IRA GREENBERG

CREATE CODE ART, VISUALIZATIONS, AND INTERACTIVE APPLICATIONS WITH THIS POWERFUL YET SIMPLE COMPUTER LANGUAGE AND PROGRAMMING ENVIRONMENT.

Foreword by Keith Peters

LEARN HOW TO CODE 2D AND 3D ANIMATION, PIXEL-LEVEL IMAGING, MOTION EFFECTS, AND PHYSICS SIMULATIONS.

Processing

Creative Coding and Computational Art

friendsof ED™
DESIGNER TO DESIGNER™
an Apress® company

TAKE A CREATIVE AND FUN APPROACH TO LEARNING CREATIVE COMPUTER PROGRAMMING.

Processing

Creative Coding and Computational Art

Ira Greenberg

friendsof ED™

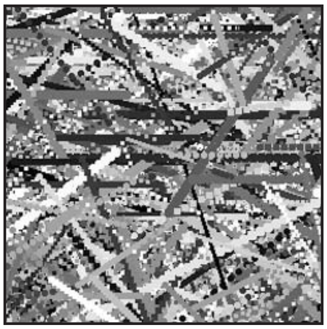
DESIGNER TO DESIGNER™

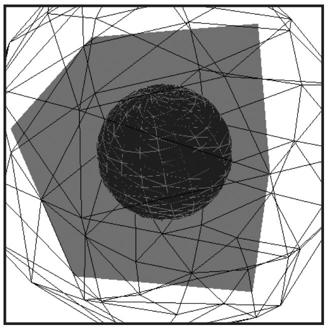
an Apress® company

PART ONE **THEORY OF PROCESSING AND COMPUTATIONAL ART**

The creative process doesn't exist in a vacuum—it's a highly integrated activity reflecting history, aesthetic theory, and often the technological breakthroughs of the day. This was certainly the case during the Renaissance, when artists, engineers, scientists, and thinkers all came together to create truly remarkable works of art and engineering. Over the last few decades, we've been experiencing our own Renaissance with the proliferation of digital technology—creating radically new ways for people to work, play, communicate, and be creative. Processing was born directly out of these exciting developments.

In Part 1 of the book, I'll look at some of the history and theory behind Processing, computational art, and computer graphics technology. I'll also discuss some of the challenges and exciting possibilities inherent in working with such a new and evolving medium. You'll learn about a number of pioneers in the computational art field, and also take a tour of the Processing environment. Finally, you'll explore some of the fundamental concepts and structures involved in graphics programming.





1 CODE ART

Ask the average person what they think computer art is, and they'll likely mention the types of imaging effects we associate with Photoshop or maybe a blockbuster 3D animated film like Shrek. I still remember the first time I cloned an eyeball with Photoshop; it was totally thrilling. I also remember getting a copy of Strata Studio Pro and creating my first perfect 3D metal sphere (you know, the one with the highly reflective ship plate texture map). However, once we get a little more experience under our belts and have tried every freakin' filter we can get our hands on, the "gee whiz" factor subsides, and we are stuck with the same problem that all artists and designers experience—the empty white page or canvas. Of course, each year I have many students who believe that they have found the perfect combination of filters that will produce remarkable and original works of art—without the need to exert too much effort (or leave their game consoles for very long). In the end, though, the stylistic footprints left behind by these filters is unavoidable. That is not to imply that the filters are the problem; I couldn't do my job without the genius of Photoshop. It is the approach to using them that is the problem, or the belief that all that power will make the process of creating art any easier.

The fact that these filters are so captivating and that my students are drawn to them is significant, and I don't think it is only because they are a quick fix. I think what we see in those amazing filters, as we do when we stare at a cool seashell or a dramatic sunset, is algorithmic beauty. Now, I can hear some of you beginning to moan when you read a phrase like that, but I promise you I won't lapse into too many pages of new age touchy-feeliness.

For me, algorithmic beauty is just an apparent mathematical pattern that we find engaging. It can be a passage in music, a strange coincidence of events, a crystal, or, yes, even a Photoshop filter. We see a pattern of forces at work, creating something uncommon, yet familiar at the same time. When we can predict the patterning too readily, and in a sense crack the code, the effect tends to wear off, and we have almost the opposite reaction to awe; we feel cheated or at least unimpressed—thus my disdain at spotting too many filters. The really cool and mysterious thing is that in some sense, in detecting beauty, our brains seem to be doing a strange type of intuitive math, where we are measuring intervals, comparing ratios, and computing some kind of algorithmic aesthetic solution.

I experienced an example of this when I was in grad school studying painting. A bunch of us were eating breakfast at the quintessential greasy spoon in center city, Philadelphia. Somebody had found this pretty cheesy visual literacy quiz. The idea was to look at a series of diagrams and pictures and say which one worked the best. There were about ten separate questions on the page, and everyone at the table took the test independently, between forkfuls of grease heaven. Surprisingly, we all got the same exact answers. Now it could have been due to the common grease we were ingesting, or sheer coincidence, but I like to think we all got the same answers because we all shared a common visual algorithmic literacy. The test wasn't about what we liked, but which images worked best. Had the test instead been about which images were better works of art, not only would none of us have agreed, but we most likely would have been thrown out of the restaurant for disorderly conduct. The decision whether an image, object, or design "works" from an aesthetic standpoint may be as close to a quantifiable measure as you can (or probably should) bring to a piece of art.

The problem with a Photoshop filter is its limited range of expressive possibility. Each filter has a rational consistency, which is absolutely necessary for generating predictable results and overall reliability. But the rationality built into the tool makes it hard to find new solutions. Coding allows you much greater freedom to build in levels of control, and also levels of randomization and even irrational processes that don't always work but can lead to exciting, unexpected results.

Using Processing, it doesn't take too long playing with some code to generate truly evocative images and visual experiences, gain some insight into how a program like Photoshop works, and go way beyond any preprogrammed filters.

Aesthetics + Computation

Aesthetics + Computation is the name of the now famous research group at MIT's Media Lab, where Casey Reas and Ben Fry, the creators of the Processing language, worked as grad students under John Maeda. However, interest in aesthetics and computation goes back way before 1996, when Maeda began teaching at MIT.

If we're a bit flexible in how we translate the term "computation," we can go really way back—as far back, in fact, to when people first learned to count and draw (ironically, skills that some would argue are in sharp decline today because of computers). The term **computation**, according to `www.dictionary.com`, means "the procedure of calculating; determining something by mathematical or logical methods." If you take the term **mathematical** out of the definition (already I can feel some readers' comfort levels increasing), the definition could pretty much account for most human decisions. For example, I have a faculty meeting coming up. Therefore, I need to remember to bring my sketchpad to the meeting to be able to draw annoying caricatures of my colleagues so that I don't fall asleep. See, a perfect example of aesthetics + computation.

Serious interest in aesthetics + computation as an integrated activity is evident in all cultures and is manifest in many of the objects, structures, and technologies of the times in which they were created. Regardless of whether the technology is an engraving stick, a loom, a plow, or a supercomputer, the impulse to work and play in an integrated left-/right-brain way is universally evident, and the technical innovations of the day most often coincide with parallel developments in aesthetics. Early astrological and calendar systems, across many cultures, combined observed empirical data with richly expressive, mythological narratives as a way of interpreting and ultimately preserving and disseminating the data. Weavings, textiles, engravings, mandalas, and graphs from cultures around the world employ complex algorithmic patterns based upon mathematical principles, yet most often are not developed by mathematicians (see Figures 1-1 through 1-4). Rather, these developments seem to reflect a universal human impulse to integrate right-brain and left-brain activities, combining qualitative notions of aesthetic beauty with analytical systems for structuring visual data.

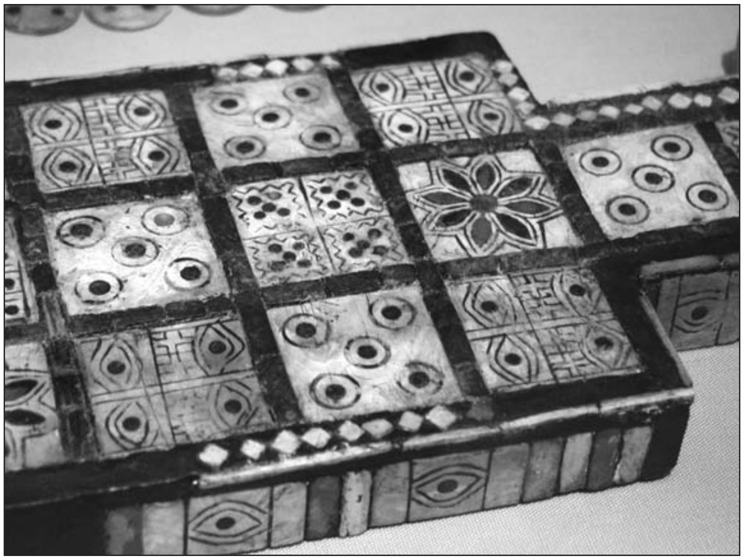


Figure 1-1. Ur game board, found in grave, 2600 BC

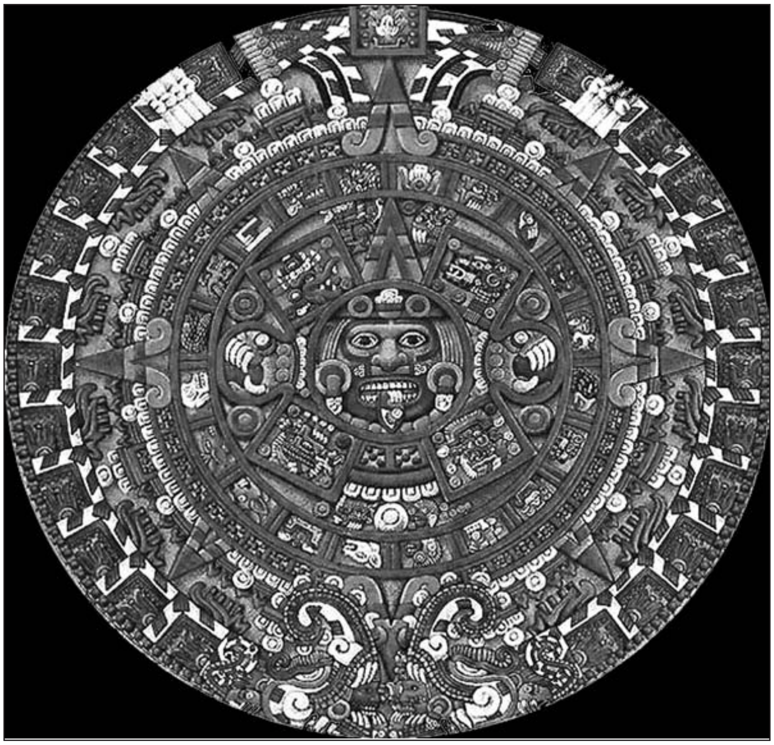


Figure 1-2. Replica of the Aztec stone of the sun, from the original found in the city formerly known as Tenochtitlán

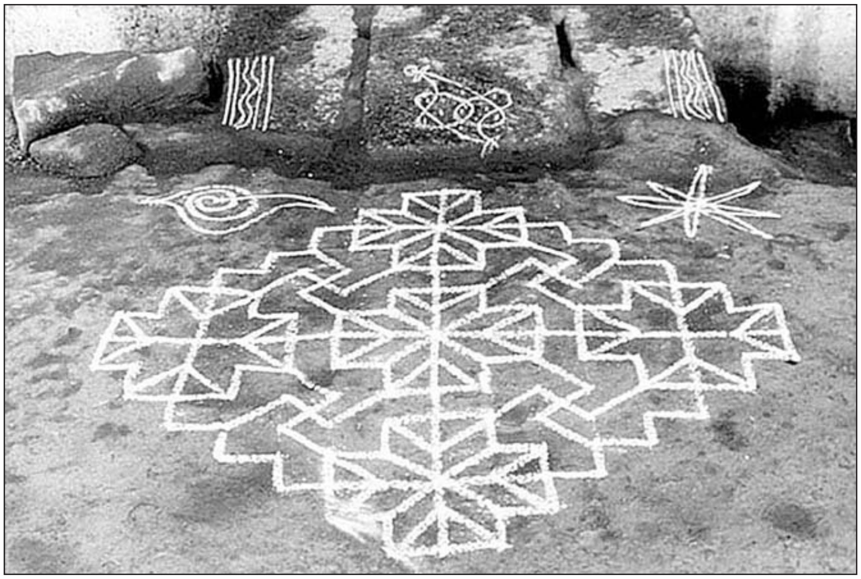


Figure 1-3. Example of ancient art of Kolam, southern India, 1970s

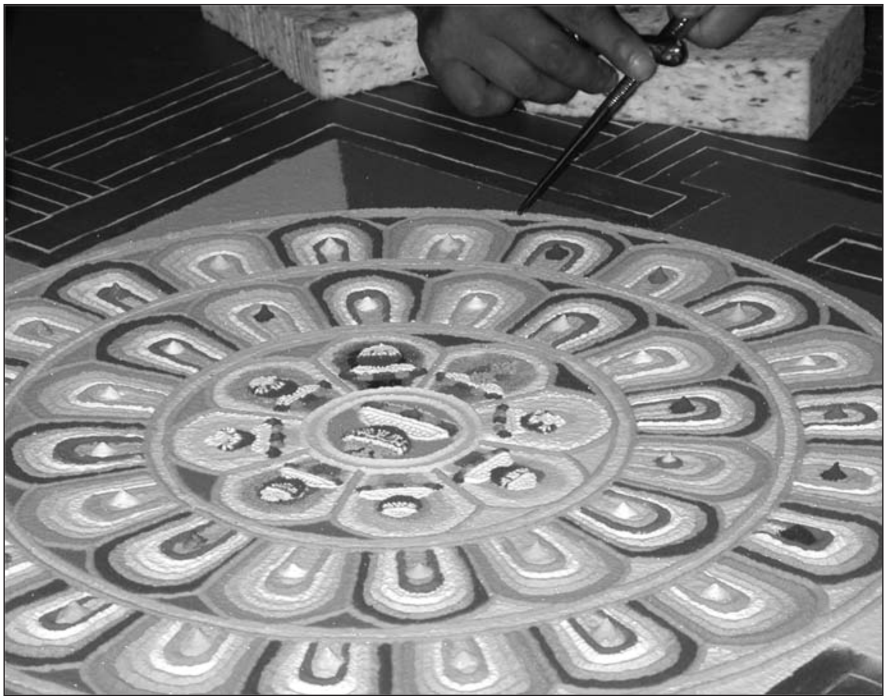


Figure 1-4. Tibetan monks making a temporary “sand mandala” in the Kitzbühel City Hall, Austria

Renaissance biographer Giorgio Vasari tells us that the Florentine architect Filippo Brunelleschi (1337–1446) took up painting to apply his newly developed theory of perspective, based upon Greek geometry. Additionally, other major Renaissance painters including Piero della Francesca, Albrecht Dürer, and Leonardo da Vinci not only experimented and applied principles of geometry in their work, but published treatises on mathematics. Dürer even developed several drawing machines that could be used to teach perspective. Especially during the European Renaissance, the division separating art and science blurred to the point that many of its greatest practitioners made nearly equal contributions to each.

