sections, materials, etc. The experiment was mean as of driving structural components by a series of replacement rules where performance-driven variations of structure are possible.



Above are variations based on 4 different initial strings 1, 2, 3, 4 using the replacement rules. Below is a result using the initial string that generated design solution 1.

Concept For Generating Basket Structure

L-systems are formalisms capable of representing growth patterns. This experiment's focus was to explore mapping of user defined letter strings into a membrane system. The algorithm recursively breaks a string entry into separate letters, applies replacement rules, then concatenates the result into a new string that will be used for the following generation. These strings define the the system depth. Finally, each string is interpreted as geometry via orientation, and thickness parameters



Design System

String Builder

Tokenizer
Input= LLLWMMWLW!LWW
Output= {L,L,L,W,M,W,L,W,L,W,W}

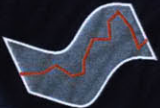
Replacer
Replacement rules: L->W, W->L, M->M
Input= {L,L,L,W,M,W,L,W,L,W,W}
Output= {W,W,W,L,M,L,W,L,W,L,L}

Concatenator
Input= {W,W,W,L,M,L,W,L,W,L,L}
Output= WWWLMLWLWLL

Moderator

String Interpreter


Origin Maker


Translator


System Components

Replacement Rules
C R L
= = =
L C R

Interpretation Rules
Depth * { L C R}

Turtle Graphics

Representation of L-System Growth On A Surface

10.0 Summary:

L-systems can package a variety of behaviors in a few alphabets and rules. In terms of programming, these systems are also very easy to implement for they deal with discrete units. A user only needs to provide rules and an initial string. Then, the string will be tokenized, rewritten and concatenated iteratively based on the specified number of generations. L-systems have not yet been deeply implemented within an architectural context. Their structure does not match the complexity found in many architectural design problems.

11.0 References:

Alfonseca, M. and Ortega, A. "A Study of the Representation of Fractal Curves by L Systems and Their Equivalences." IBM Journal of Research and Development 1997, Volume 41, Number 6.

Chosmsky, Noam. The Logical Structure of Linguistic Theory. Univ. of Chicago Press, 1985.

<http://www.biologie.uni-hamburg.de/b-online/e28_3/lsys.html>

Minsky, Marvin. Computation: Finite and Infinite Machines. New Jersey: Prentice-Hall, Inc., 1967.

Lindenmayer, Aristid and Prusinkiewicz, Przemyslaw. The Algorithmic Beauty of Plants, New York: Springer-Verlag, 1990.

Papert, Seymour. Mindstorms: Children, Computers, and Powerful Ideas. New York: Basic Books, 1980.

Cellular Automata

Introduction:

In the previous section, I presented L-systems as the first of the four formalisms I chose to discuss in this thesis. The reason was that it was the least flexible of all systems. Its symbols are limited to one type of meaning, alphabets. Cellular Automata systems offer a richer environment for its symbols are not limited to one type of meaning. A symbol in CA (cell) can refer to “Color” with its variations (black, white, etc.), or size (with various numbers), location (in reference to many axes), etc or even different objects.

1.0 History and Background

2.0 System

3.0 Components

4.0 Units

5.0 Inheritance, Constraints and associativity

6.0 Mechanics

7.0 Representation

8.0 Solution space

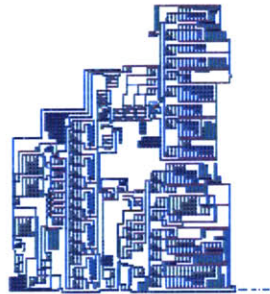
9.0 Experiments

10.0 Summary

11.0 References

1.0 Historical Background:

In 1940s John von Neumann was working on the problem of self-replicating systems. His initial goal was to build robots that could produce copies of themselves. Through the process, he faced major difficulties in devising methods for recognition so a robot can identify and pick different parts. Later, he worked with an abstract environment (mathematics) where he could select and recognize parts simply by indexing. He was able to build a universe of cells where reproduction behavior is simulated in discrete actions performed by each cell. Thence was the birth of Cellular Automata. His system was built on 29 states per cell ¹(Batty, 2005). Each cell's behavior is driven by the states of its neighbors through sequential rule application (This was explained in the section on Algorithmic systems). It is not a surprise that CA, and many other formalisms in computing, inspired the field Artificial Intelligence where behavior is being driven by networking a set of discrete tasks performed by agents (among other things).



Neumann First Cellular Automata. Image reproduced from < http://en.wikipedia.org/wiki/Image:VonNeumann_universal_constructor.PNG >

As a provisional model to Neumann's CA, John Conway invented the Game of life back in the 1970s to be the earliest CA automated by computers. Martin Gardner popularized this game in *Scientific American* articles². The game displayed an impressive diversity of behavior, fluctuating between apparent randomness and order, what Stephen Wolfram noted as Complex and considered a universal Turing machine.

1 M. Batty, Cities and Complexity: Understanding Cities with Cellular Automata, Agent-Based Models, and Fractals. (Cambridge: The MIT Press, 2005) 229-258.

2 M. Gardner, A Quarter-Century of Recreational Mathematics, *Scientific American*, 1998.

Since 1983 Wolfram has been publishing papers systematically investigating one-dimensional cellular automata, which he called “*elementary cellular automata*”. Their simple structure and unexpected complex behaviors led him to suspect that complexity in nature could also be generated by similar mechanisms. In the *New Kind of Science (2002)*³, Wolfram represented cellular automaton as a new domain for exploring different fields and branches of science including chemistry, geology, biology, physics, and many others. He repackaged almost all written literature on Cellular Automata and added a major contribution resembled in the taxonomy of their behaviors.

In architecture, cellular automata were mostly explored as generative systems for pattern generation. Within design, John Frazer presented some of the most known experiments of using CAs for form generation. However, some designers like Michael Batty utilized cellular automata as universal computers that can reproduce (reorganize) themselves ensuring certain conditions. He produced a considerable amount of work on implementing cellular automata for generating neighborhoods of different land uses.

2.0 System:

Cellular Automata simulates reproduction behavior by sequentially applying replacement rules to cells’ patches (neighborhoods). These rules manipulate the states of a cell and its neighbor once at a time. A neighborhood definition may change based on various embedded rules. For example a cell may consider a neighborhood of three if its state is A, but considers only one cell if its state is B. Rules may also be related to time steps. CA systems are composed of replacement rules, cells, and initial states. Rules are only applied if their (left side) matches an initial condition (states of a cell and its neighbors). This will be explained in more detail in section 6. Below is an example of CA rules.

3.0 Components:

Like L-systems, Cellular automata design systems are composed of separate data generators interpreters and evaluators. The difference is that Cellular automata are only context-sensitive; there is always a need for evaluation components to check states of cells and adjacent cells within a neighborhood. Thus, evaluators are always embedded within generators. Interpretation components can map cells’ states directly as outcome. It is also possible to embed evaluators within interpreters.

3

S. Wolfram, *A New Kind of Science*, (Champaign: Wolfram Media Inc., 2002)

For example, given a CA generation of 20 cells, an interpreter might map cells' states as colors on a surface, or might check (evaluate) indexes and states of certain states, and translate that data into various levels of glass properties for a building skin.

4.0 Units:

Similar to L-systems, Cellular Automata is a composition of discrete units (cells). However, they do not have to be of the same type. CA cells can carry a variety of "things". They may contain geometric descriptions, colors, numbers, etc. These descriptions may only change in values, but not in type. Thus, Neighborhoods, relationships and cells' topologies are always maintained. The fact that CA units can include multiple meanings makes them capable of generating more variation than L-systems.

5.0 Inheritance, Constraints and associativity:

Unlike L-systems, initial inputs (cells' states) do not affect most of Cellular Automata systems behavior. This is apparent specifically in the periodic automata where the system will always generate a definite pattern (this will explained in the following section). This is due to the fact that rules are applied sequentially where neighborhoods (update) each other. In other words, neighbors' conditions are constantly changing based on changes in the previous neighborhoods. A piece of information does not extend beyond one generation. This means that it is not possible to ensure the existence of an attribute or property along through out generations. Thus, Inheritance is not possible. Constraints are usually embedded implicitly within the generation step through defining the number of iterations a certain cell may undergo, or types or locations of cells that are allowed to change. Associations may be established in the interpretation components where various parameters may drive the creation of results. These parameters may include: states, time steps, history of cells states, etc.

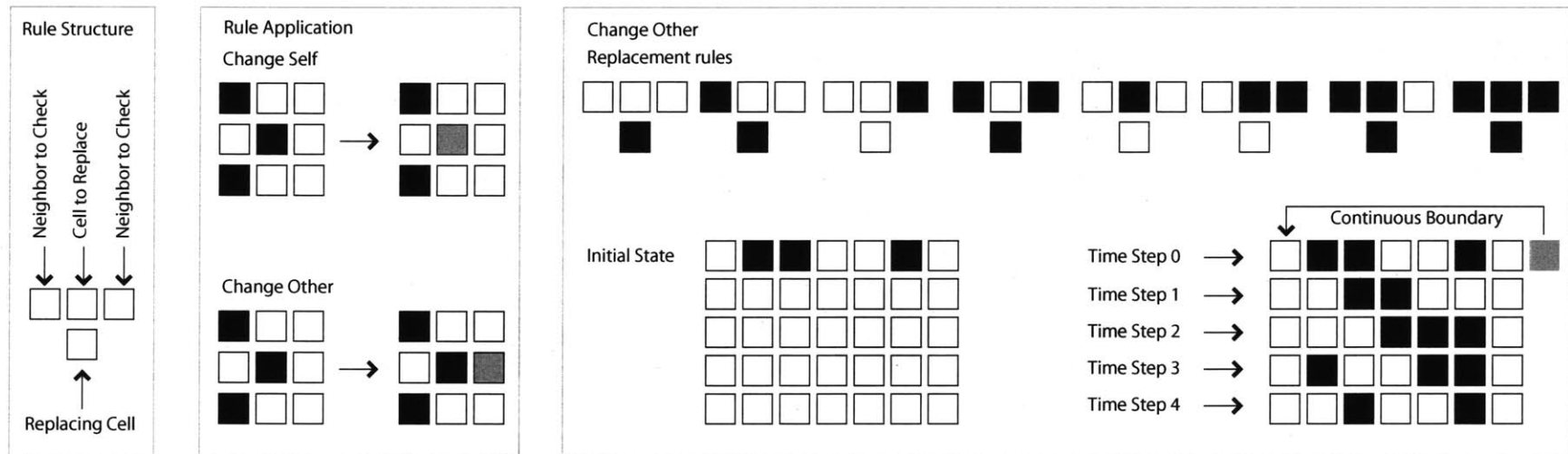
6.0 Mechanics:

Wolfram described four types of behavior in Cellular Automata systems in reference to the time period needed for them to mature (stop or repeat themselves). These are: Fixed, Periodic, Chaotic and Random. A Fixed Point behavior is identified as the sudden halting after a very short period of time. Periodic behavior shows repetitive pattern within a fixed period of

time. A Chaotic, also known as *Aperiodic*, or *Complex*, seems to repeat its behavior within different durations of time steps. Complexity can be visualized as a mixture of the periodic and random behaviors. The last type of behavior, *Random*, does not seem to repeat its behavior within any specific time period. Wolfram classified 256 rules for elementary cellular automata. The number of initial conditions is calculated by: $\text{States}^{\text{Neighbors}}$, which is $2^3 = 8$. Below are eight neighborhoods with two states.



