Bootstrapping a Computational Discourse

MAYA PRZYBYLSKI
University of Waterloo

> Computer Science is no more about computers than astronomy is about telescopes. Edsger Dijkstra[1]

INTRODUCTION

Bootstrapping: a self-sustaining process that proceeds without external help

System Stalker Lab (SSL) is an introductory exploration of design computing, aiming to instill awareness of the structures, processes, and opportunities necessary to develop a design practice inclusive of computational strategies and techniques. SSL is offered as a third year undergraduate option studio at the School of Architecture at the University of Waterloo. The students participating in the studio are self-selected and enjoy a favorable class size of around twenty students, allowing for an intensive, focused semester. It is assumed that students coming into the studio do not have any computer code writing or reading experience.

COMPUTING CONTEXT

The goal of SSL is to seed a computationally-oriented design culture in the school by clarifying and speculating on the opportunities existing within computing in relation to architectural design. Central to this potential is the architect's relationship to *processing* and, in turn, *processing's* relationship with computing.

Andrew Kudless has identified *processing* as a central skill of the architect.[2] Architects' activities are characterized by creating sets of instructions to be used by others. In the creation of these instructional sets, the designer organizes diverse sets of knowledge into a comprehensive collection of procedures that (coupled with intensive efforts by related disciplines) result in a physical construct. Data collected across numerous dimensions of a project, such as site, program, materials, culture and budget, are transformed into a clear set of drawings, models and specifications that comprise this instructional set that others will use. In such a context, the term *processing* is not dissimilar to what we commonly mean by the term *design*, but perhaps recasts it with more emphasis on the analysis, parsing and filtering of information in order to uncover opportunity for invention, intervention and mobilization.

Computing might be considered to be the automation of processing. Dr. Peter J. Denning, an American computer scientist known for his expertise in communicating computing principles, states that "the fundamental question underlying all of computing is, *What can be (efficiently) automated*."[3] This question can be naturally extended to include the challenges of implementing, analyzing and applying these automated processes. In the architectural context there are numerous processes that can benefit from automation. The scope of such automation is wide. At one end, there are automated processes, such as a door schedule generator within a Building Information Modeler, that speed up familiar (and often mundane) tasks; at the other are powerful exploratory processes, such as environmental performance simulators and evolutionary algorithms, that afford new levels of engagement and expose new potentials.

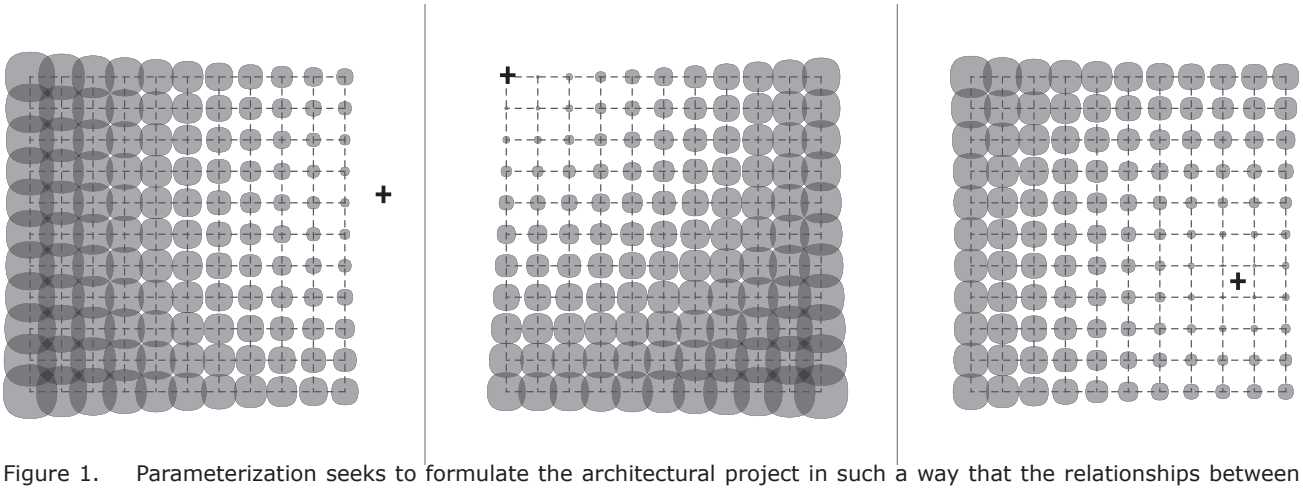


Figure 1. Parameterization seeks to formulate the architectural project in such a way that the relationships between elements remain active, or influential throughout the process. The most common form of parametric design occurs at the level of geometry – often referred to as associative geometry. The example show here outlines a simple parametric relationship where a parameter, attribute, of one object – namely the diameter of each circle is related directly to a second parameter – the distance from the circles' centers to the attractor point.

In order to fully engage with these deeper uses of automation, designers must be capable of authoring their own tools. Such capacity has been characterized as allowing engagement with *computation* rather than (mere) *computerization*.[4] Computerization implies a view of the computer as simply an advanced tool that enables, through the use of out-of-the-box software, the digitization of that what is already achievable and familiar through the use of a limited, predetermined set of procedures; in contrast, design computation expands the relationship designers have with the machine, whereby designers engage with the underlying principles of the computer's automating capabilities in order to explore formal and organizational strategies by way of processing.

