Rulebuilding
Exploring Design Worlds through End-User Programming

by Yanni Loukissas

Bachelor of Architecture (1999)
Cornell University

Submitted to the Department of Architecture
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Architecture Studies
at the Massachusetts Institute of Technology
June 2003

© 2003 Yanni Loukissas. All rights Reserved

The author hereby grants to MIT permission to reproduce
and to distribute publicly paper and electronic copies
of the thesis document in whole or part.

Signature of Author:

Certified by:

Accepted by:
Rulebuilding
Exploring Design Worlds through End-User Programming

by Yanni Loukissas

Submitted to the Department of Architecture on May 22, 2003 in Partial Fulfillment of the Requirements for the Degree of Master of Science in Architecture Studies

Abstract:

Compositional rules have been proposed as the generating mechanisms of form in architectural treatises since Vitruvius. Recently, computational approaches to architecture have pursued a language for rulebuilding rather than the rules themselves. However, architects have resisted adopting computation as a means of expression, presumably because of the embedded culture of two-dimensional representations. A recent change in the construction industry from manual to automated fabrication techniques suggests a parallel shift in architectural representation from drawings to procedural descriptions of design. As such, computation can help architects to relate creative design to a process of manufacturing and assembly. In order to accommodate this, it is necessary to develop a new vocabulary for describing compositional rules which relies on an understanding of both design process and products as computational objects.
Acknowledgements

I would like express thanks to all those who have supported my work over the past two years.

To my thesis advisors, Bill Porter and Larry Sass for their wisdom and enthusiasm.
To my reader, Ann Pendleton-Jullian for her unwavering support.
To the Center for Bits and Atoms which funded my research position with Larry Sass.
To John Maeda, for showing me that computation can be beautiful.
To Sherry Turkle and the Initiative for Technology and Self for raising the intellectual stakes with respect to technology.
To Peter Testa for encouraging me to come to MIT.
To Gehry Partners and Foster and Partners for their openness.
To my fellow SMArchS students (James, Javier, Lieza, and Stelios)
To my parents, my sister and Macksi for always believing in me.
TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION
1.1 Context
1.2 Methods
   1.2.1 End User Programming
   1.2.2 Digital Fabrication
   1.2.3 Observation

CHAPTER 2: EXPLORATION
2.1 Exercises
   2.1.1 Shape Computation
   2.1.2 Symbolic Computation
2.2 Education
2.3 Practice
   2.3.1 Gehry Partners
   2.3.2 Foster and Partners

CHAPTER 3: DISCUSSIONS
3.1 Design Rules
   3.1.1 Implicit Rules
   3.1.2 Explicit Rules
   3.1.3 Meta-Design
   3.1.4 Rationalization
3.2 Representations
   3.2.1 Process Objects
   3.2.2 Physical Objects
   3.2.3 Assemblies
3.3 Design Types
   3.3.1 Operators
   3.3.2 Constraints
3.4 Control Mechanisms
   3.4.1 Explicit
   3.4.2 Random
3.4.3 Pattern
3.4.4 Numeric
3.4.5 Gesture
3.4.6 Environment
3.4.7 Control Combinations
3.4.8 Rules without Control
3.5 Variation
3.5.1 Topology and Geometry
3.5.2 Optimization
3.5.3 Paradigm Shifting
3.6 Evaluation
  3.6.1 Performance Evaluation

CHAPTER 4: CONCLUSION

APPENDIX
Appendix A: Examples in Rhinoscripting
  A.1 Surface Tiles
  A.2 Light Boxes
  A.3 Facade Strips
  A.4 Surface Reflectors
  A.5 Light Catchers
  A.6 Unfold Surface
  A.7 Join Strip
Appendix B: Projects by Foster and Partners
  B.1 SwissRE
  B.2 The Greater London Authority
Appendix C: Sample Rhinoscript
CHAPTER 1: INTRODUCTION

1.1 Context

In 1987, The Art of Computer Graphics Programming: A Structured Introduction for Architects and Designers, proposed the procedural structure of computer programming as a generative mechanism for architectural design. This book followed a tradition of treatises since Vitruvius which establish underlying compositional rules for architectural design. The Art of Computer Graphics Programming described a language for rulebuilding rather than the compositional rules themselves. Architects have resisted adopting computation as a means of expression, presumably because of the embedded culture of two-dimensional representations. However, a recent change in the construction industry from manual to automated fabrication techniques suggests a parallel shift in architectural representation from drawings to procedural descriptions of design. As such, computation can help architects to relate creative design to a process of manufacturing and assembly. This thesis describes a new vocabulary for discussing compositional rules which relies on an understanding of both design process and products as computational objects.

The methods outlined here are derived from experiences in three related contexts: 1. a personal exploration into the physical expression of computational design ideas; 2. involvement as researcher and instructor in a computational design workshop at

---

the Massachusetts Institute of Technology (MIT); 3. discussions with technologists and designers in architectural practice. This thesis discusses computational issues raised in these three contexts through a framework of themes including the nature of design rules, the structure of computational representations, conceptual design types, control mechanisms, variation, and evaluation.

1.2 Methods

1.2.1 End User Programming
An end-user program (EUP) or script is a set of instructions written in computer code and executed within a specific software environment. Scripting languages are written using the same basic structures as full-fledged programming languages; variables, loops, conditionals, and functions. Conventional programs can run independently because they are compiled or translated into machine code (0’s and 1’s) at ‘design time,’ the time when they are written.\(^2\) This process produces an executable file which can be continually loaded without recompiling. A script always has to compile at run time. That means that a script executes more slowly than a conventional program. In addition, scripts can make use of functions already coded into the parent software environment. However, scripting the basic functions improves the speed and control of scripts.

Scripting languages enable one to encode new functionality into an existing program, as opposed to creating new software from scratch. There are limitations to what can be scripted in any given software environment. However, scripting provides just

\(^2\) Clark, Susanne et. al. 1999. VBScrip\text{t} Progr\text{ame}\text{'s} Reference. Birmingham, UK: Wrox Press Ltd.
the right amount of access to underlying structures which allows one to develop personal and project specific tools. The personal and student explorations described in this document were written in RhinoScript, the scripting language of Rhinoceros, a 3D modeling environment. RhinoScript is based on the Visual Basic Programming Language developed by Microsoft. Nevertheless, the techniques described here can be applied to architectural design using any end-user or conventional programming language.

1.2.2 Digital Fabrication

This exploration has been guided by an intention to design physical objects through procedural programming. It has been informed by multiple ways of fabricating physical objects from digital instructions, including 2D cutting techniques (laser, water jet), additive 3D methods (prototyping) and subtractive 3D methods (milling). The majority of the personal examples were developed using a consistent means of 3D additive fabrication using a Z-Corp 3D printer. The Z-Corp builds up solid models from many layers of bonded plaster. This machine houses two columns of plaster, a feed and a build, in side by side containers. Before printing, the feed container is filled below the surface of the machine with enough plaster to build a solid plaster block the height and width of the given model. The build container starts out nearly flush with the top of the machine. As the machine runs, the feed column rises and a mechanical arm spreads a thin layer of plaster from the freed across the surface of the build column. The cross section of the 3D model is printed onto this thin layer in bonding liquid. The build column is then depressed slightly to allow another layer of plaster to be spread on top. The feed container rises again and the process repeats. As subsequent layers are shifted from feed to build and the next cross section of plaster is bonded to the previous one, a solid
model of bonded plaster is slowly assembled out of thin slices, surrounded in the build container by loose plaster.

1.2.3 Observation

My work has been structured around a series of workshops at the Massachusetts Institute of Technology focused on integrating considerations about fabrication into the conception of designs through the use of digital technology. These workshops have often drawn expertise in this area from collaborations with well known design firms. Gehry Partners was involved when I first started examining these workshops in the spring of 2002. I conducted an ethnographic study of the Gehry workshop entitled Digital Mockups which brought together a wide range of technological tools from on-line collaborative systems to parametric modeling and rapid prototyping in order to tease out the central issues of digitally driven fabrication machinery. I observed another workshop peripherally in the fall which paired digital tools with a specific construction material, ceramics. From these two workshops, I learned that when students learn to fabricate objects from computational instructions which encode constraints, they learn about the implicit way in which all media, virtual and physical, are constrained.

While writing this thesis, I have participated as research staff, teaching assistant, and ethnographer on the most recent in this series of courses, entitled Generative and Parametric Tools for Design and Fabrication. This course invited Foster and Partners as consultants to explore constraint-based modeling and scripting as opposing methods for architectural design and fabrication. As research staff, I have sought to develop teachable methods for fabricating models from scripts. As a teacher I have tried to coach students towards unique approaches to these media. Lastly, as an ethnographer I have tried to absorb the range of approaches that students and instructors contribute to the
exploration of these media. My ethnographic observations stem from in class observation and personal interviews with students, academics and professionals. Throughout the text, the use of quotes from these interactions will inform the way that computational objects are discussed.
CHAPTER 2: EXPLORATION

2.1 Exercises

2.1.1 Shape Computation

![Figure 4: SGTools Variations, Fall 2003.](image)

This exercise in rulebuilding began within the framework of an extension to AutoCAD called *Shape Grammar Tools*\(^3\) (SGTools). This program defines design rules in terms of shape transformations. It is one of several software implementations of Shape Grammars\(^4\). In SGTools, rules are described by example. The user draws a before-and-after representation of each transformation rule. Multiple rules can be stored and later called into use.

SGTools allows users to make three-dimensional shape patterns quite easily from visual rules. Rules can be cumulative. The user can take the entirety of what is created with one set of rules and act on it with another set. It is also possible to backtrack and redeploy old rules differently. However, rule making is limited to defining the local relationships between shapes. As such, it is not possible to define rules that address the design in a holistic way. In addition, rules cannot be generalized and cannot be used

---

\(^3\) SGTools was developed by Luis Romeo, PhD student in Design and Computation at the Massachusetts Institute of Technology, 2002.

\(^4\) Shape Grammars were developed by George Stiny and James Gips
to allow variation within the forms produced by a single rule. Because of the visual way in which rules are recorded, there is no record of the design history which underlies all transformations. All these factors made this environment highly constrained and suited to a particular family of designs (spirals, domes, etc.) that can be generated from a linearly repeating rule.