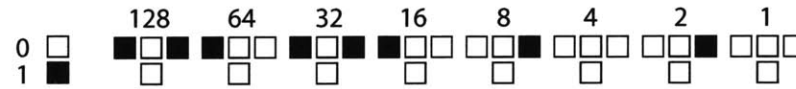
In the above eight initial states combinations, you may only generate two new states (black or white). So that the total number of possible combinations becomes: $\text{States}^{\text{Neighbor Combinations}} = 2^8 = 256$ rules. Cells' initial combinations and new states are called Rules. They are represented numerically through a binary system where white means zeros and black means one. For example, a state of 01101011 equals rule 107. I generated a series of experiments to represent the four classes of behavior. ⁴



⁴ G. W. Flake, *The Computational Beauty of Nature: Computer Explorations of Fractals, Chaos, Complex Systems, and Adaptation*. (Cambridge, The MIT Press, 1998), 229-258.

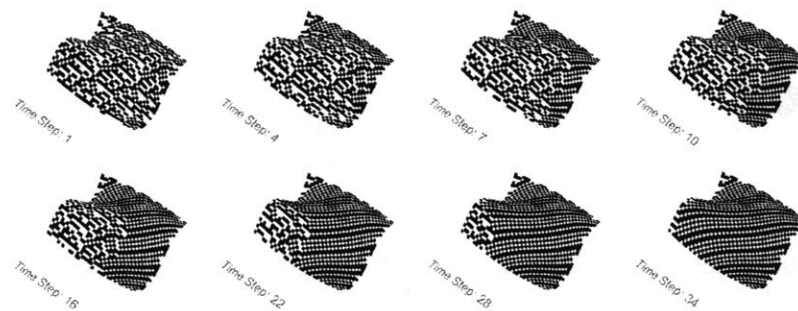
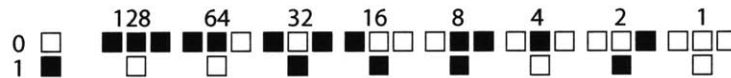
6.1 Class one:

An example of a fixed-point behavior is rule 8. Here, the system reaches to a fixed state instantly after one generation.



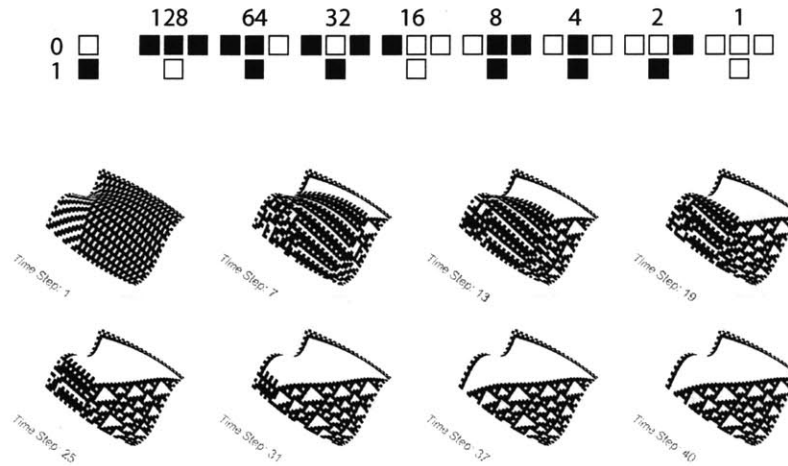
6.2 Class two:

A periodic behavior can be generated using rule 109. CA repeats a behavior in similar time periods.



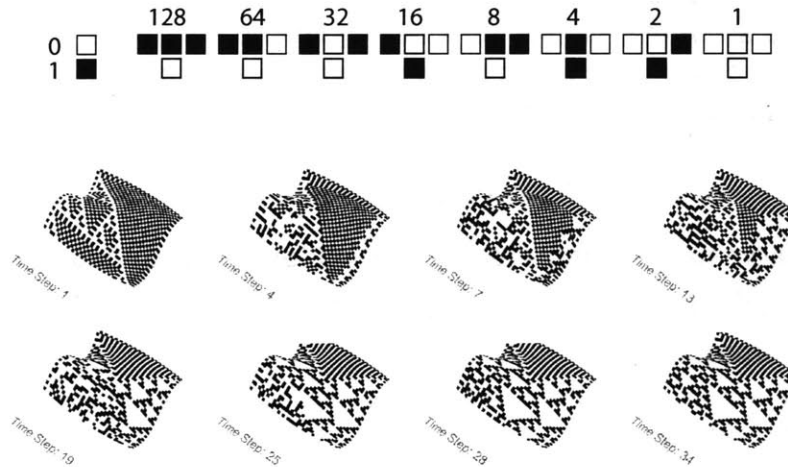
6.3 Class three:

An example of Complex behavior is rule 90. Complex behavior seems to repeat certain triangulation patterns through different time periods.



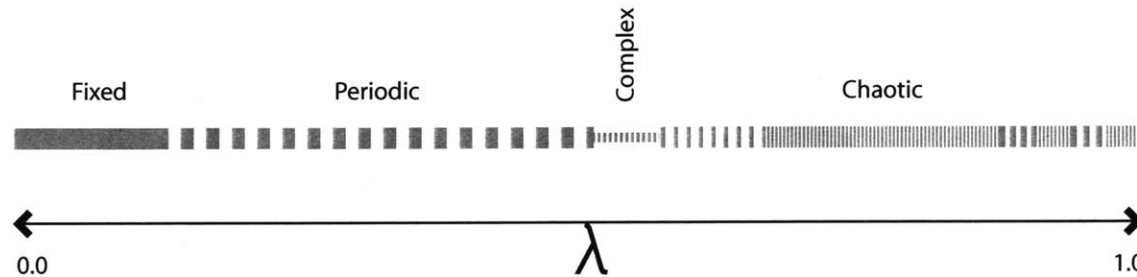
6.4 Class four:

An example of random behavior is rule 22. The pattern does not seem to repeat any specific pattern within a fixed time period.



Chris Langton proposed a diagram to map the transition of CA behavior from Fixed Point to completely Random. The logic relies on representing both extremes: Fixed Point where rules will generate fixed type of cells, Random where rules will generate a variety of cells' types. These two types of rules will make the total rules set. ⁵

$$\lambda = (\text{Total Rules} - \text{Fixed Point Rules}) / \text{Total Rules}$$



The value of λ reflects the type of behavior on the scale from Fixed, to Periodic, to Complex to Chaotic (random)

CA rules can be automated iteratively within a fixed number of time steps, or progressively until a global condition is achieved. The first type of application is computable for rules, cells number, and time periods, which are fixed. The second type is not necessarily computable in abstract since its halting condition may not be achieved. Consequently, any solution space in CA is always computable by virtue of repeating behavior or by inherited limitations in computers. For this reason, CA Class four behavior, Random, may be considered as an extended complex behavior that computers were never able to capture. These various behavior types show that CA systems deliver limited variations. More of its behavior is periodic, (patterned). For this reason, these systems do not lend themselves to complex architecture problems. This pushes the application of CA to organizational aspects or patterns generation.

7.0 Representation:

Like L-systems, Cellular Automata systems are usually implemented through two separate representations for design purposes: One for generating data (states), and one for interpreting these states as outputs. Evaluation is usually merged within the generation and interpretation.

⁵ G. W. Flake, *The Computational Beauty of Nature: Computer Explorations of Fractals, Chaos, Complex Systems, and Adaptation*, (Cambridge, The MIT Press, 1998), 229-258.

Ibid.

8.0 Solution space:

Solution Space in CA is the set of possible variations in cell's states over time. It is calculated as a function of $Time\ Steps \times States^{Cells}$. A complete solution space can only be identified when the behavior of cellular automata matures by repeating itself or by halting.

9.0 Experiment:

9.1 Cellular Automata As A Texture Map

In the first experiment, the system implements rule set 126 (as identified by Wolfram) and directly maps the solution as colors. The context of this experiment only required a generator and an evaluator engine. There was not an interpretation engine. A system-controlling unit was implemented to handle the computation.



9.2 Porous Skin design:

The concept in this experiment was to use Cellular Automata Systems as data calculators to A) arrive at a general global condition of zeros and ones, B) then interpret that data as variations of canonical-truncated-pyramids. These pyramids include different sizes of openings to simulate a porous skin. An external system component was implemented to automate the process based on a defined number of time steps. This experiment utilizes two main engines: a generator and an interpreter. Unlike L-systems, the evaluation component is embedded in the generator engine to help guide rule selection based on cells states.

The system requires the user to select a starting condition from a predefined set of One-color, random, or checker board⁶. Then it asks for a number of iterations where it will apply replacement-rule-22 and keep track of cell's history. Then it asks the user to select a surface where it populates a patch-grid. After a final solution is generated, the system uses cells' final states and update history to create vectors with different magnitudes. It finally constructs canonical pyramids with various opening-sizes. This system can be extended to control various types of geometric constructions based on locations of cells in relation to each other, or a certain architectural program, etc.

6 Some Cellular Automata Rules are sensitive to the starting condition. The rule used in this experiment (Rule 22) is one of them.



Cellular Automaton For Porous Skin

This exploration deals with a CA generative system as a representation for design. The goal was to create a self-organizing neighborhood of units leading to a skin with controlled porosity. In this structure local conditions drive the global behavior based on states, cells history and surface orientation.

10.0 Summary:

Most of cellular automata behaviors generate ordered patterns like periodic or aperiodic, and very few generate universal patterns (not bound to certain order) like random or chaotic. Since architecture is a) not driven by only sequential processes; b) neither it is about patterns; c) nor is it about unpredictability; Cellular automata applications within architecture become very limited to certain confined tasks. The implementation of such systems requires a great deal of appropriation and customization for design as they utilize a specific structure, and representation, which might not always match the complexity of architecture design problems.

11.0 References:

Batty, M. *Cities and Complexity: Understanding Cities with Cellular Automata, Agent-Based Models, and Fractals.* (Cambridge: The MIT Press, 2005).

Flake, G. W. *The Computational Beauty of Nature: Computer Explorations of Fractals, Chaos, Complex Systems, and Adaptation.* (Cambridge, The MIT Press, 1998).

Gardner, M. *A Quarter-Century of Recreational Mathematics.* *Scientific American*, 1998.

Wolfram, S. *A New Kind of Science.* (Champaign: Wolfram Media Inc., 2002).

Fractal Systems

Introduction:

We saw in the previous sections that L-systems and Cellular Automata maintain the size of their smallest units. Rules replace alphabets or cells without breaking them to smaller ones. The concept of the “smallest unit” is not applicable to Fractal Systems for they are based on mathematical models of recursion. Fractal algorithms will recursively fracture elements first, and then replace them by new ones.

1.0 History and Background

2.0 System

3.0 Components

4.0 Units

5.0 Inheritance, Constraints and associativity

6.0 Mechanics

7.0 Representation

8.0 Solution space

9.0 Experiments

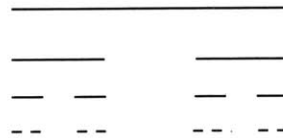
10.0 Summary

11.0 References

1.0 History and Background:

Coined by Benoit B. Mandelbrot¹, the term “Fractals” describes what was known as mathematical monsters explored between the end of 19th and beginnings of 20th century. These include Cantor (1872), “Cantor set”; Giuseppe Peano (1890), “Peano Curve”; David Hilbert (1891), “Hilbert Curve”; Helge von Koch (1904), “Koch Curve”; Walclaw Sierpinski (1916), “Sierpinski Carpet”; and Gaston Julia (1918), “Julia Set”; among many others. These “monsters” were considered “exceptional objects” for they don’t fall under any theoretical framework². They were generated to explore fundamental concepts like topology and continuity. The most famous example of fractals is the Cantor set. So, let’s take it as an entry point to the subject.