Implied in the description of *processing* is the inherent complexity of the architectural problem space, only amplified by increasing access to seemingly relevant information. Architectural problems, regardless of scale, can be thought of as existing within ecologies of multilateral interactions between physical, virtual, political, cultural and technical agents. Unfortunately, our human ability to manage these increasingly complex networks of causal relationships is limited, and complexity can become overwhelming and disabling for the designer. To combat such inhibition, abstraction can be used to make the problem more manageable through a process of reduction; but such filtering, efficacious as it may be, runs the risk of losing the essence of the problem at hand.

In his thesis work entitled Design Exploration through Bidirectional Modeling of Constraints, Axel Kilian points to the potentials of computing to overcome this barrier through associative modelling.[5] Starting with an abstracted basic model, the designer can incrementally add behaviors and relationships between agents in the project. The designer defines the nature and behaviors of parameterized relationships, and the computer manages their interactions, thereby overcoming the human limits of managing complex causal chains (Figure 1). The computational designer is able to abstract a problem for initial action and then, relying on the machine as an automatic accountant, incrementally rebuild the lost complexity thereby allowing the original richness of the problem space to be maintained.

THE BOOTSTRAPS

It is within the context of this discourse that SSL operates. SSL focuses on the investigation and exploration of the structures, processes and opportunities central to design computing. Such a practice requires that designers expand their notion of digital methodologies to include the fundamental paradigms of computer science. At the core of such a practice is close attention to the organization of information and the use of rule-based logical processes to automate (or compute) in a meaningful way. A long view is taken, where it is hoped that students leave the studio with a foundation, or a set of bootstraps, of computing literacy and best practices with which they are able to pull together

their own design practice – to which computation can now contribute.

Owing to the fact that SSL is delivered in a studio environment and not simply as an elective, the lab is able not only deliver the bootstraps but also to test what opportunities emerge from pulling up on them. On the one hand, introduction to and operation within an unfamiliar domain such as design computing requires a commitment to skills-based instruction where students can learn the conceptual and technical frameworks of the discipline. On the other hand, the traditions of the design studio require that students engage in their projects in a critical, robust and rigorous manner. To accommodate both of these imperatives the course is divided into two phases: The first phase focused on encouraging the students to *become computational* and the second phase testing the potentials of this new state of being. While the continuity of these two phases within a single studio sparks constructive dialogue, the focus of this paper is on the first phase -- the development of a workable and expandable foundation in design computing.

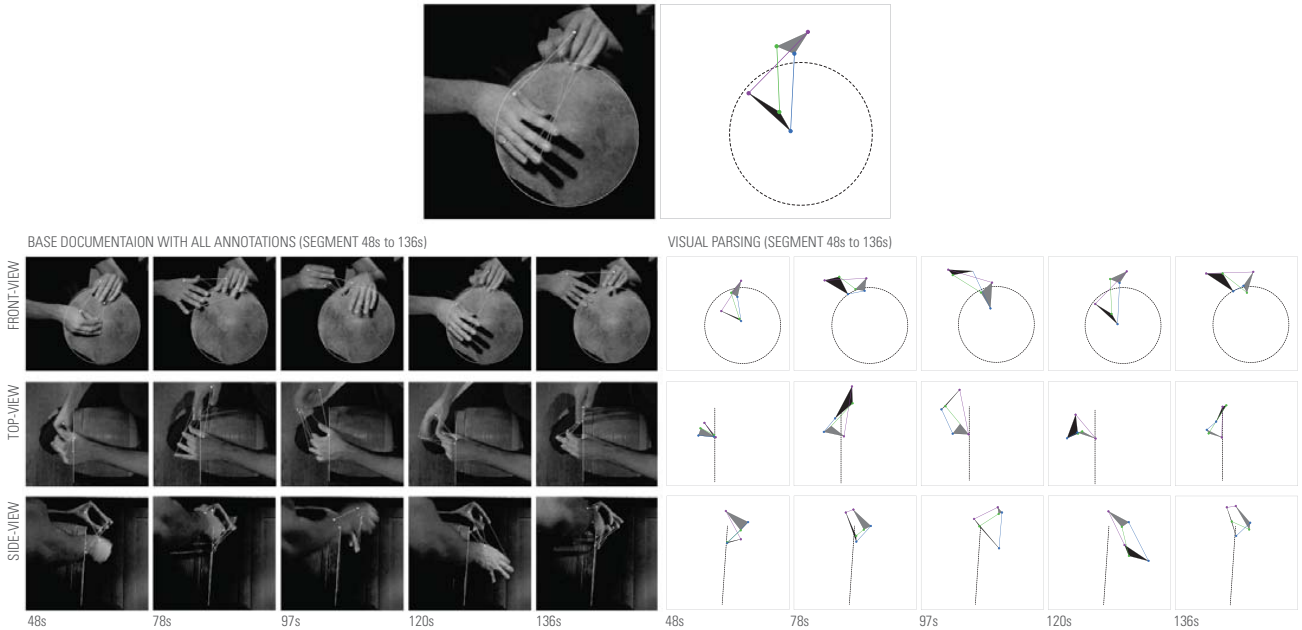
In order to develop these bootstraps for designers working with computation it is instructive to once again turn to Denning as he advocates for the development of skills in four basic areas: *algorithmic thinking*, *representation*, *programming* and *design*.[6] Loosely, an *algorithm* is understood as a finite set of well-defined procedures that translates an input into an output. Algorithmic thinking calls on us to calibrate our expressions in such a way that our understandings and our actions are formulated in terms of a discrete set of procedures, each step delivering an unambiguous result when executed. *Representation* deals generally with how data is stored or communicated so that it can be useful (i.e. become information). *Programming* is the skill that allows designers to express their algorithmic thinking and representations in a specific form or syntax, resulting in a piece of software that causes a computer to perform in a prescribed way. Note that, of the three foundations noted thus far, programming is first one that directly implies a computer. Finally, *design* synthesizes the previous three skills into a coherent solution for a specific problem addressing a particular set of concerns. For Denning, considerations include "practical issues" such as engineering, context, constraints, and performance requirements, all understood from the perspective of the human user.[7]