Interestingly (or maybe sadly), it may be one of the unique aspects of our contemporary culture that the need for professional hyperspecialization has made left-/right-brain integration more difficult and rare. The amount of training and specialized information we each feel we need to develop "competency," and ultimately, success, in our fields forces a vocational myopia, which then reinforces many of our erroneous assumptions: "I'm not good at math," "I can't draw," and so on. The situation isn't helped when we create totally inscrutable private vocabularies around our fields (especially the most technical ones), making them practically inaccessible to outsiders. This is certainly the case in computing, and especially programming. I think one of John Maeda's hopes in creating the Aesthetics + Computation group at MIT, and the reason he created the programming language Design by Numbers (DBN; the precursor to Processing), was to reveal the aesthetic beauty and power of computation (and math) to artists and designers. By using a really simplified programming language and approaching programming instruction in a straightforward and creatively centered way, he hoped artists would feel free to begin to express themselves through this fascinating medium and challenge some of their unproductive assumptions (such as "artists don't do math").

I think Maeda believes, as I do, that the core expressive element of computing is not at the software application level, using sliders, spinners, dialog boxes, and filters; but at the lower level of computation, most accessible through direct programming. I like to think of programming as the material level of computing, the place to get, at least metaphorically, your hands dirty. Casey Reas and Ben Fry, Maeda's students, understood this vision and built Processing as an extension of (or at least homage to) DBN, adding the features of a powerful programming language while retaining DBN's ease of use to facilitate its application as an expressive creative medium.

Computer art history

The history of computer art goes back further than most people realize. However, before diving into computer art, it's helpful to create some historical context by examining just a little computing history. There are many places to begin the story of computing; we could go back 4,000 years to the ancient Egyptians and Babylonians and the origins of mathematics. I suspect of interest to some readers would be the Egyptian Ahmes Papyrus, which contains some of the first known written equations for, among other things, the formula for beer—one approach to keeping students interested in math. We could then methodically move through the entire history of mathematics, looking at the Babylonian Salamis tablet, or counting board, circa 300 BC; the Roman hand abacus (see Figure 1-5) a little later; and then the *suan pan*, or Chinese abacus, getting us to around 1200 AD. We could then jump 400 years to 1614 and John Napier, the Scottish supergenius who developed logarithms (which allow multiplication and division to be handled like simple addition and subtraction), as well as a system of using metal plates for performing multiplication and division. For this, Napier gets credited with developing the earliest known attempt at a mechanical means of calculation. Building upon Napier's work, Edmund Gunter gets credited in 1620 with developing the second most important geek tool of all time, the slide rule (see Figure 1-6). Leaping ahead 200 years or so to the 1830s, we come across a very significant moment in computing history and some extremely interesting characters: Charles Babbage, Ada Lovelace, and the Analytical Engine.

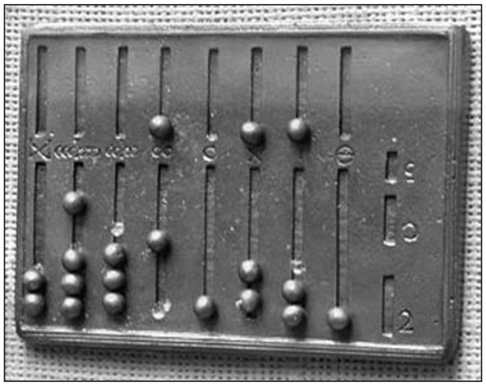


Figure 1-5. Reconstruction of a Roman abacus

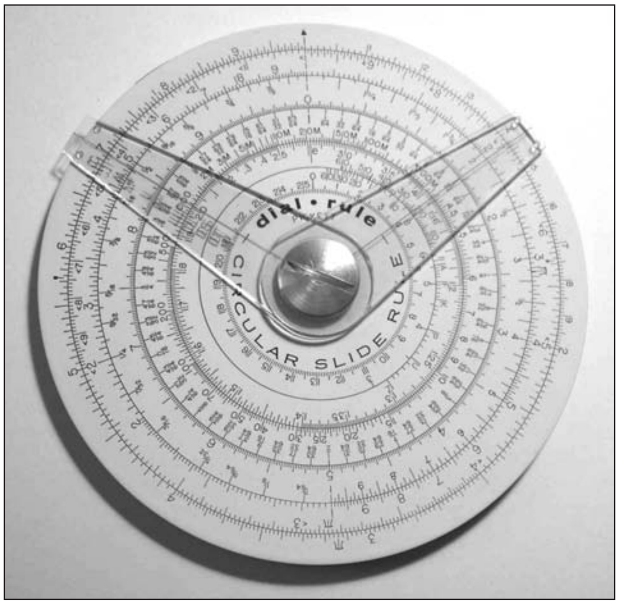


Figure 1-6. Pickett circular slide rule with two cursors

Babbage was a professor of mathematics at Cambridge, who interestingly held the same Lucasian chair as Newton, which is now held by present day supergenius Stephen Hawking—not a bad lineage. Babbage was an obsessive classifier, regularly measuring and collecting all the relevant (by his standards) data he came across, such as the heartbeats of farm animals, the rate at which a man could saw wood, and even the frequency and cause of broken windows around town. He supposedly went so far as to bake himself in an oven for a couple of minutes to figure out the amount of heat that the human body could withstand. Babbage was also a vehement hater of street musicians (not sure why).

Babbage believed, perhaps naively but brilliantly, that his obsessive data analysis would lead to a sort of empirical truth about, well, everything. Babbage designed a complex calculating machine called the Analytical Engine (see Figure 1-7). It was designed as a steam run, automatic, programmable, universal machine capable of solving nearly any algebraic function. Amazingly, Babbage's design included almost all the logical features of a modern-day computer. Unfortunately, as with many visionaries, his wild ambition for the engine was ultimately beyond the reality of its actualization, and it was never completed.

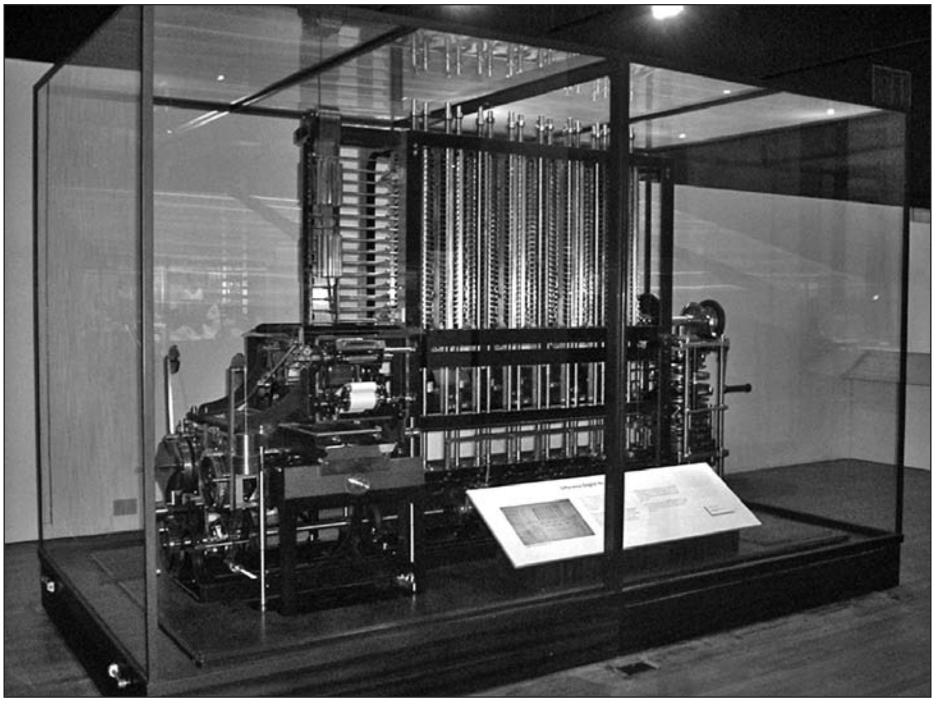


Figure 1-7. The London Science Museum's replica difference engine, built from Babbage's design

Ada Lovelace, the daughter of poet Lord Byron, who never actually met her famous father, was raised to be a mathematician and scientist. Her controlling mother was determined to rid her of any inherited poetical tendencies from her estranged famous husband (she divorced Byron shortly after her daughter's birth). This plan of course failed, and Lovelace's fertile imagination remained intact, along with her phenomenal analytical talents, making her a great exemplar of the power of an integrated mind. The story goes that Lovelace eventually heard about Babbage's analytical engine at a dinner party in 1834, and was quite taken by the idea. She saw many potential applications for the powerful engine, when most everyone else did not, including the wild possibility of having the machine compose music and even generate graphics. Babbage and Lovelace developed a correspondence, and she eventually wrote a document to Babbage suggesting a plan to have the engine calculate Bernoulli numbers. There is no simple way to explain Bernoulli numbers, so I'll just say that they are quite significant in mathematics—here's a link if you're interested: `http://en.wikipedia.org/wiki/Bernoulli_numbers`. The plan, or algorithm, that Lovelace wrote to Babbage is now regarded as the first computer program—100 years before the invention of the computer! (How cool is that?)

In spite of Babbage and Lovelace's prescient work, computing history went nowhere for nearly 100 years. It wasn't until the mid-1930s that German engineer Konrad Zuse developed the Z1 (see Figure 1-8) and got credited with developing the first computer.

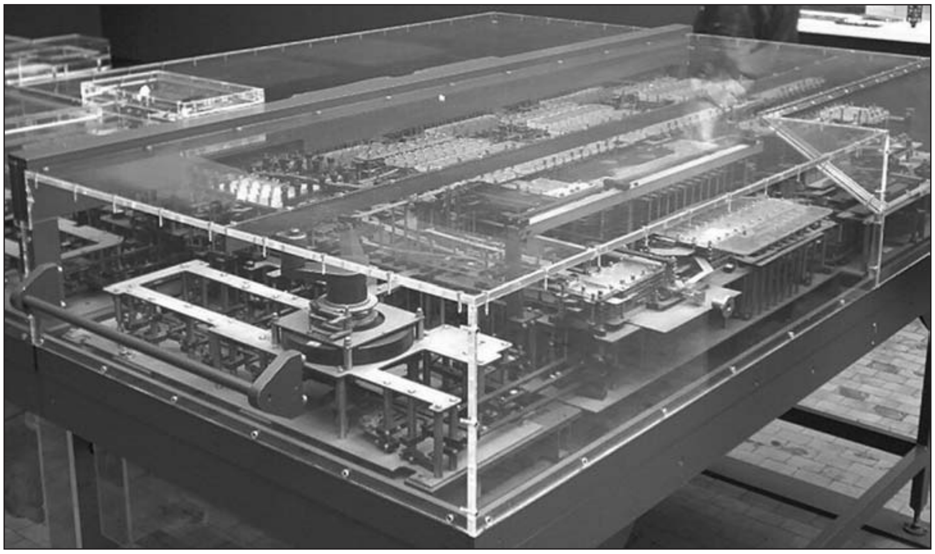


Figure 1-8. The Z1 was a mechanical computer created by Konrad Zuse in 1937. A reproduction of this machine (pictured) is currently housed in the Deutsches Technikmuseum Berlin.

Between the '30s and mid-'40s—thanks in part to the very unfortunate need for things like firing tables and atomic bomb calculations—numerous modifications were made to the Z1, and other significant computer initiatives were begun, including John Atanasoff's ABC computer, the Mark 1 at Harvard, and the ENIAC at Penn (see Figure 1-9). These initial machines were very difficult to program, and the Mark 1 and the ENIAC were absolutely enormous. The Mark 1 was 51 inches long, 8 feet high, and contained 17,486 vacuum tubes. The ENIAC weighed in at 30 tons and used so much power that it was said to cause the lights of Philadelphia to flicker.