Cantor set, also known as no middle-third set³, (Cullen 1968, pp. 78-81), is generated by removing the open-middle -third interval of a set recursively. This can visually illustrated as shown below:



In design, one of the first examples on procedures that generate such objects occurred in the work of Albrecht Dürer 1525 who published “The Painters Manual” which contained a section on “Tile Patterns Formed by Pentagons”⁴.



Durer Pentagon. <http://ecademy.agnesscott.edu/~lriddle/ifs/pentagon/image301.gif>

1 Benoit Mandelbrot, “How Long Is the Coast of Britain? Statistical Self-Similarity and Fractional Dimension,” *Science*, 5 May, 1967, New Series, Vol. 156, No. 3775: 636-638.

2 Hartmut Jürgens, Heinz-Otto Peitgen, and Dietmar Saupe, *Chaos and Fractals*, 2nd Edition (Springer: 2004), 63.

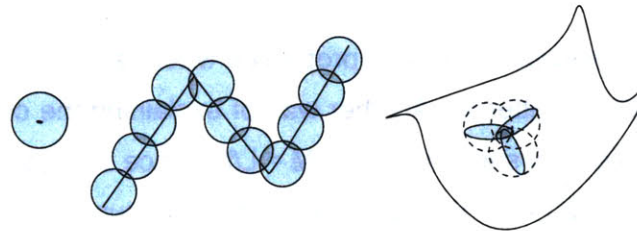
3 H. F. Cullen, *Introduction to General Topology*, (Boston: Heath, 1968), 78-81.

4 Lee Makowski, “An Unreasonable Man in a Quasi-Equivalent World,” *Biophysical Journal*, January 1998, Volume 74: 534 –536.

Fractals, as Mandelbrot defines them, are self-similar objects at different scales. Just like any other geometric property, self-similarity is expressed by a dimension called: Fractal dimension. It is more than the object's Topological Dimension and less than its minimum Euclidian dimension. What does this mean?

Our typical understanding of dimensions is built on the Euclidean dimension, (D_E); That is, the minimum number of axes or directions needed to describe an object in relation to the space that contains it. For example: a line in can be described by one direction only if it was straight. Otherwise, it needs two directions in reference to a plane, or three in reference to a volume.

Statements like: "a line is one dimensional or a surface is two dimensional regardless of space", are based on topology. A Topological dimension (D_T), usually known as the covering dimensions, relies on the minimum number of the smallest-disks-possible needed to cover an object. It is the number of disks generating "an" intersection -1. In the case of a point, a topological dimension is 0. In the case of a curve, there are no more than 2 covering disks per intersection. Thus, the topological dimension of a curve is 1. In the case of a surface, there are 3 spheres per intersection. Thus, the topological dimension of a surface is 2.⁵



The figure above shows covering disks and various objects, a point, a line and a surface.

A simpler way of describing topological dimensions is to think of a moving point. Ask yourself: How many possible directions would a moving point have on an object? In the case of a line it is one direction: forward/backward. In the case of a surface, it is two directions: forward/backward, and left/right. In the case of a solid, you add a third direction: Up/down. The concept of how many units needed

to cover or measure an object is the essence of the Fractal dimension, or Similarity dimension (D_s). It is a measure of how similar an object is to itself. Does not make much of sense! Does it? The following paragraph will hopefully give a clearer description.

Empirical evidence suggests that the smaller the increment of measurement, the longer the measured length becomes, and the more accurate it is. If you were to measure a stretch of coastline with a long ruler, you would get a shorter length than if you were to measure the same coastline with a smaller ruler (broken into pieces) even though you are using the same measuring units. The second measure can be mapped more accurately than the first one like shown in the following figure. Thus, it is said to be more “similar” to real measure of the coastline. Is there a way to gage how similar these measure are to the coastline?

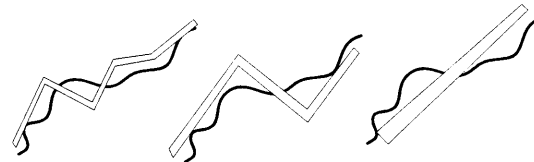
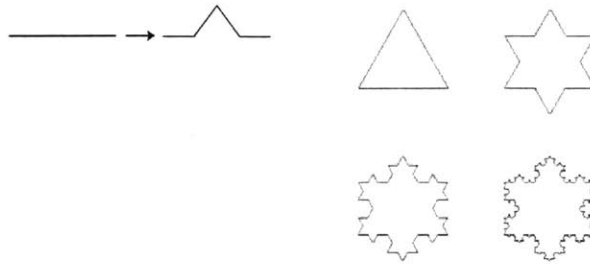


Figure showing different measuring sticks for a curve. The smaller the measuring device is, the more accurate the measurement is.

In a broader sense, many objects in nature possess self-similar properties where a certain pattern re-appears at different scales. A fractal dimension assumes the re-appearance of patterns. Another way of explaining the concept of re-appearance is the following: Given a line length L of unit 1, a number of division N such that a unit $M = L/N$. Since $L = 1$, then $M = 1/N$. Thus $L = M * N = 1$, Similarly, an Area will be presented as $A = L * L = (M * N)^2 = 1$. In (Euclidean terms) the super script refers to the dimensionally level. This definition will always generate an integer for the relationship between the number of divisions and the measure of the unit at different scales is always proportional. Thus, if you scale M up by a certain ratio, then N will be scaled down by the same ratio.

Fractal objects do not maintain this property, thus the dimensionally level is a non-integer. The equation to calculate the fractal dimension is usually written in the following form: $D_s = \text{Log}(\text{number of measuring units}) / \text{Log}(1/\text{Number of divisions})$. In the case of a line, $D_s = \text{Log}(M)/\text{Log}(1/N) = 1$. In the case of one of the mathematical Monsters, a Koch Curve, where the re-appearance of a four segments scaled to a third is evident at different scales, $D_s = \text{Log}(4) / \text{Log}(3) = 1.261\dots$ The following fractal system shows the generation process of the Koch Curve.

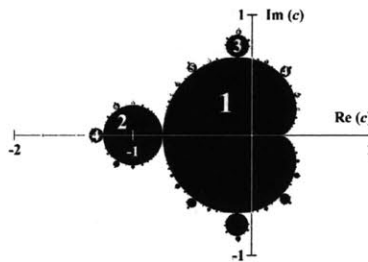


A curve is replaced by a four-sided polygon where each side is scaled by the ration of 1:3.

2.0 System:

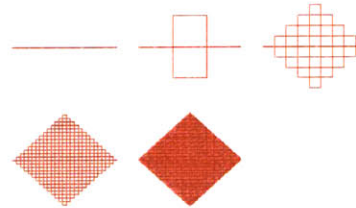
A fractal system is capable of generating objects with self-similar parts where pattern reoccurs at various scales. Such a system can be algorithmically achieved through recursion. Recursive algorithms are procedures that “calling themselves” for a given object or a set of objects “until” a base condition is matched. This general framework draws three methods for generating fractal objects:

The first, deals with recurrence. The most famous fractal object generated by this method is the Mandelbrot set. It is represented by the following formula: $Z_{n+1} = Z_n^2 - \mu$ where $Z = X + iY$; X, Y are real numbers, i is a fixed parameter, and μ is a complex number⁶. This type relies on reusing the last result (Z) as initial data for the following calculation. Thus, the value of Z reoccurs mathematically in the equation.



Visualization of Mandelbrot set. Image Credits: <http://www.fractalus.com/kerry/articles/area/overall-mandelbrot.gif>

The second method generates fractal object through Iterated Function Systems (IFS). These include fixed replacement functions. Fractals generated by this method include Cantor set, Sierpinski carpet, Sierpinski gasket, Peano curve, and Koch snowflake among others. Iterative functions include: Displacement, Scaling and Rotation ⁷.



The generation process of a Peano Curve. Image Credits:<http://home.hia.no/~byrgeb/png/Peano.png>

The third method is stochastic (random). A recursive algorithm is randomly applied on different parts of a given set of objects. These include simulations of mountains, clouds and many other natural plants representations.



The generation process of terrain. This fractal was generated by random application of mid-point displacement algorithm. Image Credits: <http://www.cs.clemson.edu/~mvannin/805/P4/>

Since fractal Systems are built by recursive process, they are hierarchical by nature. A main function will trigger an internal one within the system structure. For this reason also, fractal systems can be described as parametric for a change in a parameter in the main function, typically a base condition (halting condition), will affect the generation process through the number of times it will trigger an internal function. For example, a fractal system may run until a segment length is smaller than a minimum (integrated evaluation), or it may run for n number of iterations (separate evaluation).

⁷ Gary William Flake, The Computational Beauty of Nature: Computer Explorations of Fractals, Chaos, Complex Systems, and Adaptation, (Cambridge: The MIT Press, 2000), 103-106.

3.0 Components:

Unlike L-systems or cellular automata, Fractals have generators and interpreters combined in one engine, which I will refer to a generator/interpreter as “Constructor”. This proposes a different system component layout. Fractal systems also include three engines, but coined differently. These are: initiators, Constructors, and evaluators.

Initiators are those functions that call Constructors for as long as a stopping condition is not matched. Constructors are responsible for constructing (generating/interpreting) data as long as they triggered by the initiator. Evaluators are responsible for finding a stopping condition for the system to halt. Currently available computers have limited capacity. Thus, the third engine is essential to make recursion computationally possible as it registers and evaluates for stopping conditions. Like the previously discussed systems, Fractal compositions are composed of discrete units. Thus, recognition algorithms don't go beyond calling objects by their indexes.

4.0 Units

Like L-systems and Cellular automata, Fractals systems deal with units as discrete. This is due to the fact that fractal algorithms always break elements into a predefined number of smaller units, similar to what we saw earlier in the Koch curve. This fact draws an interesting distinction between fractal systems on one side, and L-systems and Cellular Automata on the other. Fractals are not bound to the concept of the smallest unit because they recursively break elements into smaller ones before applying replacement rules, where L-system and Cellular automata are limited to one size because they only replace whole units without breaking them. However, from an implementation point of view, the smallest unit for geometric fractal systems is a point, and zero for mathematical ones.

5.0 Inheritance, Constraints and associativity:

Although hierarchical, fractal systems also do not allow for inheritance like L-systems and cellular automata for data is constantly replaced through recursion. Constraints take the form of a) stopping conditions like “run until area = A”, b) or context-descriptions like “apply rules to objects on first and last objects only”, c) or certain rule behavior like “divide by ration of X”. Associativity can only be found within the constructors, as they are responsible for manipulating given objects.

6.0 Mechanics:

Fractal systems' behaviors generate from: Recurrence, Iteration or Random. These include: Exact Self-Similar by recurrence, Semi Self-Similar by iterations, and Statistically Self-Similar.

6.1 Exact Self-similar:

These are strictly self similar in all levels. Proportion is maintained. This type includes exact scaled down copies of the whole in every part of the object.

6.2 Semi Self-similar:

These include self similar parts, but may vary in scale, or proportion. Usually deferred to as distorted fractal.

6.3 Statistically Self-Similar:

These include any object or set of objects that have a non-integer similarity dimension as a minimum description. Thus, such type of fractals is included in the definition of both the Exact and Semi self-similar types. These two types belong to the fractal family as they share the Self-similarity and non-integer dimension.

7.0 Representation:

In the previously explored systems within design, there was always a distinction between the representation of various components: generation, interpretation and evaluation. In Fractal generative systems, generation and interpretation are merged in one representation for they both happen in one step.