The first phase of the studio, lasting five of the term's thirteen weeks, focuses on building a workable foundation in algorithmic thinking, representation and programming. The structure of this phase consists of two parallel streams, a project stream and a workshop stream, that converge in a final exercise. Weekly day-long workshop sessions immerse students in the practice of algorithmic thinking and its requirement of discrete description. During the early sessions, the specifics of particular programming languages are suppressed and *pseudocode* is used as an accessible language for communicating algorithmic thinking. Pseudocode is a high-level description of an algorithm intended to be easily readable by humans so as to communicate the key steps of the algorithmic process, while retaining fundamental structures of programming languages such as conditional execution and repetition. Writing pseudocode is analogous to sketching in that the basic structure of a script or program is expressed before any of the coding takes place. Working with pseudocode simultaneously reinforces two properties of algorithmic thinking: the need to express problems and processes precisely with no ambiguity, and the fact that algorithmic thinking is independent of its specific implementation as programming in a specific language.

Only after a grounding in pseudocode do students proceed to their implementation in a specific programming language in order to test the algorithms they are developing. Since students are beginning the term with some familiarity with the Rhinoceros 3D modeling tool, it is natural to extend their toolset through exposure to related computing tools such as VB/RhinoScript[8] and eventually Grasshopper[9]. The following weeks' workshops further introduce students to the fundamental structures of algorithmic thinking and script development such as variables, logical operators, conditional execution, repetition, basic data structures such as arrays, and modular organization through the use of functions. In each case, students are introduced to the topic using pseudocode and then proceed to the specific implementation in VB/RhinoScript. In addition to covering basic programming structures, later workshops introduce students to more advanced topics such as non-deterministic execution and recursion. Each session concludes with an in-workshop exercise where students first analyze and then incrementally augment an existing piece of code in order to develop more complex behavior.

While some geometry is used during these workshops in order to visualize the outcomes of the algorithms, the role of geometry is kept to a minimum, using mostly primitives such as points, lines, boxes and spheres. The resistance to involving more complex geometry during these early stages is twofold. First, it is assumed that students already exposed to the basics of algorithmic thinking and its qualities of discreteness and precision will later be able to quickly grasp the applications of an advanced architectural geometry course. Second, while manipulation of geometry is clearly well suited to computational techniques, it is not necessary that an architect's relationship with computation is based only on geometry; other less explicitly formal areas of exploration might include simulation, digital fabrication, interpretation of site data, and responsive behaviors. Given the goal of seeding a computationally-oriented design culture, it is hoped that, equipped with a solid computational foundation, students will have the opportunity in subsequent course offerings to address elements in this expanding territory with more specificity and thereby develop a critical architectural practice.

The workshop topics begin with VB/RhinoScript and conclude with a pair of days committed to Grasshopper.[10] This ordering is deliberate. As mentioned earlier, expressing algorithms with pseudocode is analogous to sketching. Thinking through a problem by expressing it in pseudocode is an essential prerequisite, especially for novice algorithmic thinkers, before implementing the process in any language, be it a VB/RhinoScript text file or a Grasshopper definition. The pseudocode algorithm is structurally similar to an implemented text-based script, with the latter fleshed out by syntactical specificity and pragmatic considerations such as memory management. The programmer's task thus becomes one of translation; once the pseudocode – which is a precise, unambiguous, statement of a set of processes – is worked out the implementation in a scripting language is relatively straightforward. Such a direct relationship does not hold



TEXT BASED PARSING (SEGMENT 48s-136s)

time (s)	thumbLeft(x)	thumbLeft(y)	thumbLeft(z)	thumbRight(x)	thumbRight(y)	thumbRight(z)	indexLeft(x)	indexLeft(y)	indexLeft(z)	indexRight(x)	indexRight(y)	indexRight(z)	jointLeft(x)	jointLeft(y)	jointLeft(z)	jointRight(x)	jointRight(y)	jointRight(z)
48	49.08	26.09	59.63	55.50	36.03	39.60	53.96	24.10	56.34	56.74	35.96	37.96	56.77	37.32	68.57	41.66	28.70	45.49
78	47.14	28.46	62.20	34.65	44.81	71.67	45.00	22.83	60.91	41.30	31.40	58.37	55.33	38.27	70.46	23.44	44.32	68.91
97	54.77	29.84	58.59	24.30	18.58	73.04	53.00	24.15	47.47	39.56	15.62	76.00	54.25	36.28	67.30	20.74	29.47	77.06
120	47.4	25.96	69.96	48.52	22.59	43.22	54.57	22.28	66.95	52.99	14.38	35.46	60.28	36.29	75.79	35.93	20.99	50.95
136	54.71	25.45	49.65	36.11	38.93	74.28	52.08	24.02	64.88	44.62	31.10	60.65	59.58	35.42	75.38	23.57	36.04	74.29

