“The nature of form in the digital age is trapped in the invisible realm of code. Form+Code makes that world visible to the community that stands to gain the most from it: artists and designers.”
—John Maeda

“At long last, here is a publication that looks comprehensively at the contemporary digital medium with clarity, at its recent past and into the future.”
—Greg Lynn

“Elegant as an algorithm, clear as a program, and as enthralling as a video game, Form+Code is a powerful tool, both as a tutorial and as an in-depth analysis of the aesthetics of the information age.”
—Domenico Quaranta

“This incredibly rich study of the history and possibilities of creating media through code is a must-have reference collection.”
—Karsten Schmidt

www.formandcode.com

Once the exclusive domain of programmers, code is now being used by a new generation of designers, artists, and architects eager to explore how software can enable innovative ways of generating form and translating ideas.

Form+Code in Design, Art, and Architecture offers an in-depth look at the use of software in all creative disciplines. This stunning visual survey introduces readers to more than 300 significant works of the past 60 years in the fields of graphic design, typography, data mapping, art, digital fabrication, interactive media, gaming, artificial intelligence, artificial life, and all forms of new media and expression.
This book is dedicated to the students in the Department of Design Media Arts at the University of California, Los Angeles.
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTRODUCTION</td>
<td>8</td>
</tr>
<tr>
<td>WHAT IS CODE?</td>
<td>10</td>
</tr>
<tr>
<td>FORM AND COMPUTERS</td>
<td>26</td>
</tr>
<tr>
<td>REPEAT</td>
<td>42</td>
</tr>
<tr>
<td>TRANSFORM</td>
<td>66</td>
</tr>
<tr>
<td>PARAMETERIZE</td>
<td>92</td>
</tr>
<tr>
<td>VISUALIZE</td>
<td>118</td>
</tr>
<tr>
<td>SIMULATE</td>
<td>146</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>172</td>
</tr>
<tr>
<td>IMAGE CREDITS</td>
<td>173</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>174</td>
</tr>
<tr>
<td>DESIGNER/ARTIST/ARCHITECT INDEX</td>
<td>176</td>
</tr>
</tbody>
</table>
Software influences all aspects of contemporary design and visual culture. Many established artists, such as Gilbert and George, Jeff Koons, and Takashi Murakami, have totally integrated software into their processes. Numerous prominent architects and designers use software extensively and commission custom programs to realize their ideas. The creators of innovative video games and Hollywood animated films also write software to enhance their work.

While these exciting developments are taking place at the highest levels of the creative professions, integrating them into design education is a challenge. Even the most motivated student will find the technical boundaries difficult to overcome. As a comprehensive first introduction to software development within the arts, this book seeks to encourage the enthusiasm that the field requires. It will not, however, teach you to program computers. To satisfy that urge, see Processing: A Programming Handbook for Visual Designers and Artists by Casey Reas and Ben Fry.*

In Form+Code in Design, Art, and Architecture we define form as visual and spatial structures; code is defined primarily as computer programs, but we extend the definition to include instructions beyond computer code. The book is organized into seven chapters: What is Code?, Form and Computers, Repeat, Transform, Parameterize, Visualize, and Simulate. The first two chapters set the foundation by defining terms and introducing basic concepts. The themed chapters that follow are deeply linked to code. Each begins with an essay to define the territory, continues with images and captions to clarify and explain each theme, and concludes with two illustrated examples of programs. The corresponding source code is available in multiple programming languages and can be downloaded for free from the book’s website: http://formandcode.com.

We’re tremendously excited about the potential for creating form with code. We hope this book will inspire readers to think further about the relationships between these topics.
Codes typically serve three main purposes. They are used for communication, clarification, or obfuscation.

In Morse code, a word is transformed into short and long pulses so that it can be communicated over a telegraph. The word my is encoded by the sender into "- - -," the resulting sound is then decoded back into my by the receiver.

Genetic information is encoded in sequences of deoxyribonucleic acid (DNA), such as "AAAGTCTGAC," with A standing for adenine, G for guanine, T for thymine, and C for cytosine. The genetic code is a set of rules that use these sequences to build proteins.

The California Health and Safety Code is a set of written laws that codify the rules set forth by the State Legislature. For example, section 12804 states, "Flammable liquid means any liquid whose flashpoint is 100 degrees Fahrenheit, or less."

Ever since the origins of writing, codes have been used to protect messages from unwanted eyes. For example, a code can be as simple as replacing each letter of the English alphabet with a number: A is 1, B is 2, C is 3, etc; with this code, the word secret becomes "19, 5, 3, 18, 5, 20."
THE ALGORITHM

There are many types of code. Within the context of this book, we're interested primarily in codes that represent a series of instructions. This type of code—often called an algorithm, procedure, or program—defines a specific process with enough detail to allow the instructions to be followed.

While the word algorithm may be unfamiliar to you, it's the same idea. It is commonly used within the context of computer instructions. While most people wouldn't refer to a pattern for knitting a scarf as an algorithm, it's the same idea.

While the word algorithm may be unfamiliar to you, its meaning is not. It's just a precise way of explaining how to do something. Algorithms can be defined as having four qualities. These qualities can be easily understood when defined in relation to travel directions.