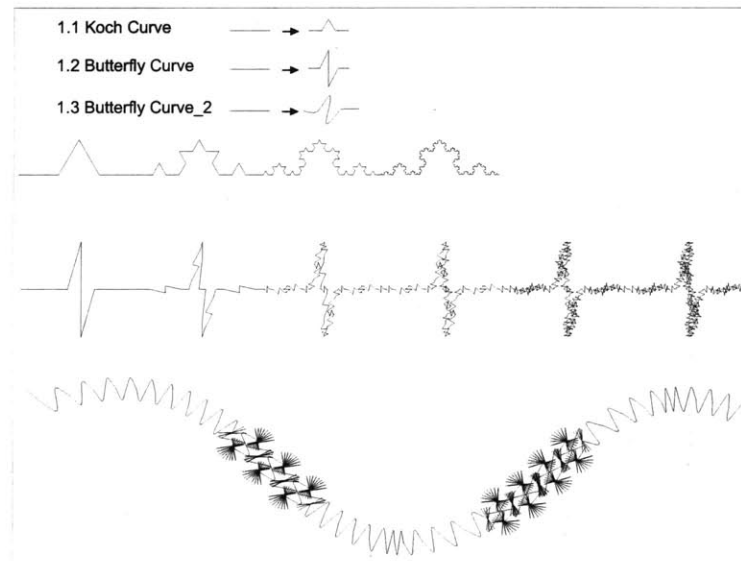
8.0 Solution space:

Fractal systems ability to generate variation can be described by the following function: Number of replaced objects X number of replacing objects X number of recursive operations (controlled by halting conditions or global number of iterations).

9.0 Experiments:

9.1 Fractal Curves:

In fractals and shape grammar design systems, generators and interpreters are combined in one engine. This experiment shows the generation process of a series of fractal curves. The system includes a constructor, an embedded evaluator and an initiator. The first deals with breaking and constructing shapes, the second evaluates for a stopping condition, and the third handles the automation of the process.



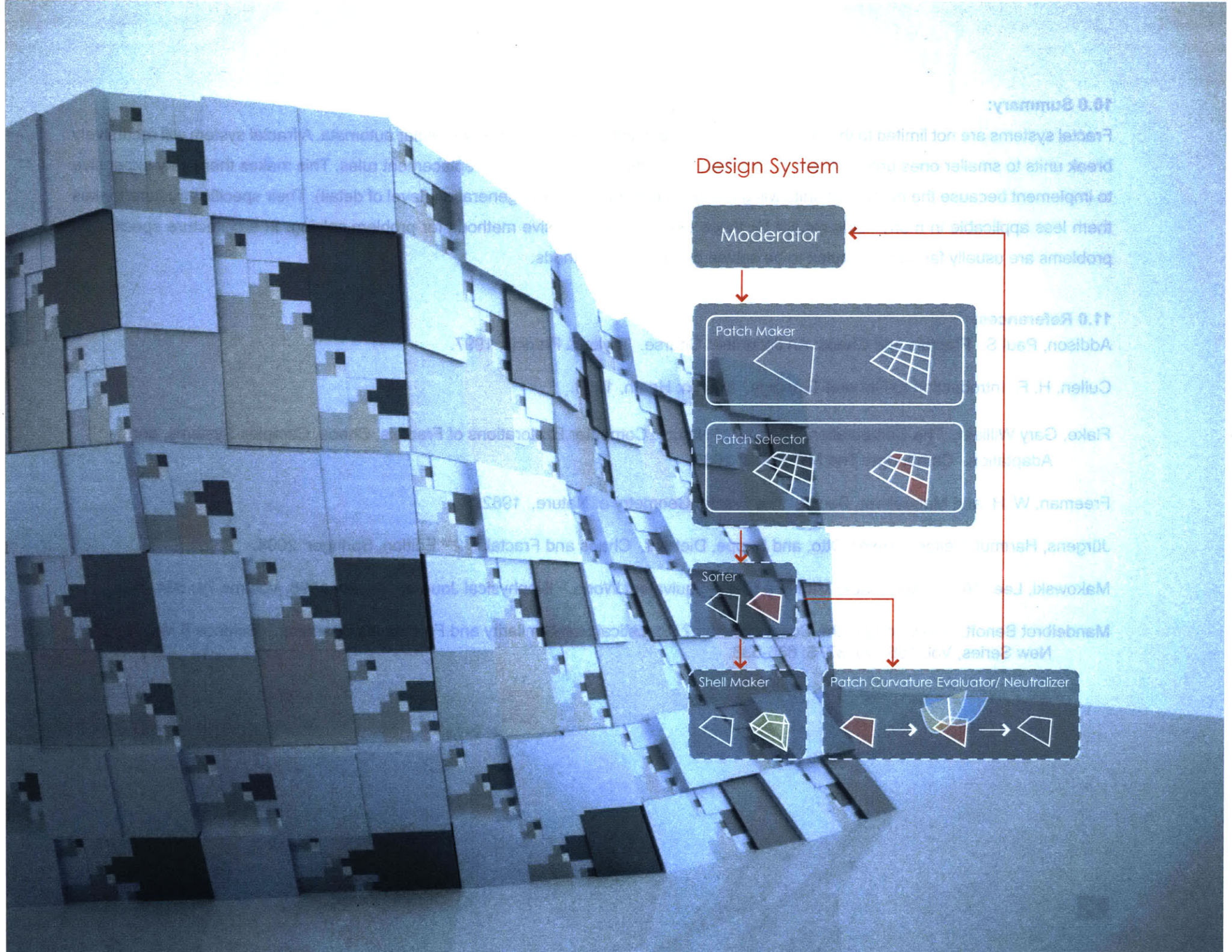
9.2 Paneling System Based on Surface Curvature:

This experiment utilizes a similar structure to the previous one, however the context is more complex. The goal was to devise a system that can generate optimized panels for complex curved surfaces. The algorithm will break a surface to a number of optimized patches. Then construct a series of (slightly) optimized ones in the same region recursively until the evaluation engine declares a stopping condition, which in this case is a minimum surface area or an acceptable curvature. Finally each panel is given a thickness as a suggestion to material properties. Like the previous experiment, an Initiator is needed to handle the automation of the computation.

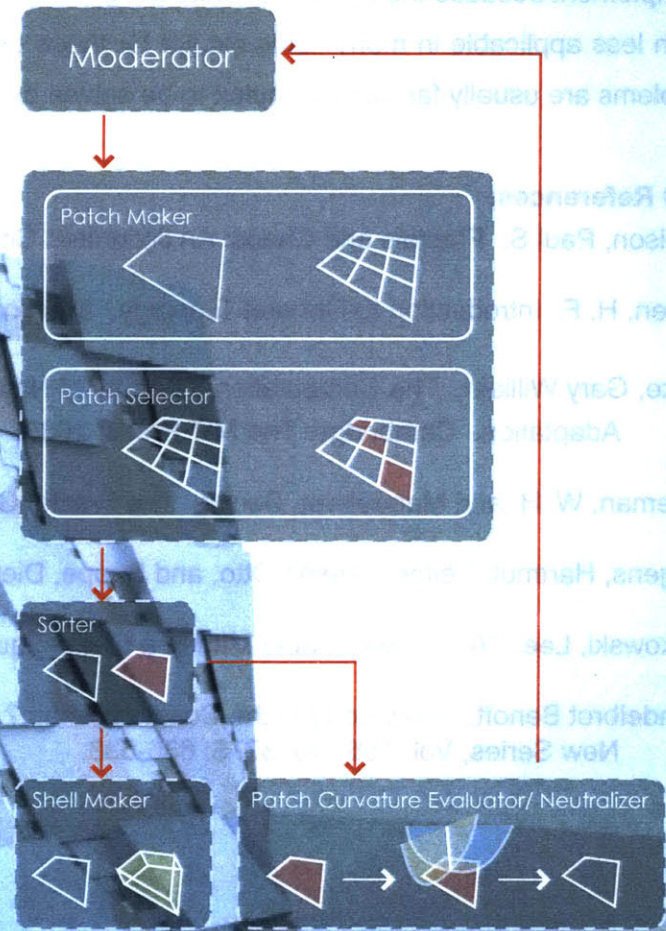


Concept For Panelling System

In this project, the goal was to explore a generation method for a panelling system based on surface curvature. A fractal subdivision algorithm was developed to break a user defined surface into patches, then evaluate the area and Gaussian curvature of each. Based on the generated data, the algorithm will recursively break each patch until a stopping condition of area or curvature is met. Finally, thickness is being added as suggestion of possible material properties for each patch.



Design System



10.0 Summary:

Fractal systems are not limited to the boundaries of “smallest units” like L-systems or Cellular automata. A fractal system will recursively break units to smaller ones until a stopping condition is reached, and then apply replacement rules. This makes them very expensive to implement because the number of units will grow exponentially with every generation (level of detail). Their specific structure makes them less applicable in many fields except for those dealing with recursive methods for problem solving. In architecture specifically, problems are usually far more complex to be solved by repetitive methods.

11.0 References:

Addison, Paul S. *Fractals and Chaos: An Illustrated Course*. Taylor & Francis, 1997.

Cullen, H. F. *Introduction to General Topology*. Boston: Heath, 1968.

Flake, Gary William. *The Computational Beauty of Nature: Computer Explorations of Fractals, Chaos, Complex Systems, and Adaptation*. Cambridge: The MIT Press, 2000.

Freeman, W. H. and Mandelbrot, Benoit, *The Fractal Geometry of Nature*, 1982.

Jürgens, Hartmut, Peitgen, Heinz-Otto, and Saupe, Dietmar. *Chaos and Fractals*. 2nd Edition, Springer: 2004.

Makowski, Lee. “An Unreasonable Man in a Quasi-Equivalent World,” *Biophysical Journal* January 1998, Volume 74: 534 –536.

Mandelbrot Benoit. “How Long Is the Coast of Britain? Statistical Self-Similarity and Fractional Dimension.” *Science* 5 May, 1967, New Series, Vol. 156, No. 3775: 636-638.

Shape Grammars

Introduction:

The previously discussed formalisms (L-systems, Cellular Automata, and Fractals) recognized units as discrete in reference to their locations (boundaries) assuming they have fixed identities throughout the computation process. In Shape Grammars, units are recognized both by fixed and flexible definitions. The first relies on “identity” (like other systems) where the second relies on “embedding” (intrinsic to shape grammars). These systems were built to capture visual calculation processes in design. They handle recognition through the human ability to see. Mapping such a concept to the world of discrete units that computers understand requires very sophisticated algorithms. Ones that can pick shapes wherever they may be. This fact limited the implementation of shape grammars to analog processes performed by humans, or computer-automated ones working with discrete units only.

1.0 History and Background

2.0 System

3.0 Components

4.0 Units

5.0 Inheritance, Constraints and associativity

6.0 Mechanics

7.0 Representation

8.0 Solution space

9.0 Experiments

10.0 Summary

11.0 References

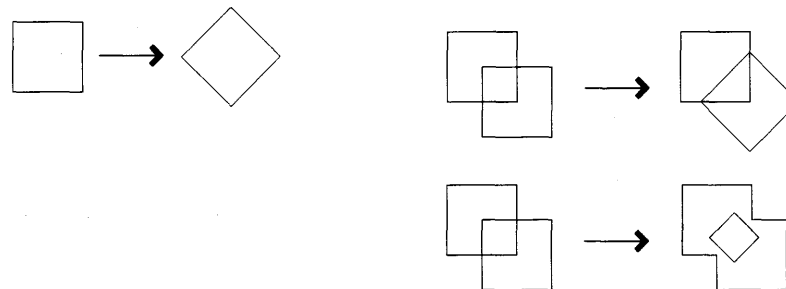
1.0 History and Background:

“Design is more than sorting through combinations of parts that come from prior analysis (how is this done?), or evaluating schemas in which divisions are already in place. I don’t have to know what shapes are, or to describe them in terms of definite units — atoms, components, constituents, primitives, simples, and so on — for them to work for me. In fact, units mostly get in the way. How I calculate tells me what parts there are. They’re evanescent.”¹

George Stiny
Shape

The main concept of shape grammars can be encapsulated in the adverb “how”, as opposed to “what”. George Stiny,² along with James Gips, created a system for visual computing termed ‘Shape Grammars’ which is based on the decomposition of shapes into embedded parts, boundaries and relationships.