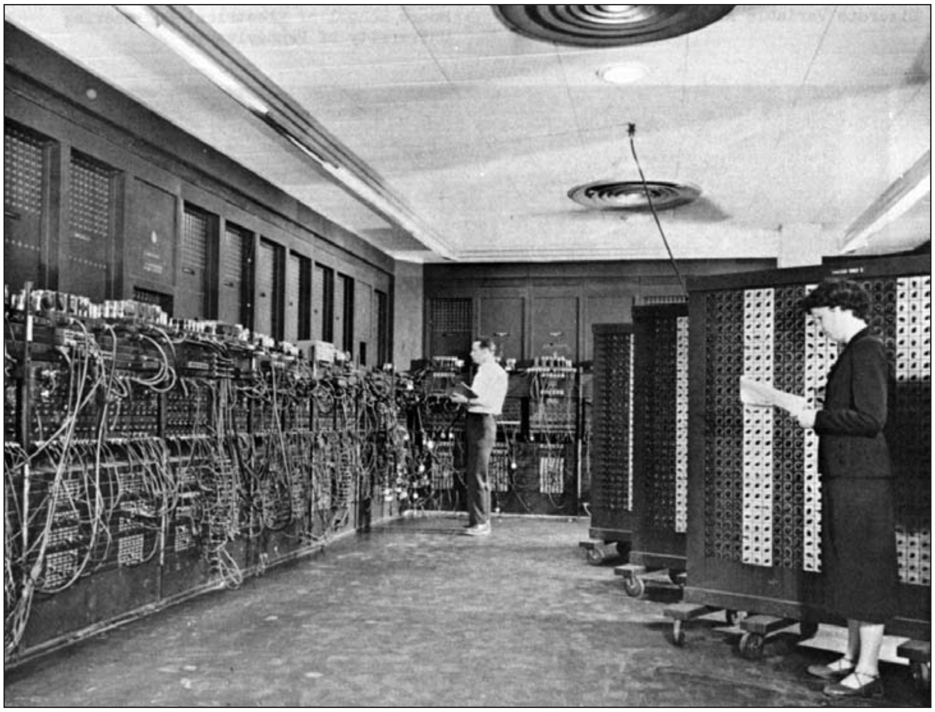


Figure 1-9. The ENIAC

Finally in 1951, the first general-purpose computer, the UNIVAC, was developed, with a relatively easy-to-use programming language, including some programming standards. The UNIVAC was used by the Census Bureau in 1951, and famously predicted Eisenhower's victory in 1952. The programming language the UNIVAC used was developed by the amazing Grace Murray Hopper (see Figure 1-10), the first woman to be awarded the rank of rear admiral.

Figure 1-10.
Commodore Grace M. Hopper

As an aside, it's worth observing that computer programming, which has gotten the stigma as a male-dominated pursuit, had two brilliant women as its founding pioneers. Contemporary artist Mary Flanagan addresses this issue in a very interesting way with her RAPUNSEL project, which I'll elaborate upon a little later.

Admiral Hopper is, among many other distinctions, also attributed with coining the term "software bug," when she actually found a dead moth in one of the Mark 2 computers, preventing it from working. She had programmed the Mark 1 at Harvard in the very early '40s, and also developed the first compiler. Compilers are used to convert the mnemonic code that a programmer writes into machine code. Machine code is a binary system composed of zeros and ones that is, in a sense, the native language the computer speaks. Mnemonic code is a higher-level system of symbols and natural language-like constructs, which we use to program the computer with. For example, as it is near lunchtime I am beginning to think "I'm getting hungry." Converting these words into binary machine code, I get the following:

```
01001001001001110110110100100000011001110110010101110100011101000110100101
1011100110011100100000011010000111010101101110011001110111001001111001
```

I don't know about you, but I find it a bit (no pun intended) easier to understand the symbolic code "I'm getting hungry," rather than the 144 zeros or ones in the binary code.

As computers have evolved, and the problems we throw at them (such as art making) have steadily increased in complexity, the mnemonic codes, or programming languages, that we've developed have steadily gotten higher-level. **Higher-level** means moving the symbolic abstraction of the language further away from the underlying binary machine code and nearer to our natural language—and ultimately (hopefully) nearer to how people think. This idea of creating higher-level symbolic abstractions away from the zeros and ones, allowing the coder to think more naturally and thus gain programming literacy more easily, has led directly to the development of languages like Processing.

It didn't take very long for some pioneering artists to see the expressive potential and fascination in computers. As early as 1956, just five years after the UNIVAC was developed, artists began experimenting with computing as an expressive medium. Of course, computers were inaccessible to most artists in the late '50s. According to Jasia Reichardt in her book *The Computer in Art*, there were no computers in any art departments or art colleges by the end of the 1960s. Although at a few places, notably Ohio State, artists had access to the university computer, as well as some technical staff. Jasia then goes on to state in her 1971 book, "one can assume that there are probably no more than about 1,000 people in the world working with computer graphics for purposes other than the practical ones."

Dividing computer graphics pioneers along practical vs. nonpractical applications in 1971 struck me as amusing. I'd argue that anyone dabbling in this black science back then was pioneering very experimental applied applications at best. I think this same erroneous distinction between applied vs. fine art continues to linger within the digital arts today, as many "hybrid" types (myself included) have struggled to define their places in universities, galleries, and industry. Fortunately, there were times in history when scientists, artists, philosophers, engineers, and so forth were all seen as integrated creative practitioners—not divided solely by their perceived utilitarian value in the marketplace. And there seems to be a growing recognition in industry, thanks in part to companies like Apple, that creativity and design are good for business. Universities are also recognizing and responding to students' interests in blurring old delimiting disciplinary and curriculum boundaries, and digital media and the electronic arts are at the center of this movement. The recent dramatic and swelling popularity of Processing is a testament to this contemporary renaissance.

Code artists

In concluding this chapter, I've included some brief biographies of 12 code/electronic artists, many of whom are still practicing, exhibiting, and teaching. Some have worked as fine and applied artists throughout their careers, many with traditional materials, which in most cases has informed the digital work. All of these artists apply computation and technology to their creative practice in a formally rigorous way, utilizing code and algorithmic approaches. Their work is not solely a commentary on technology, nor illustrated or simply facilitated by it, but is rather *of* it. I also want to preface this list by stating that I have left out at least as many worthy individuals as I have included. I have tried to include those individuals who represent a wide range of approaches and whose work most directly informs the varied directions artists are pursuing with Processing. Following the biographies is a list of additional noteworthy artists with links to their work.

Ben Laposky, 1914–2000

Both mathematician and artist, Ben Laposky was a pioneer among computer art pioneers. He created the first electronic abstract images in 1950 by using an electronic analog device called a cathode ray tube oscilloscope. To put Laposky's pioneering role in some context, it wasn't until 1960 (ten years later) that the term "computer graphics" was first coined by William Fetter, a graphic designer at Boeing. Laposky created images of beautiful mathematical curves he named "oscillons." The curves were based on the basic waveforms utilized in analog computers. Analog computers, now mostly unknown, were utilized as early as the 1920s and were capable of very fast, complex calculations. They used continuous variations in voltage, allowing real-time calculations, as opposed to distinct finite signals, which are employed in digital computers. To learn more about analog computers, check out `http://en.wikipedia.org/wiki/Analog_computer`. By the 1940s, analog computers began to be replaced by digital computers, which were more affordable to produce and more universal as general-purpose tools. I'll discuss more about the type of waveforms Laposky generated later in the book, including how to generate some of them. To capture the curves, Laposky photographed the images directly off the display with high-speed film. What I find striking about his images is their elegant simplicity and organic quality, in contrast to the technical means from which they arose. Beyond Laposky's pioneering technical virtuosity is his steadfast focus on the aesthetic quality of his work. This combined focus on the technical and aesthetic is an important enduring legacy that unites all the artists mentioned in this chapter. To learn more about Laposky and view some images of his work, check out `www.dam.org/laposky/index.htm` and `www.atariarchives.org/artist/sec6.php`.

John Whitney Sr., 1918–1995

John Whitney's work involved photography, film, and eventually also music. In the 1940s, he began to study images in motion, along with his brother James. This interest would remain with him the rest of his long career and eventually involve the integration of motion and sound. Whitney combined a very successful career in commercial film and television with his more personal and experimental investigations. He even produced the title sequence for Hitchcock's *Vertigo*. By the 1960s, Whitney had formed a company, Motion Graphics, to produce commercials, at which he pioneered early computer-based animation and typography with a mechanical analog computer that he invented. Whitney gained wide recognition for his work with the analog computer, as well as for a sample reel entitled "Catalog," which showcased his innovative motion work. Between 1966 and 1969, Whitney got to work with powerful digital computers as the first artist in residency at IBM. From this period through the '70s, he continued to push the complexity of his work, creating numerous experimental films and documentaries while also refining his focus to achieve what he called "harmonic progression." Throughout his entire career, Whitney remained an innovator, eventually developing a computerized instrument that allowed him to compose visuals and music in real time. Additional information on Whitney, including images, can be viewed at `www.siggraph.org/artdesign/profile/whitney/nowhitney.html` and `http://en.wikipedia.org/wiki/John_Whitney_(animator)`.

Herbert W. Franke, b.1927

Franke is a brilliant generalist, with formal training in physics, mathematics, chemistry, psychology, and philosophy. He is a widely published author—his works include seven science fiction titles, essays, articles, and scripts on subjects as wide ranging as physics, theoretical chemistry, scientific photography, visual perceptions, art and technology, futurology, and speleology (the scientific study of caves). His early work paralleled Loposky's, utilizing oscilloscopes to generate images based on waveforms. Since 1962, Franke has remained active in "experimental aesthetics," lecturing and creating new work in the field. He cofounded the Ars Electronica Festival in 1979, arguably the most prestigious international electronic arts festival in the world, held annually in Linz, Austria. In 2005, for their work on Processing, Casey and Ben won a Golden Nica—the festival's top prize. Franke has remained on the bleeding edge of emerging technology throughout his career, and beginning in the mid '90s began developing algorithmic animations using the Mathematica software system. In 2002, he released a book on the subject, entitled *Animation with Mathematica*. To learn more about Franke (in English), see `http://en.wikipedia.org/wiki/Herbert_W._Franke` and `www.dam.org/franke/index.htm`; to learn more about him in German, see `www.zi.biologie.uni-muenchen.de/~franke/`.

Lillian Schwartz, b. 1927

Lillian Schwartz has been a central and pioneering figure in the history of computer art, making broad and significant contributions in computer-generated art and computer-aided art analysis. Her work and research have led to advances in graphics, film, video, animation, special effects, virtual reality, and multimedia. She gained prominence in the late '60s when one of her kinetic sculptures, *Proxima Centauri*, was included in the groundbreaking 1968 Machine Exhibition at the Museum of Modern Art (MoMA), New York. She is also the first artist to have a piece of computer art acquired by the MoMA. In addition to the MoMA, her work has been exhibited at and is owned by the Metropolitan Museum of Art, the Whitney Museum of American Art, the Moderna Museet (Stockholm), Centre Beaubourg (Paris), the Stedelijk Museum of Art (Amsterdam), and the Grand Palais Museum (Paris). Her films have been shown and won awards at the Venice Biennale, Zagreb, Cannes, and the National Academy of Television Arts and Sciences; and she has been nominated for and received Emmy awards. Like many of the early pioneers, Schwartz has made substantial contributions within industry, academia, and the art world. She was a consultant at AT&T Bell Laboratories, IBM's Thomas J. Watson Research Laboratory, and Lucent Technologies Bell Labs Innovations. Her research has led to advances in the understanding of visual and color perception, sound, perspective, historic preservation, and applications of the computer in the philosophy of art—using computation to investigate the creative process. She has taught in both fine art and computer science departments throughout the United States, as well as internationally. Schwartz has received numerous awards, including Computerworld Smithsonian awards in three categories. She has also been the subject of articles, books, and television news and documentary programs, and is the coauthor of *The Computer Artist's Handbook* (together with Laurens R. Schwartz). To learn more about Schwartz and see images of her work, go to her personal website, `www.lillian.com/`.

Harold Cohen, b. 1928

Cohen, probably more than anyone else, made a wide and dramatic migration from the traditional arts to computer art. He was a celebrated painter in the '60s, including representing Great Britain in the Venice Biennial in 1966. In 1968, he moved to Southern California as a visiting faculty at the University of California, San Diego. He soon became interested in artificial intelligence, and in 1971 was invited to spend two years at the Artificial Intelligence Laboratory of Stanford University as a guest scholar. Since then, his work has focused on the integration of artificial intelligence with the visual artist's creative process. Over three decades, Cohen developed an expert drawing system, a rule-based software program capable of simulating aspects of human creativity and intelligence. He named the system he developed AARON. AARON is capable of creating original works of art on its own. Although all the work AARON produces is stylistically similar, the range of subject matter, composition, and color is not. What is so surprising about AARON, and I believe what makes it groundbreaking, is that the work it produces is representational. Most algorithmic systems generate abstract or color field–oriented images. AARON creates, in a sense, freehand drawn images of people, still lifes, and landscapes. Cohen's (and AARON's) work has been shown around the world, including in the Los Angeles County Museum, Documenta 6, the San Francisco MoMA, the Stedelijk Museum in Amsterdam, the Brooklyn Museum, the Tate Gallery in London, and the IBM Gallery in New York. Cohen has a permanent exhibit in the Computer Museum in Boston, and represented the United States in the Japan World Fair in 1985. In 2000, artificial intelligence entrepreneur and inventor Ray Kurzweil licensed the code to AARON and created a screen saver version of the software, which he hosts at his Kurzweil CyberArt Technologies website (`www.kurzweilcyberart.com/`). Cohen continues to push his work and process, developing even more naturalistic and organic representational images based on machine intelligence algorithms. To learn more about Cohen and to see images of his (and AARON's) work, check out `www.kurzweilcyberart.com/aaron/hi_cohenbio.html` and `www.viewingspace.com/genetics_culture/pages_genetics_culture/gc_w05/cohen_h.htm`. Here's an interesting blog entry he wrote about "creative" programming: `http://grandtextauto.gatech.edu/2003/06/17/harold-cohen-on-artist-programmers/`.