Figure 2. Acquire and Parse. The first phase of the studio deals with the development of students' ability to structure their observations parametrically. Students are challenged to stalk an existing, observable, dynamic phenomenon and pursue its description as a parameterized system. Students need to seek out and invent associative properties within their chosen phenomenon. Here, Susan Han and Sheida Shahi (2010) set out to pursue the relationships between the position and orientation of hands while drumming.

true for pseudocode and a Grasshopper definition. Grasshopper is a powerful visual programming tool that allows designers to generate complex parameterized models very quickly and easily. So quickly and easily, in fact, that the results of a specific implementation may be unexpected, sometimes compelling, and thereby derail the developer from the original intent. While these unexpected results can undoubtedly provide opportunities for further exploration, especially for a more seasoned developer, it is essential that novice developers engage with the tools with a high degree of control. This is crucial in developing an expandable foundation where the tools are exactly that and not overpowering drivers themselves. By insisting on the role of pseudocode and its precise alignment with the outcome of a script, it is hoped that students carry this workflow forward into their explorations in Grasshopper where control of the outcome, at least in the beginning, is paramount.

SYSTEM STALKING: AQUIRE, PARSE, MINE & DEPLOY

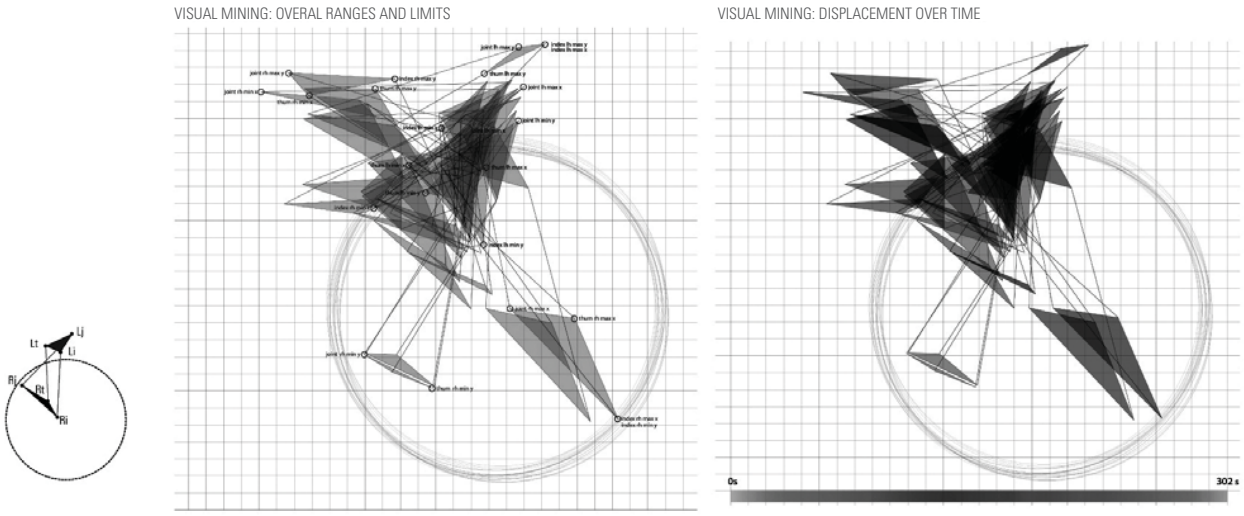
Concurrently with this series of workshops, students develop a project where they explore the processes and challenges of algorithmic thinking when confronted with a subject that may not, at least at first glance, be suited to such methods. The project, *Popping Up | Out of Nowhere*, asks students to choose an existing, observable dynamic phenomenon and pursue its investigation (or stalking). Some families of phenomena include: *Basic motion*, where the subject is one object moving as an assembly of components; *Series* where the investigation identifies a typology and explores its various instantiations; and *Swarms* where the phenomenon is seen as many objects moving as a whole. Once the phenomenon is selected students investigate ways in which the subject matter can be understood discretely. This process of understanding is comprised of four steps adapted from Ben Fry's Visualizing Data.[11]

The first step in the project is that of *acquisition*. Students are required to document their phenomenon. In its most basic form this documentation, or base data, consists of a series of photographs taken from two axes – not unlike Eadweard Muybridge's nineteenth century photographic series studying animal locomotion. Students are encouraged to explore additional dimensions of their phenomenon such as sound, intensity and time, as well as other methods of documentation. At the end of this first step students have a qualitative set of data describing their chosen phenomenon.

