# Design Computation With Eyes and Hands

**Author** Derek A. Ham
*NC State University College of Design*

## INTRODUCTION

Most design education curricula have not caught up with the computing demands of the profession. While the tools designers use are becoming more digital, the same can be said about the products designers are taught to create. The posters and books of yesterday have been replaced with digital artifacts in the form of websites, eBooks, apps, and now virtual reality (VR). As the domain of graphic design expands, many schools have responded by shifting their focus away from the things designers make and place greater emphasis on the things designers do. As a result, design students are lead to believe that their place in the world is in being an organizer, researcher, and developer of the "big idea." This pedagogical shift in design education has left students empty in their ability to actually make anything. Years past, a graphic design student would actually make their designs because they were all physical based prototypes. The physical world was a domain the educational community could handle. Today however, students do not actually "make" their projects. Apps are presented as "digital mockups" taking on the form of animated videos or illustrated images. In fact, in many undergraduate design programs, projects that step into the territory of digital or computationally driven fall short in becoming fully actualized. The cause for this is simple. Coding is not yet considered a core competency in design education.

For a long time, designers have placed a great emphasis on tactile approaches to learning such as hand drawn sketching, diagramming, and model building. In many cases, designers used these techniques to separate themselves from other professions; it was the skill that gave them creative superiority over other professions (Allsopp, 1952; Lawson, 1990; Lawson, 2004). While sketching is not being removed from design education, this method of process thinking is in desperate need of a reboot. Designers must adopt new methods to reach places where they are not only just "users" of new technology but hackers and creators of new technology themselves.

Beyond project-based learning and beyond the STEM to STEAM movement, there are many concepts found in computing education that could help expand design education and provide students with the skills necessary to be both visually artistic and analytically systematic. Computational thinking expressed through visual systems is a way to bridge the gap between different modalities of thinking. We can use rules and algorithms along with intuition to be artistically creative. To further this pedagogy, I have documented what happens when students are introduced to computing ideas through shape grammars as a scaffold approach towards learning to code. This research gives insight into how students studying design acquire computing concepts and adopt them into their creative process. For educators, this sheds new light on how we can build algorithmic and analytical problem solving approaches into foundational art and design studies.

## MOVING BEYOND CODING TOWARDS COMPUTATIONAL THINKING

Design education has historically kept distance from methods that involve calculation and computing. Having one foot in the arts and another foot in the analytical has made design education fickle. Computation is only brought into the discussion when a project's outcome is situated in a digital domain. Even then, many designers prefer the methodology of prototyping through representation rather than wrestling with computer code to make the prototype work.

Computing in design education should not be limited to coding activities centered on the production of digital artifacts (apps, websites, etc.). Computational thinking offers a much broader approach to design activities that are useful for students. As Wing states, *"Computational thinking is using abstraction and decomposition when attacking a large complex task or designing a large complex system. It is separation of concerns. It is choosing an appropriate representation for a problem or modeling the relevant aspects of a problem to make it tractable"* (Wing, 2006).

A design approach that embraces computational thinking allows students to create solutions that are flexible, dynamic, and responsive to change. This approach requires designers to place an emphasis on systems that are parametric having several interlocking components.  Conceptually this might be easy to understand in the context of web-design and digital based projects, but the usage of computing approaches towards aesthetic decision making in the creation of letter forms, logos, and other analog design systems are often overlooked.  Educators often equate computing with digital, and this is an unfortunate mistake.

We can use computational thinking in almost every human endeavor once we expand our understanding of what it means to compute. In the broadest sense, computing can be defined by three primary components: variables, rules (or procedures), and schemas. Computational thinking is our ability to process complex thoughts based on the development of new rules and schemas. The first stage of computational thinking begins with the assignment of discrete "thought variables." A variable can come in any form, from the concrete interpretation of our body's senses all the way to the abstract notion of the formless, such as the concept of time.  In computational thinking, we take these variables of thought and assemble them with spatial arrangements. Each thought-arrangement becomes a computational rule that can further be processed to form generic schemas of thought. We use variables, rules, and schemas to drive our actions or internally create new thoughts that can loop into a much deeper level of computational thought.