The ability to “pick” shapes or parts of shapes is possible through the concept of embedding. In shape grammars, if you can see it, then you can pick it. Following is a basic example on a Shape Grammar system where a rule says: Pick the a square and rotate it from a given set of shapes. The designer may pick any shape as shown below:



Left side is the rule, right side are two possible results. The first is generated by applying the rule the lower square where the second is generated by applying the rule to the square in the middle.

2.0 System:

Shape Grammars looks at computing as a composite of counting and seeing. Counting provides a methodology to measure or describe whole shapes and seeing devises a framework to pick out embedded parts from larger shapes. Shapes are sets of maximal elements. These include all possible embedded smaller elements. For example, there are indefinitely many line segments in any given line segment, indefinitely many surfaces in a surface, and solids in a solid. There are two relationships by which shape grammars recognize, or ‘pick’, elements: embedding and identity. Embedding appears in the case of seeing parts *within* maximal shapes. For any given rule,

1 George Stiny, Shape. (Cambridge: The MIT Press, 2006).

2 J. Gips, and G. Stiny, ed. C. V. Freiman, Information Processing 71. (Amsterdam, 1972), 1460.

which operates on a line segment, embedding allows one to *pick* either a whole line segment (maximal) or any part thereof. This way, the composition of shapes can be continually reconfigured into smaller and larger parts. The exception is points; since a point can only ever contain itself, points can not be enlarged or reduced. Maximal elements only contain shapes with similar topologies³. For example, it is not possible to pick (or embed) a point element in a segment. However, it is possible for them to overlap such that a point exists in the space of a segment. Identity allows for seeing maximal elements only. It means that any element can be embedded in itself. Here is another way of expressing these two concepts:

Given three Sets:

$C = \{\mu_1, \mu_2, \mu_3\}$, $B = \{\mu_1\}$, $D = \{\alpha\}$. It is true that:

- 1- $C \subset C$, $B \subset B$ (Identity relationship)
- 2- $B \subset C \therefore C$ is a Maximal Element
- 3- $D \not\subset C$ for D has different types of basic elements (Shape grammars do not mix topologies)
- 4- $C \not\subset B$ for $C - B \neq B - C$

In the above example, multiple symbols in set C suggest multiple embedded elements, but of similar topology.

In shape grammars, the boundary of an element with dimension n is identified by elements of dimension $n-1$. For example, two points bound a segment. Three or more segments, or three or more points bound a surface. A minimum of four surfaces, or six line segments, or four points bound a solid. Furthermore, any element of dimension n must exist in a space of a similar, or higher dimension. A point may exist in a point space, line space, plane space or volume space. A segment may exist in a line space, or volume space, but not point space. Points are the only elements that can exist in point spaces, thus they can be used for counting. This may be best presented in the following Algebra Table, represented as U_{ij} .

$$\begin{array}{cccc}
 U_{00} & U_{01} & U_{02} & U_{03} \\
 & U_{11} & U_{12} & U_{13} \\
 & & U_{22} & U_{23} \\
 & & & U_{33}
 \end{array}$$

“U” Refers to the element, “i” refers to its dimension (used for Boolean part relationships); and “j” refers to dimension of space containing it (used for Euclidean transformations). For example, U_{00} is a shape with dimension 0 (a point), in a 0 dimensional space (point-space).

³ Topology is a continuous description over two states of before and after a Boolean operation, a transformation or a deformation. For example, it is true to state that a square maintains its topology after a rotation/shearing (even twisting) for it still contains the same description of 4 sides. If you subtracted a circle from a Square, then you changed its topology because you introduced new edges. For further reading on topology, see: Dover, Experiments in Topology.

In shape grammars, the empty shape represents the null set of symbolic math. Any transformation of an empty shape results in an empty shape. Any transformation of a nonempty shape contains the transformation of each of the basic elements in the shape.

a) The rule $X \rightarrow X$ means X is X . This rule may sound redundant as it redefines the element X . But in fact, it expresses the concept of picking (seeing) elements by identity relationships. In any other context, X is considered a part-shape or a maximal shape.

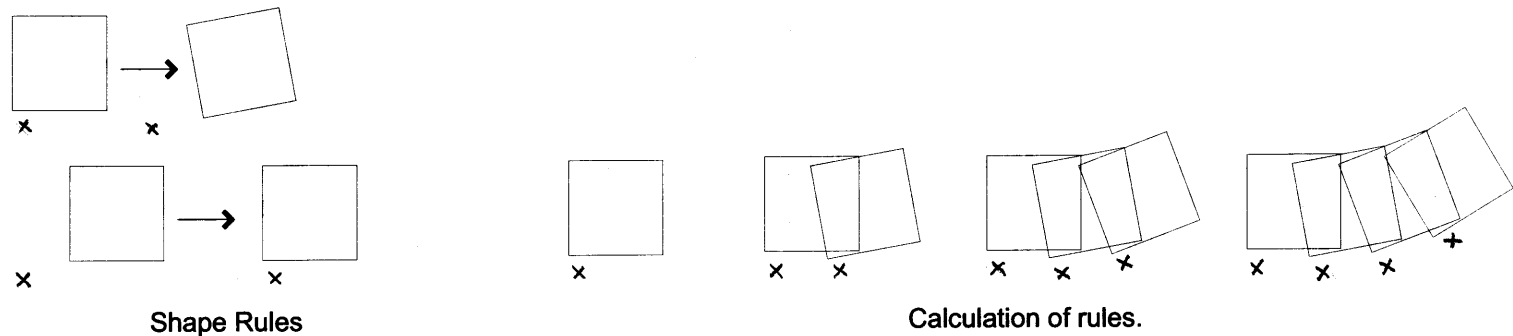
b) The rule $X \rightarrow X'$ means X is under transformation, thus $X' = X + \text{Transformation}$.

c) The rule $X \rightarrow$ means X is being erased. This rule expresses the Boolean operations of subtraction.

d) The rule $X \rightarrow X+Y$ means that element Y is being added to element X

e) The rule $X \rightarrow Y$ means X is being replaced by a new element Y

Below is an example on applying shape grammars rules.



3.0 Components:

Shape grammars components are packaged differently in comparison to the previously discussed systems. They include: Constructors, Recognizers and Initiator. A Constructor is similar to that of fractal systems. It combines generator and interpretation processes. A Recognizer deals with temporary decomposition of existing shapes to find embedded initial shapes (left side of grammars' rules). Upon successful findings, the Recognizer will report (pick) the results and inform the Initiator. Initiators deal with system management. They select which rules should be applied and pass them along with the picked shapes to the constructors. Recognition in shape grammars can be performed by human or by a computer. For a computer to recognize initial shapes for a given set of rules, it must be able to decompose the existing shapes into boundary elements and embedded basic elements, identify any possibility to construct an

initial shape, then finally “pick” the element(s). The current status quo in computer science is not fully capable of implementing shape grammars due to the limitation of making computers recognize any shape anywhere.

4.0 Units:

Before discussing the units in shape grammars, I would like to reiterate the following facts:

- 1- Any shape may be a part of other shapes (embedding)
- 2- Shapes are made of basic elements (collections of topologically similar elements)
- 3- Shapes can be manipulated through Boolean operations like subtraction, addition, or intersection, and transformed through translation, rotation, scaling and reflection.

In the previously discussed formalisms (L-systems, cellular automata, and fractals) you have noticed that units and rules were fixed (continuous) through out the computation. New types of units or rules cannot just appear. In shape grammars, rules can be written on the fly and units can be recognized anywhere in any form. Recognition is usually performed by human. Thus, they can pick anything anywhere, and make any rule for what they picked on the spot.

In shape grammars, units are made up of basic elements of a single kind: points, segments, surfaces, or solids. Points have no divisions. But segments, surfaces, and solids can be broken into discrete pieces, (smaller line segments, triangles, and tetrahedrons). More generally, every basic element has a distinct basic element of the same dimension embedded in it, and a boundary of other basic elements from a lower dimension. Points have no boundaries. The above-mentioned properties of basic elements are presented in the following table:⁴

Basic element	Dimension	Boundary	Content	Recognition
Point	0	none	none	Identity
Line	1	two points	length	embedding
Plane	2	three or more	area	embedding
Solid	3	four or more	volume	embedding

The abstract premise of shape recognition by embedding allows shape grammars to handle units with flexible or fixed definitions (like algorithmic or parametric systems in abstract). However, once implemented through computers, designers have to force distinctions and fixed descriptions of units' locations so that they can be picked or recognized easily.

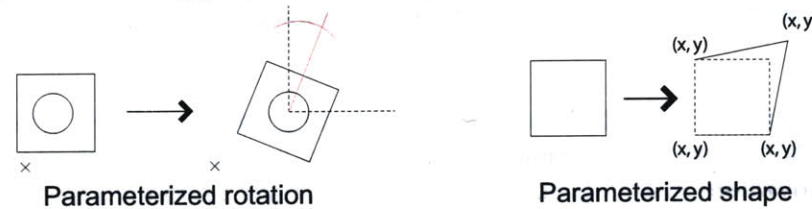
4

This table is taken from George Stiny, Shape, (Cambridge: The MIT Press, 2006).

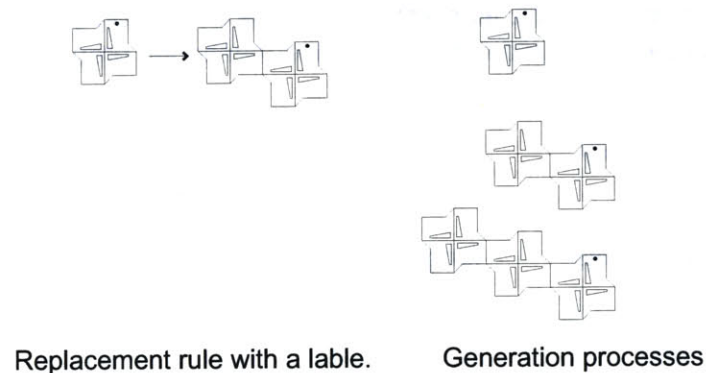
5.0 Inheritance, Constraints and associativity:

It is possible within a shape grammar to inherit properties because replacement rules can be applied to parts of elements where other parts continue to exist throughout the computation in various generation steps.

In shape grammars, like other generative systems, associativity defines dependencies between rules, shapes, and parameters. For example, the rule: $X \rightarrow Y$, can only be implemented if an element matching the topology of X exists, or is generated by other rules. Associativity is also possible by parameterizing properties of shapes like in parametric shape grammars. or by parameterizing rules.



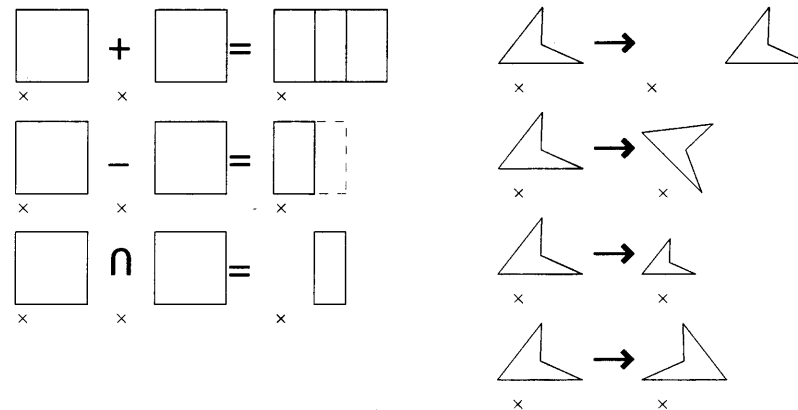
Constraints can be applied through aids that describe contextual conditions when locating initial shapes. For example, adding labels or colors. In the following page you will find an example on adding labels to shape grammars rules constrain their behavior.