2.1.2 Symbolic Computation

I turned to the use of programming, at first with the intention of making my own tool, similar in scope to SGTools but with a different underlying rule making methodology. However, I realized that the limitations of SGTools would underlie any specialized tool. Like a jeweler who crafts the instrument for the cut, designers revise their work process with each new design brief. Architecture is resistant to generic methodologies. It is an "ill-defined problem"5 because each project has unique circumstances. Every architectural problem is an opportunity to rethink the design process.

I have developed many examples of concept specific computational instruments defined and encoded in scripts. These instruments were developed in an exploratory manner as a way of clearing paths for future work. As such, many of the scripts that I produced had an intention to be educational. During the process of developing these scripts, I sought to bridge gestures, rules, and fabrication as distinct but relatable ways of making. I have placed an emphasis on varying control systems in scripts. The full description of each script can be found in the appendix. From this set of examples I hope to give some insight into the characteristics of computational, explicit rule-making as part of design.

---

A.1 Surface Tiles
Input: Two profile curves
Output: A tiled surface with ribs from two profile curves.

A.2 Light Boxes
Input: None
Output: A grid of light modulating boxes.

A.3 Facade Strips
Input: Number (Minimum Surface Area)
Output: A light modulating surface of planar strips.

A.4 Surface Reflectors
Input: Two profile curves, light ray, point
Output: Small surfaces that reflect incoming light towards a single point.

A.5 Light Catcher
Input: Two profile curves, light ray
Output: A surface which funnels light into a space from one direction.

A.6 Unfold
Input: Two profile curves
Output: Unfolded planes from a faceted curved surface.

A.7 Joint
Input: One profile curve
Output: A series of independent flat panels connected with dovetail joints.
2.2 Education

Workshops in the Department of Architecture at MIT offer space for students to concentrate on specialized issues while retaining an exploratory perspective. Underlying this is a tension between the rationalist paradigm of technical innovation and the constructivist meandering path of design. Axel Killian, Terry Knight and Larry Sass were the lead instructors in the *Parametric and Generative Tools for Design and Fabrication* workshop. Carlos Barrios and myself were research assistants and specialized instructors for CATIA and Rhinoscripting respectively. As part of the workshop, I had the opportunity to work closely with both instructors and students in making sense of computation as both process and product.

The students were introduced to three digital creation platforms; CATIA, Rhinoscripting, and Ecotect. CATIA is a 3D modeling program developed to help the aerospace industry precisely design and fabricate aircraft. In contrast to the other 3D modeling programs, CATIA is parametric and gives priority to the relationships between objects rather than the objects themselves. Rhinoscripting, as mentioned in the *introduction*, is a form of end-user programming within the Rhinoceros 3D modeling environment. Lastly, Ecotect is a platform for simulating the effects of environmental factors on architecture. Learning to work effectively with these three tools meant a steep learning curve for the students. However, one of the main objectives of this course was to provide an arena in which students were not expected to master all of these environments but could develop knowledge through personal exploration. Students took independent paths of development in which they encountered unique problems. Throughout the semester, we
worked together to establish methods of generalization for discussing all the projects.

The students in the workshop were all graduate students in the Department of Architecture. They had prerequisite knowledge of 3D modeling. However, only one student had extensive experience with programming. Members of the architectural office, Foster and Partners, were also involved as organizers, critics of student work, and as a model of architectural practice. We visited their office at the beginning of the semester and met with them several times using a video conference system.

Students were given a brief introduction to programming but very few step-by-step tutorial sessions. We sought to avoid turning the class into a programming course by requiring the students to learn a whole set of techniques up front. Instead, our intention was to allow students to approach the material through three hands-on projects in which they were to learn techniques as needed. We provided them with examples that could be read and expanded from by experienced programmers or modified in simple but important ways by novices.

The first assignment was an introduction to all the tools in the context of a study on lighting. Students were asked to produce a sequence of varying lighting conditions inside a small box (enclosed but for one side) by making three different light mediating membranes for the open face of the box. Students were asked to choose a pair of attributes like open / closed, solid / scattered through which to explore a range of possible conditions. Students worked with three basic architectural devices; reflectors, diffusers and absorbers. As mentioned above, the students were provided with example scripts including the light boxes script, the facade strip script and the
surface reflectors script⁶. Variations of the final membranes were prototyped using the Z-Corp printer.

In the second project students designed and made various prototypes for a pavilion on the grounds of the Museum of Fine Arts in Downtown Boston. Students were again asked to deal with lighting issues. Many of them applied their initial projects as surface making techniques in the generation of full enclosures for the pavilion.

The third project of the workshop focused on developing construction details for the pavilion. Students were asked to work at a large scale. A reliance on z-corp prints gave way to more complex assemblies from laser cut components. Some models were fabricated using a combination of components from the 3D printer and the laser cutter.

Based on class discussions and interviews conducted individually with students, I was able to understand the range of ways in which students developed scripts in relation to other means of design investigation. There was a give and take between the students and the media. Student development of scripts mirrored design in more conventional media. They revisited and revised scripts while continually shifting design criteria.

*Workshop student:* “I made a lot of sketches. The first one, because I didn’t know the scripting. I didn’t know where to start, I want this cell to do this, You do a stupid relationship thing... you make little things that you can’t use later. So I go back to sketching and rewrite it. Later, you know which part to write first so that it can expand more... yeah back and forth...”

⁶ See sections A.2, A.3, and A.4 in Appendix.
Many times during the process, students reformulated their design problems in response to what they were able to generate through scripts.

Much of the work completed by the students was done with heavy support from instructors in the class. Some of students did not have a complete understanding of what they produced in the first exercise. At the beginning of the second exercise, students found themselves with scripts that they could not manipulate effectively. They worked together to develop generalized Rhinoscripts which they could trade with each other and reuse in different conditions.

Workshop student: “The first thing I did basically was copy one of (another student’s) codes. (name of student) had a matrix of cubes and once you know how to make a matrix you can make a whole set - like you can make a circle in the matrix”

Students who spent the majority of their time working on one script rather than developing several study scripts were in danger of being locked into an initial strategy. Sometimes students modeled the form that they wanted to script as a precursor to writing code.

Workshop student: A lot of times I would draw it in Rhino using the tools and then work backwards... and I borrowed a lot of things from peoples scripts. I would test out their scripts and then pirate pieces.

There is typically development of implicit to explicit representations within architectural practice. However, the students in the workshop were prompted to start with explicit
rules and work backwards towards more ambiguous forms that could be evaluated and interpreted as architecture.

“Our designers usually build physical models then build a digital model and finally use rapid prototyping. We have asked you to do the reverse process. We were trying to set up a process where students went in reverse.”

Students found that the path to developing scripts was not linear. For many students, scripting involved more pre-planning than they were used to.

Workshop student: “I have to be clear of the system that I want. It just made me think more in the beginning. Usually, I do a lot of stuff that is unnecessary to get to that point. If I want to go from point A to point B I don’t see the map so I have to take many passes to get there. It’s like, okay, I have to see the map first (when scripting)”

Students used a combination of writing, sketching, discussion, and sometimes modeling to work through their scripts.

Workshop student: “I had to do a lot of sketching. I had to draw it in 3D to understand where the coordinates are. I would draw what I want - the idea. I would draw the elevations of it so that I could figure out how to input the coordinates.

Q: Did you write in the coordinates?

---

Yes, otherwise it was too confusing for me to think about it in my head. It's like - I draw it in 3d space on the page I draw 0,0,3,0,2,0 once I got all those points fixed I would write the code on a piece of paper the basic geometry.”

Eventually the structure of scripting became the foundation for conceptual departure as students became familiar with the medium.

Workshop student: “It was like I want to make this object. How can I make it? As it started progressing, I got ideas from the pre-made functions.”

Although students designed scripts and produced prototypes, these techniques changed most students’ normal work habits. This educational experiment has revealed some of the limitations and affordances of scripting in the context of a focused design problem. These issues will be address in the discussion in chapter 3.

2.3 Practice

Computers have become preliminary construction sites in which architects must understand conventions, limitations and possibilities for innovation. They are the dominant method for storing, communicating and manipulating the data that is necessary to guide the construction of buildings. In the least technological practices, digital media allow architects to efficiently revise construction drawings and transfer documents to geographically remote clients, consultants and contractors. Technologically savvy architects are adopting techniques for
manufacturing architectural components straight from digital representations.

Frank Gehry Partners and Foster and Partners have achieved international recognition for constructing buildings of unprecedented complexity. Both of these offices now rely on advanced computational representations but were in practice long before they adopted computer technology. I have had the opportunity to visit both these offices and learn about how new technologies have been adopted in each environment. Both of these offices have participated in workshops at MIT.

The following section is a brief presentation of how technology has influenced the design process of Foster and Partners and Gehry Partners.

2.3.1 Gehry Partners

One could say that the major problem of Gehry Partners is the translation between explicit and implicit ways of making. In order to fabricate buildings of the scale and complexity produced by Gehry Partners, it is necessary to have an explicit, sharable, searchable model of the construction. However, Gehry’s process of design is driven almost exclusively by physical models. The transition between the physical models and the computational models is a difficult one. It is also one that happens many times within a single design process.

My understanding of Gehry’s process draws heavily from discussions that I have had with members of Gehry’s team as well as faculty and students at MIT who have worked with Gehry Partners. Throughout this investigation, I have been particularly interested in how media, digital and the physical are discussed in relation to one another. My central question in looking at this office has been “how does the computer frame the
way that physical media are understood and manipulated in design and construction?"

"Early in the process we bring in construction systems at a detailed level. We keep in close communication with the fabricators. The computer is a communication tool between these two sides of the process."  