Howard Gardener's work, that expanded our understanding of intelligence, presents a fine example of how we might situate computing education beyond analytical domains [5]. Before the *Theory of Multiple Intelligences,* most spoke about intelligence as something that manifested itself through linguistic and analytical expertise. Gardner expanded intelligence manifestation into new categories including musical, spatial, bodily kinesthetic, inter-personal, and intra-personal. In each of these domains we can articulate computational systems comprised of variables, rules, and schemas. As Gardner states, *"One might go so far as to define a **human intelligence as a** neural mechanism or **computational system** which is genetically programmed to be activated or "triggered" by certain kinds of internally or externally presented information"* [5].  Educators then can understand computational thinking as something to be embraced by any discipline. In the case of design, which relies heavily on spatial logic, computational thinking will manifest itself through visual calculation.

## TEACHING DESIGNERS TO THINK COMPUTATIONALLY

The first step towards getting design students to be more computational in their creative process is to get them to embrace computational thinking. One methodology that has proven useful to do this is by teaching them "shape grammars"(Ham, 2015). Shape grammars, created by MIT professor George Stiny (2006), can be used to explore art, architecture, and design through formal rules and algebraic descriptions.

### *Shape Grammars Background*

Shape grammars have three major components: variables, rules, and schemas. The variables found in shape grammars come in the form of shapes from the zero-dimension all the way to the third dimension. Points are zero-dimensional variables, lines are one-dimensional, planes are two-dimensional, and solid-volumes are variables of the three-dimension. Variables grow in dimensionality by embedding. Within every dimension (excluding zero-dimensions) we can always find sub variables within them as the necessary components to articulate that level of dimensionality. Embedding is a key component to shape grammars and what makes computing in the visual domain so special.

In traditional computing systems, variables are often handled in a combinatorial manner. Variables are articulated at the start of a function and then never change. With shape grammars this is not the case. The shape variables we use can be changed at any time, as can the rules we create to play with them. We can essentially erase previous statements. Consider the following shape grammar rule where "shape x," an irregular hexagon, is rotated around a fixed point.

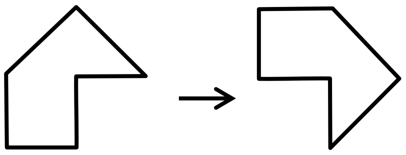


*Figure 1: Shape rule showing a rotation of an irregular hexagon*

The graphic shown in Fig. 1 demonstrates the way shape grammar rules are traditionally represented, with the depiction of the shape following the rule detonated on the right hand of the arrow. The same rule can be expressed algorithmically in shape grammars with the following schema: $x \rightarrow t(x)$.

If I decide to change the rule to represent not just the rotation of my initial variable, but to include the copy and rotation, I would use the following schema: $x \rightarrow x + t(x)$, with the following rule (Fig. 2).

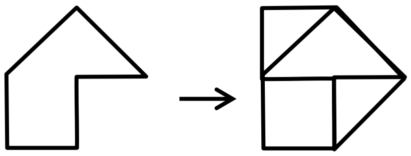


*Figure 2: Shape rule showing a copy and rotation of an irregular hexagon*

This is where we find embedding in shape grammars. The rule shown in Fig. 2 yields much more to our eyes than what was put into it. We can still see the original irregular hexagon shapes, but now, we also find several other shapes (squares, triangles, etc.). Any of these additional shapes can be our new variables to play with, and we can create new rules for them. In many cases the emergent forms become more visually dominant than the original shape variables we started with. This is how computing in shape grammars works.

*Shape Grammar Design Activity*

The kit of parts is an exercise given to design students to introduce them to computational thinking with shape grammars [6]. Students are presented with variables in the form of four identical shape pieces. These shapes can be arbitrary, but the size and material quality is key. First, their size is no larger than the equivalence of playing card (44 x 60mm). Secondly, these shape pieces are printed or drawn on a transparent material to allow the students to slide them over each other through the act of embedding (Fig. 3).