6.0 Mechanics:

It is important to formalize a design language before building a generative system for it. Once a language is formalized, design objectives, architectural forms, and rules extraction become easier. Rules might be nested in stages to ensure a certain control of the design's development, in a bottom-up or top-down process. It is also possible to parameterize rules such that shapes can have variations within them. For example, a 4-sided shape might be drawn as a square, rectangle parallelogram, diamond, etc. All these

shapes can be generated by parameterizing angles or sides' lengths. Like the previous three formalisms, Shape Grammars are also built by replacement rules. In this context, replacement rules include initial shapes, spatial relations (typically presented as an arrow), and replacing shapes. Initial shapes make the left side of a grammar. Spatial relations make middle. They handle: a) Boolean Operations (Union, subtract, intersect), and b) Euclidean transformations (Move, Rotate, Scale, Reflect). Replacing shapes make the right side of grammars. A replacement rule $x \rightarrow y$ is read as: "x goes to y" where x is the initial shape (right side of the equation), "goes to" is a spatial the relation, and y is the result or replacement for x (left side of the equation). In the opposite page you will find examples on these concepts.



Examples of Boolean Operations and Euclidean Transformations

7.0 Representation:

Representation in shape grammars is built through geometric shapes, labels and colors. Just like Fractals systems, there is no distance between generation and interpretation processes. Both are combined in one representation.

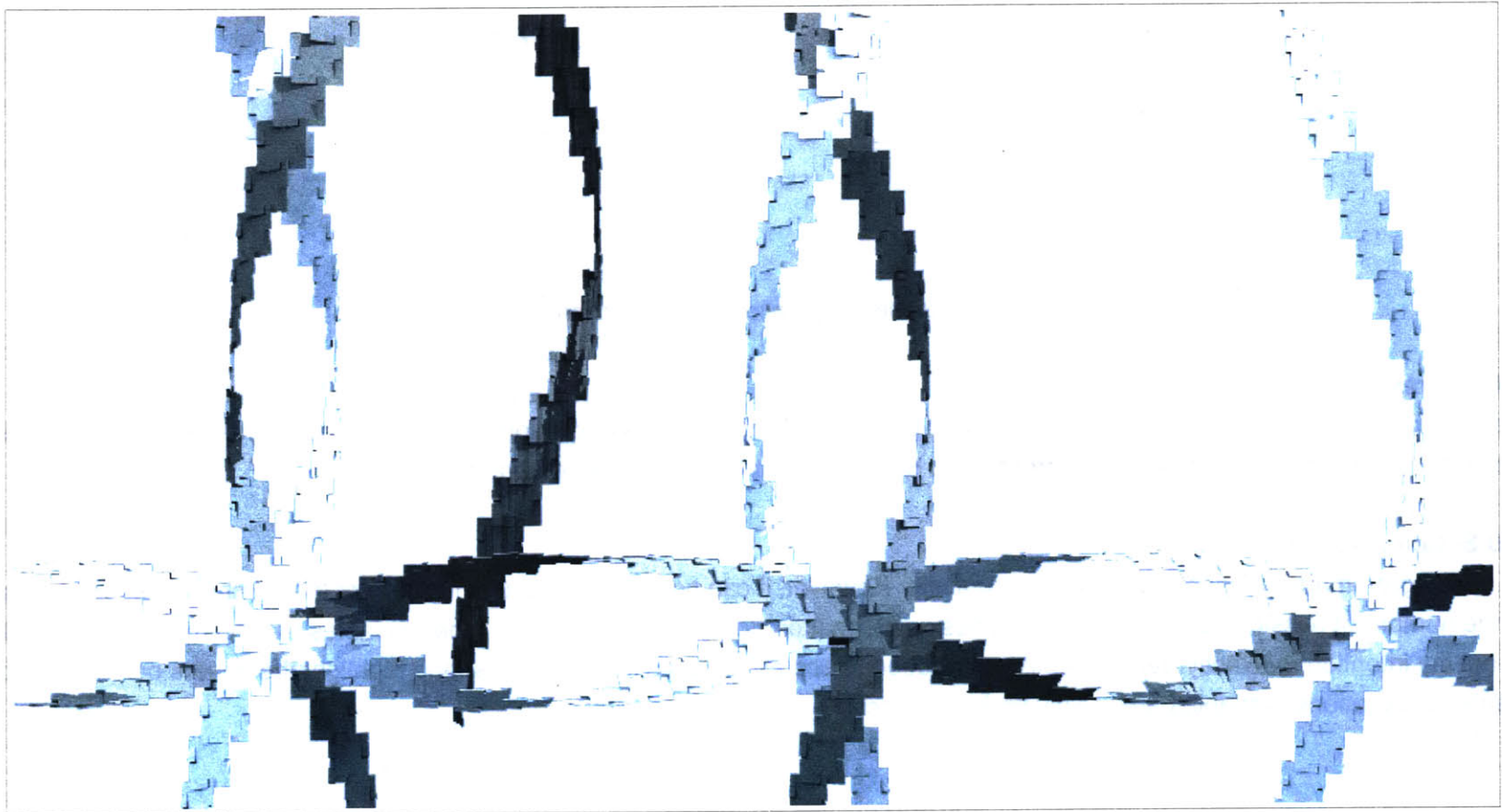
8.0 Solution space:

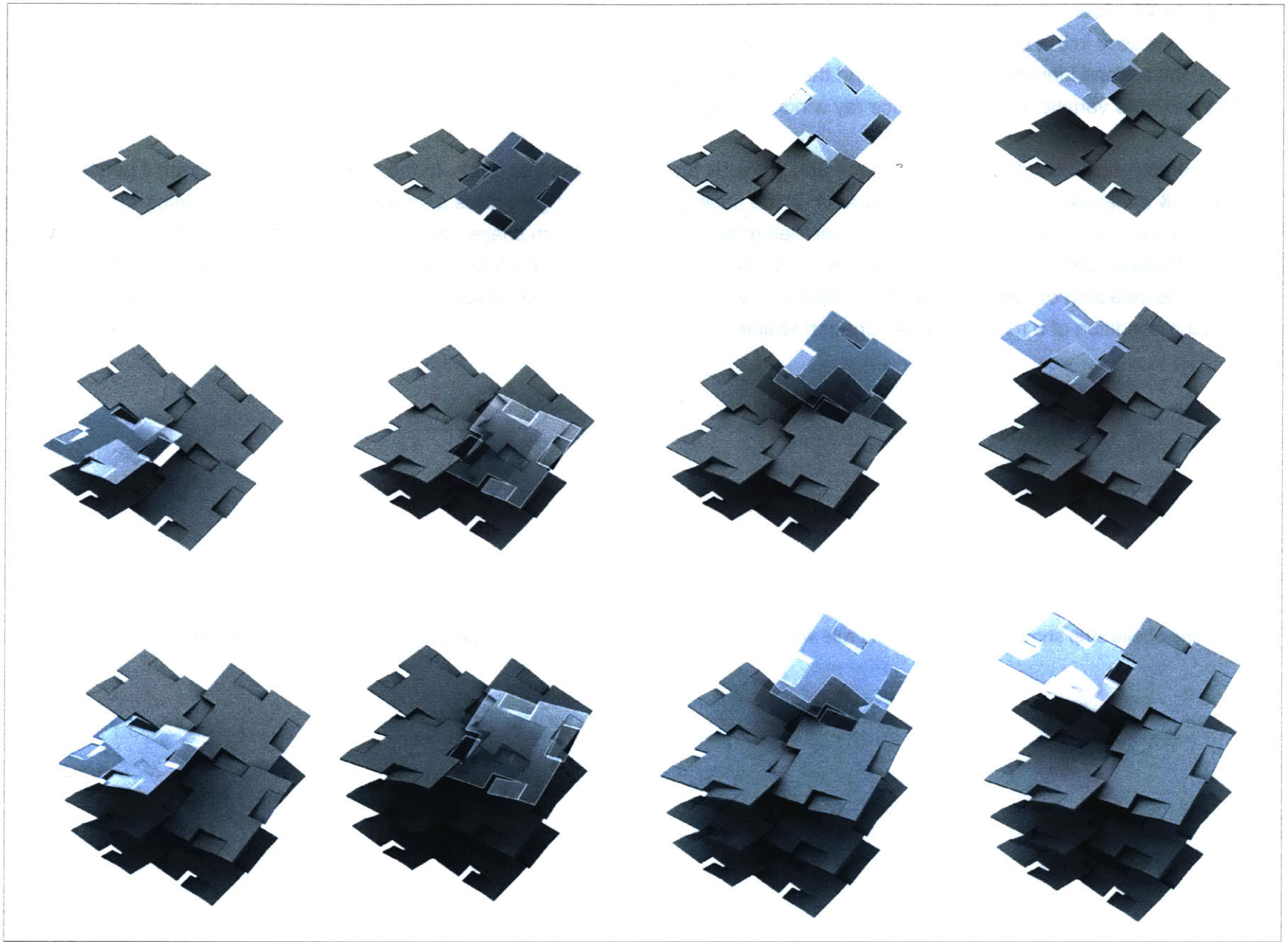
In shape grammars, solution space is almost infinite. However, it can be understood as a function of the number of recognized shapes X the number of possible replacing shapes X design steps (iterations). There are many other variables that I eliminated for simplicity. It is not my intention to exactly define the size of solution space as much as it is to suggest and highlight the drivers behind the ability to generate variations. Variables are mainly problem specific.

9.0 Experiments

9.1 Art Installations:

In this experiment shape grammars was used to generate designs for an art installation. The system utilized additive grammars where units build on top of each other growing A) weaved arms, or B) twisted columns. Labels were used to guide the growth patterns. This system was built as analogue where the designer handles selection and application of rules. One of the intentions behind this exploration was to integrate material properties like thickness or weight as one of the design parameters where rotations, connectivity, joints and aesthetics play a major role in arriving at a design. Below is an example on the weaved arms. Opposite is an example on the twisted columns.