- There are many ways to write an algorithm. In other words, there are always multiple ways to get from point A to point B. Different people will create different sets of directions, but they all get the reader to their intended destination.

- An algorithm requires assumptions. Hiking directions assume that you know how to hike, from knowing to wear the right shoes, to understanding how to follow a winding trail, to assuming that you know to bring plenty of water. Without this knowledge, the hiker may end up lost and dehydrated with blistered feet.

- An algorithm includes decisions. Directions often include instructions from different starting locations. The person reading the directions will need to choose a starting position.

- A complex algorithm should be broken down into modular pieces. Directions are often divided into small units to make them easy to follow. There may be separate directions for coming from the North or South, but at a certain point the directions converge and both groups follow the same instructions.

Likewise, directions to get from one place to another, instructions for assembling kit-of-parts furniture, and many other types of guidelines are also algorithms.

Hiking Directions to Point Break

From the North:
- Follow the trail from the Nature Center
- Turn right at the Water Tower
- Walk until you see the Old Oak Tree
- Follow directions from the Old Oak Tree

From the South:
- From the Pines Grove, follow the Boggy Trail
- Turn right on the South Meadow Trail
- Turn right on the Meadow Ranch Trail
- Walk until you see the Old Oak Tree
- Follow directions from the Old Oak Tree

From the Old Oak Tree:
- Follow the path under the tree
- Turn right onto the Long Hill Trail
- Follow the trail until you reach Point Break
In computer programming, code (also called source code) is used to control the operations of a computer. It is an algorithm written in a programming language. There are thousands of programming languages, and new ones are developed every year. Although the words and punctuation used in programming languages look different from written English words, the codes they use are intended to be read and understood by people. Specifically, computer-programming languages are designed for the way people are taught to read and write from an early age, with the precision necessary for constructing a computer. Human languages are broad, ambiguous, and contain large vocabularies. Code is terse, has strict syntactical rules, and small vocabularies.

Writing in a human language allows the author to utilize the ambiguity of words and to have great flexibility in constructing phrases. These techniques allow multiple interpretations of a single text and give each author a unique voice. Each computer program also reveals the style of its author, but there is far less room for ambiguity. In fact, there can only be one interpretation of every piece of code. Unlike people, computers are not able to guess or interpret a meaning if it's not stated exactly. There are rules of grammar in every language, but if I misspell a word or two, you will still understand, but the computer won't. Fortunately (or maybe unfortunately), people are highly adaptable, and for many individuals it's easy to learn how to structure code.

Before a piece of code can be run on a computer, it must be converted from a human-readable format to a computer-executable format; these are sometimes called machine code, binaries, or executables. This conversion transforms the code into software. It can now run (or execute) on a computer. Machine-based code is usually represented as a series of 1s and 0s:

```
0011 0011 0001 1000 0001 0000 0001 1001 0000 0001 0000
0010 0010 0010 0000 0010 0000 0010 0000 0010 0000
0010 0010 0010 0000 0010 0000 0010 0000 0010 0000
0010 0010 0010 0000 0010 0000 0010 0000 0010 0000
```

While this series of 1s and 0s looks different from source code, it's a literal translation of the human-readable code. This translation is required for the computer to be able to follow the instructions. We think you'll agree that understanding what these sequences of 1s and 0s mean is more difficult than reading the source code. This format instructs the computer's operations at the lowest level. Each bit (1 or 0) is grouped into bytes (a sequence of eight bits) that define how the computer makes calculations and moves data into and out of the processor.
THINKING IN CODE

Software is a tool for the mind. While the industrial revolution produced tools to augment the body—such as the steam engine and the automobile—the information revolution is producing tools to extend the intellect. The software resources and techniques at our disposal allow us to access and process enormous quantities of information. For example, the science of genomics (the study of the genome) and the collaborative scholarship of Wikipedia were not possible without the aid of software. But using software is not only about increasing our ability to work with large volumes of information; it also encourages new and different ways of thinking.

The term procedural literacy has been used to define this potential. Michael Mateas, an associate professor in the computer science program at the University of California, Santa Cruz, describes procedural literacy as "the ability to read and write processes, to engage procedural representation and aesthetics." One component of procedural literacy is the idea that programming is not strictly a technical task; it's an act of communication and a symbolic way of representing the world. A procedural representation is not static. It's a system of rules that define a space of possible forms or actions. Video game designer Ian Bogost defines this elegantly in his book Persuasive Games: The Expressive Power of Videogames:

To write procedurally, one authors code that enforces rules to generate some kind of representation, rather than authoring the representation itself. Procedural systems generate behaviors based on rule-based models; they are machines capable of producing many outcomes, each conforming to the same overall guidelines.

A video game like Spacewar! is a good example of a procedural representation. Playing the game requires understanding the spatial and kinetic relationships between two opposing spaceships. Each player controls a ship by rotating left and right and by thrusting the rocket with the goal of shooting down the other ship. To write the game, a procedurally literate individual had to break the behaviors into modules with enough detail so that they could be programmed. The primary complexity involved in creating the game is not technical; it's about choreographing all of the components into a coherent and enjoyable experience.