The second step – *parsing* – involves translating this qualitative representation into a quantitative one. Here students are required to structure some aspects of their qualitative set discretely. Most students do this by identifying a set of markers and then tracking their progression through the photographic series. Students choose a combination of explicit and implicit markers. The explicit ones, such as a discrete point or edge, are easily identifiable within each frame, while the implicit ones, like the center of a swarm, require some interpretation. At the end of this step students have two types of representation of their phenomenon in addition to the original photographic series: the annotated version of the series tagged with a set of attributes, and a tabular version where each marker in each frame is expressed in terms values such as x-/y-/z-position, color saturation, or length, just to name a few (Figure 2).

The third step introduces the idea of associative behavior or *parameterization*. The goal is that students analyze their phenomenon; leveraging the various representations they have created in order to understand or expose relationships between their markers. Strategies of filtering and data mining[12] are used to interrogate the data to define patterns, limits and ranges of behavior. On the one hand, students begin to define the behavior of one marker across the series by understanding its quality with respect to parameters such as range, displacement and rate of change; on the other hand students seek out associative behaviors between markers, defining parameterized relationships where an attribute of one marker affects an attribute of another. Once again students complete this step with two different representations: the filtered versions of their marked-up set where the relationships are graphically expressed, and a text-based expression of the extracted behaviors or rules (Figure 3).

Up until this point, students have been exploring algorithmic ideas without the use of computation-specific tools. Much of the work is in fact pen and paper in nature – carried out either through tracing and drawing or through writing out logical expressions. This again reinforces the notion that working computationally is as much about a way of struc-



NUMERICAL MINING

time (s)	length(Lt,Li)	length(Li,Lj)	length(Lj,Lt)	length(Rt,Ri))	length(Ri,Rj)	length(Rj,Rt)	Δlength(Lt,Li))	Δlength(Li,Lj))	Δlength(Lj,Lt))	Δlength(Rt,Ri))	Δlength(Ri,Rj))	Δlength(Rj,Rt))	dist(Rt,Lt)	dist(Ri,Li)	Δdist(Rt,Lt)	Δdist(Ri,Li)
0	10.35	16.62	9.71	9.54	5.45	14.37	-	-	-	-	-	-	42.42	32.06	-	-
8	4.02	9.23	11.32	3.80	18.42	15.10	-6.32	-7.38	1.62	-5.75	12.97	0.73	19.02	20.57	23.40	11.49
18	6.38	7.04	10.18	6.61	13.28	16.41	2.35	-2.19	1.14	2.82	-5.14	1.31	16.71	25.94	2.31	5.37
26	24.76	35.73	11.74	18.58	27.11	11.93	18.38	28.69	1.56	11.97	13.83	-4.48	34.26	65.45	17.55	39.51
27	12.05	4.67	7.77	9.11	23.09	15.37	-12.70	-31.06	-3.97	-9.47	-4.02	3.44	18.30	29.52	15.96	35.93
48	5.89	12.55	11.79	2.06	16.86	15.04	-6.17	7.88	4.02	-7.06	-6.24	-0.33	21.03	18.59	2.73	10.94
78	2.50	14.07	11.63	14.87	20.74	11.54	-3.39	1.52	-0.16	12.81	3.88	-3.50	15.67	4.49	5.36	14.10
97	12.25	19.87	7.73	15.54	18.85	5.37	9.75	5.80	-3.90	0.67	-1.89	-6.18	33.31	31.54	17.63	27.05
120	7.78	10.52	14.14	8.96	23.04	14.77	-4.47	-9.35	6.41	-6.59	4.19	9.40	26.76	31.53	6.54	0.01
136	15.46	12.90	26.19	16.07	25.08	12.54	7.68	2.38	12.05	7.11	2.04	-2.23	30.86	8.58	4.10	22.95
142	5.99	9.92	15.89	3.86	18.42	15.03	-9.46	-2.98	-10.30	-12.20	-6.66	2.49	15.80	22.57	15.06	14.00
148	10.80	21.25	11.53	21.16	22.78	10.10	4.80	11.33	4.36	17.30	4.36	-4.92	8.25	7.84	7.55	14.74
154	15.22	19.97	10.89	6.06	19.68	14.17	4.43	-1.28	-0.63	-15.10	-3.10	4.07	10.39	4.36	2.15	3.48
172	6.87	11.61	14.76	3.04	17.61	14.57	-8.35	-8.36	0.76	-3.02	-2.08	0.40	14.98	23.28	4.59	18.91
222	7.42	10.17	13.69	27.62	33.83	13.26	0.55	-1.44	-1.07	24.58	16.23	-1.31	24.11	43.76	9.13	20.49
241	8.12	10.29	18.27	4.48	20.93	17.28	0.70	0.13	4.58	-23.15	-12.90	4.01	16.61	15.67	7.51	28.09
247	11.89	21.78	13.58	19.47	27.41	11.58	3.76	11.49	-4.69	14.99	6.48	-5.70	39.90	48.98	23.29	33.31
267	8.61	12.22	14.33	13.29	9.77	9.87	-3.28	-9.57	0.75	-6.18	17.64	-1.71	8.02	13.50	31.88	35.48
288	22.70	34.18	11.48	19.60	23.56	9.94	14.09	21.96	-2.85	6.31	13.80	0.08	8.10	6.24	0.08	7.26
302	10.23	14.97	9.12	4.26	5.89	10.11	-12.47	-19.20	-2.36	-15.34	-17.67	0.17	43.89	35.11	35.79	28.87
MAX	24.76	35.73	26.19	27.62	33.83	17.28	18.38	28.69	12.05	24.58	16.23	9.40	43.89	65.45	35.79	39.51
MIN	2.50	4.67	7.73	2.06	5.45	5.37	-12.70	-31.06	-10.30	-23.15	-17.67	-6.18	8.02	4.36	0.08	0.01
AVG	10.46	15.48	12.79	11.40	19.59	12.92	-0.01	-0.09	0.38	-0.28	0.02	-0.22	22.42	24.48	12.70	19.58