Roman Verostko, b. 1929

Roman Verotsko has had a long, distinguished, multifaceted career. For 16 years, beginning in 1952, he lived as a Benedictine monk. Toward the end of this monastic period, he was sent to New York to develop himself as an artist, with the goal of bringing this experience and training back to the abbey. While in New York, he received an MFA from Pratt and then took graduate courses in Art History at both Columbia and New York University. Eventually struggling with his beliefs in revelation and the dogmas of the church, he left the monastic life in 1968 and took a faculty position at Minneapolis College of Art and Design, where he remained throughout his academic career. Verostko spent the first 30 years of his artistic life painting. This early work dealt with the search for an internal dynamic force in a work of art. He wrote the following about his process:

> *Through this process, in a kind of spiritual quest—one has to empty the self of "thinking," be entirely present to the moment, and strive to be one with one's world. To be one with the brush, the crayon, the panel, the universe—in a free flowing gesture was indeed the goal. Being most "free" was also being most "joined."*

In 1970, he received a Bush Fellowship grant, which provided him the opportunity to work with Gyorgy Kepes at the Center for Advanced Visual Studies at MIT. At the time, Verostko's interest in computer technology related more to how artists could humanize the emerging medium than how it could be applied directly to the creative practice; it would be 14 years (1984) before Verostko began utilizing computer code within his own creative practice. Eventually, code would become his primary creative medium, extending (rather than supplanting) his earlier interests in spiritual practice and painting.

Verostko was influenced by artists such as Mondrian, Kandinsky, and Malevich in his pursuit to uncover the underlying unity of nature, based on an integration of opposing forces within the image. He developed a software/hardware system he called "Hodos." Hodos generates both screen-based and physical images created with brushes mounted on a pen plotter's drawing arm. Verostko's algorithms control the range of expression of the system, allowing for his personal style and interest in opposing forces to emerge. In 1995, Verostko and a group of artists including Jean-Pierre Hébert and Ken Musgrave formed an informal group they named "the Algorists," for those artists who use their own algorithms for creating art objects. Verostko has received numerous awards and recognition for his work including the Golden Plotter (first prize, 1994), Gladbeck (Germany), and Prix Ars Electronica (honorary mention, 1993). Verostko maintains an excellent site about his work and life at `www.verostko.com/`.

George Legrady, b. 1950

Chronologically, Legrady represents a generation of digital artists born between the earliest pioneers—Laposky, Whitney, Franke, and Schwartz, who depended upon (and often were) scientists and technical experts—and the current generation of digital artists in their 30s and 40s, who grew up with the ease and convenience of personal home computing. Paralleling this is Legrady's own personal story. He is a two-time immigrant, born in Budapest in 1950, who fled to Montreal in 1956 during the Hungarian uprising, and then eventually settled in California in 1981. This recurring theme of being between or crossing generations, space, cultures, and even time runs throughout his work. Trained originally as a classical musician, Legrady's musical interests eventually crossed over, and in 1969 he worked as a rock and roll keyboard player. A product of the counterculture revolution of the '60s, Legrady created socially conscious documentaries in the 1970s and '80s on the Cree Indians of northern Quebec, communist iconography of central Europe, and hand-painted advertising billboards in China. In 1981, Legrady began working with digital artist pioneer Harold Cohen (whose bio is included in the preceding text), experimenting with computer code as a creative medium. By the mid-'80s, Legrady, then an assistant professor at the University of Southern California, began to receive recognition and awards for his digital work, which at the time consisted of digital prints. In the following decade and through the present, Legrady's work has become more site-specific and computationally intensive, involving interactive and algorithmically-based installations. For example, his well-known piece *Pockets Full of Memories* involves an interactive space with large-scale projections. Here's an excerpt about the piece, taken directly from Legrady's site (`http://www.mat.ucsb.edu/~g.legrady/glWeb/Projects/pfom2/pfom2.html`):

"Pockets Full of Memories" is an interactive installation that consists of a data collection station where the public takes a digital image of an object, adds descriptive keywords, and rates its properties using a touchscreen. The data accumulates through-out the length of the exhibition. The Kohonen self-organizing map algorithm is used to organize the data, moving the images of the objects into an ordered state according to similarities defined by the contributors' semantic descriptions.

George Legrady is Professor of Interactive Media, with joint appointments in the Media Arts and Technology program and the department of Art at UC Santa Barbara. Additional information about the artist and his work can be found at `www.georgelegrady.com/`.

Mark Napier, b. 1961

Napier originally studied engineering before switching to studio art. He graduated with a BFA from Syracuse University and began his art career as a painter. A self-taught programmer, he supported himself as a software engineer, developing database systems and web-based tools for the financial industry. In 1995, he merged these two pursuits, ending his painting career and focusing his creative work exclusively, at the time, on Internet-based art. In recent years, some of his work has expanded into gallery and museum settings. Napier pioneered innovative and boldly conceptual web-based pieces such as *Shredder* (`www.potatoland.org/shredder/shredder.html`), *Digital Landfill* (`www.potatoland.org/`

`landfill/`), and *Feed* (`www.potatoland.org/feed/`). These projects, while technically sophisticated and finely crafted, purposefully revealed the limitations, political implications, and chaos of the Web. *Shredder* is an alternate browser that constructs web pages, not as ordered predictable pages, but in Napier's words, "as a chaotic, irrational, raucous collage." Besides the implicit politics and software engineering in pieces like *Shredder* and another browser he developed called *riot*, one perceives an interest in the visual. These algorithmically-generated pieces are chaotic—at times boisterous, but always aesthetically engaging, informed by a trained painter's eye. It is these opposing forces of the raw and the refined that give Napier's pieces their distinctive energy and appeal. Napier's work is in numerous major collections, including the Guggenheim in New York. He's also had work commissioned by both the Whitney Museum of American Art and the San Francisco Museum of Modern Art. His work has been shown widely, including at the Whitney Biennial (2001), Ars Electronica, the Kitchen, ZKM net_condition, the Walker's AEN show, and many other venues and festivals around the world. Additional information, images, and code can be found on Napier's personal site, `http://potatoland.org/`.

John F. Simon Jr., b. 1963

John Simon combines a highly refined visual aesthetic with industrial grade coding skills. He graduated from Brown in 1985 with degrees in studio art and geology, and then received two more advanced degrees in earth and planetary science from Washington University in St. Louis, and an MFA in computer art from the School of Visual Arts in New York. His dual interests in analytical systems and aesthetics is almost always apparent in his work. For example, in his *"art appliances" software and LCD panels, 1998–2004*, Simon sets up algorithmically-based, complex software systems that examine abstraction vs. realism, color theory, viewer perception, and other traditional painting concerns. The work often has art historical connections and references (e.g., to Bauhaus, Klee, Kandinsky, Mondrian, and Lewitt). Beyond Simon's impressive software engineering skills and art historical knowledge is a subtle, often poetic, aesthetic sensibility; many of his pieces are beautiful objects. The work, both in concept and execution, is precise and economical. Simon extends his work from the Web to handheld devices to wall-mounted integrated hardware/software systems that he builds. He also produces software-driven laser-cut Plexiglas objects. Simon is redefining how artists exist in the marketplace, pioneering the sale of affordable works of art directly from his site, at `http://numeral.com/`.

John Maeda, b. 1966

John Maeda is currently the E. Rudge and Nancy Allen Professor of Media Arts and Sciences at MIT. He is also one of the most well-known designers and digital arts pioneers in the world, awarded in 2001 both the US and Japan's top design career honor: the National Design Award and the Mainichi Design Prize—not bad for someone who recently turned 40. Maeda's work and teaching have had a profound impact on the digital design landscape and, perhaps more than anyone else, Maeda has influenced the present generation of code artists. Between 1996 and 2003, he directed the Aesthetics + Computation Group (ACG) at MIT, which conducted research into the application of computer code and computation as a primary creative medium. A number of Maeda's students have become leading designers/artists and (creative) technologists, including Golan Levin, Jared

Schiffman, and of course Casey Reas and Ben Fry. ACG was an outgrowth of an earlier research group at MIT called the Visual Language Workshop (VLW), created in 1973. VLW was created by Muriel Cooper and Ron MacNeil. Muriel Cooper was a renowned designer, like Maeda, who became interested in applying artificial intelligence to the traditional design process. In 2003, Maeda changed directions, transforming ACG into the Physical Language Workshop (PLW), a design-oriented group that according to the site overview, `http://plw.media.mit.edu/`, "designs tools for creating digital content in a networked environment, and the means by which the content can be leveraged as creative capital within an experimental online micro-economy that we call OpenAtelier." For Maeda, PLW was a return to the core ideas in Muriel Cooper's VLW.

In addition, Maeda codirects SIMPLICITY, a new media lab–wide research initiative aimed at redefining users' relationships with technology in their daily lives. This statement on the SIMPLICITY site says it all: "How do you make something powerful, but simple to operate at the same time? This is the challenge." Maeda's extraordinary range of talents and his critical approach to the development and implementation of technology in our daily lives have contributed to him being included in *Esquire*'s 1999 list of the 21 most important people of the 21st century. Here are some links to learn more about Maeda and view his work: `www.maedastudio.com/index.php`, `www.media.mit.edu/people/bio_maeda.html`, and `http://weblogs.media.mit.edu/SIMPLICITY/`.

Mary Flanagan, b. 1969

Mary Flanagan is an artist, producer, designer, technologist, activist, writer, and theorist, and her work reflects an integration of all these interests. Her multidimensional projects are created primarily for the Net or installation, and thematically involve the influence of technology—net.culture, computer gaming, and mundane technological tools—on our daily lives. Flanagan offers a fresh, alternative voice, inspiring through her work underrepresented populations to cross the digital divide. Prior to her academic appointments, Flanagan spent a number of years working at Human Code, an Austin-based software developer. Although she was a highly regarded and award-winning producer and designer within the gaming industry, she was frustrated by the lack of titles being developed for girls and minorities, so she left the industry for academia to pursue her social activist/artistic vision. Two projects she has since developed directly address this concern. *The Adventures of Josie True* (`www.maryflanagan.com/josie/`) is the first web-based adventure game for girls. The game's characters include Josie, an 11-year-old Chinese-American girl; a female African-American aviator called Bessie Coleman; and Josie's science teacher, Ms. Trombone, who is also an inventor. Flanagan is also involved in the development of a collaborative and highly ambitious project: RAPUNSEL (`www.maryflanagan.com/rapunsel/`). The RAPUNSEL project team is made up of a number of leading computer scientists, artists, and educational theorists who are researching and building a software environment to teach programming concepts to kids. Ultimately, RAPUNSEL will become a multiuser 3D game to teach middle school girls computer programming. Flanagan's work has been exhibited internationally at museums, festivals, and galleries, including the Guggenheim, the Whitney Museum of American Art, SIGGRAPH, the Banff Centre, the Moving Image Centre in New Zealand, the Central Fine Arts Gallery in New York, Artists Space in New York, the University of Arizona, the University of Colorado, Boulder, and many other international venues. Her essays on digital art, cyberculture, and gaming have appeared in periodicals such as *Art Journal*, *Wide Angle*, *Intelligent Agent*, *Convergence*, and *Culture Machine*, as well as several books. She has received funding by the National Science Foundation, the Pacific Cultural Foundation, and the National Endowment for the Arts. Flanagan teaches in the Department of Film and Media Studies at Hunter College, New York. Her research group and lab at Hunter is called tiltFactor. Visit it online at `www.tiltfactor.org/`.

Casey Reas, b. 1970

Casey Reas is a renowned designer/artist, lecturer, thinker, and of course one half of the Processing design/development team. He was a student of John Maeda's at the MIT media lab, where he and Ben Fry helped develop the DBN programming language/environment. Processing was, to a degree, a natural outgrowth of DBN. Reas originally studied design at the University of Cincinnati before attending MIT, and was one of the founding professors at Interaction Design Institute Ivrea. At Ivrea, Reas worked with an international student body to develop a new arts pedagogy. It was during this period in Ivrea that he and Fry initiated Processing. Reas's work is software-based—code and coding are his primary medium. However, he manifests his code-based work in a variety of executions, including kinetic, reactive, and printed pieces. Some of his most recent work employs ideas explored in conceptual and minimal artworks, such as the wall drawings of Sol LeWitt. In a commission he received in 2004 from the Whitney, entitled {*Software*} *Structures*, Reas explored the relationship between conceptual art and software art. For the project, Reas created three unique descriptive structures that merely described relationships between elements in the pieces. He purposely left the structures general, without any code notation, to allow other coders to implement the pieces in different programming languages. Three other leading code artists—Jared Tarbell of Levitated (`http://levitated.net/`), Robert Hodgin of Flight404 (`http://flight404.com/`), and William Ngan of Metaphorical.net (`http://metaphorical.net/`)—created the 26 code implementations. The project can be viewed at `http://artport.whitney.org/commissions/softwarestructures/`. When Reas is not lecturing around the world and developing Processing, he teaches as an assistant professor in the department of Design/Media Arts at UCLA. Reas is represented by bitforms gallery in New York, (`www.bitforms.com`), the BANK gallery in Los Angeles (`www.bank-art.com/index.html`), and the [DAM] in Berlin (`http://dam.org/`). His work can be viewed online at `http://reas.com/`.