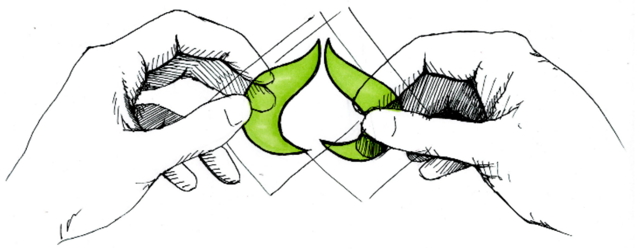


*Figure 3: Hands at play working with shape variables*

Students play with their shapes and begin to articulate specific rules that communicate design gestures. These rules are explicitly written in the form of a shape rule or/and algebraically as a shape grammar schema. The flexibility to represent rules visually as a graphic or algebraically in the form of a schema makes shape grammars easy to grasp and record. Design students are used to capturing ideas in their sketchbooks, but never have they recorded their ideas with such formal precision. This step is essential for the students to adopt computational thinking in their practice. They are learning to "teach their design strategies to a machine" (Knuth, 1984).

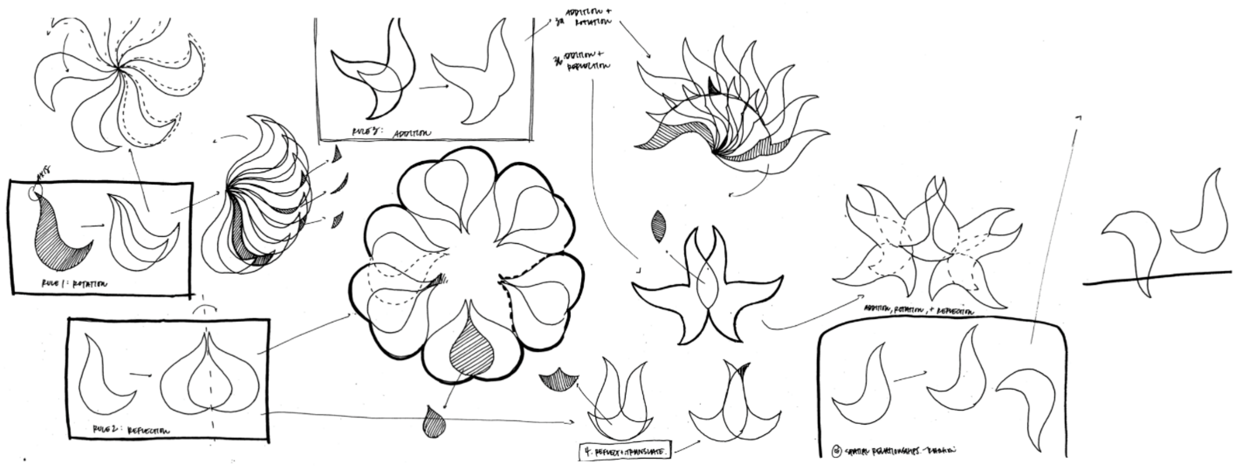


*Figure 4: Student sketchbook documenting shape grammar rules and schemas in their thought process*

## Studio Task 1: Using Shape Grammars For Analysis

Students in a second year graphic design studio were given a project structured to present them with a primer on shape grammars and computational methods of design activities. This design activity was completed over one week, where students met in a course commonly called a "design studio" named after the same space the class is held. Here students meet with a faculty member for three hours on three separate days. Through active learning, students would learn how to generate and then use a set of rules as guiding principles for the development and analysis of a graphic design system. Although we were calling the system "computational," students would learn the process of working with shape grammars by hand through what Terry Knight (2012) calls "slow computing." It is with this method that design students can learn best how to create shape-rules and schemas that either serve as the building blocks of a new design or as a descriptive way to discuss an existing design. Another term introduced to the students (from shape grammars) was visual computing.

Underneath this instructional approach lies a framework built on visual calculation. The premise behind visual calculation is that design can carry with it a set of formal descriptions that articulate the methods carried out to achieve its visual form. We can think of these formal descriptions as recipes or algorithms; they are procedural steps that communicate the core components of a design system. Skilled designers know that the process to a solution is not linear. Spatial thinking allows for an expanded view on computational thinking, making it flexible, and allowing for the user to take into account of the larger set of known and unknown factors.