Figure 3. Mine and Limit. Han and Shahi (2010) continue their analysis of their phenomenon by describing relationships between elements as well as limits in their behavior.

turing thoughts, investigations and methodologies as it is about specific tools.

After these initial steps, the project converges with the skills acquired during the workshop sessions. Working with the workshop exercises and the set of quantifiable behaviors or rules extracted from their analysis, students design and deploy a custom algorithm to execute a sequential transformation. Students pick a subset of the geometries emerging from their analysis and consider them as components in a series of transformation operations (Figure 4). Two kinds of transformation tend to emerge. The more basic approach transforms the position and orientation of the component at each step through translation, rotation and scaling, while the shape of the component itself remains constant; the second, more complex, approach has a transforming component where the geometry of the component also changes across the series. These two strategies allow students of all skill levels to design and develop their code, first through sketching (now understood as a combination of traditional visual expression and its new counterpart pseudocode), and then through implementation in VB/RhinoScript. In each case the logic of the transformation is rooted in the attributes and behaviors extracted during the analysis phase. It is not the intention of the designed transformations to replicate or simulate the original phenomenon. This is reinforced by the fact that the reference for the sequential transformation is no longer the original documentation but instead a synthesis of the parsed, filtered and mined representations that were created earlier.

Through these four steps of acquiring, parsing, mining and deploying, students are exposed to and become practiced in the four basic skills of computing (algorithmic thinking, representation, programming, design) as well as the overall concept of automation as outlined by Denning. Algorithmic thinking becomes familiar to students as a way of

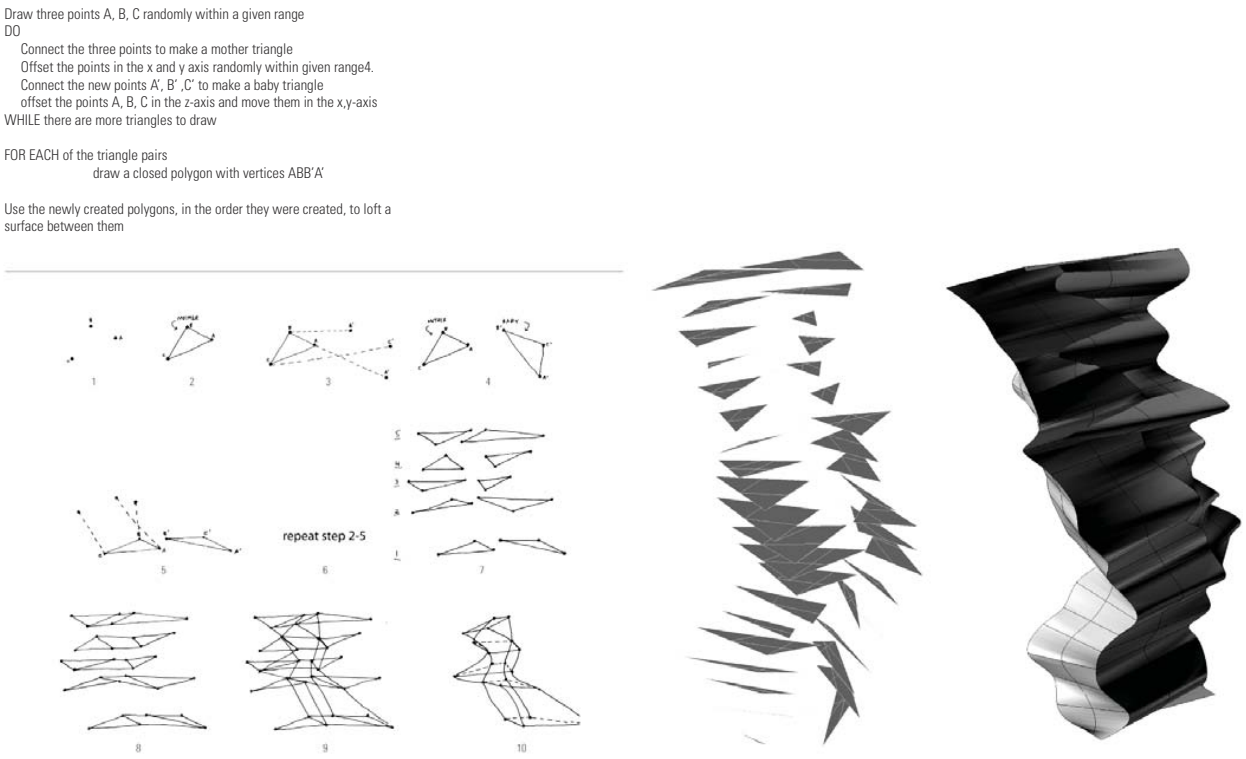


Figure 4. Design and Deploy. Han and Shahi (2010) define a sequential transformation with basic geometry and behaviors extracted from their analysis.