Jared Tarbell, b. 1973

Jared Tarbell's code art is sensuously analytical, combining gorgeous tonal subtleties and implicit complex mathematical structures. This fine integration of aesthetic and analytical concerns sets his work apart. He gained prominence with his early ActionScript work and his inclusion in the friends of ED *Flash Math Creativity* books. In recent years, Tarbell has been creating with Processing, and the increased low-level capabilities of Processing and Java have allowed him to create even more organic, complex, expressive work. Most of this work reflects his interest in visualization and emergent behavior, as well as his commitment to open source development; he freely distributes his source code in modifiable form. Jared holds a BS in computer science from New Mexico State University, is a frequent lecturer at international conferences, and is a contributing author to another friends of ED book, *New Masters of Flash, Volume 3*. He maintains the websites Levitated (`http://levitated.net/`) and Complexification (`http://complexification.net/`), where his work can be viewed, and in the case of Complexification, purchased directly from the site.

Ben Fry, b. 1975

Ben Fry, along with Jared Tarbell, represents a generation of young artists who explore computational processes and structures as fundamental creative modalities. He's also of course the cocreator of Processing. Born the same year as the Altair 8800 was introduced, Fry and his generation never knew a time without personal computers. By the time Fry et al. were entering elementary school, the Macintosh computer was emerging, issuing in a revolution in desktop computer graphics. This generation was able to develop a fluency in computing that would have been impossible for previous generations (especially for artists). I think this fluency (and comfort) with computing is evident in Fry's work, which is less about the phenomena of computation and more about advanced and poetic applications of it.

Fry earned his undergraduate degree from Carnegie Mellon, double majoring in graphic design and computer science, and his PhD from MIT (Media Lab), studying under John Maeda and alongside Casey Reas in ACG. On the online title page of his PhD dissertation, he proposes, "To gain better understanding of data, fields such as information visualization, data mining and graphic design . . . be brought together as part of a singular process titled Computational Information Design."

Much of Fry's work deals with visualizing large data sets, including the human genome. After completing his PhD, he worked at the Eli & Edythe Broad Institute of MIT & Harvard, developing tools for the visualization of genetic data. His personal work also deals with visualization. For example, his well-known and visually engaging piece *Valence* is custom software he wrote about "building representations that explore the structures and relationships inside very large sets of information." You can read more about the piece at `http://acg.media.mit.edu/people/fry/valence/index.html`. Fry's work has been shown in galleries and museums throughout the world, including the Whitney Biennial, the Cooper Hewitt Design Triennial, the MoMA in New York, and the Ars Electronica in Linz, Austria. His work has also appeared in the feature films *Minority Report* and *The Hulk*, and in print publications, including the journal *Nature*, *New York* magazine, and *Seed*. Fry has been the recipient of numerous awards, including the Golden Nica from the Prix Ars Electronica in 2005 for his work on Processing with Casey Reas, and a Rockefeller Foundation New Media fellowship. He was also included in the "The I.D. Forty: Forty Designers Under 30 Years of Age." Fry currently holds the Nierenberg Chair of Design for the Carnegie Mellon School of Design. When he's not winning awards, helping students at CMU, or creating visualizations, he can be found answering users' questions on the Processing discourse board. You can see more information about Fry and view his work at `http://benfry.com/`.

And many more . . .

While I have included some wonderful and very noteworthy code art pioneers, there are many others of equal distinction I was forced to omit because of space concerns. I actually struggled quite a bit with this section. The following are some of these prominent folks with links to their work. I'm sorry I wasn't able to include more.

- **Charles Csuri**: `www.siggraph.org/artdesign/profile/csuri/`
- **Joshua Davis**: `www.joshuadavis.com/`
- **Andy Deck**: `http://artcontext.org/`
- **Amy Franceschini**: `www.futurefarmers.com/`
- **Ken Goldberg**: `www.ieor.berkeley.edu/~goldberg/index-flash.html`
- **Jean-Pierre Hébert**: `http://hebert.kitp.ucsb.edu/studio.html`
- **John Klima**: `www.cityarts.com/`
- **Mario Klingemann**: `www.quasimondo.com/`
- **Ruth Leavitt**: `http://dam.org/leavitt/index.htm`
- **Golan Levin**: `www.flong.com/`
- **Manfred Mohr**: `www.emohr.com/`
- **Colin Moock**: `www.moock.org/`
- **Ken Musgrave**: `www.kenmusgrave.com/`
- **Yugo Nakamura**: `www.yugop.com/`
- **William Ngan**: `http://metaphorical.net/`
- **Josh Nimoy**: `www.jtnimoy.com/`
- **Josh On**: `www.futurefarmers.com/josh/`
- **Robert Penner**: `www.robertpenner.com/index2.html`
- **Ken Perlin**: `http://mrl.nyu.edu/~perlin/`
- **Keith Peters**: `www.bit-101.com/`
- **Amit Pitaru**: `http://pitaru.com/`
- **Paul Prudence**: `www.transphormetic.com`
- **Daniel Rozin**: `http://smoothware.com/danny/`
- **Karsten Schmidt**: `www.toxi.co.uk/`
- **Manny Tan**: `www.uncontrol.com/`
- **Martin Wattenberg**: `www.bewitched.com/`
- **Marius Watz**: `www.unlekker.net/`
- **Mark Wilson**: `http://mgwilson.com/`

Summary

I wanted to create some code art historical context in this chapter and also hopefully provide some inspiring role models. As you progress as a coder, you'll want to study the work/code of others, and this list is a great place to begin. I think you'll find, as I have, that the "creative coding" community is made up of a wonderful group of generous, intelligent, highly creative people. Proof of this can be seen daily on the Processing Discourse board (`http://processing.org/discourse/yabb_beta/YaBB.cgi`), where a devoted cadre of code-helper zealots—JohnG, st33d, seltar, TomC, mflux, metaphorz, arielm, fjen, blprnt, flight404, shiffman, toxi, fry, REAS, and many more—stand ready to assist new and experienced coders alike. Thanks to all of you! This book has benefited immeasurably because of your generosity and combined wisdom.

2 CREATIVE CODING

I have a friend, Mark K., who was a web developer who had artistic yearnings (don't they all). I met him a number of years ago when he enrolled in a 3D animation course I was teaching. Mark worked for a sizable, very businessy type company that will remain nameless, doing pretty dull programming. Mostly he was working on legacy code—stuff other people had written—and trying to keep things together with band-aids and string. On top of this, his company was outsourcing most of the interesting development work overseas, and Mark also had the fun job of trying to coordinate video conferences at all hours of the night (big time difference), while integrating their code (in a programming language/ development environment he wasn't too familiar with) back into the company's legacy spaghetti (disorganized and poorly structured) code. Is it any surprise Mark was pretty sour on coding and dreamed about painting rainbows and unicorns in the Elysian Fields? Well, actually it was more like working in the game industry. We met at the perfect time.

While Mark was losing his passion for coding, I was losing my passion for pretty much everything else. I was dreading teaching the 3D course at the time, an area just a few years prior I had been passionately involved in. However, my spirits lifted on the first day of class when I met Mark. As a first day ice-breaker, I usually have each student tell me a little about themselves. When Mark described his programming background, I immediately imagined us having these long, intense discussions about the joys and intricacies of coding. Of course this fantasy was shot down quickly as Mark continued to explain his interest of eventually entering the game or animation industry and his waning interest in programming. A little disappointed, I summoned encouragement for his new interest and ambition and settled in for a long semester. In the end, the semester turned out to be great. Mark discovered an innate gift and interest in design, and the discussions I originally imagined having with Mark did indeed happen, but of course in a different and actually far more interesting way. Mark and I also developed a real friendship that continues to this day.

One of the subjects that Mark and I often discussed was the intuitive vs. analytical nature of coding. Mark had been formally trained as a programmer and thus had developed a "look before you leap" approach to coding that spilled into his other creative work as well. He was trained to analyze a problem, develop a plan, and then implement it. I on the other hand, trained as a fine artist, was taught to let go of the need to understand everything, and to learn to find structure and meaning in the process. Obviously, the utilitarian aspects of programming and the pretty much complete lack of utility of painting contributed to the approaches adopted in our training. At this point, I was mostly coding little graphics experiments with ActionScript. I would sit on the couch with the TV on, the laptop burning my legs (PowerBook G4), snacks of course, and tweak code until cool things started happening. Eventually one of my convoluted, incomprehensible equations would start yielding something I liked. Then I'd do a "save as" and keep tweaking until I found something else. I was able to hack out tons of art, watch bad reality TV, and still IM Mark during renderings. Mark was always amazed that I could approach programming that way. He just couldn't see it as an intuitive medium. We eventually tried some collaboration, but would often get stuck in our planning vs. implementing discussions. In time, though, we both began to learn from one another. I began planning some stuff before attacking by the seat of my pants, and Mark began to "throw some dung against the side of the barn." I really began to see the virtue in Mark's analytical approach when my projects began to get more complex and I started developing larger applications and reusable classes (which I'll cover a bit later). I hope Mark was able to learn from me as well, which seems to be the case, as he eventually landed an art director's position, and also has begun to enjoy coding again—with Processing, of course.

I think what was reinforced for me through my collaboration with Mark was the idea that to be really effective, you need to approach a problem both analytically and intuitively or even holistically. It is relatively easy to understand how to approach a painting or even design problem intuitively. As children, most of us scribbled with crayons and learned to make things without much planning. We learned to express ourselves with marks and gestures and weren't too self-conscious about making a mess. Our experiences in math class may have been quite different, as problems got framed by Boolean logic—the answer is either true or false. How you were feeling during the process of solving the problem had little bearing on the answer. When a non-expert looks at a design or painting, especially a representational image, they believe they can understand it (of course, whether they really can is a whole other issue). When that same non-expert looks at a page of computer code, it seems totally inaccessible and very likely overwhelming. They are used to looking at pictures and gestures, for which there doesn't seem to be a right or wrong answer. Looking at code seems like really hard work, and based on those early experiences in math classes, maybe even scary hard work. You do certainly need to put in some amount of effort to learn to decipher the code. But the same thing applies to being able to read a painting or design. It's just that many people believe that just because they have eyes, they can intuitively understand the visual formal language—the underlying pictorial structure—in a painting or design. But seeing, of course—especially seeing critically—involves the brain as well. Now I could very easily lapse into many pages of the most vitriolic prose about the current plague upon our nation because of rampant visual illiteracy, blah, blah, blah . . . but my editor will surely put the kibosh on that. So let me sum it up by stating that in my humble opinion it is much, much, harder to learn a foreign language than it is to learn a programming language, and having taught both painting and programming, it is much, much, much harder to (really) learn to paint or even draw a figure than it is to gain basic fluency in programming.

If you haven't thrown the book at the wall yet, a logical next question might be—especially if you are totally new to programming—"Exactly how much do I need to know before I can begin to start hacking out some art on my couch, and of course what snacks do you recommend?" Well, obviously, unlike drawing, you can't just grab the tool and begin scribbling. Well, if you start with some sample code (which Processing responds quite well to), you can, and that is one valid approach to learning to code. However, I think a much more efficient path, and I guess the main reason for the book, is to quickly (with minimal stress) develop a base coding literacy that will optimize your hack-per-minute efficiency—which translates into making more good art. Regarding snacks, I'm a big fan of Newman's organic stuff, especially the mint Newman-O's. (Yuummm!)

For many years, I tried the steal/tweak/pray coding approach, especially with JavaScript. However, there was always this fear and guilt in the back of my mind: the code will break, somebody will find out exactly how little I know, and I really can't take credit for this. Usually, freelance deadlines beat those feelings back. But using appropriated code that I didn't totally understand to make art for myself didn't feel as good. Also, as a professor I hate being asked questions I don't know the answer to. Of course, this happens practically every day when you have bright, talented students—which I've mostly been blessed to have—and are teaching emerging technology. Teaching Flash for many years, I would get asked practically every ActionScript question imaginable, and I eventually got really tired of BSing my way through half-assed answers. I eventually decided to formally train myself in programming. I got a mountain of books on programming languages (mostly Java and

ActionScript), object-oriented programming, and basic math (mostly algebra and trig), and I took the leap. It was based on this experience that I first got interested in writing a book on creative coding.

I spent many long, lonely nights with those books, especially the Java API (application programming interface—basically a description of the language's rules and tools). Over time I've come to realize that most programming how-to books are excellent for teaching you how to program if you already have a basic understanding of programming. The Java API, although indispensable to me now, was pretty scary, inscrutable, and downright frustrating back then. When I eventually began teaching entire semester courses on creative programming, I really learned a lot about how non-technical people learn technical stuff, and most importantly how to make it engaging. I knew from my own experiences that code was a powerful creative medium, but how to convince my wide-eyed art students was another thing altogether, and I knew most sane people wouldn't do what I did—tackling this scary stuff on their own. I should also point out that besides being obsessive compulsive, I was a science student prior to being an art student, so I had developed some tolerance and coping strategies for studying dense, unfriendly stuff. This book is a tribute to all my coding students who taught me along the way how to be a better teacher and (most importantly) what to do to keep them from dropping my classes.

The origin of Processing

Since around 1945 (or 1834 if you go back to Ada Lovelace's letter to Charles Babbage), many, many programming languages have been developed—in fact, more, I think than most people would guess. Unless you are deep in the computer science world, you maybe have heard of at most five to ten different programming languages. According to a somewhat comprehensive language list at `http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm`, more than 2,500 languages have been developed. My guess is that the number is actually much higher. Interestingly, at the time of this writing, Processing isn't on the list. This might be due to the list not being properly maintained or because some purists might not see Processing as a separate language. *What*? Well, rest assured, I sent an e-mail telling them all about Processing. But there is a valid argument to be made that Processing is not technically a language, but more of a programming environment—what is commonly referred to as an IDE, or integrated development environment. I'll try to make this argument and then I'll refute it. I love arguing with myself. However, before I get into more Processing nuts and bolts, a more conceptual over-the-hood discussion about Processing and its forebear, DBN, will be helpful, or at least (I hope) mildly interesting.