The movement of physical to digital is not a simple unidirectional conversion. A typical work cycle at Gehry Partners starts with physical modeling. Physical objects are often built using materials which have some affinity to actual construction materials. These models are scanned into the computer as a cloud of points. Once in the machine, a number of computational constructs are developed in order to approximate the most important features of the point cloud. It is important to note that the point cloud is not directly turned into a digital model. Gehry Partners works with physical and digital representations in parallel. Through the use of varied computational techniques, technologists in the office try to develop rules of constructability which can be "fit over the design intent." After a suitable shape has been digitally formed, another physical model is extracted from the computer through laser cutting or printed templates for evaluation. Although this physical/digital cycle is split up into technical and design tasks, there is a considerable amount of feedback which propels the cycle to continue and develop.

2.3.2 Foster and Partners

The Specialist Modeling Group (SMG) is a small technical collective directed by Hugh Whitehead that works as an in-house consultant for Norman Foster’s office. SMG operates outside of

---

8 Dennis Sheldon. Director of Computing, Gehry Partners. Comments made at a Massachusetts Institute of Technology lecture, Spring, 2002.

9 Dennis Sheldon

Figure 11: Greater London Authority, Foster and Partners.
the generic structure of design teams in the office; tackling technical tasks deemed too complex to be handled by conventional means.

"Curved Buildings present a huge management problem. Small mistakes add up quickly to unacceptable tolerance violations."  

The SMG may start at the beginning of a project, help out for a short time, or bring a difficult process to resolution. Whitehead and his group are responsible for developing new methodologies which can guide other members of the office through complex tasks. SMG tries to present design teams with a package of tools and not a pre-packaged answer. Whitehead describes their task on each project as building an “option generator” through tools and methods that are appropriate to the situation. Usually the procedural nature of programming provides an ideal medium within which to build new tools. The group works on getting into the mindset of the designers and translating design intentions into computer code.

"Most designers already think programmatically but they don’t have the time or desire to program. We help designers to express programmatic thoughts without having to program"  

Whitehead and his group address the issue of process in the office. They combine close observation, analysis and technical skills in order to design new work paths for the office. They define their synthesis of form in terms of functional, spatial, sculptural, structural and environmental considerations. SMG work towards an understanding of the architectural design
process in the office as a means of developing computer based tools which can aid and, in some cases, replace traditional methods.

The Specialist Modeling Group at Foster's office often creates custom tools for use with Microstation as opposed to buying more specialized software. This allows them to develop more task-focused tools which are then easily usable by anyone in the office. This prevents the development of bottle neck processes which must pass through a limited and expert set of tools or people. For the SwissRE project, the Specialist modeling group built an adjustable model using scripting which could be controlled by specifying dimensions or manually manipulating the profile within a constrained geometry. The group had to work with designers to arrive at a set of parameters by which the form of the building was to be generated. The head designer didn’t like an initial idea for using the curvature of an ellipse so the group developed a method for defining a spiraling curve for the building’s profile derived from a complex of arcs (all adjustable in the scripted model). The script co-coordinated the floor plate drawings with the overall profile of the building. When the building profile was changed, floor plates would update automatically.

"The system helped us to formulate the shape out of geometric constraints. The form emerged out of this process."\textsuperscript{12}

This relational model allowed Foster and Partners to test many possible variations without having to redesign all the details for

\textsuperscript{12} Judit Kimpian, Member of the Specialist Modeling Group, Foster and Partners. Comments made at the Massachusetts Institute of Technology workshop, Generative and Parametric Tools for Design and Fabrication, Spring, 2003.
each. Unfortunately, it is not clear as to whether the relational model was used in the design process or as a parallel experiment.

"Sometimes I make macros by just recording a modeling process and tweaking it. I use a lot of shortcuts because many times I need the code by the next day." 13

Although specialized tools are useful, they require a lot of time and energy. SMG strives to achieve a balance between generalized and specialized tools. They try to make the tools more generic than they need to be. In addition, tools are often built in modules so that parts can be reused in the creation of later tools.

According to SMG, writing programs for design is not a clean, well understood process. Tools range from quick fixes to general solutions for common design problems. The first step in the process is often just observing the design teams. In the end, SMG must develop an effective workflow for the design teams which can be broken down into a series of operations, the most explicit of which are executed by computer code.

"From a mathematical perspective, a problem correctly stated is a problem solved." 14

Whitehead himself is quick to point out that this claim is only true in well-defined disciplines. A problem can only be correctly stated if there is some certainty about what the unknowns are. In a typical architectural problem there are countless variables related to site, client and program. The kind of optimization that this statement implies can only be calculated for a limited set of

14 Anonymous
variables. Any optimized solution necessarily does not account for some significant variables.

“A designer in the office told me that computation is for communication and sketching for thinking. Later on I thought, it is the exact opposite.”

This quote from Hugh Whitehead reveals that there is a lot of confusion about the nature computational representations. As SMG tries to meet deadlines and solve immediate technical problems, they are rewriting the office’s process of architectural design. Although it appears as if Whitehead is aware of some of these shifts, it is not clear whether the rest of the office is.

The following are some of the most salient changes that SMG has perceived. Firstly, the office is able to produce sets of building information with much smaller teams. This is an immediate change in the size and funding structure of design teams. One tool can do the work of many people laboring manually. Secondly, the office is able to do more precise simulations and arrive more easily at partially optimized solutions. Thirdly, the abstractions built by SMG allow designers in the office to manipulate form manually record dimensions / coordinates as they play. Fourth, when confronted with many variables, a parametric model can help designers to see the interference / intersection between them.

For SMG, dimension-driven design has forced them to think out geometry in first principles. A set of rules ends up taking precedence over the geometry. Making a set of rules forces SMG to define the rational behind designs - theirs and others’.

\[15\] Hugh Whitehead
"When preparing a parametric model, you are forced to think in a very structured way and define your design in ways that you don’t normally do." 16

Another source of anxiety in the office is “the apparent loss of tactile or action knowledge” in designing on a conventional computer. Despite the advantages of symbolic computation, the office still finds it important to build physical models. According to Whitehead this allows designers to take advantage of a unique hand/eye/brain coordination that is unattainable with current day computer applications. Working with physical media has other advantages as well. There is a considerable amount of overhead involved in developing a new digital tool. Imbedding a particular rationale in the tool can backfire when you find that you’ve “crystallized the rules too soon.” 17 Or worse, you may unnecessarily accept a set of self imposed limitations.

"Optimizing and sub-optimizing are not good for lateral thinking." 18

When the office does accept the task of building a constrained or customized design kit in the form of a scripted tool, a whole new set of considerations come into play. “Power is nothing without control. The generative process can produce a lot of useless stuff.” 19 Designing a useful control system for the tool is never easy. One has to arrive at a balance between the ease of programming the tool and the ease of use of the tool itself.

16 Judit Kimpian
17 Hugh Whitehead
18 Hugh Whitehead
19 Francis Aich
"A lot of things that we find out from the tools are common sense. We find ourselves running simulations that are really unnecessary." 20

In the end the tools do not give absolute answers. Sometimes the best thing these tools can do is to help the office pose more informed questions to their fabricators and consultants.

20 Judit Kimpian
CHAPTER 3: DISCUSSIONS

3.1 Design Rules

3.1.1 Implicit Rules

"Rules in design are largely implicit, overlapping, diverse, variously applied, contextually dependent, and subject to exceptions and critical modification."21

The act of designing a building can be understood as the establishment of order-inducing rules on an existing site. According to former MIT Professor Don Schon, designers formulate and test rules implicitly in the form of drawings and models.22 In this investigation, programming has served as an instrument through which to study the affect of using design rules which are purposefully external, unambiguous and perhaps capable of generating unexplored physical forms.

"Foster’s office was made famous by using optimization as an aesthetic. Now we actually have the tools to optimize."23

The conflict between implicit and explicit means of representing design concepts permeates this work. I do not presuppose the dominance of one technique over the other. I have tried to explore the translation from one to the other and in doing so expose the characteristics of both as means of expression. In my investigations, I have found that it is difficult for most architects to separate these two ways of working.

---

22 Schön (1988)
23 Judit Kimpian
Workshop student: “Programming helps you or forces you to make your rules concrete. I used rules in the past but they were more vague, or I didn’t realize that I was using them but I wouldn’t be able to understand the difference between how I make rules now and how I used to make rules unless I went back to my old way of working, and that would be very difficult after having programmed rules.”

3.1.2 Explicit Rules

Computation describes ‘how to’. It is the expression of logic from an imperative point of view. Scripting supports the structured development of form through repetitive procedures while allowing for variation through the adjustment of parameters on which those procedures rely. Computational ideas can be expressed as process objects through computer code and more recently as physical objects through rapid prototyping. This thesis explores the meeting between a way of thinking and a way of making which gives rise to an aesthetic of structure and variation.

Computational representations allow designers to define the conditions by which designs may be derived. This is a means of externalizing a process in an unambiguous way. However, transferring design ideas into computer code should not be misinterpreted as the recording of design knowledge. Knowledge about designs is a subjective construction that might be linked to a shared set of rules but ultimately evolves from the interaction of designers, clients, and consultants.

Designers who have collaborated using scripts have described this process as more transparent than collaborating through a conventional design representation. Conventional drawings
make the shape of a design explicit, but leave out a definition of how the shapes are derived and related.

*Workshop student: “My partner and I shared code.... It was easy to see what the other person was trying to do in code. You could see exactly what had been added and you could leave comments for each other.”*

3.1.3 *Meta-Design*

The process of coding design rules has been eloquently framed by Mark Burry\textsuperscript{24}, a visitor to the workshop. Burry calls this process “meta-design” or the “design of design”. “Meta-design is the process in which you map your route.”\textsuperscript{25} Burry notes that this is seen amongst many architects as the loss of something essential to design. “Actually”, he asserts “it can help architects to become more disciplined, a necessary step for many practitioners.” However, he also notes that in choosing the appropriate process for a design, the only options are not computational ones. “In many cases the most appropriate way to start is through sketches.”\textsuperscript{26} Computational design is just another medium of expression and representation that is being added to the list of approaches in offices like Foster and Partners.

In requiring this sort of preplanning, the computational mindset helps to frame the way that all media are used in the design process. Conventional tools like paper\textsuperscript{27} and pencil\textsuperscript{28} are describable in terms of the same rules which explicitly govern computational design tools. In watching students in the

\textsuperscript{24} Mark Burry. Professor of Innovation at The Royal Melbourne Institute of Technology (RMIT) in Australia. Comments made at the Massachusetts Institute of Technology workshop, Generative and Parametric Tools for Design and Fabrication, Spring, 2003.