In the first step of the design-computing exercise, students analyzed an assigned corporate logo through shape grammars. The design student's task was to deconstruct the logo into a set of shape variables that were assembled through a set of rules. Shape grammars served as a visual schema to describe the visual identity of the company.

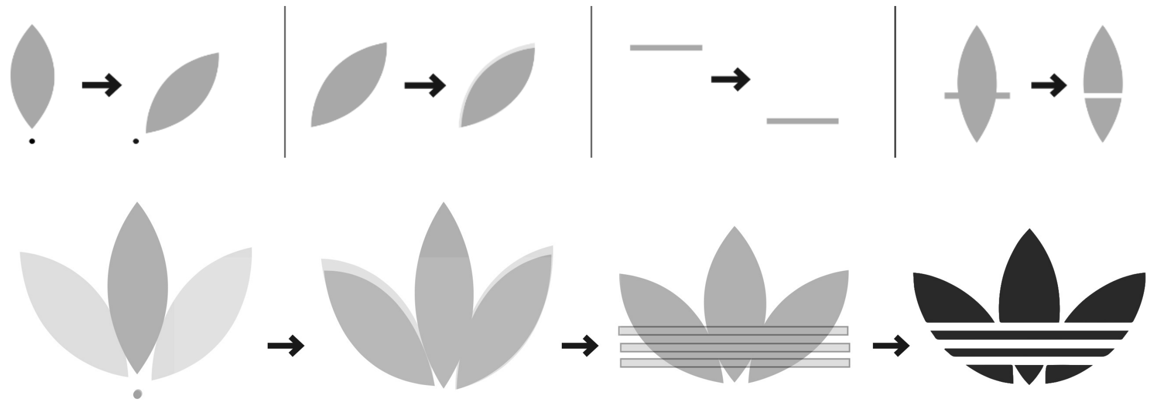


*Figure 5: Student work using shape grammars to deconstruct the "Adidas Logo"*

Studio Task 2: Using Shape Grammars For Synthesis

The second task of the design project allowed students to use the design grammar they had created as a catalyst for new visual designs.  This component of the design exercise involved the application of shape grammars as a generative tool to create new designs. Once students can articulate a grammar system pulled from an existing logo or brand system, they can then use it to generate new designs of the same kind. Students use shape grammars to create alternative design-brand strategies for the company.

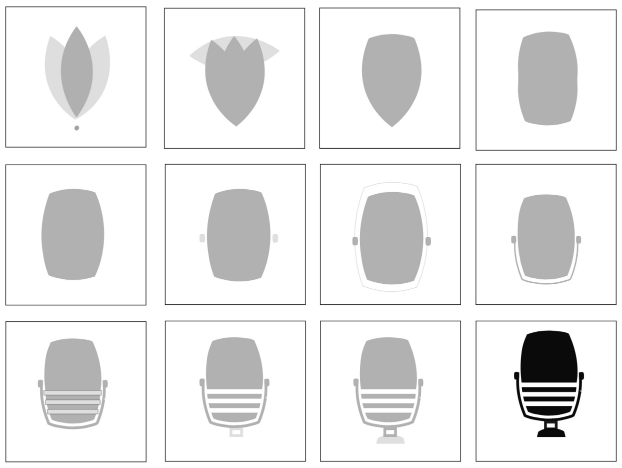


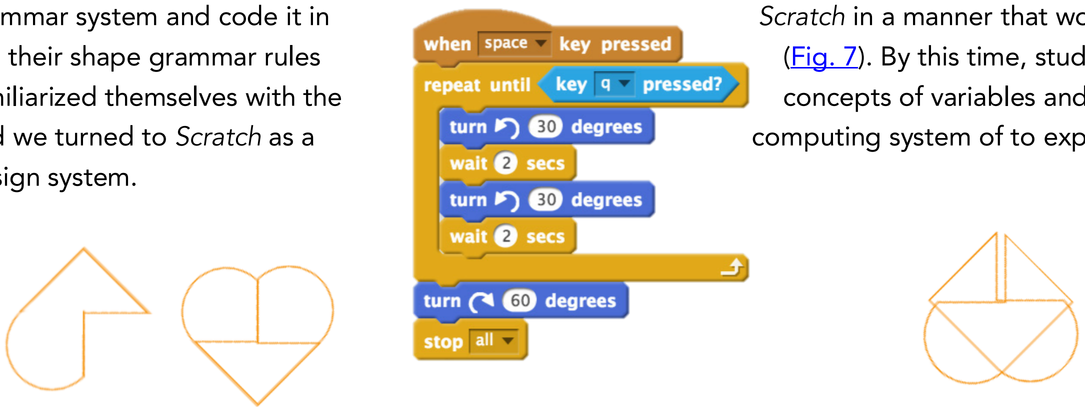
***Figure 6:*** *Student work using "Adidas" shape grammar schema to design an icon for a music app*