structuring thoughts and methodologies. Students are required to devise structured processes that enable the preparation of a set of interests for computational exploration. Developing satisfying methods of translating qualitatively understood aspects of a chosen phenomenon into a digitally understandable quantitative expression is a central issue. For many students, confronting the disparity between their intentions and their ability to structure these intentions algorithmically is frustrating; especially at first, when the necessity of abstraction reduces the phenomenon to an apparently oversimplified rendition. For example, one student initially interested in the complex of intensities, geometries and movements in a cloud-filled sky became focused on orthographic bounding shapes and their geometric centers. While seemingly reductive compared with more qualitative or fluid design exploration, such abstraction proved powerful in that it enabled action by making the problem more manageable; later, after seeking out associative relationships between their aspects of their analysis to describe systems of relations, the student was able to incrementally increase the complexity of their study and rely on the machine to manage the causal relationships they defined – greatly diminishing any sense of oversimplification as the project progressed.

Representation becomes a rich dimension of the work as students invent strategies for encoding phenomena to allow for algorithmic thinking. In this context the challenge of representation is twofold in that two constituencies are being addressed: the project's human audience which seeks to understand the project against numerous qualitative criteria such as experience, materiality and tectonics, social impact, site impact, economy, technical performance, to name a few; on the other hand, for purposes of computation, the project needs to be encoded in a discrete and unambiguous way. Students practice moving between multiple parallel modes of representation over the course of the project. In the end, they have created a collection of artifacts ranging from hand-drawn analytical tracings and their digital equivalents, to hand-written pseudocode segments and their formalized VB/RhinoScript implementation, to the outputs of the automation such as digitally rendered images or

digitally fabricated models. Each mode is related to the others: the pseudocode is often supplemented with graphical sketches of behavior, while examination of script outputs often leads students to return to their code to refine, correct and expand. Code, and its implication of automation, becomes part of the students' array of communicative tools.

Assuming the challenges of algorithmic thinking and representation are addressed, the challenges of programming are relatively straightforward. As a first step, students develop a working knowledge of a programming language, and learn to take advantage of program development aids such as editors and debuggers. The combination of workshops and project-making provides students with an effective context in which to learn these skills. Students translate algorithmic processes expressed in pseudocode into VB/RhinoScript instructions, assess the results, and reiterate the process while building in added layers of complexity. While the general logical flow of these processes is easily grasped, more abstract concepts such as the use and design of data structures are more difficult to learn and require more attention. Students can also expand their working knowledge to include more than one language, as each has its strengths and weaknesses. Understanding that the language is a variable in the process further decouples the foundations of algorithmic thinking and representation from the specifics of a given particular programming language.

The design/deploy exercise allows students to synthesize the previous three skills into a coherent solution for a specific problem. The domain of an architectural design problem is vast, rich with both qualitative and quantitative concerns. Students are confronted with the challenge of defining and prioritizing their own set of considerations. Within a computational context they also need to decide which dimensions of the problem could benefit from computing (read automation) and begin to