\textsuperscript{25} Mark Burry

\textsuperscript{26} Mark Burry

\textsuperscript{27} Sheldon, Dennis R. 2002. *Digital Surface Representation and the Constructibility of Gehry’s Architecture*. Cambridge, MA: MIT.

\textsuperscript{28} For further description of the pencil as a computational device see the work of George Stiny.
workshop develop explicit representations through scripts, it was apparent that these highly structured means of expression can be tuned to design intentions. However, it also seems that computational structures significantly change the way that design happens. I asked students to discuss computational rules in comparison to the way they normally work.

Workshop student: “I think there are always rules. I think in the case of like, I can think of like a lot of projects where there have been iterative qualities. Even drawing pencil studies, it's like a series of spaces. An example, a theater study of the lighting quality and I did like four drawing act 1, intermission, act 2, the end. Same size sheet of paper size, same techniques, same plan of the space and then I would render it differently so I guess the rules are like... you set up rules for yourself. It's a more intuitive way, in some senses you don't have to spell them out, you don't have to write a computer program. In the case of doing a drawing study, or if you are building a model you make a triptych of models, same size, same use of materials but they deal with different vocabularies, or they deal with different variations on a theme.”

Workshop student: “In programming, rules always have to be mathematical, or about the geometry. This is not always true in rules outside of the machine.”

3.1.4 Rationalization
Hugh Whitehead from Foster and Partners defines two approaches for developing constructible designs through computational representations; “pre-rational” and “post-
rational." Post-rational is the approach most commonly associated with Gehry Partners. In a rigorously post-rational system, the formal design is conceived in a process that is for the most part divorced from considerations about construction. A construction system is then retroactively imposed on the design. Certain compromises inevitably have to happen in order for the design to conform to any systematic means of construction. The opposite process is a pre-rational system in which the construction system is defined before the design process happens. Design is constrained to happen within the limits of what is constructible under the adopted system. This system is extremely well controlled but can impose conceptual limitations. Although the original constraint-based system can be revisited and revised as much as its internal structure allows, developing assumptions about the limitations of a construction system too early can lead to an underdeveloped design.

Dennis Sheldon, director of computing at Gehry Partners, defines a similar set of approaches: "approximating a desired surface" is analogous to a post-rational process and "a pre-analytic surface" which is akin to a pre-rational design strategy. Sheldon argues that pre-analytic design worlds can be built computationally, like at Foster and Partners, or using physical materials. Sheldon has drawn behavioral analogies between paper models and buildings constructed of sheet material.

"There's nothing like a piece of paper. Paper is automatically pre-rationalized. Breaking the surface allows a double curvature to play out." 

---

29 Terms mentioned by Axel Kilian and referenced to Hugh Whitehead in a discussion at the Massachusetts Institute of Technology workshop, Digital Mockups. Spring, 2002.
30 Gehry's process of approximating a desired surface using digitizing techniques has been described in the Practice section of this thesis.
31 Sheldon (2002)
32 From a discussion with Dennis Sheldon at the Massachusetts Institute of Technology workshop, Digital Mockups. Spring, 2002.
Paper and sheet material have a similar ability to curve in one direction without an expensive stamping process. By producing physical models from paper, the designers can understand what is and isn’t buildable in the corresponding full scale materials.

3.2 Representations

Workshop student: “In the beginning I had to run script. I had no idea what they did. If someone would have just shown me the (generated) object, I wouldn’t read into it in terms of variations. The code is much more meaningful. You have to describe it as code. It becomes much more rigorous, how you vary each step to make a different object.”

Three primary ways of representing designs in a computational form have sustained this exploration; computer code, computer models, and physical prototypes. Each of these representations has unique structural properties which determine how they might be composed and further manipulated. Computer code has an explicit language of expression. It is composed of imperative statements which are arranged in a modular structure and are parametrically adjustable. In the context of this exploration, 2D screen-based computer models have served as an intermediary between coded processes and physical products. Physical prototypes have a double structure; an underlying generative logic and an intrinsic material behavior.
3.2.1 Process Objects

Workshop student: The script was more interesting than the result... it was just a process, pure process. It wasn’t making form in the beginning."

Creating scripts in Rhinoceros is largely the manipulation of symbols which refer to locations specified by (x, y, z) coordinates. At the base of any system created through scripting is a Cartesian reliance on points. Objects are defined in terms of points and must be manipulated at the coordinate level of description.

Workshop Student: There is a disconnect between what you need to know to design and what you need to know to code. You have to know too much to code all points... example of apertures. I can sketch a shape, but to codify it I need to know all of the geometry that makes up that shape.

This has many advantages in terms of precision and the ability to apply a generalized solution to many objects defined by the same number of points. However, many of the workshop students had difficulty thinking outside of the terms of higher level conventional modeling functions. In asking for help, students spoke in terms of such high-level behaviors as “rotation”, “stretching”, and “solid boolean operations”. Eventually it was necessary for students to explicitly define each of these behaviors in terms of how each point changes in the object of manipulation. Eventually it is necessary to build one’s own abstractions which hide the details of how points are specified allowing one to concentrate on higher level objects and behaviors.
Workshop student: “It’s easy to determine the relationship between two objects through code. It is much more difficult to code the shapes that you want. The tools are not good for molding shape.”

3.2.2 Physical Objects

Workshop Student: “The files that I produced had to be explicitly for this printer. We had to print it three times and rewrite the script to accommodate the printer.”

Scripts must be tailored for specific output devices. When fabricating from a digital file, one must take physical considerations into account. For example, in order to generate a printable model for the Z-corp, a script must specify only solid objects. Objects must be above a minimum thickness (usually 1/8” depending on the shape of the object) and beneath a maximum size. The allowable thickness of a model is partly a function of the overall proportion of the model. If the surface of the model is fairly large, its thickness can be less. However, elements which are small in all dimensions are easily broken. If a model is too large for the Z-corp, it must be split into parts that can be printed and reassembled. Surfaces or incomplete solids generated by the script will result in a damaged print. When developing the light catcher script, I had problems printing the model because of some unintentionally generated surfaces that were not part of solid objects. This made the model impossible to print.

Models generated by scripts cannot usually be scaled and printed at different sizes easily. The physical characteristics of constructions change as scale increases. A script must be altered when the model is scaled in order to account for increased self-
weight and new joint conditions. The programs that I developed all had to take account of the physical nature of the 3D printer. When the geometry that I wanted to produce went beyond the ability of the printer, I had to find ways to rewrite the code which would enable my designs to be printed. It was only possible to understand the constraints of the fabrication system after extensive testing.

"Some materials are easy and even in those there are rules. So flat things can be built out of flat stuff but often the edges are where you have to think hard right? If you are building something out of formed stuff, first of all form-making is expensive. So it turns out to be the form not the material that dictates the part and the rules tend to come from the forming and shipping, how heavy is this thing can you pick it up with a crane."

During the workshop, we tried to encourage the perception of digitally fabricated models as computational objects, with appropriate structural characteristics. However, as soon as students had a physical model they reverted to discussing the design as a fixed object.

Workshop instructor: "Right now it is just an object. How do you get back to a stage in which you can find variations or is that counter productive when you are just trying to get to fabrication?"

3.2.3 Assemblies
Architectural constructions are generally component-based. However, the Z-corp produces only monolithic, solid objects. In

34 From a discussion with Dennis Sheldon at the Massachusetts Institute of Technology workshop, Digital Mockups, Spring, 2002.
order to study component-based designs, it is necessary to fabricate multiple objects that are computationally related but physically separate. The joint strip script can generate multiple components from one initial gesture. Several students in the class faced this issue when studying their models at larger scales. The nature of physical assemblies is not easily simulated. Throughout this work it has become obvious that designers need to develop close relationships with the production of physical objects. Physical materials have behaviors which require iterative testing and must be allowed room for tolerance even in the construction phase.

"Think of design as an assembly process rather than a product. This is the only way to distribute the design to those who fabricate it."  

One is not limited to the use of one of these fabrication technologies. Students tested many ways of fabricating their projects including 3D printing, molds, and flat patterning. Many students in the workshop used several technologies in conjunction to produce design components with different properties. Using 3D printing alone poses many problems. The z-corp printer lets one produce solid objects with complex curves but not thin or flexible surfaces. Another problem with 3D printing is that it yields the same structural resolution everywhere when you might get by with less material in places and consequently increase the speed of production or flexibility of the model. A kit-of-parts ensures that you have the behavior that you need in the right places.

If one understands the nature of a fabrication system, a design script can be written which encodes these necessary formal

---

35 See section A.7 in Appendix.
36 Hugh Whitehead
properties. One group of students modeled their design by hand and concentrated solely on fabrication through scripting. They wrote a script which created lattice of 2D ribs from any surface. This lattice could be fabricated on the laser cutter. After having developed this script, the students were able to physically model multiple complex surfaces in a very short period of time. This pairing of computation and fabrication allowed them to cycle through design, implementation and evaluation extremely quickly.

3.3 Design Types

Two ways of thinking about rules in computation have permeated this study. These two types have been called by various names. In the workshop, they are referred to as "generative" and "parametric" rules. However, these terms have become inaccurately associated with the software tools Rhinoscripting and CATIA respectively. Computational design types are most powerful as models of design thinking distinct from the media of implementation. For the sake of clarity in distinguishing these two modes of thought, I would like to return to two older and more theoretically grounded definitions; operators and constraints.