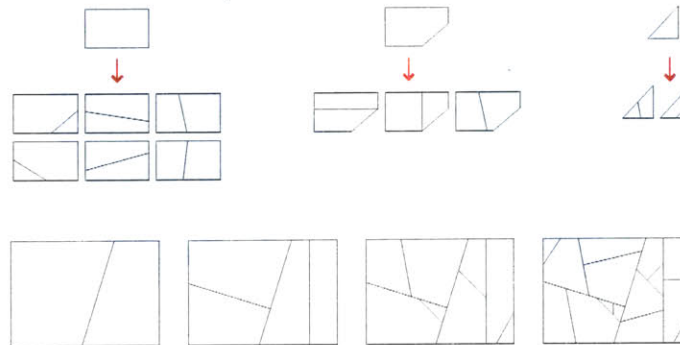




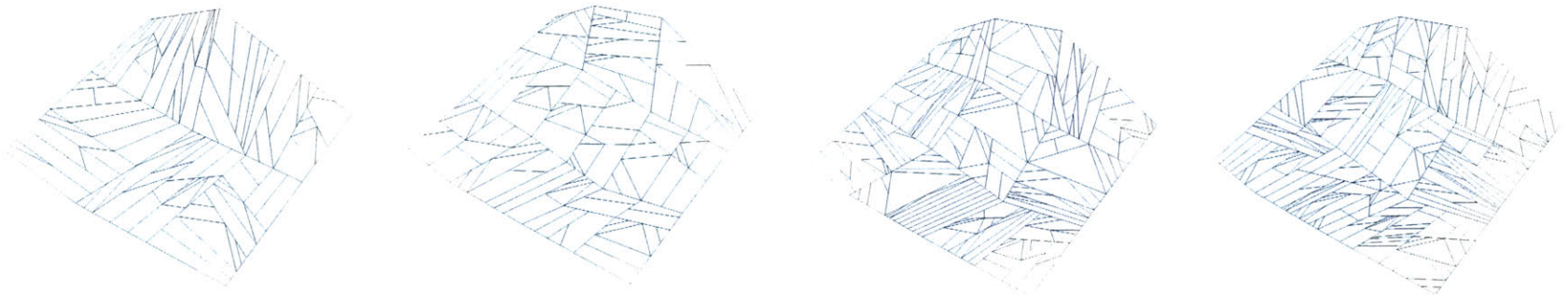
9.2 Ice-ray Windows

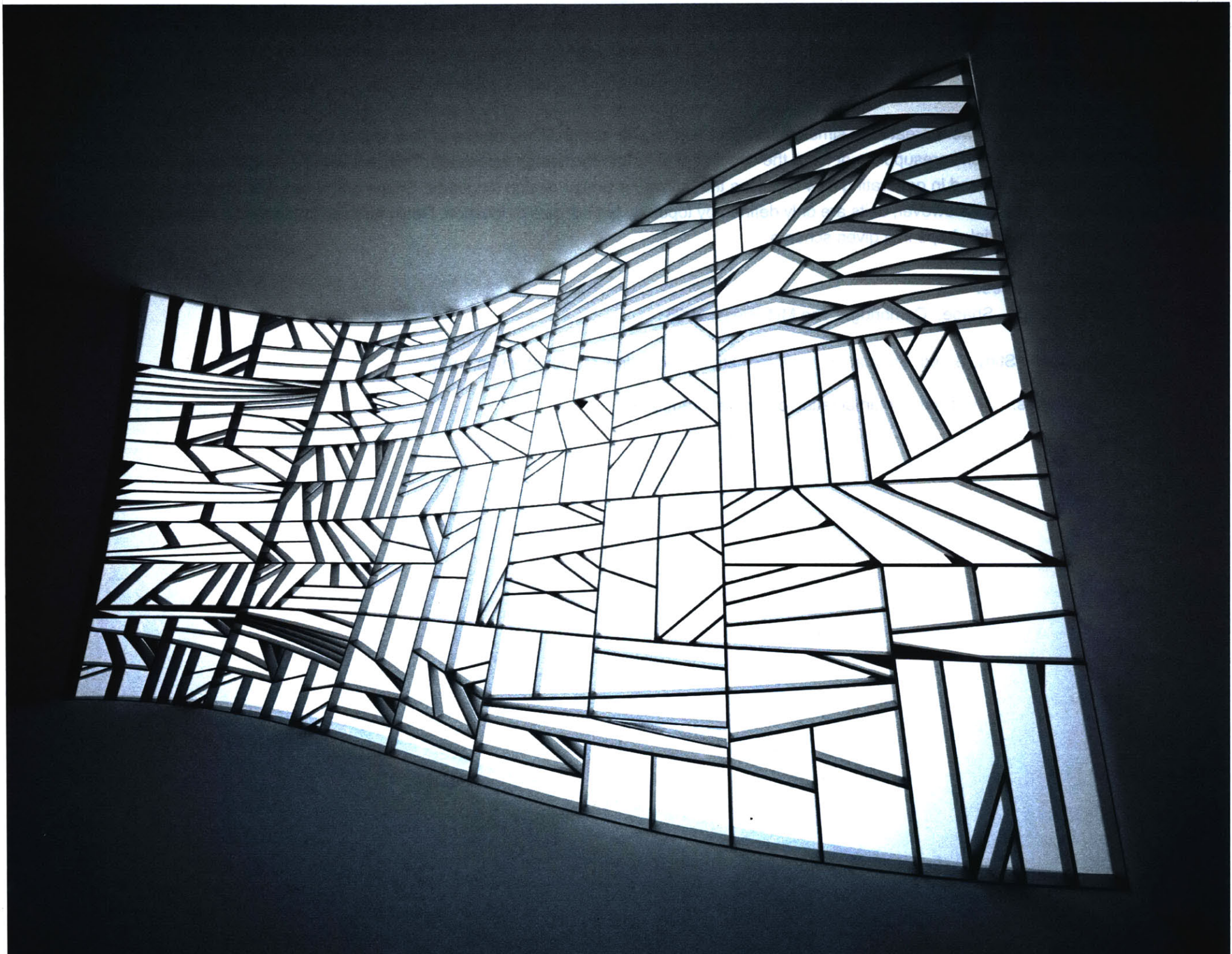
The systems used in this experiment included three components: a constructor, a recognizer and an initiator. The first component builds shapes; the second identifies their types, and the third picks rules then passes them to the generator. The goal behind this experiment was to generate variations of Chinese-Ice ray window designs.

The generator receives rules from the initiator and starts by dividing shapes into duals. In the case of a parallelogram it generates: A) a pentagon & a triangle, or B) two parallelograms. In the case of a triangle, it generates a smaller triangle & a parallelogram. In the case of a Pentagon, it generates: A) a triangle & a parallelogram. The fact that each shape generates a defined number of new shapes made recognition easy to implement. The recognizer identifies the topology of two at a time. Then it passes the information to the initiator that handles the selection of rules. This system requires the user to select an initial shape (a parallelogram, a triangle or a pentagon), and a stopping condition of a minimum area for each shape.



The above strip shows the replacement rule where the lower shows the calculation processes. Below are some generated variations.





10.0 Summary:

Shape grammars were created to simulate visual computing for design. They devise of how to see shapes instead of what shapes to see. Thus, they don't presuppose or force the concept of units into the design processes. They are usually used in reverse engineering existing designs; and in generating new solutions that convey a similar design language. Shape grammars, like other formalisms, rely on rules and units. However, units are only defined by topology but not size or location. Designers can implement a rule wherever they recognize its initial shape in a given schema.

11.0 References:

Stiny, George. Shape. Cambridge: The MIT Press, 2006.

Gips, J. and Stiny, G., ed. Freiman, C. V. Information Processing 71. Amsterdam, 1972.

Gips, James. "Computer Implementation of Shape Grammars."

3.0 A Provisional Taxonomy Of Generative Systems:

Earlier in the second chapter, I defined generative systems as structures composed of: Algorithms that are expressed in various representations, and capable of producing many results. I also defined formalisms as a context-sensitive type of generative systems for they work by rules. Then I showed a proposed model for integrating generative systems within a design process. Finally, I introduced a series of sections on: Algorithmic, Parametrics, L-systems, Cellular automata, Fractals and Shape grammars, where I highlighted shared and intrinsic properties to each system.

This chapter complements the previously covered material as it offers a summary table of generative systems properties and a discussion on relationships among them leading to a provisional taxonomy.

In the previous chapter you have read about various generative systems, their structures, components, units, etc. I believe it will be helpful to recapitulate these findings in a table where you can reflect on them side by side.

	Algorithms	Parametric	L-systems	Cellular Automata	Fractals	Shape Grammars
Context in Computing	Varies	Regeneration	Botanic Growth	Reproduction	Mathematical Monsters	Visual Calculation
Computation	Varies	Varies	Parallel	Sequential	Varies	Varies
Design Components	Generators/Interpreters Evaluators	Generators/Interpreters Evaluators	Generators/Interpreters Evaluators	Generators/Interpreters Evaluators	Initiators/Constructors Evaluators	Initiators/Constructors Evaluators
System Behavior	Varies	Possible Inheritance	Deterministic or Non Context Sensitive of Non	Periodic, Aperiodic Chaotic, Random	Recursive	Deterministic or Non
Rules Type	Varies	Varies	Replacement	Replacement	Replacement	Replacement
Units Type	Varies	Varies	Symbols	Symbols	Symbols, Numbers	Shapes
Smallest Units	Varies	Varies	Alphabet	Cell	Zero or points	Basic Elements
Means to Recognition	Varies	Varies	Counting	Counting	Counting	Seeing
Basic System Components	Tasks and units	Algorithms & Associations	Algorithms & Replacement Rules	Algorithms & Replacement Rules	Algorithms & Replacement Rules	Algorithms & Replacement Rules
Inheritance	Continuous	Continuous	Discrete	Discrete	Discrete	Continuous
Constraints	Relationships	Relationships	Adjacent Letters	States, Neighbor, Location	Location, Stopping Conditions	Color, Labels, Axes
Solution Space	Units, Tasks, Relationships	Parameters, Variables, Relationships	Rules, Generations Replacing Alphabets	Rules, Time Steps Cells, Neighborhood Size	Rules, Stopping Conditions,	Rules, Recognized Shapes, Replacing Shapes

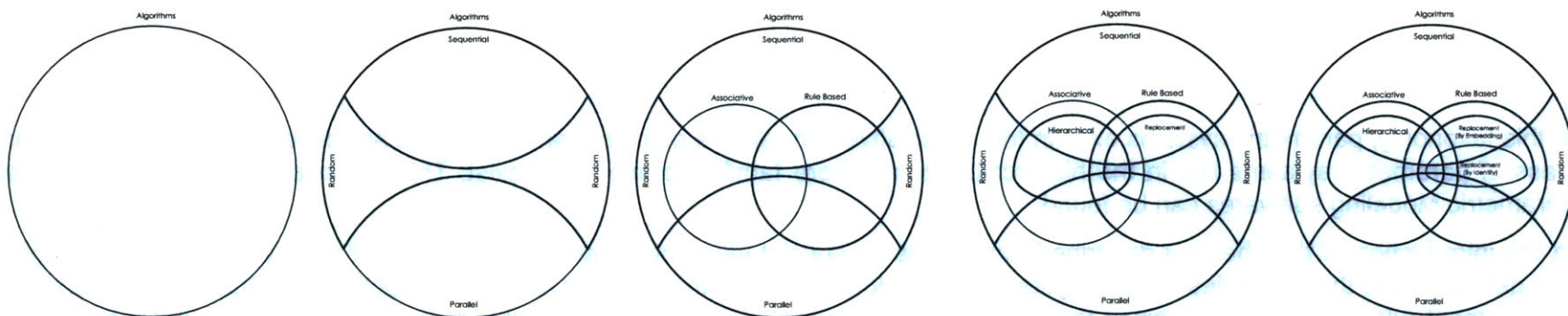
From the previous discussions and the above table, you may have realized already that all generative systems build on top of one core component, algorithms. They add different layers of properties devising distinct categories. These layers may also overlap highlighting finer classes within a category. In other words, every generative system is an algorithmic system.