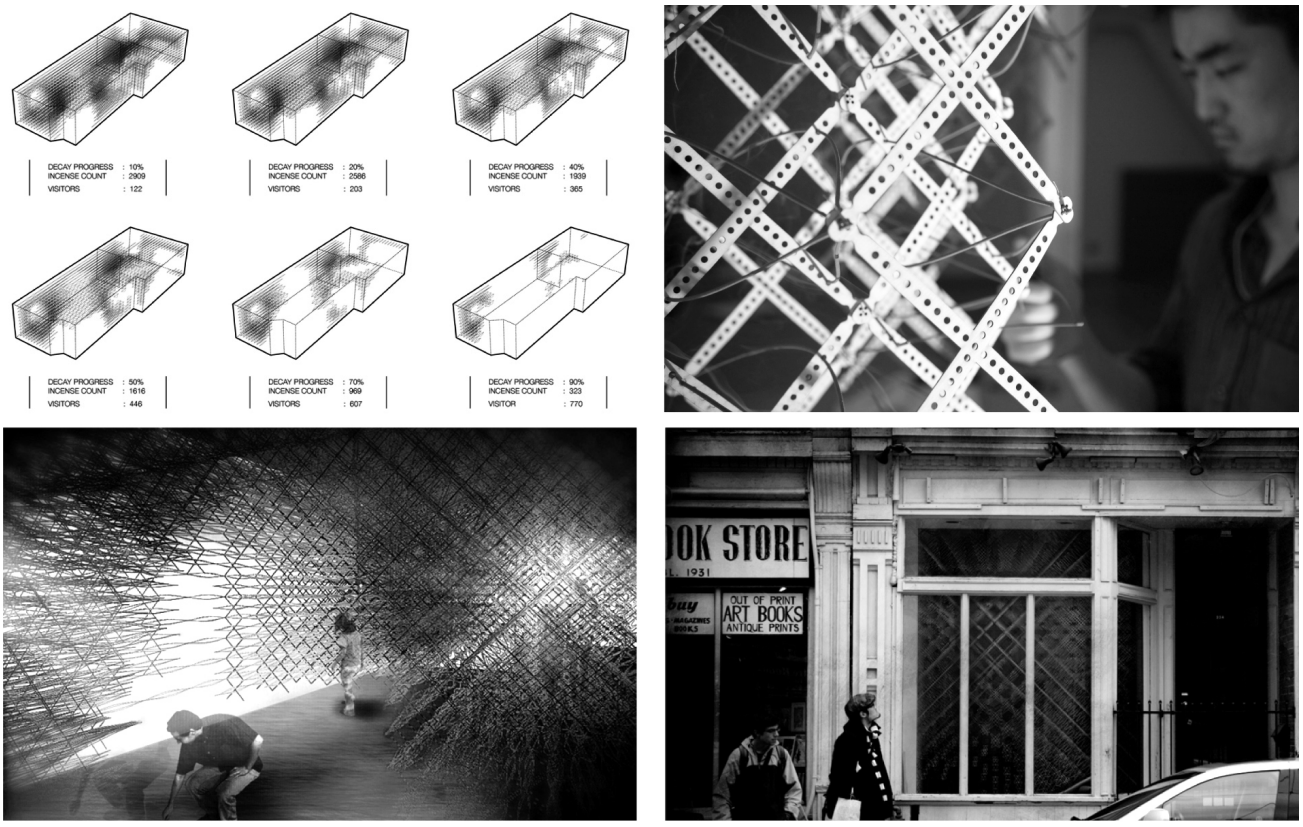


Figure 5. InSense Space: An SSL project by Connor O'Grady and Harry Wei (2010). The students conceived of an architecture that would emerge as an index of the inhabitation of a space over time. They began by developing a detail and exploring its deployment by physical modeling at a 1:1 scale. This informed their concept of its deployment at the site, starting as a matrix built up of these simple units, and the students then developed tools to simulate the emergence of the environment in response to stimuli over several weeks.

structure its exploration. Students identify an aspect of their analysis and begin to test its potentials with respect to automation by defining a sequential transformation. At the beginning this exploration is again somewhat frustrating, as students' computational skills are not yet developed to equal their other design skills. However, many students catch on to the similarity between developing code and developing a more familiar design project. Applying recognizable ideas of iteration, progressive development and scalar shifts, students are able to establish a framework for their development and incrementally introduce more complex behavior. The analogy of students' non-computational design process of moving from a state of abstraction to resolution is invaluable in communicating the process of developing algorithmic processes.

Thus the first phase of SSL sees the students engage with computation, enabling them to develop project-specific tools to structure their work as a system, and then explore the space of that system and develop it in an iterative manner to arrive at the final proposition. The process exposes them to the skills necessary for the conceptualization, design and execution of a project operating within a computational discourse in a highly structured way. At this point in the term, roughly week 6 of 13, the foundations necessary to becoming computational have been delivered, allowing the remaining time in the studio to be used to test, reinforce and expand on these foundations in a more open-ended project (Figure 5).

ENDNOTES

1 Edsger Dijkstra (1930-2002), a Turing Award (1972) winning computer scientist known for his contributions to the development of programming languages, graph theory and distributed computing.

2 Andrew Kudless expressed much of this relationship between architectural work and processing in the introduction to "Generative Design" an advanced elective course offered at the California College of Arts, MEDIAlab in 2008.
(http://mlab.cca.edu/2008/12/generative-design/)

3 Denning, Peter J. "Computer Science: The Discipline" (1999): p 3. Web. July 14, 2011 (http://cs.gmu.edu/cne/pjd/PUBS/ENC/cs99.pdf).
This article was prepared by Denning for the fourth edition of the Encyclopaedia of Computer Science (A Ralston and D. Henning, Eds).

4 Kostas Terzidis elaborates on the potentials of computation versus computerization as related to nondeterministic processes in his book Algorithmic Architecture (2006, Oxford: Architectural Press).

5 Kilian Axel. Design Exploration through Bidirectional Modeling of Constraints. (2006): p 23. Web. July 05, 2011. (http://hdl.handle.net/1721.1/33803)
This book was submitted as Kilian's Ph. D. Thesis at The Massachusetts Institute of Technology, Dept. of Architecture, 2006. Advised by Takehiko Nagakura.

6 Denning. p.3.

7 Denning. p.4.

8 RhinoScript is a scripting tool, developed by McNeel, based on Microsoft's VBScript language. With RhinoScript, you can quickly add functionality to Rhinoceros 3D, or automate repetitive tasks.
(http://wiki.mcneel.com/developer/rhinoscript)

9 Grasshopper® is a graphical algorithm editor tightly integrated with Rhinoceros 3D's modeling tools.
(http://www.grasshopper3d.com/)

10 Mode Lab conducted the 2-day grasshopper workshop.
(http://modelab.nu/)

11 Fry, Ben. Visualizing Data.(2007): p. 5-18. Cambridge: O'Reily Media Inc. Print.

12 In Visualizing Data (page 5), Fry defines mining as the application of "*methods from statistics or data mining as a way to discern patterns or place the data in mathematical context*". Data mining is understood as the non-trivial extraction of implicit, previously unknown and potentially useful information from data.