3.3.1 Operators

Operators are nothing more than instructions. They rely on a combinatorial view of making which assumes that designs are composed of discrete objects which are arranged and transformed throughout the design process. Operators have a long history of use in architectural design. Manuals of classical architecture rely on the description of classical components

---

such as columns and cornices in terms of operator rules. In his book, *The Logic of Architecture*, William Mitchell defines operators as “tools for manipulating shapes in a design world.” Mitchell’s operator rules have five generalizable manifestations. An *instantiation* is the establishment of a design component as an instance of an abstract component type. A *transformation* is the unary alteration of a design component through rotation, reflection, scaling, or translation. *Combination* is the binary addition or multiplication of two design components to form a third. *Replacement* is the substitution of one design component for an equivalent component. Finally, an *algebra* is the design world comprised of all the operators that are brought to bear in a design process.38

### 3.3.2 Constraints

“If one assumes that everything is possible at the outset of designing, the designer must continually constrain the problem in order to arrive at a solution.” 39

Constraint rules act as boundary conditions. Constraints define the space of a design by continually narrowing the scope of possible solutions. These rules are excellent for optimization and sub-optimization. Constraint-based rule systems are already widely used in engineering. Several software packages developed for engineers have implemented some form of constraint-based modeling. Among these packages are CATIA (Dassault Systemes), TriForma (Bentley), and Solidworks (Dassault Systemes).

The process of developing constraint rules in architecture has been outlined by Mark Gross and Aaron Fleisher in their article, “*Designing with Constraints.*” This article

---

39 Mark Burry
describes a means of working with constraints which involves developing several constraint-based models, resolving their internal conflicts, and judging among them. According to this paradigm, design is not only problem solving but also problem setting. Seeing design as a series of constraints allows the designer to set and express the problem clearly.

"Designing is understood as a process of incrementally defining an initially ill-defined question, and concurrently proposing and testing possible answers."  

These two rule types have obvious implications for design as both a thought process and a physical object. Operator rules offer an object-oriented view of process. As a means of manufacturing operators suggest a product assembled out of components with definite states and characteristics. The constraint-based view of design privileges relationships over objects. Objects in a constraint-based environment are in an unresolved state. As such, the implication for fabrication is that designs are fluid and their final states are a function of material and construction flexibility as opposed to the assembly of static parts.

3.4 Control Mechanisms

Scripts might be developed with one of two types of control mechanisms, closed systems or open systems. Closed systems are scripts which take no user input and are completely deterministic. This type of system executes flawlessly because it

---

40 Fliesher, Aaron and Gross, Mark. 1988. “Designing with Constraints” Design Studies 9, no. 3
41 Fleisher (1988)
always proceeds in the predefined manner. The *Light Boxes*\(^{42}\) and *Facade Strips*\(^{43}\) are both examples of this type of script. These examples were appropriate for beginners because they are complete as code and easily traceable. Open systems afford more variable output but ultimately unpredictable results. Open rule sets respond to an arbitrary input. They must be defined more generally in order to respond to many possible parameters. Often, conditions must be specified which halt the development of the script if the input results in unsatisfactory forms. Below, I describe six control types which have been used in my own work, the work of the students and in practice.

### 3.4.1 Explicit

In explicit scripts the geometry of forms is numerically specified. Almost every RhinoScript uses some explicit instructions to specify variables that are generated independently of any external control system. However, purely explicit scripts are completely rigid and can only be used to generate a single result.

### 3.4.2 Random

In computer programming randomness is never uncontrolled. In Visual Basic, the parent language of RhinoScript, there is a predefined function, *Ran*, which generates random numbers between 0 and 1. It is possible to scale this range to any desired set of numbers through a simple multiplication. Therefore, the range of random numbers is precisely controlled by the designer. As a result, the parameters of a model which are random are quite controlled. In the example of the *Light Boxes*, I chose to specify the majority of the dimensions explicitly including the number of columns and rows and the overall size of the model. Meanwhile, the size and depth of the individual grid cells

---

\(^{42}\) See section A.2 in Appendix.
\(^{43}\) See section A.3 in Appendix.
remains variable. In other words, some aspects of the model are set but others still allow room for play.

One particular student, who was already an advanced programmer, produced a script that was so complex that it appeared confusing to all eyes.

Workshop student: "(My) planar surface is purely mathematically described with a parametric function; The idea of randomness but not."

This caused a bit of controversy among reviewers. Is it meaningful to use a deterministic function if in fact the results appear to be random? Determinism and randomness seemed at this point only relevant in as far as they could be discerned by viewers. In this situation, there was a gap between the explicit instructions of the script and the inferred logic of the result.

3.4.3 Pattern
One commonly used closed system is a pattern. Patterns are geometric compositions based on progressions, regressions, or compositions generated by explicit mathematical descriptions. Patterns are often the first programmed models that people build simply because they allow variation with minimal effort. In the workshop, scripting allowed students to create patterns of apertures early on. Students produced patterns using linear, quadratic, and sinusoidal wave functions in order to control the way in which repetitive elements changed across the distance of a surface. The Light Boxes script has been implemented both with a random control and a pattern control. Foster and Partners also spent a lot of time experimenting with geometric progressions as control mechanisms for their work. In the example of the SwissRE building, Foster and Partners found the
curvature for the building’s profile in a series of superimposed circles diminishing in size according to the Fibonacci sequence.

### 3.4.4 Numeric

Scripts which solicit numeric input as a means of control are also common. Instead of relying on a function to generate the form of her first assignment, one student wrote a script which solicited input from the user for each aperture in the surface. This input was filtered through a rule designed to inhibit the amount of light that penetrated the surface in any one area. The intent of the inhibition rule was to balance out light levels across the surface. The user determined the size of the openings on one side of the surface. However, the inhibition rule dictated how those inputs would register on the back surface.

### 3.4.5 Gesture

Programs which respond to gestural input are ubiquitous within the world of graphical user interfaces. Developing a set of rules which can respond directly to input from the user was one of my first interests. The Surface Tiles script was the first gesture-responsive script that I wrote. Gesture-responsive scripts are open systems in which the rule set is defined in relationship to an existing shape. Gestural inputs may be in the form of scanned images, digitally drawn shapes, or digitized three dimensional objects.

---

44 See section A.1 in Appendix.
3.4.6 Environment

“When you watch an ant follow a tortuous path across a beach, you might say, "How complicated!" Well, the ant is just trying to go home, and it's got to climb over little sand dunes and around twigs. Its path is generally pointed toward its goal, and its maneuvers are simple, local responses to its environment. To simulate an ant, you don't have to simulate that wiggly path, just the way it responds to obstacles.”

This means of generating organization comes through defining a general approach to the environment which may produce complex results. Patterns might arise from such a strategy, however these will be informed contextual patterns and not based on abstract geometry. In the example of the Light Catcher, the generated form is extremely complex and would be difficult to model manually on a computer let alone physically. The Light Catcher script can produce hundreds of light responsive apertures, each one different and each one with a unique geometry. This is not accomplished by isolating each local condition and resolving it. It is done by specifying a general method which is applied to many different conditions. As long as each condition has the appropriate definition and parameters, the general method can be applied, although it might result in a different final geometry for each condition.

---

46 See section A.5 in Appendix.
47 The procedure which makes the Light Catcher specifies a series of surfaces that span between two squares. However, the nature of these squares is unspecified in this method; it is specified in another method. This kind of abstraction allows the method to take any two squares as input and generate a tube with one square at each end. Then, when I create the method that describes these squares, I don't have to worry about how they make the tube. I can figure out the size and positioning of the two ends of the tube without worrying about the dimensions and geometry of the tube's surfaces. This allows me to set the relationships which I know in advance and continually
Modulating a script in order to handle complexity can become a problem. When too many abstractions have been defined, it can be difficult to reconsider their underlying relationships at a later time. This becomes increasingly challenging as abstractions are constructed, one upon another. However, a conscientious designer is able to see these methods as building blocks which can be taken apart and rearranged. Thus, scripting can elevate the design process to the level of relationships as opposed to individual components.

3.4.7 Control Combinations
In the scripts listed in the appendix there are many instances in which multiple inputs and controls are used to orchestrate the final outcome of the generative process. The *Light Catcher* script employs all of these control methods. It uses explicit coding to define the thickness of the model’s surfaces. It uses patterns to determine how reinforcing structural components are deployed. It relies on gestural input to specify the conditions of the surface on which the light catchers are generated. Finally, it uses environmental parameters to calculate the orientation of each light catcher and the generation of supports with respect to the ground plane. Programs which employ several control types can be described in terms of independent and dependent parameters.

3.4.8 Rules without Control
Students had many uncoded rules that strongly guided their projects. Components were often designed to account for orientation with respect to the sun and in relationship to other objects inside and outside of the pavilion using control mechanisms that were not directly related to these variables. Throughout the process students struggled to hold onto these rearrange other relationships without worrying about having to rebuild the entire system from scratch.
uncontrolled rules while separately trying to gain increased control over their scripts. Some of the students were working with scripts on a phenomenological basis, reacting to what the script did without complete understanding.

Workshop student: “We started over again because we didn’t have control over the script...it was a great experience. In the previous model we have many variables which we could have played with for months.”

3.5 Variation

Designers make many objects along the way to a finished design. These objects are independent investigations which “can embody, symbolize, and mean in ways that are identical to the cultural artifacts we identify as buildings or paintings or other finished works.” 48 Computational objects can be seen as objects in this right, with their own associated ways of being interpreted and manipulated.

3.5.1 Topology and Geometry

There is a fundamental difference between geometry, the exact location of points in space, and topology, the relative relationship of points in space. This distinction is important in the creation of computational objects. By iteratively changing the parameters used to generate models one can create multiple possibilities for consideration. Variations can be produced in the geometry without disturbing the underlying topology. Variations in topology can also be arranged by changing the number of parts or underlying relationships between parts. There are two evident directions in which to sort through these variations; optimization and exploration.

3.5.2 Optimization

Optimization is a continual narrowing of the search to a specific goal. This type of search is commonly called “hill climbing”\textsuperscript{49} and is used extensively in engineering applications. Engineers often optimize for a limited number of variables which control important issues such as the strength of a system, its speed or its cost.

“If you find the right representation, the solution falls right out.”\textsuperscript{50}

In relation to a similar axiom\textsuperscript{51}, Hugh Whitehead has pointed out that a problem can only be correctly stated if there is some certainty about what the unknowns are. The kind of optimization that this statement implies can only be calculated for a limited set of variables. In a typical architectural problem there are countless variables.