The projects yielded from these design exercises allowed students to develop computing strategies in a way that was intuitive and flexible. Students could review their "design recipe" and alter the formula at any stage resulting in a different visual outcome. The concept of "parametric design" is foreign to designers whose methods are void of articulating variables, rules, and an overall design grammar. This process of shape grammars goes beyond system thinking and situates design education right at the front door of computing education. The chasm between the two does not need to be as vast as it has historically been.

## TEACHING DESIGNERS TO CODE

Studio Task 3: Working With Scratch

A final component to the graphic design studio’s design exercise prompted the students to go into *Scratch* and create a “logo-generating application.” Students were instructed to take a shape from their grammar system and code it in *Scratch* in a manner that would carry out their shape grammar rules (Fig. 7). By this time, students have familiarized themselves with the concepts of variables and rules, and we turned to *Scratch* as a computing system of to expand their design system.



```
when [space] key pressed
repeat until <key [q] pressed?>
    turn ↺ (30) degrees
    wait (2) secs
    turn ↺ (30) degrees
    wait (2) secs
end
turn ↻ (60) degrees
stop [all]
```

***Figure 7:*** *Student project example using Scratch to extend their shape grammar rules*

While the compositions carried out by *Scratch* allowed for embedding and shape emergence, the system was not robust enough to target the new shapes that emerged from the geometric transformations. Scratch still runs off of combinatorial logic. This is even more evident in the way the code blocks snap together. The interface not only borrows from *Lego’s* mechanics but also has strong visual similarities. Combinatorial play is often a great starting point for young learners but should not be seen as the end-all. Even within the realm of coding, there can be several in-between moments that are wonderful and should be explored. If students only limit themselves to seeing variables as 0-dimensional units, then they will miss opportunities to see the emergence of new design solutions.

*Making Coding Visible*

Visual Programming Languages (VPL’s) work for designers because of the inherent spatial thinking component attached to them. Providing a visual and diagrammatic understanding for what a line of code is saying must come before the student learns the syntactical arrangement of any given programming language. This can then be reinterpreted into code line indentation. Indentations might be secondary for skilled coders, but for designers and novice learners, it visually reinforces the understanding of the hierarchy presented in their command lines, something the early coding consoles lacked. Contemporary IDE’s have now adopted a color based notation system that assisted in the further dimensional qualities of the programmer’s visual model of the program. VPL’s take this a step further in the creation of “code blocks” giving designers the visual tools necessary to understand the logic behind the code. Together, the design qualities of integrated development environments (IDE’s) have allowed programmers to develop mental models of their work allowing them to think both spatially and computationally.

For designers, the core concepts of coding must be expressed spatially. Educators should give students visual equivalents to all coding concepts.  In doing this task the computing educator might explain concepts like conditional statements (if and else) and looping events with visual equivalents. When they describe these concepts, they might move their hands around and when convenient draw diagrams on a white boards. VPLs build on this spatial connection, by providing ready-made tools to allow users a visual and intuitive method to code.  This should not be perceived as a distraction from traditional IDE's, but understood as building foundational understandings for computing.