Processing grew out of research in the Aesthetics + Computation group at the MIT Media Lab in around 2001. Prior to that, John Maeda, the group's famous leader, with the aid of some of his students, developed DBN. DBN was a very simple programming language and environment, begun in around 1999. Casey Reas and Ben Fry also participated in the development of DBN. Processing can be thought of as the very advanced, production-grade child of DBN. DBN was usable on the Web or as a stand-alone application, and was designed to massively simplify the coding process for visual artists. Unlike Processing, DBN didn't have much application outside of the classroom, as the screen output was limited to about 1.5 by 1.5 inches and there was no color. Yet many of the basic processes we expect in a programming language were there, including the ability to set and recall values, create customized commands, and run loops. It even had some basic networking and event behavior (mouse and keyboard detection). If you're interested to learn more, as of this writing, the DBN site can still be found at: `http://dbn.media.mit.edu/`. DBN was developed to simplify the massive complexity involved in graphics programming. Later on in the book, I will provide some sample code illustrating the minutiae and verbosity of code needed to generate the simplest graphics, even in a high-level language like Java. By "high-level," I mean a language that is supposed to be easier to work in than a low-level language. You can think of the lowest-level language as zeros and ones. DBN was in a sense a reaction to this problem, and it went, some would argue, overboard in addressing it. Processing, having the benefit of standing on the shoulders of DBN, as well as being a second go-around for Ben and Casey, cleverly addressed this problem by providing a multimodal approach, allowing some users to work in a very high-level way—like in DBN—while other codeheads (myself included) could work in pure Java and still others could work with a hybrid approach. This totally inspired design, providing *choice*, is I believe what makes Processing a very special and cutting-edge tool/environment/language. Today, Processing, free from the confines of the Media Lab, is supported by Ben and Casey, with a growing cadre of devoted (some might say obsessed) "Processors" around the world.

Programming language comparisons

Now for some nuts and bolts. If you get a little lost in this section, don't panic; it will become clearer in time. Also, it is not critical info to get started coding. If you really hate it, just skip it. But I strongly recommend coming back to it eventually. I put it here in the beginning of the book because I do think it is important.

The argument to be made that Processing is not actually a programming language, but rather a programming environment, is based on the fact that Processing is built on top of the Java programming language. Of course, parts of Java itself were created with other programming languages as well (C and C++), so what's the point? The point is that when you run Processing, you are actually running a Java program. Processing allows you to write much simpler code than you would need to write if you were working in pure Java. It also allows you to **structure** your code in a **function-based** (procedural), **non–object-oriented** style. But once you **compile** your code, the output is converted to **Java class files**—indistinguishable from any other Java program—and the class files are **interpreted** within the **Java Virtual Machine** as your program runs. Now don't freak out! I know I just dropped a bunch of terms you may not be familiar with. We'll take it slow, and deal with each term, one at a time. Again, my recommendation is to read this stuff and not get too stressed. If it's really bugging you out, skip it and return to it later; you'll still be a good person. I promise, in time, it will all be clear (or at least clearer).

If you have some prior programming experience with another language, or if you're thinking, "Why should I learn Processing instead of another language?" the Processing site includes a good comparison and contrast of a competitive set of languages to Processing (`http://processing.org/reference/compare/index.html`). Of course, I don't really think these languages compete in any real sense, as each has its own strengths and weaknesses. Additionally, if this coding thing sticks for you, it is very likely that you will learn some if not most of these languages in the future. It actually gets pretty easy to learn a new programming language after you grasp basic programming theory. As I mentioned earlier, learning a programming language is much easier than learning a verbal or natural language. Now let's tackle some of the new terms I just threw at you and look at just a little processing code.

First, what does it mean to "structure" your code?

Here is a simple Processing program that draws a black rectangle on a white background; it's not too ground-breaking, but it is a nice rectangle (see Figure 2-1 for an image of the rectangle that the code produces).

```
size(400, 400);
background(255);
noStroke();
fill(0);
rect(width/4, height/4, width/2, height/2);
```

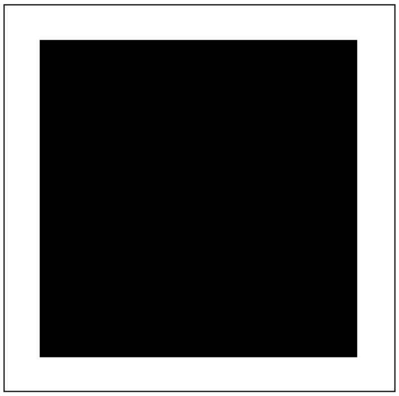


Figure 2-1. Rectangle created using an unstructured coding approach

A program this simple wouldn't necessitate structuring your code any further. So, at this point, if all you need to make are single squares, you're done. Congratulations! However, imagine instead of having 5 lines of code, you had 50, or 500, or 5,000? It becomes very difficult to understand and maintain a program when it reaches a certain scale. There are ways of structuring your code, kind of like using paragraphs to organize a chapter, that make a large program more manageable. In addition, there are structural ways of simplifying your program, reducing redundant lines of code, and ultimately increasing your coding efficiency and possibly the program's performance (these are all good things).

Function-based (procedural) vs. object-oriented structure

There are a number of ways of structuring your code—I'll discuss two ways it's done in Processing. The first way uses functions (also referred to as procedures, or subroutines—different words essentially describing the same technique). Functions are reusable blocks of code that work like processing machines, which you call when you need them. The second way uses an object-oriented approach, a far more ambitious and abstract approach that models a programming problem using concepts from the real world. In object-oriented programming (OOP), the code is divided into smaller (and independent) blocks of code called objects.

Before I describe and contrast each of these approaches in more detail, it is worth stating that Processing allows you to structure a program using either a function-based, object-oriented, or hybrid approach. In fact, if you want to, you can avoid structure altogether. This amazing flexibility, accommodating different types of coders with varied experience levels, is one of the features that makes Processing such a unique environment. Later in the book, I will go into a much more detailed explanation of this material (I hear your heart fluttering with delight), but for now I'll just describe them in very broad strokes.

Let's go back to the simple five-line program. This program executes from top to bottom. You could test this by moving the `fill(0);` line below the `rect(width/4, height/4, width/2, height/2);` line.

```
size(400, 400);
background(255);
noStroke();
rect(width/4, height/4, width/2, height/2);
fill(0);
```

You wouldn't see anything if you now ran this program—can you guess why? It's because the color used by the program to draw the rectangle starts out as `white`, the same color as the background (255 equals white), and is only changed to black (0 equals black) after the rectangle is drawn. So you are actually drawing a white rectangle on a white background. Now, imagine that you have 5,000 lines of code—wouldn't it be easier to not have to put every line in an exact sequential order to get the program to run properly? Isn't the whole digital thing about not having to be so sequentially linear? Function-based, or procedural, programming can help solve this problem by breaking a program up into reusable code structures that you call as you need them. These blocks are called **functions** (as opposed to objects in OOP); you can think of them as independent processing units waiting for action. Functions are explicitly called when they are needed and can optionally be passed values, which the functions can act upon. Calling a function can be considered the entry point of a function. Once called, a function can either end when it completes its job, or optionally return values. For example, if I wrote a program to keep attendance in my classes (something I've never done very consistently), I might create a function that simply assigned a value of `present` or `absent` for each student on a specific date. Then in class, I could call the function and pass in the value `present` or `absent` for each student. I wouldn't need the function to return any value. On the other hand, if I wanted to find out which students were absent on a specific day, I might create another function that when called would return the actual names of the students who were absent.

Let's make the current rectangle drawing program a little more complicated. Don't worry about trying to memorize or even understand the code for now. It is the theory that is most important. Here's the program structured using functions:

```
void setup(){
  size(400, 400);
  background(255);
  noStroke();
  float myColor = getRandomColor();
  fill(myColor);
  rect(width/4, height/4, width/2, height/2);
}
float getRandomColor(){
  return random(255);
}
```

The program still draws a rectangle on the white background, but instead of the rectangle always being black, the program calculates a random value between white and black for the fill color of the rectangle. The line `return random(255);` generates and returns a random number between 0 and 255. 0 equals black and 255 equals white, and anything in between equals a variation of gray. You'll notice the program now has these strange brackets,{ (open) and } (closed). These are called **curly braces**, and they are used to group the lines of code needed by the function to do its job. You'll also notice that the program is broken into two groups of code: `void setup(){...}` and `float getRandomColor(){...}`. These groups are both functions. The lines `void setup(){...}` and `float getRandomColor(){...}` are the function declarations. It is a good idea to use a verb in the name to describe what the function does. That way, it is easy to understand what that function's job is when reviewing the code. Also, a function is followed by open and closed parentheses. This is the place where any arguments (values) the function will need to do its job are included—commonly referred to as "passing arguments." Even if no arguments are required, as in the preceding case, you still need to include the parentheses.

There is another very common programming structure called a **variable**, which I'll discuss in detail a little later in the book. A variable is simply a location in memory that will hold a piece of data. For instance, in the left part of the expression `float myColor = getRandomColor();`, `float myColor` is a variable, to which a numeric value can be assigned. The word "float" just specifies the type of numeric value the variable can hold.

The line `float myColor = getRandomColor();` is a function call. Actually, the right part of the expression, `getRandomColor();`, is the function call, and the left part of the expression (`float myColor =`) simply assigns the value that the function call returns to the `myColor` variable. Here's where things get a little interesting. When the `getRandomColor()` function is called, it takes over—before the line below the function call (`fill(myColor);`) is dealt with, the code inside the actual `getRandomColor()` function is run. Only when the function called finishes will the code below the function call be run. Inside the `getRandomColor()` function, one of Processing's built-in functions, `random()`, is called. Once the `random()` function does its job, it returns a number back to where the `getRandomColor()` function was called from, and this returned value is assigned to the variable `float myColor`. I realize this may seem a bit overwhelming if this is your first time dealing with coding; I promise it will make (more) sense over time.

OOP was developed later than function-based or procedural programming, and was designed to try to improve on some of procedural programming's perceived limitations. In a general sense, OOP was designed to allow the types of modules that functions introduced to become larger, more complex, and more independent entities. OOP facilitates code reuse—which will become more important as you get more experienced—and enforces certain programming rules. OOP is a much more complex approach to programming that delves into very interesting but quite challenging abstractions. I will cover OOP in Chapter 8, although I include a brief overview of it in the next chapter, which focuses on code grammar. OOP is such a big subject that it would be worthwhile to devote an entire book to creative OOP with processing—maybe if I'm still standing when I'm done writing this one . . .

If you're interested, here is a Processing OOP version of the little rectangle program, with some comments—the descriptions following the two forward slashes (//). I am not going to do a full analysis of the program here, as I'll be covering these concepts in painstaking detail later in the book.

```
// Processing Object-Oriented approach: Create a rectangle
void setup(){
  // set display window to 400 x 400
  size(400, 400);
  // set background color to light gray
  background(225);

  //instantiate a MyRect object and assign it to the reference ➥
    variable rectangle1
  MyRect rectangle1 = new MyRect(width/8, height/8, width/4, ➥
    height/4, 200, -1);

  /* call drawRect() method using MyRect object reference, rectangle1
     calls the method using  dot syntax (literally
     the period between the object and the method call).*/
  rectangle1.drawRect();
}

// MyRect class definition
class MyRect {

   // public instance fields - each MyRect object will have
   // their own set of these properties
  float x, y, wdth, ht;
  int fillCol, strokeCol;

 /*constructor - called when the object is instantiated, using the new
   operator. the parameter list between the parentheses needs to match
   the argument list in the instantiation call */
  MyRect(float x, float y, float wdth, float ht, int fillCol, ➥
    int strokeCol){
   // initialize instance fields - assignment happens form
   // right to left, e.g. the value of x is assigned to the
   // instance property this.x
    this.x = x;
```

```
        this.y = y;
        this.wdth = wdth;
        this.ht = ht;
        this.fillCol = fillCol;

        /* check for stroke - if the last argument passed to the
           constructor is not -1 then use that value to set the
           stroke color. If it is -1, don't draw a stroke */
        if (strokeCol!=-1){
          this.strokeCol = strokeCol;
          stroke(strokeCol);
        }
        else {
          noStroke();
        }
    }

    //drawRect method
      void drawRect(){
        //assign fill color
        fill(fillCol);
        // draw rectangle
        rect(x, y, wdth, ht);
      }
    }
```

The other terms I threw at you earlier were “class files,” “compile,” “interpret,” and “Java Virtual Machine,” which I’ll define as I introduce Java and describe Processing’s special relationship to it.

Java

I want to cover a little of Java’s history (sorry), but I think you may find it somewhat interesting, especially as a lesson in the virtue of happy mistakes. I actually cover happy coding mistakes a little later on as well. Java’s history is also relevant in regard to a deeper understanding of Processing. Have I convinced you? Hope so.

Java, originally named Oak, was developed in the early 1990s as a language to be used to program devices other than personal computers—things like microwave ovens with embedded processors—not terribly sexy stuff, but as you’ll see, pretty visionary. However, appliance manufacturers didn’t embrace the idea. So to save face, the Sun engineers went back to their drawing boards and came up with a plan to remarket Oak as a language to control “set-top boxes” for interactive TV. But alas, the cable companies passed on that idea as well. Things were not looking good for Oak. To make matters even worse, it was discovered that the name “Oak” had already been trademarked by another technology company. Before I tell you how Java was reborn out of this very inauspicious beginning, let me explain the basic problem the Sun engineers were trying to address in the design of Oak in the first place. Then you can try to guess (if it is not already obvious) what saved Java.