“Computers may introduce a false sense of having optimized a design, which may be fundamentally ill-conceived.”\textsuperscript{52}

\textsuperscript{50} Winston (1992)
\textsuperscript{51} See the section on Practice.
3.5.3 Paradigm Shifting

"A topologist always confuses his coffee and his donut." 53

For the topologist, the coffee cup and donut are geometrical variations of the same structure. But of course this is a huge conceptual and pragmatic distinction. If you actually tried to bite into your coffee, you are likely to end up with scalding coffee all over yourself. Like this example, exploratory variations seek to open up the possibilities rather than narrow them. Explorations sometimes result in paradigm shifts which reframe how the initial problem is understood.

3.6 Evaluation

3.6.1 Performance Evaluation

Generative systems developed through scripting can produce many design variations in a short period of time by adjusting the controlling parameters. The final problem of making variations is evaluation. Many times in scripts, conditionals are used to evaluate data as the script is running. In the Facade Strips 54 example, I use an evaluative function to collect data from the script. This script regenerates multiple times until the criteria of this function are met. This is an optimization technique for evaluation.

It is possible to take a purely performance driven view of evaluation. This can be helpful in narrowing the acceptable range of variations. However, the definition of satisfactory performance should not be a means to pass off the responsibility of design. Designers who accept this means of validating designs must investigate performance criteria critically. High

53 Anonymous
54 See section A.3 in Appendix.
performance in one area may mean low performance in another. As we have seen, it is only possible to optimize a few variables at a time. There is also a tendency to weight the criteria most heavily which can be expressed explicitly. Foster and Partners encouraged the students to experiment with performance-driven design by producing numerically evalutatable representations. However, many of the students ended up concentrating on the poetics of light rather than its performance.

"Most of the emphasis is on the object rather than the effects inside. Performance evaluation lapsed." 55

Many architects are currently working on ways to connect design systems and environmental analysis more strongly. As a result the computer is positioned somewhat uncomfortably between good design and good policy. This sentiment has been voiced in many discussions about technology by architects. During a lecture at MIT, Branko Kolarovic, a Professor of Architecture at the University of Pennsylvania, said that his ideal computer tool would be "something like a slider" that would allow him to experiment in the space between environmental optimization and design goals.

Students who relied on metaphorical sources of control in their work produced results that could not easily be interpreted in terms of success and failure. These schemes were in danger of becoming meaningless in the context of the optimization paradigm.

55 Hugh Whitehead
CHAPTER 4: CONCLUSION

End-user programming can help architects to relate creative design to a process of manufacturing and assembly. This thesis identifies methods for discussing computational objects as processes and products in parallel. Throughout my investigation, the development of computational objects has shown to be structured both by design decisions and by the nature of the representations. Scripts, computer models, and physical prototypes have independent structures yet all stem from the same underlying computational ideas.

Conceptual frameworks such as operators and constraints help to frame the way in which designers think about computational objects. Scripts generate form through both closed instructional systems and responsive rule sets which follow numerical, gestural, and environmental input. Computational solutions can be directed inwards towards optimization or outwards towards conceptual exploration.

Workshop student: “I work very intuitively. If I think too much about it I get really confused so I just start drawing. I don’t understand why that object is the way it is. Scripting it forces you to describe every move that you make. It helps you to describe your intuitive process a little bit cause you start setting up relationships.”

In the transition from implicit to explicit representation, the nature of architectural design rules is likely to change. The computer will not eradicate older media. It will shift the way that these media are understood and applied. The computer is a reflective medium which frames how designers work and in doing so allows architects to see themselves critically.
Appendix A:
Examples in Rhinoscripting

A.1 Surface Tiles

Input: Two profile curves
Output: A tiled surface with ribs from two profile curves.

This script takes two arbitrary curves as the starting point to generate a curved surface made out of straight components based on a simplistic model of straight frame construction. The resulting surface geometry depends on the initial profile curves. This script shows one approach to a fundamental problem that computers can solve better than any designer: the task of breaking a complex surface into small, constructible components. This script relies on interpolation to find intermediary points that compose the surface between two profile curves. This points matrix belonging to the complex curved surface can be used to rebuild the surface in space from components that the script can specify as alternately planar or curved. The nature of the components must ultimately be related to the manufacturing method that is intended. In this example, the script assumes the materials of conventional wood frame construction as a starting point. It develops a set of independent, underlying relationships based on how the components of wood frame construction might come together. This system is applied and sometimes distorted by a range of surfaces that it is applied to. Some of the surfaces violate the ability of the script to maintain the wood-frame-like relationship of the components. In these cases, there are unintentional tears and fragmentation. This is an example of an open rule system because the script can be
applied to a range of arbitrary starting profiles. Like any system it has limitations. However, codifying the relationships of a system like this helps designers see the constraints and either work within them or recognize how they can be extended. It is easy to get absorbed in the process of making one set of rules work and lose track of the larger, and more important picture, which might be attainable in an easier, cheaper, more elegant way following a different set of rules. The key issue that this raises is how can rule building help people to think about and be critical of rule systems in general, both programmed and not.

A.2 Light Boxes

*Input:* None

*Output:* a grid of light modulating boxes.

This script is designed as an introductory example for architects with no prior knowledge of programming. It is built using two repetitive structures called loops. The first loop builds rows from a sequence of unit elements repeated along one axis. The second loop builds the full grid from a sequence of these rows repeated along a perpendicular axis. This double loop structure is called a nested loop and is useful for building multi-dimensional structures. Each element used to build the grid has its own independent properties. In this case, the elements are extruded tubes with a depth and a frame thickness. Both the depth and the frame thickness are parameters to determine the amount of light that can pass through this grid at any one cell. The relationships between the code, frames, tubes, rows, and grid define the basic structure of this model. However, the dimensions and numbers that determine the resulting geometry are adjustable without destroying the underlying structure. Thus, a designer can easily produce geometric variations which all share the same basic relationships. These relationships are
independent of the final geometry of any one variation. They define the topology of the model.

This underlying structure can be made to generate a configuration where all the tubes have identical frames and are extruded to one consistent depth. The same structure can produce a model where each tube has a different frame size and a different depth. The possible variations are virtually unlimited. This might make it difficult for an architect to choose any one configuration. However, it is important to note that the range in which this model can vary, without changing the underlying structure are numerous but constrained. This underlying structure sets the relationships that are important or necessary to produce the desired form. It is possible to develop variations in relationship to some conditions outside of the set structure of the script. These conditions might stem from structural imperatives, environmental concerns, relationships to adjacent functions, or aesthetic interests.

The designer of a script specifies which relationships in the design should be independent. These relationships are the underlying structure. In addition, the designer specifies the dependent relationships. These relationships are open to the kind of variation that are discussed above. This relationship between independent and dependent relationships must be mediated by another structure. This mediating structure is the control. Here I have set the control to be a sequence of random numbers generated by a computer function. These random numbers create varied models which express the range of what the script can produce. After seeing the results of a designed randomness, the designer might decide to make more explicit choices about the final geometry. In other words, the designer might link the controlling structure to a rational objective. We presented this model to the students with the hope that they would find the
random variations alluring but ultimately guide the model to assume a final form by remaking the control structure, thus separating form or order from the meaningless but rich possibilities generated in the random model.

A.3 Facade Strips

Input: Number (Minimum Surface Area)
Output: a light modulating surface of planar strips.

The program builds a solid/void pattern of two-dimensional strips which can be exported to a laser cutter. The script is controlled by the length and number of strips in each row. In the examples shown here, a random number generator controls the dimensions of the strips. However, these values can just as easily be controlled in a more directed way given site, programmatic and material constraints. This script is equipped with an evaluative function which keeps track of the total area created and can automatically rerun the script if the pattern does not meet adjustable lighting requirements.

This is another simple script developed as an introductory example for students. Like the grid version of this series of light mediating facade generators, this script is built using a nested loop system. However, the dimension of the base elements is variable. This results in a striated or non-gridded matrix of elements. The base elements of the facade are two dimensional strips. The height of these strips is fixed as are the number and length of the rows. This results in varying patterns that all occur within a fixed boundary like in the grid example.

This script has an additional behavior that is one level advanced from the grid script. The evaluative function is an independent agent which assesses the results of each randomly generated surface. This self-regulating ability of scripts suggests a new positioning of the random number generator. In this
example, the random number generator produces not just one meaningless variation, but a range of potential solutions. The solutions are efficiently evaluated by this new evaluative function. It is possible for one script to judge many more variations than would be possible by an entire design office. This should not be a surprising. This script uses a simplistic conditional to determine whether or not each randomized facade meets some basic criteria in terms of a solid/void relationship. If the random generator produces the appropriate relationship, then the script stops, otherwise it generates a new random facade. This repetitive sequence continues until the conditions are satisfied. As a result, the script is able to find alternatives, perhaps undiscovered, or undiscoverable by human designers. This is a simple version of a self-regulating system. A more complex version can search through a vast variety of configurations. As foreign as this may seem to designers who are used to formulating and critiquing their own work by eye, this process has already been widely accepted in engineering as a means of optimizing solutions. Although design is not a problem-solving discipline, design often involves some problem solving. Optimization tools are a necessary part of a designer’s toolkit.

A.4 Surface Reflectors

*Input:* Two profile curves, light ray, point

*Output:* Small surfaces that reflect incoming light towards single point.

This script follows the rule: angle of reflection = angle of incidence. This is my first script to use vector math in order to derive the position of points in space. This script was written as an advanced example for the class. It is a demonstration of a lighting principle which would prove useful for the students.
This particular script does not suggest a piece of a building, wall, ceiling, etc. as some of the other scripts might. It is a study device which helped me, and hopefully the students, to understand how light might function in relationship to form and how a model might be driven by environmental factors. It was also a great stepping point into the world of complex structures where the complexity is in the relationships and not in the data structure. The rules that guide the reflectors are very simple. However, complex results are achieved when these rules are applied to a complex surface. One of the powers of symbolic programming is the ability to achieve manageable complexity with simple rules. This script leads to many others in which variations were driven by complex environmental relations rather than complex rules.

A.5 Light Catchers

Output: A surface which funnels light into a space from one direction.