Visual programming tools amplify the visual/spatial qualities of coding by further abstracting the process away from syntax based lines of code to interactive code blocks.  Visual programming tools have the ability to deconstruct the presumably difficult and complex activities of coding for design students. Just as shape grammars merge spatial and computational thinking, VPLs open up an alternative way to think about code spatially and computationally. The barriers for entry to get started with VPLs are typically very low. The user interfaces and actions found in VPLs are noted to be much easier to navigate and comprehend than that of the programing environments found in text based IDEs. *Scratch* uses color-coded blocks that fit into each other like *Legos* to run its functions (Resnick, 2001). The advantage of this system is a built in editor that prevents students from assembling code logic that is incoherent. Students still need to go through the process of debugging, but much of the legwork is completed by the system that only permits certain blocks to work together.

## *RESULTS and FURTHER DISCUSSION*

### Learning With VPLs

What many students come to realize after completing these exercises, is that coding is simply another tool in their design tool belt; true computing begins with thought and makes its way down into the design methods well before learning to code, Computational thinking is not about a design process that is in competition with our digital tools; nor is this about making design students more machine-like in their process. To do so would be a losing effort; computers carry out looping behaviors in much more efficient fashion. Students were able to witness this once their logo shape grammar rules were embellished with functions that animated their shape variables. The computer could perform the geometric transformations of their shapes much faster than their hands could.

After completing the entire series of exercises, students realized that computational thinking is about how designers can co-create systems with digital tools. We can create much more elaborate patterns to solve problems and be creative even in the visual domain. This thinking creates solutions through the use of variables, rules, and schemas. The approach requires the student to analyze the task and plan out the system that will compute it. This approach is carried out through reflective thought and constant debugging until the optimal performance is met.

Giving students a methodology to visualize code outside of traditional syntactical structures enabled them to think about code structures universally in ways that are both spatial and computational. Students could think about code from new angles, and make mental connections with their variables and rules. Spatial and visual thinking become the forefront in work in parallel with computational thinking. Visual programming is intrinsically useful to design students who already think spatially, but it can also be useful to anyone.

Once students of design embrace computational thinking, the jump to learning to code is not so severe. For many design students, the most intimidating feature of coding is the abstracted and foreign nature of programming languages and the screen environments in which we write them. IDE's lack the tactile and dynamic qualities found in most design creation software. Adobe Photoshop and Adobe Dreamweaver are worlds apart. A scaffold approach to traditional IDE's has proven to be the best approach with design students beginning with VPLs.

## CONCLUSIONS

Understanding "computing knowledge" as design as opposed to "computing knowledge" as information gives a much more expanded view allowing us to desegregate design and computing disciplines from being separate taught subjects.  Shape grammars are a very helpful mechanism that bridges the gap between traditional computing education and that of art and design. The approach allows educators to scaffold in key ideas that bolster creativity and provide easy access to working with design projects that require coding and analytical thinking. Shape grammars help articulate and categorize variables playfully. When we allow students to pair their visual thinking with computational thinking we find new opportunities to expand both educational domains. Together, both ways of thinking fuel the way we can be creative and design iteratively, while fostering reflective practice.

## REFERENCES

1. Allsopp, B. (1952) Art and the nature of architecture. London, England: Sir Isaac Pitman & Sons.

2. Lawson, B. (1990) How designers think. London, England: Butterworth Architecture.

3. Lawson, B. (2004) What designers know. Oxford, England: Elsevier.

4. Wing, J. (2006) Computational Thinking. Communications of the ACM. March 2006, Vol. 49(3) 33-35. Gardner, H. (1983) Frames of mind: The theory of multiple intelligences. New York, NY: Basic Books.

5. Ham, D. (2015) Playful Calculation. In Revolutionizing Arts Education in K-12 Classrooms Through Technological Integration, N. Lemon (Ed.). IGI Global, Hershey, 125-144.

6. Stiny, G. (2006) Shape: Talking about seeing and doing. Cambridge, MA: Massachusetts Institute of Technology. Knuth, D. (1984) Computer Science and Mathematics. In Campbell, D. M., & Higgins, J. C. (Eds.), Mathematics: people, problems, results Wadsworth International, Belmont, CA,  25-37.

7. Knight, T. (2012) Slow computing: Teaching generative design with shape grammars. Computational Design Methods and Technologies: Applications in CAD, CAM and CAE Education. IGI Global, 34-55. Web. 27 Aug. 2014. Resnick, M. (2001)  Closing the fluency gap. Communications of the ACM, 44(3), 144-145.