Without getting too geeky, the main objective in the design of Oak was to develop a relatively simple-to-program, high-level, universal programming language. The really key word here is "universal." A proprietary technology (and a refusal by companies to respect and adhere to standards) creates a difficult landscape to work in. This problem is not limited to the computer industry. A number of years ago my wife and I invested in a water filter system that attaches directly to our kitchen faucet. Over the years we've moved a lot, dragging our filter with us. In almost every new home, we've had to contact the filter company to request yet another odd-sized adapter (we've got a quite a collection by now). Lack of standardization has up until recently been especially frustrating and difficult for web designers, who were forced to develop multiple sites, limit design concepts, and create crazy workarounds to ensure that their sites worked across all the different web browsers (this is no longer so much of a problem because of the advent of web standards, better development practices, and better and more consistent support for those standards across browsers).

One of the benefits of a program like Flash is a universal Flash player that ensures consistent content delivery across different platforms (operating systems and web browsers). The history of Flash, originally called FutureSplash Animator, arguably would have been quite different had Java not a few years earlier pioneered the concept of a portable, universal language.

So the emergence of the Internet saved Java. Java's now well-known slogan, "Write once, run anywhere," turned out to be the perfect strategy for the Internet. And by 1996, Java was also embedded in Netscape's browser, ushering in a revolution in online multimedia-based content and development. Interestingly, in the ten years since this development, Flash has far surpassed Java as the online multimedia-based (front-end) content tool of choice, and Java has moved to the back end of the Web—to the server side. Java is also widely used on mobile and wireless devices, such as PDAs and cell phones. With the development of Processing, Java is again being used for online content creation. So arguably, Processing has saved (or at least resurrected) the Java applet. Although unlike Flash, which is used to enhance/develop entire websites, Processing (currently) is being used primarily for aesthetic exploration. Of course, as more Processing tools and code libraries are developed, this could change. OK, so that's a nice story, but how do Java and Processing work?

Java is considered both a compiled and interpreted language. Unfortunately, the terms "compiled" and "interpreted" don't quite mean the same thing in programming as in general parlance. To compile, in computer-speak, is to convert a (usually high-level) programming language into something lower level. Remember, a computer likes zeros and ones (the lowest-level language), but most people find it frustrating to communicate in binary code. It is easier to write programs in the same natural language we speak (the highest-level language), but a spoken language is extremely complex and would demand tons of processing to be able to interpret it. Therefore, in the design of a programming language, there is a trade-off between ease of use and performance.

Java was designed with this trade-off in mind. Sun's Java compiler, javac (Processing actually uses a similar open source compiler, developed by IBM, called jikes), converts a programmer's code (the stuff we write) into a middle-level language called bytecode (that the compiler writes). This bytecode (also known as class files) is universal, meaning that I can take my class files compiled on my Mac and run them on a Windows machine, as long as the Windows machine has a Java Virtual Machine (JVM). As most operating systems and browsers have a JVM, I can use my class files in any of these environments, regardless of where they were written. Thus, you can see the power of Java's "Write once, run anywhere" strategy.

In Java's early days, it was considerably slower than non-interpreted languages like C or C++, which don't use a middle compilation state (the bytecode). These languages compile directly to machine code specific to the native underlying system. Thus, these languages are highly efficient, but not easily portable. If you write a program in C++ on a Mac, it will only run on a Mac. You have to literally go in and change some of your code to create a second version of your program (referred to as porting your program) if you want it to run on a different platform. Today, based on certain advances in compilers, such as "just-in-time" compilers (in which some of the Java bytecode is compiled to native machine language instructions during runtime), Java's speed is comparable to lower-level, non-interpreted languages like C and C++. This gap continues to narrow, or become insignificant, as computers continue to get faster. So Java's strategy seems to be really paying off.

Processing has all the benefits of Java with a much simpler and optimized environment for creative coding. As an example, following is the original five-line rectangle drawing program written in Processing. I've also written the same program as a Java applet.

Here's the original Processing program:

```
size(400, 400);
background(255);
noStroke();
fill(0);
rect(width/4, height/4, width/2, height/2);
```

Here's the same program as a Java applet:

```
import java.awt.*;
import java.applet.*;

public class MyStage2 extends Applet{
  public void init(){
    setSize(new Dimension(400, 400));
    setBackground(Color.white);
  }
  public void paint(Graphics g) {
    g.setColor(Color.black);
    g.fillRect(getSize().width/4, getSize().height/4, ➥
      getSize().height/2, getSize().height/2);
  }
}
```

The Java applet requires about twice as much code as the Processing sketch. However, as you write more complex programs in Processing, the number of required lines of code to write the same program in pure Java begins to increase exponentially. So if Processing is essentially Java, how does it do the same thing with less code? It does it by using a software framework that does a lot of the extra (non-creatively oriented) work under the hood, leaving you to just concentrate on the fun stuff.

A framework is just a concept for bundling precompiled code together for use in developing a related family of programs. From a programmer's standpoint, a good framework allows you to write less code and focus more on the higher-level concerns of your program. Java relies on many frameworks. For example, Java's Collections framework allows different types of groupings of objects (collections) to be manipulated in a uniform way, and, as Sun says, "independently of the details of their representation." This is a huge advantage to a coder, who now only has to learn one approach (a single pattern) to getting or setting a value within a collection of objects, even though internally these collections may use very different implementations. The Processing framework does a similar thing, but in the area of creating and manipulating graphics and images. Graphics programming is complex, and using pure Java involves ascending a pretty steep learning curve. In addition, after learning how to create some graphics (shapes and text), you would still need to learn a whole new set of procedures for working with images, not to mention video, animation, and so on. Using the Processing framework, these disparate and complex operations are standardized into a uniform and simplified set of commands.

Procedural OOP ("poop") approach

I suspect by now, if you've stuck through this chapter, you're in need of a little diversion, so a short story might be refreshing. A number of years ago I was preparing for a show of my paintings. I was very excited about the show, as I was only a few years out of grad school and had this romantic vision of being discovered, and picked up by a major New York gallery. The fact that my show (actually a two-person show with my friend David) was at a small cooperative gallery in Piermont, New York, didn't seem to have any effect on my fantasy. In preparing for the show, I stretched up a number of large, impressive looking canvases on which to create my "serious" work. Besides preparing for my show, I also had a brain-numbing job doing mostly print production work for an aesthetically bankrupt (though not at all financially bankrupt) corporation. My one fun indulgence was a Thursday night drawing group I frequented in Manhattan.

When I first moved to New York City, I looked up an old professor who invited me to a weekly drawing group at his studio. I was of course thrilled and went to my first session expecting to bask in all the intense aura of professional, established New York artists at work, but instead found a party-like atmosphere with free-flowing wine, loud music, and continuous conversation. In time, I became quite accustomed to this new approach and had a great time listening to the older artists tell stories about the good old days, and of course we did make art.

So each week I would go to the drawing group, crank out a bunch of work, come back to my studio, and pile the work in the corner. During the days, when I wasn't slogging through Quark at the mind-numbing job, I painted my "serious work." This went on for nearly a year, until I began final preparations for my show. When it got down to selecting the actual paintings to put in the show, I had a major revelation (it actually felt more like a kick in the head). My serious work, well, kind of sucked. It was self-conscious and forced. I was pretty desperate and had no time to create new work, so I went to the corner with my "non-serious" work piled high from the Thursday night drawing parties and started rifling through the pile. I was shocked—the work was really interesting and fresh, but also accomplished in its own way. The work had spontaneity and energy, but it also had structure and clarity. The "serious" day painting that I thought was my real work was actually something else altogether. In retrospect, although the work was self-conscious and forced, the discipline of doing the serious painting contributed to my skills and technique, allowing me to let go on Thursday night to really express myself. It was this combination of discipline and freedom that I attribute to the quality of the work. I ended up only putting the Thursday night work in the show, and got a very positive response. Of course I didn't get famous, but I certainly kicked butt in Piermont.

So what the heck does that story have to do with Processing? Well, when it comes to writing creative software, there are a number of ways to approach it. There is the "serious" way and there is the Thursday night party way. It is not so much a specific set of techniques I'm talking about, but rather an attitude. Some programming languages and environments demand a highly structured and rigid approach that doesn't favor spontaneous expressive shifts. OOP, which is a beautiful construct that I will go into later in the book, requires planning and a well-thought-out plan; Without such a plan, a "by the seat of your pants" OOP approach can lead to piles and piles of code and excess complexity, taking you away from self-expression and burying you in pure digital craft, or worse—damage control. The other extreme, pure gut hacking of unstructured spaghetti code, can yield fast effects and little teasers, but it becomes overwhelmingly difficult to carry a creative impulse through to a more realized form. So there needs to be a middle way, combining structure with fast prototyping capabilities. This yin/yang approach is possible in Processing, which provides a fast, loose, procedural coding environment with a highly structured and powerful object-oriented backbone. That is not to say there aren't pitfalls to working in such a flexible environment, in which it's easy to develop nasty hacking habits. My recommendation, and the way I've structured the upcoming tutorial sections in the book, is to begin coding with passion. Don't let the fear of doing it *right* or learning everything first get in the way of expressing yourself, but at the same time, study the language (first Processing and then eventually Java) and OOP; it will ultimately empower you to create larger, more expressive works.

Algorithms aren't as scary as they sound

1. Buy a new copy of *Processing: Creative Coding and Computational Art*.
2. Open the book to page 1.
3. Begin reading at the top of page 1.
4. Stop reading at the bottom of the last page of the book.
5. Write a glowing review of the book on Amazon.com.
6. Go to step 1.

These six steps constitute an algorithm. They are not a computer program—they are just specific directions on how to achieve something. The term *algorithm* comes from Abu Abdullah Muhammad bin Musa al-Khwarizmi (referred to just as al-Khwarizmi), a Persian supergenius (with a really long name) from the early part of the ninth century. We get the word *algorithm* from the al-Khwarizmi part of his name. al-Khwarizmi worked near Baghdad as a scientist, mathematician, astronomer/astrologer, and author. He is also commonly referred to as the father of algebra. You may have first been introduced to the word *algorithm* in a math class in high school, but you may have been daydreaming at the time (or, if you're anything like me, drawing a less-than-flattering picture of your math teacher). Algorithms are fundamental to mathematics, but they are also essential to pretty much any other planned activity. If you go to Google and type define:algorithm, as of this writing, 28 definitions come up. One definition I like in particular, from `http://images.rbs.org/appendices/d_glossary_geometric.shtml`, is "A step-by-step procedure used for solving a problem." I like this definition because it is so incredibly simple and universal, and can be applied to practically any problem. The only thing I might add to the definition is the limit on the number of steps. My new definition is "A *finite* step-by-step procedure used for solving a problem." When you're dealing with programming—or really, any task—an infinite number of steps is almost always a bad thing. For example, in the preceding algorithm I wrote, experienced coders probably saw a major logic problem; the algorithm never ends. Intellectually this might be an interesting idea, like a hall of mirrors endlessly reflecting one another, but in programming we call this an infinite loop. Loops are central to programming, but we usually want them to end when some condition is met. For example:

1. Buy a new copy of *Processing: Creative Coding and Computational Art*.
2. Open the book to page 1.
3. Begin reading at the top of page 1.
4. Stop reading at the bottom of the last page of the book.
5. Write a glowing review of the book on Amazon.com.
6. If Ira's still not a millionaire, go to step 1.
7. Stop following this algorithm (and thank you)!

The logic in the algorithm will no longer generate an infinite loop. Instead, at step 6, only if the condition is not met (alas, I'm still not a millionaire) does the algorithm repeat. But eventually the condition will be met, step 6 will be skipped, and step 7 will end the loop. Unless there is some specific direction to go somewhere else (e.g., go to step 1), the program keeps executing from top to bottom. This is why the program falls through step 6 when the condition is no longer met and executes line 7, ending the loop.

Computers are inherently dumb machines. They are super-quick at processing data and have awesome memories, but they have no innate intelligence. For example, if I asked a computer to hold a 1,000-pound weight over its head (just assume computers have arms), it would eventually destroy itself by dropping the weight on its head when its arms gave out. Why? Because computers are perfect literalists; they do only and exactly as they are told, without any assumption, reflection, or self-awareness. Unless I instructed the computer to put the weight down when it felt its arms getting tired, it wouldn't do it. One of the difficulties in learning to code is not that we have to think so brilliantly, but rather that we have to think so mind-numbingly literally—without the benefit of emotion, assumption, or intuition. For example, if I want to ensure that you will continue reading this book, I could tell you that you *must* keep reading this book. However, because people are intelligent and not robotic literalists, the majority of you will likely tell me to go to hell. Generally speaking, if we want human beings to do something, we need to make an appeal with a more complex strategy, taking into account feelings, social conventions, personal history, favorite baked goods, and so forth.

Humans are excellent at intuiting. We can meet someone for the first time and within a couple of minutes build a complex initial profile of the person; not that all these assumptions hold up—but it's amazing, with so little data, that any do. We seem to be able to remember subtle general patterns of behavior that let us build composite types. Psychologists have named this organizing function *gestalt*. Gestalt is also commonly defined as the sum of the parts being greater than the individual parts. For example, when you show someone a picture of Elvis, they don't think "two eyes, a nose, lips, sideburns, sequins, etc." They don't even think "man" or "human being"—they think, "It's Elvis!" The gestalt allows our minds to generalize object-oriented models based on our memories and sensory data. The gestalt makes us highly efficient at processing vast amounts of complex information without going insane. Imagine if we were constantly aware of all the sensory data surrounding us? However, this amazing organizing ability we have can also cause major problems when we think our limited mental models reflect all of reality. Teaching drawing for many years, I felt my hardest job was convincing the students to let go of the way they thought they saw the world, to be able to learn to see it again freshly and expand their perceptual paradigm. When people first start out learning to program, they typically make some common intuitive assumptions because of the gestalt. These types of assumptions, which work more or less in the real world, fail miserably when applied to programming. Computers innately lack an internal gestalt to discern the whole from the parts. They are essentially just dumb (although ridiculously powerful) calculators. If computers have any intelligence at all, it is at the software level. Software—operating systems, programming languages, and applications—to some degree create a gestalt-like reality in the computer, the so-called spirit in the machine. Thus, perhaps the passion people feel for their machines is really a connection to the spirit imbued within the machine (the software)—which of course is the human presence.