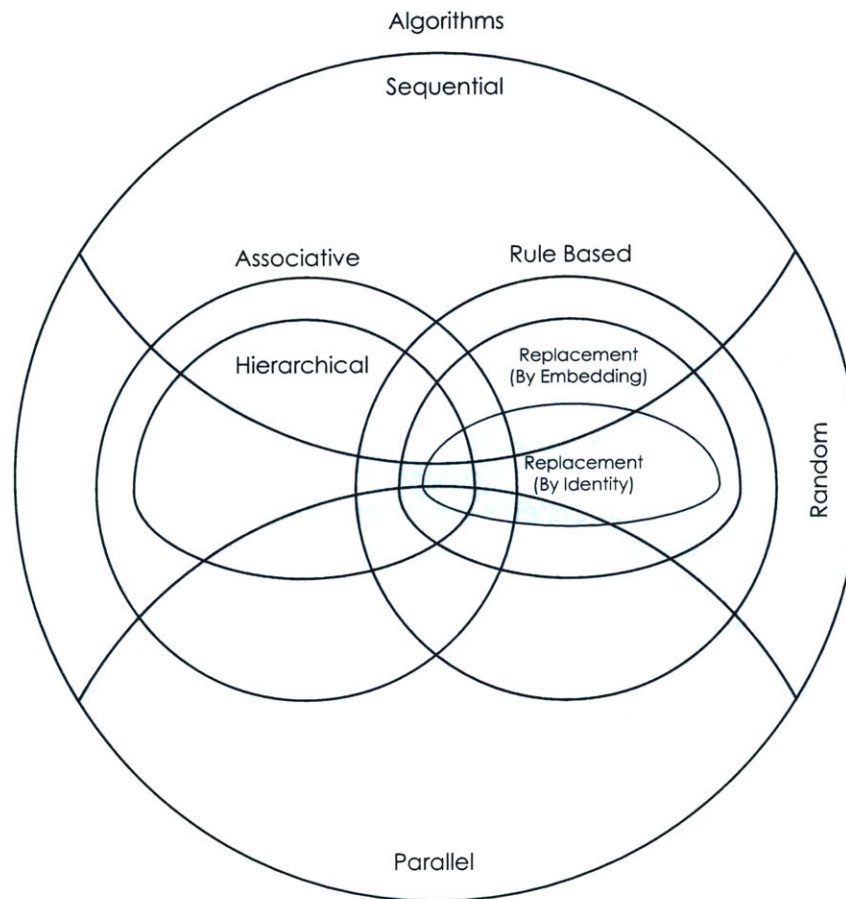
We found earlier that Algorithmic systems may run in a Parallel, and/or a sequential, and/or random fashion. Parallel Algorithms deal with elements simultaneously and separately without affecting each other. Sequential Algorithms deal with elements one by one such that a predecessor affects its successor (this was explained in the previous chapter). Random algorithms do not follow any specific order.

We also found earlier that some systems included associations (relationships) such as linking parameters to one another, or an action to a series of actions, etc. Associations are usually viewed as constraints for they impose definitions of how a system behaves.

A special type of associative-algorithmic-systems is the Hierarchical systems (explained earlier in section 2.2.2). In this type, elements are arranged within a ranking structure. A (higher) main element will control a (lower) sub element. This arrangement is usually visualized as a family of components. Hierarchical systems allow for only one-directional associations where “Parents” drive “children” and not vice versa. Non-Hierarchical systems, by their definition, do not possess any structure. So, every hierarchical system is an associative system, and not every associative system is hierarchical.

Whether associative or not, hierarchical or not, some systems are context-sensitive (Rule-based). This property is usually expressed in the form of “if...then” statements. Rule-based systems include functions or tasks that only work if a defined context (condition) is matched. There are many types of rules like interpretation, transformation, deformation, replacement, etc. In general, any rule can be written as a replacement rule. This type of rules can handle many algorithms in its representation for it includes: a left side (input), an operation (algorithm), and a right side (output). The first defines a condition to match (element to replace); the second contains descriptions of tasks; and the third describes the result (replacing element). Below is a series of diagrams showing various layers of the above-mentioned properties.





In this taxonomy, parametric systems exist in the associative zone where elements and behaviors are driven by relationships. A finer class of parametric systems lies in the hierarchical zone where families of elements exist. This is the zone where Animation and parametric Modeling softwares can be found.

In the introduction of this thesis, I described formalisms as special types of generative systems that work with rules. In the light of this definition and the properties of each system, we can locate: L-systems, Cellular Automata, Fractals and Shape Grammars in the replacement-rule-based category since they work by replacement rules. It is also possible to locate a finer class in the associative-replacement-rule-based algorithms zone like: Parametric L-systems, Parametric CA, Parametric Fractals and Parametric Shape Grammars. In the following paragraphs, I will use “formalism” instead of “rule-based-algorithmic-system”.

These formalisms lie within specific locations in the Replacement-Rules zone based on how they run. For example, L-systems are

located in the intersection of parallel application and replacement rules. CA is located in the sequential one. Fractals and Shape grammars can be located in Parallel, Sequential or Random zones.

If we look at these replacement-formalisms from a shape grammar point of view, we can circle a finer sub category based on how replacement rules identify elements. For example L-systems, Cellular automata and fractals operate on Identity, where Shape grammars operate on identity and embedding. Thus, they can pick explicitly defined or embedded elements. This makes shape grammars a more general class of replacement-rules for it includes all the other formalisms.

Here is a question for you. Is it possible to build hierarchical-replacement-L-system? I actually discussed this issue briefly in each section in chapter 2, specifically, in the fifth segment on (inheritance, associativity and constraints).

Let's reiterate a few concepts. A hierarchical system has a specific type of structure where elements are arranged in ranks, mainly families. In a hierarchy, there is a top and a bottom. Parents can inform, drive, update children but not vice versa. This might happen in the form of rules driving rules, or objects supporting others, or relationships updating others, etc. In L-systems, or any system, a hierarchy only channels the flow of information in one direction. It controls the context in which a rule can be triggered, but not how it works. Thus, any system can be built with a hierarchy.

Here is another question. Do hierarchical-associative-replacement-formalisms have inheritance?

Hierarchy allows for inheritance where attributes reappear in many generations as they flow from parents to children and not vice versa. For inheritance to take place, there needs to be a common placeholder, or carrier, or (gene) existing in all generations. In general, a Replacement-formalism works by "replacing" elements with "different" ones. Thus, there is always a possible discontinuity where elements might disappear. This is especially true for L-systems, Cellular Automata and Fractals because they deal with symbols as discrete units. These formalisms replace whole elements every time rules are applied. However, this is not the case with Shape Grammars.

As I stated earlier, Shape Grammars finds elements by identity (whole elements) or by embedding (part elements). Thus, replacement rules may replace a whole element or a part of it while keeping the rest for the following generation. This makes it possible for inheritance to exist in shape grammars.

In fact, Palladian Grammars are inheriting-hierarchical-formalisms. They are structured in ranks where top-down rules drive bottom-up rules (hierarchical). They also allow for continuity as they can replace parts of shapes (inheritance). For example, you start by building a skeleton, then build a series of walls on top, then find rules to carve windows and doors inside walls, etc.

It is important to note that the previously shown taxonomy is provisional. It repackages systems' properties only discussed in this thesis. My goal was to propose a departure point for others to situate these systems in relation to one another, read their potentials and constraints, maybe create new types of systems, and hopefully edit or arrive at totally new diagrams.

4.0 Conclusion:

Most of the current implementations of generative systems in architecture are skin-deep. They are usually geared towards solving simplistic problems, or designing in confined contexts. This can be explained by many reasons, some of which are discussed below.

As shown earlier, algorithmic and parametric systems offer less constrained environments for design since they do not utilize any specific structure or rule or representation. However, in order for these systems to be applicable to a certain problem, they needed to be customized to provide more specific, and less generic, contributions. The downside to this being that the more they are tailored to a specific context, the less useful they may be for others. A standard generative system is not possible.

Contrastingly, formalisms like L-systems, Cellular Automata, or Fractals were tailored for specific contexts and purposes. Thus, they have their own inherent limitations like: structures, rules sets, and representations. Such limitations make it hard to appropriate these formalisms in design. For example, L-systems are based on parallel replacement of symbols only, which is one approach for synthesizing solutions that may not always match the complexity of architectural problems. Cellular automata are sequential systems, thus they are hard to control or guide. Most of their behavior is periodic or random, which in many cases is not applicable to design processes. Fractals systems are also hard to impose on any design context for they are (purely) recursive. To a greater extent, those formalisms display their behavior graphically, which some architects took at face value. For example, L-systems are usually used to generate branching or tree-like geometric shapes, where most of Cellular Automata and Fractals systems are used to generate geometric patterns.

While the implementation of these formalisms is fairly possible by computers, shape grammars impose a hardedge problem, which is recognition. Shape grammars deals with units in terms of topology and embedding where computers understand them as discrete numbers only. The contrast between these two opposing worlds is hard to blend. It requires very sophisticated algorithms to simulate and embed artificial intelligence in computers so that they can pick any shape. Even if were implemented by designers themselves, Shape grammars require very detailed descriptions to carry a full design from start to finish. The formalism allows designers to pick any shape, anywhere, or apply any rule, or even create new ones. Such a universe of possibilities is hard to select from without detailed descriptions of how and what to apply rules to.

The design process itself can be found to be another reason that many of these systems are implemented in a very superficial way. Architectural design problems are non-linear, and decision-making is always driven by many factors simultaneously. In most cases, a design problem slowly gains its definition over the course of the design process. Thus building certain generative systems specific to ill-defined problems becomes time-consuming and in many cases unfeasible.

Finally, most of the implementations of generative systems in architecture are being channeled towards synthesis of forms without the guidance of thorough performance-based criteria. Thus, it is possible for a generative system to create variations of unfit design candidates for it cannot detect or evaluate deficiencies.

Current implementations of generative systems in architecture provoke many questions about their future. Will they continue to be used? Should we seek deeper integration methods within our design processes? Are we going to create new ones? Since design problems seldom present themselves in the same fashion, customization of generative systems becomes inevitable. I believe this requires us (architects) to gain higher levels of mastery in building analysis methods, performance-based evaluation criteria and robust synthesis processes.

5.0 Bibliography:

- Alexander, Christopher. A Pattern Language: Towns, Buildings, Construction. USA: Oxford University Press, 1977.
- Artificial Intelligence for Engineering Design, Analysis and Manufacturing, 20, Cambridge University Press, 2006.
- Coello Coello, C.A. and Cortés N.C, "Solving Multi-Objective Optimization Problems Using an Artificial Immune System," Genetic Programming and Evolvable Machines June 2005, vol. 6, no. 2: 163–190.
- Eisenman, Peter. "House X," Rizzoli 15 April 1983.
- Kalay, Yehuda E. Architecture's New Media: Principles, Theories, and Methods of Computer-Aided Design. Cambridge: The MIT Press, 2004.
- Koder, Selim. "Interview with Peter Eisenman." Ars Electronica 1992.
- Krishnamurti, Ramesh. Artificial Intelligence for Engineering Design, Analysis and Manufacturing, 20, (Cambridge University Press, 2006), 95ñ103.
- Gips, J. and Stiny, George, ed, Freiman C. V. "Shape Grammars and the Generative Specification of Painting and Sculpture." Information Processing 71. Amsterdam, 1972.
- March, Lionel. The Architecture of Form. Cambridge: Cambridge University Press, 1976.
- Mandelbrot. Fractals and Chaos: The Mandelbrot Set and Beyond. 1st edition, Springer; 2004.
- Mitchell, William J. The Logic of Architecture: Design, Computation, and Cognition. Cambridge: The MIT Press, 1990.
- Roth, Leland M. Understanding Architecture: Its Elements, History, and Meaning. HarperCollins Publishers 1993.
- Simon, Herbert. Sciences of the Artificial, 3rd edition. Cambridge: The MIT Press, 1996.
- Stiny, George. "Production Systems and Grammars: A Uniform Characterization," Environment and Planning 1980, B, volume 7: 399-408.
- Stiny, George and Mitchell, W. J. "The Palladian Grammar." Environment and Planning B: Planning and Design 1978.

Sullivan, Louis, ed. Menocal, Narciso G. and Twombly, Robert. The Poetry of Architecture. 1st edition, W. W. Norton & Company, 2000.

Turing, A. Undecidable p. 118; footnote Davis 2000.

Wolfram, Stephen. A New Kind of Science. Wolfram Media, 2002.