This script combines the facade grid system with the tiled curved surface. It is another variation of the light receptive or responsive surface. This model collects light as opposed to redirecting it, as is done in both the examples of the reflector and the shade. This example builds light boxes on a complex curved surface. One end of the each box is co-planar with the surface, the other end is parallel to a specified direction of the sun. The result is a wide variation of boxes which stretch and distort in order to bring in the sun at varying points on a billowing surface. Below, shafts of light stream through the fractured, but contiguous surface competing for precedence within a spatially unified form.

A.6 Unfold Surface
Input: Two profile curves, light ray
Output: Unfolded planes from a faceted curved surface.

This script was originated in order to output models with the laser cutter that could not be printed with the Z-corp. Here is an example of a script that can be seen as almost purely utilitarian. However, it has demonstrated some interesting properties in translating data from one organization into another. To explain, this script analyzes faceted curved surfaces and reconstructs them as flat two-dimensional patterns. This is a means of translating data from one form into another and can be a powerful tool for design. The ability to translate forms between states can be used to describe and design kinetic structures or as a basis for analyzing an existing landscape or site for renovation. The ability to project and distort forms into possible configurations is a powerful one. Interestingly, one of my misfired examples translated a flat surface into a series of curves. Procedural means of interpretation or analysis have proved to be open to distortion.

A.7 Join Strip

Input: One profile curve
Output: a series of independent flat panels connected with dove-tail joints

This script was an attempt to make a model out of independently printed components that can be joined together to form a single composite form. Because 3D plaster models cannot be glued or otherwise easily joined it was necessary to make an attempt to join models by interlocking their forms. This is a method commonly done in wood and other solid materials. Like a puzzle, the pieces of this model lock together in only one possible position.
Appendix B:
Projects by Foster and Partners

B.1 SwissRE

"SwissRE was conceived as an environmentally progressive building. The client wanted to create the image of an environmentally responsible insurance company."  

In the design of a skyscraper in downtown London for SwissRE, technological innovation was actively supported by the client. Foster’s office investigated many ways in which the building could reduce its environmental impact. Controlling and designing this complex curved form efficiently meant finding a way to retain control over hundreds of component details as the building form was adjusted and tested against varying considerations. Using computational tools allowed Foster’s office to explore many more design variations than would normally be allowed by a conventional design process of manual drawing and model making. Through a parametric model built in Microstation, designers were able to adjust the geometry, within a constrained space, without sacrificing any dimensional knowledge of the building form.

"The parametric model allowed us to tweak the design much more than we would normally do."  

The curved form of SwissRE while standing out symbolically as a form differentiated from its context, is physically and therefore environmentally recessive, resulting in significant benefits for its surroundings. In a dense urban site, a tall building like SwissRE

Project Architect, SwissRE, Foster and Partners

Project Architect, SwissRE, Foster and Partners
generates a large shadow over neighboring buildings. The recessive form of SwissRE results in a considerably less imposing shadow than other rectilinear buildings of similar size. In addition, the curved form of SwissRE presents a minimal surface to the wind and the significant structural loads that the wind imposes. According to the architects, SwissRE actually improves wind conditions around its base. The accompanying slimming of the curved form of SwissRE’s base allows for considerable open space to share the same site. Of course, as you approach SwissRE from any direction, it recedes from you, rendering it far less dominating than its actual size would suggest. Finally, the continuous nature of SwissRE’s curved surface allows structural forces to be directed along more than one path to the ground. As a result, damage to a few structural members will not be catastrophic to the entire building.

A desire to maximize natural light and airflow through the building led Foster’s to develop a scheme in which the floor plates were articulated with inlets by which sunlight and outside air could pass through the building. These inlets combine in long atrium spaces carved away from the form of the tower. As the floor plates decrease in size towards the top of the building, the inlets become smaller in order to preserve a usable amount of square footage. Vents to the outside are located along the perimeter of each floor. SwissRE sets a precedent. The architect’s office stated that it is the first high-rise building with a natural ventilation system.
B.2 The Greater London Authority

"It was clear after this work that a fundamentally different mindset was required with a curvilinear building."\textsuperscript{58}

The Greater London Authority was built just prior to the SwissRE office tower. This was the first curved building that Foster’s did. All the lessons learned through the GLA were brought to bear on SwissRE. The GLA was built using techniques developed for rectilinear buildings. That is apparent in the some cumbersome details and ill-fitting joints. This initial building concept was an object on the edge of the river. This object was shaped through metaphors including but not limited to a lens, an egg, and a pebble. These are shape ideas which were put through a filter of computational tests in order to access and judge variations based on solar gain, direct solar radiation, and possibilities for self shading. The optimal shape in terms of these conditions was then built out of components that could be easily unfolded and used to generate the working drawings.

\textsuperscript{58} Judit Kimpian
Appendix C:

Sample RhinoScript: Light Catcher⁵⁹

Option Explicit
call Rhino.EnableRedraw( vbFalse )
call Rhino.Command( "NoEcho " )

" Define global variables used by all
" functions
dim XX: XX = 0
dim YY: YY = 1
dim ZZ: ZZ = 2

" no of segments
dim globalNum: globalNum = Rhino.GetInteger( "How many rows?" )
dim globalCount: globalCount = 0
dim globalCount2: globalCount2 = 0
dim globalw: globalw = 1/10
" create new plane of reference
" perpendicular to vector n
dim n: n = light
dim p: p = array( 1, 0, 0 )
dim r: r = cross( n, p )
dim s: s = cross( n, r )
n = normalize( n )
r = normalize( r )
s = normalize( s )
call selection

call Rhino.Command( "Echo " )
call Rhino.EnableRedraw( vbTrue )

" Selects a line respresenting the direction
" of the sun find start and end points on this
" line and derive the vector
function light
  dim pt1, pt2, strSun, vi
  strSun = Rhino.GetObject( "Select a ray of light" )
  pt1 = Rhino.CurveStartPoint( strSun )
  pt2 = Rhino.CurveEndPoint( strSun )
  light = vector( pt1, pt2 )
end function

" Selects two profile curves which generate the
" surface on which everything is constructed
function selection
  dim arrCurve1, arrCurve2

⁵⁹ Some of the functions included in the Light Catcher Script were written in collaboration with Stelios Dritsas.
dim arrPoints1, arrPoints2
    dim po, pi, vt

    arrCurve1 = Rhino.GetObject( "Select profile closest to the x-axis" )
    arrCurve2 = Rhino.GetObject( "Select profile furthest from the x-axis" )

    arrPoints1 = Rhino.DivideCurve( arrCurve1, globalNum )
    arrPoints2 = Rhino.DivideCurve( arrCurve2, globalNum )
    call polyline(arrPoints1, arrPoints2)
end function

function polyline( points1, points2 )
    redim arrPoints(globalNum)
    dim arrPoints2
    dim index
    dim arrPoint
    dim count: count = 0
    dim line
    dim dist

    do while count < globalNum + 1
        for index = 0 to globalNum
            arrPoint = interpolate(points2(index), points1(index), count/globalNum)
            arrPoints(index) = array( arrPoint(0), arrPoint(1), arrPoint(2)
        next
    line = Rhino.AddPolyline( arrPoints )
    dist = Rhino.CurveLength( line )

    if count > 0 then
        " creates lightboxes along the surface
        call lightbox( arrpoints, arrpoints2, dist )
    end if

    " creates supporting ribs under the surface
    call create_Struct( arrpoints, count )

    count = count + 1
    " here i am keeping track of two seperate arrays (the two parallel curves) to build the light boxes
    arrpoints2 = arrpoints
end function

function lightbox( arrpoints, arrpoints2, length )
    dim index, dx, dz
    redim points1(3)
    redim points2(3)
redim points3(3)
redim points4(3)
dim surfaceA, surfaceB, surfaceC, surfaceD
dim offsetX: offsetX = -.5
dim offsetY: offsetY = 0
dim offsetZ: offsetZ = -.5

for index = 0 to globalNum - 1
  " chart the change in dx and adjust shaders based on slope
  dx = arrpoints(index)(0) - arrpoints(index + 1)(0)
  length = sqr(dx * dx)/2

  " chart the curvature of the surface and do not implement light catcher if the slop is too extreme
  'dz = arrpoints(index)(2) - arrpoints(index + 1)(2)
  'if sqr(dz * dz) < 1 and sqr(dz * dz) > 0 then
  " 1
  points1(0)= array( arrpoints2(index + 1)(0) , arrpoints2(index + 1)(1) , arrpoints2(index + 1)(2) - 1 )
  points1(1)= array( arrpoints2(index)(0) , arrpoints2(index)(1) , arrpoints2(index)(2) - 1 )
  points1(2)= relocation( arrpoints(index) , array( offsetX , offsetY , offsetZ ) , r , s , n )
  points1(3)= relocation( arrpoints(index) , array( offsetX , -length + offsetY , offsetZ ) , r , s , n )
  surfaceA = Rhino.AddSrfPt( points1 )
call Rhino.SelectObject( surfaceA )
call inner_surface1( points1 )

  " 2
  points2(0)= array( arrpoints(index)(0) , arrpoints(index)(1) , arrpoints(index)(2) - 1 )
  points2(1)= array( arrpoints(index + 1)(0) , arrpoints(index + 1)(1) , arrpoints(index + 1)(2) - 1 )
  points2(2)= relocation( arrpoints(index) , array( length + offsetX , offsetY , offsetZ ) , r , s , n )
  points2(3)= relocation( arrpoints(index) , array( length + offsetX , offsetY , offsetZ ) , r , s , n )
  surfaceB = Rhino.AddSrfPt( points2 )
call Rhino.SelectObject( surfaceB )
call inner_surface3( points2 )

  " 3
  points3(0)= array( arrpoints2(index + 1)(0) , arrpoints2(index + 1)(1) , arrpoints2(index + 1)(2) - 1 )
  points3(1)= array( arrpoints(index + 1)(0) , arrpoints(index + 1)(1) , arrpoints(index + 1)(2) - 1 )
  points3(2)= relocation( arrpoints(index) , array( offsetX , -length + offsetY , offsetZ ) , r , s , n )
  points3(3)= relocation( arrpoints(index) , array( length + offsetX , -length + offsetY , offsetZ ) , r , s , n )