Procedural literacy is a general way of thinking that cuts across all programming languages and even applies to thinking outside the domain of writing source code.

Each programming language is a different kind of material to work and think with. Just as a carpenter knows the unique properties of various woods, including oak, balsa, and pine, a person knowledgeable about software knows the unique aspects of different programming languages. A carpenter building a table will select the wood based on factors such as cost, durability, and aesthetics. A programmer selects a programming language based on the estimated budget, operating system, and aesthetics.

The syntax (or grammar) of each programming language structures what is possible within that language. Different programming languages encourage programmers to think about their work through the affordances (or action possibilities) and constraints of that language.

The way that a programming language encourages a certain mode of thinking can be demonstrated by comparing two very different languages: BASIC and LOGO. In each of these programming environments, drawing a triangle requires a different approach and understanding of space. BASIC relies on an established coordinate system and requires the knowledge of coordinates; lines are drawn by connecting one coordinate to another. LOGO requires a more intuitive approach to understanding space and how it is represented graphically.
The BASIC language was designed in 1964 to teach non-technical university students how to program. Variations of it were used to teach many children and gradually how to program in the early era of personal computing.

Visual programming languages (also called graphical programming languages) provide an alternative way of thinking with code. Writing a program with a visual programming language is similar to making a diagram instead of writing a text. Three of the most popular visual programming languages within the arts—Max, Pure Data, and vvvv—were influenced by the way sounds are constructed using patch cables attached to analog synthesizers. Visual cables, represented as lines on screen, are used to connect programming modules together to define the software. Visual programming languages make it easy to generate and filter images and sounds, but they are often too cumbersome for writing long, complicated programs. For example, the Max program is written in the text programming language C++, not a visual programming language.
Within four adjacent squares, each 4' by 4', four draftsmen will be employed at $4.00/hour for four hours a day and for four days to draw straight lines 4 inches long using four different colored pencils; 9H black, red, yellow and blue. Each draftsman will use the same color throughout the four day period, working on a different square each day.
to create early computer films in collaboration with artists Stan VanDerBeek and Lillian F. Schwartz. The computer-generated film, Permutations, was created in 1966 by John Whitney Sr. using GRAF, a programming library developed by Dr. Jack Citron of IBM. Both BEFLIX and GRAF were built on top of the language Fortran. From these and other early explorations, the development of programming languages written expressly for the arts has continued to gain momentum, building toward the current frenzy of activity. In the 1980s, the proliferation of the personal computer allowed programming to reach a wider audience, which in turn led to the development of HyperTalk, a programming language for Apple's unique HyperCard application (an early hypermedia system). The related Lingo language was developed for the first release of Adobe Director in 1988 (formerly Macromedia Director, and before that MacroMind Director). Lingo was the first programming language used by many designers and artists in the era leading up to the development of the World Wide Web in the early 1990s. The early days of the web fostered intense graphic programming exploration, primarily channeled through the ActionScript language. The rise of programming literacy within the arts and architecture communities has led to the current proliferation of programming options; many are featured within this book.

The influence of code is not limited to the screen and projected image. It is also felt in physical space. Code is used to control elements of products, architecture, and installations. It is used to create files that are output as prints and made physical through computer-controlled machines that cut and assemble materials, including wood, metal, and plastic. Code is rapidly moving outside the boundaries of the screen and is starting to control more aspects of the physical world. There are examples of this in the Producing Form section of "Form and Computers" (p. 37), as well as throughout this book.

Hypertalk

on mouseUp

put "100,100" into pos

repeat with x = 1 to the number of card buttons

set the location of card button x to pos

add 15 to item 1 of pos

end repeat

end mouseUp
WHY CODE?

The use of software in the arts can be separated into two areas: production and conception. In the first category, the computer is used to produce a preconceived form; in the second, the computer participates in the development of the form. Within the context of this book, we're primarily interested in the latter. (It is important to note that this distinction does not imply a value judgment but does impact the types of forms that are created.)

Using the computer to reduce the amount of time needed to create a complex, repetitive composition was often the motivation for the early adoption of software and its integration into the creative process. This was especially important in the field of animation, where subtle changes had to be repeated thousands of times to create the illusion of motion; however, this illusory technical benefit has had a profound effect. If initial production takes one-tenth the time that it would take to execute the work by hand, then the artist can create ten versions in the same amount of time. This way, many versions can be created and the best chosen. Efficiency facilitates the creative process by enabling more time for exploration as less time is needed for the final production. Eventually the computer came to be understood as more than just a production tool. People started to see it as, in the words of one computer graphics pioneer A. Michael Noll, "an intellectual and active creature, perhaps when fully exploited, could be used to produce entirely new art forms and possibly new aesthetic experiences."

Often, to realize a new or unique vision requires that artists and designers exceed the limitations of existing tools. Proprietary software products are general tools designed for the production of specific types of forms. If you are already using software for your work, why constrain yourself to the expectations of a software company or another programmer? To go beyond these limitations, it is necessary to customize existing applications through programming or to write your own software.