64
surfaceC = Rhino.AddSrfPt(points3)
call Rhino.SelectObject(surfaceC)
call inner_surface2(points3)

"4
points4(0)= array(arrpoints(index)(0), arrpoints(index)(1),
arrpoints(index)(2) - 1)
points4(1)= array(arrpoints2(index)(0), arrpoints2(index)(1),
arrpoints2(index)(2) - 1)
points4(2)= relocation(arrpoints(index), array(length +
offsetX, offsetY, offsetZ), r, s, n)
points4(3)= relocation(arrpoints(index), array(offsetX,
offsetY, offsetZ), r, s, n)
if points4(2)(1) < points4(3)(1) then
points4(2)= relocation(arrpoints(index), array(offsetX,
offsetY, offsetZ), r, s, n)
points4(3)= relocation(arrpoints(index), array(length +
offsetX, offsetY, offsetZ), r, s, n)
end if
surfaceD = Rhino.AddSrfPt(points4)
call Rhino.SelectObject(surfaceD)
call inner_surface4(points4)

'else
' points1(0)= array(arrpoints2(index + 1)(0), arrpoints2(index + 1)(1),
arrpoints2(index + 1)(2) - 1)
'points1(1)= array(arrpoints2(index)(0),
arrpoints2(index)(1), arrpoints2(index)(2) - 1)
'points1(2)= array(arrpoints(index)(0), arrpoints(index)(1),
arrpoints(index)(2) - 1)
'points1(3)= array(arrpoints(index + 1)(0), arrpoints(index + 1)(1),
arrpoints(index + 1)(2) - 1)
'surfaceA = Rhino.AddSrfPt(points1)
'end if
next
end function

function create_struct(points, indicator)
dim index
dim SrfPoints(3)
dim stringSrf
for index = 0 to (ubound(points) - 1)
srfpoints(0) = array(points(index)(0), points(index)(1),
points(index)(2) - 1)
srfpoints(1) = array(points(index + 1)(0), points(index + 1)(1),
points(index + 1)(2) - 1)
srfpoints(2) = array(points(index + 1)(0), points(index + 1)(1),
points(index + 1)(2) - 1.25)
srfpoints(3) = array(points(index)(0), points(index)(1),
points(index)(2) - 1.25)
"every third rib, take it all the way down to the x-axis
if (indicator Mod globalNum = 0) then 'and (globalCount2 Mod 2 = 0) then
  srfPoints(2) = array( points( index + 1 )(0), points( index + 1 )(1), 0 )
  srfPoints(3) = array( points( index )(0), points( index )(1), 0 )
end if

stringSrf = Rhino.AddSrPt( srfPoints )
"to avoid generating double rib on last row
if indicator > 0 then
  call Rhino.SelectObject( stringSrf )
  call Rhino.Command( "offsetsrf s" & globalw & " ")
  call Rhino.UnselectAllObjects
end if

"to avoid generating a double rib on first row
if indicator < globalNum then
  call Rhino.SelectObject( stringSrf )
  call Rhino.Command( "offsetsrf s" & globalw & " ")
end if

"get rid of all surfaces after using them to create solids
call Rhino.DeleteObject( stringSrf )

next
globalCount2 = 0
end function

function inner_surface1( Ppoints )
redim innerPoints(3)
innerPoints(0) = array( Ppoints(0)(0) - globalw, Ppoints(0)(1) - globalw, Ppoints(0)(2) )
innerPoints(1) = array( Ppoints(1)(0) + globalw, Ppoints(1)(1) - globalw, Ppoints(1)(2) )
innerPoints(2) = array( Ppoints(2)(0) + globalw, Ppoints(2)(1) - globalw, Ppoints(2)(2) )
innerPoints(3) = array( Ppoints(3)(0) - globalw, Ppoints(3)(1) - globalw, Ppoints(3)(2) )
call create_solid( innerPoints, Ppoints )
end function

function inner_surface2( Ppoints )
redim innerPoints(3)
innerPoints(0) = array( Ppoints(0)(0) - globalw, Ppoints(0)(1) - globalw, Ppoints(0)(2) )
innerPoints(1) = array( Ppoints(1)(0) - globalw, Ppoints(1)(1) + globalw, Ppoints(1)(2) )
innerPoints(2) = array( Ppoints(2)(0) - globalw, Ppoints(2)(1) - globalw, Ppoints(2)(2) )
innerPoints(3) = array( Ppoints(3)(0) - globalw, Ppoints(3)(1) + globalw, Ppoints(3)(2) )
call create_solid( innerPoints, Ppoints )
end function

function inner_surface3( Ppoints )
    redim innerPoints(3)
    innerPoints(0) = array( Ppoints(0)(0) + globalw, Ppoints(0)(1) + globalw, Ppoints(0)(2) )
    innerPoints(1) = array( Ppoints(1)(0) - globalw, Ppoints(1)(1) + globalw, Ppoints(1)(2) )
    innerPoints(2) = array( Ppoints(2)(0) - globalw, Ppoints(2)(1) + globalw, Ppoints(2)(2) )
    innerPoints(3) = array( Ppoints(3)(0) + globalw, Ppoints(3)(1) + globalw, Ppoints(3)(2) )
    call create_solid( innerPoints, Ppoints )
end function

function inner_surface4( Ppoints )
    redim innerPoints(3)
    innerPoints(0) = array( Ppoints(0)(0) + globalw, Ppoints(0)(1) + globalw, Ppoints(0)(2) )
    innerPoints(1) = array( Ppoints(1)(0) + globalw, Ppoints(1)(1) - globalw, Ppoints(1)(2) )
    innerPoints(2) = array( Ppoints(2)(0) + globalw, Ppoints(2)(1) - globalw, Ppoints(2)(2) )
    innerPoints(3) = array( Ppoints(3)(0) + globalw, Ppoints(3)(1) + globalw, Ppoints(3)(2) )
    call create_solid( innerPoints, Ppoints )
end function

" function create_solid( innerPoints, Ppoints )
    redim btmPoints(3)
    redim topPoints(3)
    dim surface1, surface2, surface3

    surface1 = Rhino.AddSrfPt( innerPoints )

    btmPoints(0) = innerPoints(0)
    btmPoints(1) = Ppoints(0)
    btmPoints(2) = Ppoints(1)
    btmPoints(3) = innerPoints(1)
    surface2 = Rhino.AddSrfPt( btmPoints )

    topPoints(0) = innerPoints(2)
    topPoints(1) = Ppoints(2)
    topPoints(2) = Ppoints(3)
    topPoints(3) = innerPoints(3)
    surface3 = Rhino.AddSrfPt( topPoints )

call Rhino.SelectObject( surface1 )
call Rhino.SelectObject( surface2 )
call Rhino.SelectObject( surface3 )
if globalCount > 2 then
    call Rhino.Command( "join " )
    call Rhino.UnSelectAllObjects
    globalCount = 0
else
    globalCount = globalCount + 1
end if
end function

function vertex_length( va, vb )
    dim dx: dx = va( XX ) - vb( XX )
    dim dy: dy = va( YY ) - vb( YY )
    dim dz: dz = va( ZZ ) - vb( ZZ )
    vertex_length = sqrt( dx * dx + dy * dy + dz * dz )
end function

function interpolate( po, pi, factor )
    dim dx: dx = pi( XX ) - po( XX )
    dim dy: dy = pi( YY ) - po( YY )
    dim dz: dz = pi( ZZ ) - po( ZZ )
    interpolate = array( 
        dx * factor + po( XX ),
        dy * factor + po( YY ),
        dz * factor + po( ZZ )
    )
end function

function relocation( origin, point, vx, vy, vz )
    relocation = array( _
        origin(XX) + point(XX) * vx(XX) + point(YY) * vy(YY) + point(ZZ) * vz(XX), _
        origin(YY) + point(XX) * vx(YY) + point(YY) * vy(YY) + point(ZZ) * vz(YY), _
        origin(ZZ) + point(XX) * vx(ZZ) + point(YY) * vy(ZZ) + point(ZZ) * vz(ZZ) )
end function

function cross( vo, vi )
    cross = array( _
        vo( YY ) * vi( ZZ ) - vo( ZZ ) * vi( YY ), _
        vo( ZZ ) * vi( XX ) - vo( XX ) * vi( ZZ ), _
        vo( XX ) * vi( YY ) - vo( YY ) * vi( XX )
    )
end function

function scale( pv, factor )
    scale = array( _
        factor * pv( XX ), _
        factor * pv( YY ), _
        factor * pv( ZZ )
    )
end function
end function

function normalize( vo )
  dim distance: distance = sqr( _
    vo( XX ) * vo( XX ) + _
    vo( YY ) * vo( YY ) + _
    vo( ZZ ) * vo( ZZ ) _
  )
  if( distance = 0.0 ) then
    normalize = array( 0.0, 0.0, 0.0 )
  else
    normalize = array( _
      vo( XX ) / distance, _
      vo( YY ) / distance, _
      vo( ZZ ) / distance _
    )
  end if
end function

function length( po, pi )
  dim dx: dx = po( XX ) - pi( XX )
  dim dy: dy = po( YY ) - pi( YY )
  dim dz: dz = po( ZZ ) - pi( ZZ )
  length = sqr( dx * dx + dy * dy + dz * dz )
end function

function midpoint( po, pi )
  midpoint = array( _
    ( po( XX ) + pi( XX ) ) / 2, _
    ( po( YY ) + pi( YY ) ) / 2, _
    ( po( ZZ ) + pi( ZZ ) ) / 2 _
  )
end function

function vector( po, pi )
  vector = array( _
    pi( XX ) - po( XX ), _
    pi( YY ) - po( YY ), _
    pi( ZZ ) - po( ZZ ) _
  )
end function

function translate( po, vo )
  translate = array( _
    po( XX ) + vo( XX ), _
    po( YY ) + vo( YY ), _
    po( ZZ ) + vo( ZZ ) _
  )
end function

69
Bibliography


*Special Note: Unless specified all images are the property of the author.*