Here's a concrete example that illustrates how our sophisticated brains can get us into trouble when we first start writing code. An exercise I always cover early in my creative coding class is how to generate a simple moving object programmatically (meaning using code). The problem is pretty simple: Create a small rectangle—we'll name her "recty" (she's a female polygon). Make recty move around the screen and have her bounce off the edges of the frame, never letting any part of her go out the frame.

The first challenge students generally encounter in this problem if they have no programming experience is simply moving recty. For this example, let's assume that recty is already on the screen in a frame and has an `x` property. This means that recty's `x` position (her horizontal position) on the frame can be set and retrieved by using her `x` property. Also assume that recty's registration point—a point or pixel on recty that the `x` property is measured from—is in her top-left corner, and there is also a variable named `speed` that controls how fast recty moves. I'll assign a value of `3` to the `speed` variable using the expression `speed = 3`, and I'll also start a loop running that continues to update the frame (and recty's position). I'll be covering all this stuff in more detail later on, so it's OK to be a little out of breath at this point—but just avoid operating any heavy machinery.

To move recty, most new coders try this first: `recty.x = speed`, which I think makes perfect sense because they're assigning `speed` to recty's `x` property. Unfortunately, this expression doesn't get recty moving; although something else happens—recty gets moved to the left edge of the screen (to 3 pixels to the right of the left edge of the frame) and stays there, because `recty.x = speed` doesn't increment recty's `x` position by `speed`, but rather assigns the value of `speed` (which is `3`) to recty's `x` position—putting her at that point in the frame. The loop keeps running and keeps assigning 3 to `recty.x`, so she doesn't move (poor recty), even though the frame is being refreshed continuously. So why do people do this (besides just to annoy their teachers)? I think it's because it is so easy for us to understand how to move that we don't waste any brain cells thinking about it.

Let's imagine you and I were standing across a room from one another and I'd like you to come join me on my side of the room. More than likely, I'd just say "please come over here." Your brain would fill in all the details on how to pull this off, which it would then communicate to your body and you'd begin walking across the room, most likely in a direct path toward me. Even though I didn't tell you a rate or direction to move, you'd know exactly what to do. Of course, my one-year-old daughter would probably not respond to the same verbal cue. However, if I held up a shiny object from across the room, she'd likely begin crawling over to me, also in a straight path no less. One of our challenges in programming is thinking beneath our intelligence. Although moving across a room may seem like a really simple task, there are a tremendous number of assumptions and calculations we make to achieve it, not to mention some pretty insane physics. When we model even simple behavior on the computer, we need to account for these assumptions—which is difficult. The correct solution to the movement problem is the following expression:

```
recty.x = recty.x + speed
```

Now this expression often looks a little confusing to people in the beginning. Why would something be equal to itself? It seems illogical. First of all, in programming, the symbol = doesn't mean *equals*, but rather *assignment*. The symbol == means *equals*—I know this seems weird. It might help if I write in English what's going on in the expression `recty.x = recty.x + speed`: "Add the value of the variable speed to recty's current x property, then assign that value back to recty's x property, and then do it again and again. . ." Thus, after each loop cycle, recty's x property will increment by 3. By writing the logic out as an algorithm, you can better understand what's happening. In this case, `recty.x` keeps increasing each loop cycle of the program, moving her across the screen. Will recty stop when she hits the end of the frame? Nope, we didn't tell her to. In fact, recty would continue moving until we quit the program, but of course you won't see her, because she's out of the frame. If this is still unclear, don't worry at all. This is difficult stuff to begin to get your head around. Another trick to help you understand this stuff is to do the calculations manually. For example

- Remember speed equals 3.
- Start `recty.x` out at 0.
- First, loop `recty.x = recty.x +speed`.
- Now `recty.x` equals 3.
- Second, loop `recty.x = recty.x +speed`.
- Now, `recty.x` equals 6.
- Third, loop `recty.x = recty.x +speed`.
- Now `recty.x` equals 9.
- See the pattern.

I hope that helps; it will come in time if it's still a bit fuzzy. If you're anything like me, It just takes some time to restart the brain after years of turpentine exposure and bad TV. When you get stuck on a programming problem, it really does help to write out a simple algorithm. Later on, when you begin cranking out thousands of lines of dense code, creating algorithmic roadmaps before you start coding might even eventually begin to feel like conceptual brainstorming or preliminary sketching—of course, by then you will also be a certifiable geek.

Happy coding mistakes

After my diatribe about algorithms, how can a coding mistake be seen as happy? Well, I don't mean "computer bug" when I write "mistake." A bug is usually not happy. Bugs are when stuff happens that is not very interesting and usually annoying. Bugs are the things in software—such as missing semicolons, misspelled variable names, or infinite loops—that can drive you insane and make you try to physically impale your keyboard. Happy coding mistakes, on the other hand, are nonfatal things that happen, often revealing unforeseen possibilities. As a painter, it took me a long time to be comfortable with my mistakes (which were many). I wanted to control the paint—which of course is pretty much impossible. In time, I began to realize that the so-called mistakes I was making were often the best parts of the paintings. This phenomenon occurs in all creative pursuits, as somehow our unconscious mind seems to assert itself when we least expect it. Arguably, this is also where our inspiration lies. I think more experienced artists just learn how to better utilize this capability as well as gain the ability to more selectively recognize the happy accidents from the bugs. Coding is a creative process, and thus code art can benefit from these happy mistakes. However, because the process of coding involves many more rigid rules than painting, it is not as easy to understand the concept of happy coding mistakes. In addition, when starting out coding, it is easy to get caught up in the technical aspects or craft of coding and lose sight of its power as an expressive medium.

There is a macho aspect to coding, as well as to painting. You begin to feel powerful and capable of describing all kinds of interesting behaviors; plus, you have access to a secret language that most people don't understand. Coding can become like a giant, multidimensional crossword puzzle, and you can easily get consumed in these really dense, abstract worlds—losing site of expressive possibilities or what you need to say (in an artistic sense). You need craft in coding; the more mastery you have of your medium, the wider the scope or range of your work. You also need a creative vision; without it, coding can become an analytical quagmire. I can't say that I always navigate clear of the quagmire myself, as I often get caught in trying to solve these endless, albeit interesting, technical problems. But when I let myself play with the code and let things happen that are not completely in my control, I often have more fun and even experience the same kind of (warning; new-agey phrase coming) transcendent feeling I get when I am in the flow of painting. I also think the work is more original and expressive. Ultimately, I think it is about finding flow in the coding process that takes you to new places and allows you to turn your so-called mistakes into new and fertile investigations.

OK, let's assume you buy some of this; a logical question for a new coder to ask is "How do you begin to find or embrace happy coding mistakes when you don't even know the fundamentals (or basic craft) of programming?" I think the answer is to always try to find the time to play at whatever level you are at. This is a really important approach that teachers are sometimes lousy at implementing. It is easy to get caught up in teaching the pursuit of mastery, focusing on achievement goals for students and checking off the requisite material covered. The school system reinforces this approach with standardized testing, creating crammers and compartmentalizers who separate work and play, craft from art. Is it any wonder that many of us (especially creative types) felt bored and frustrated in the traditional classroom, especially in subjects that required methodical drilling or rote memorization (like math)? Sadly, as we get older, we eventually label these subjects as the problem, not the non-integrated teaching approaches. My experience in the classroom has been that I find most people receptive to learning anything when it's taught effectively. And my vision of effective teaching always involves finding a way of integrating aspects of play and work into the learning process. Before I launch into yet another whiny soliloquy, I'll sum it up with the following: real code art (whatever the heck that means) can be made at any level—by a newbie coder learning the basics or a seventh-degree Java black belt—and happy coding mistakes help us see the expressive possibilities at whatever level we are at.

Before I end this chapter, I think it might be helpful to illustrate this point with a short example of how I approach happy coding mistakes in my own process.

Algorithmic tree

I used to paint a lot of trees. Originally, I painted the entire landscape, but gradually over time, I found myself zooming in closer on the trees, until one summer, I painted an entire series of large paintings based on part of one tree. It is interesting watching your neighbors' reactions as you spend more and more time standing in your backyard, staring up at the same tree.

So in honor of my favorite tree, I thought I'd illustrate some creative coding by generating a tree with code. Let me also preface this by saying that you shouldn't try to follow this code unless you already know how to program; it may seem totally overwhelming. My main point here is to reveal my creative process, not to teach the code; the rest of the book is for that. Feel free to run this sketch and make changes to the code to see what happens. If you get any really cool results, I hope you'll e-mail me some screenshots.

In considering the problem, I decided to think about trees some and to break the problem down into simple parts. In a moment of brilliant clarity, I concluded that trees have a trunk, branches, and leaves. OK, I know I can probably make a trunk (think straight vertical line). Branches could be a little harder, and leaves, well, probably too hard (actually too much work). Branches, I concluded, could be simplified as a thin trunk, splitting off to another series of thinner trunks, splitting off to another series of thinner trunks, and so on. Thus, it seemed branches could be made from a single branch machine, in a fractal progression (huh?). By fractal progression, I mean the branches are self-similar and repeat. Each branch's ending point is a starting point for more branches, with a decreasing scale. Thus, the tree thickness decreases from trunk to branch—which seemed pretty consistent with real trees. I decided to put the leaves on hold for now. I was ready for a little sketch of my branch plan (see Figure 2-2).

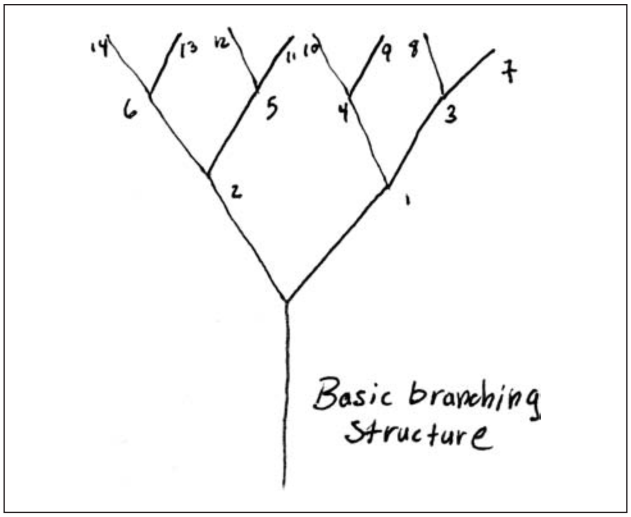


Figure 2-2. Initial schematic sketch of tree branching structure. The numbers represent the algorithmic branching order.

The drawing is a basic symmetrical tree structure in a purely schematic form. Between each branch is a node. I added some numbers for the nodes to think a little how the computer would sequentially draw the structure. So I was now ready for an algorithm that would help me write code to generate the tree form. Again, the tree trunk was just going to be a line, so I figured I'd come back to it. Here's the branching algorithm:

1. Draw two or more opposing diagonal branches from the top of the trunk.
2. At the ends of these branches, continue adding two or more opposing diagonal, slightly thinner branches.
3. Go back to step two until a specific limit is reached.

The algorithm seemed simple enough, and I noticed that it could possibly loop back on itself—or even call itself. We call such an approach in programming **recursion**, where a function calls itself. Recursion is a somewhat advanced concept, even though it is not that difficult to implement. Recursion will lead to an infinite loop (a bad thing) unless some explicit limit is set. For example, I could tell the program to run until one of the branches touched the top of the frame, or until there are 700 branches on the tree, and so on.

After a couple of well-placed slaps to my head, I finally got the basic branching code written (please note that this code fragment will not run in Processing yet):

```
void branch(Point2D.Float[] pts){
  int stemCount=2;
  if (counter2<branchLimit){
      for (int j=0; j<stemCount; j++){
        line(pts[counter2].x, pts[counter2].y, ➥
          pts[counter2].x+xg, pts[counter2].y-yg);
        pts[counter+1] = new Point2D.Float(pts[counter2].x+xg, ➥
              pts[counter2].y-yg);
                xg*=-1;
            counter++;
         }
        counter2++;
        branch(pts);
        }
    }
```

I don't want to spend too much time on the code here, as lots of coding is coming up in subsequent chapters. I'll just give a quick overview of how the function works. If you feel overwhelmed at any point, just skip the code parts. In Chapter 3, I'll begin with a very gentle introduction to the fundamentals of coding, so don't panic. What I'm really interested in illustrating is a creative process using code.

The branch function expects an array, of type `Point2D.Float`, as an argument (the stuff between the parentheses at the top of the function). An array just holds a bunch of values, as opposed to a variable, which only holds a single value. I chose to use Java's `Point2D.Float` class because it has built-in `x` and `y` properties and can handle real numbers (as opposed to only integers). The `stemCount` variable controls how many branches each subsequent branch gets. I hard-coded it to `2` just for now. The `for` loop draws the diagonal lines and updates the `pts` array with the new branch locations. The variable `xg` controls whether the branches lean left or right. By continuously multiplying `xg` by –1, it keeps toggling the branches from right to left—not terribly organic, but efficient. The counters keep track of stuff, and then I recursively call the same function, passing in the updated `pts` array. At this point, you are probably either like "Cool, that makes sense," or "Is it too late to return this book? This guy is nuts!" Well, I may be nuts, but I don't expect most of you to be able to follow this code. However, hearing some of these terms over and over will help with the "stick in your brain" part. What is really much more important and interesting is how to turn this cold schematic diagram into a tree (see Figure 2-3).

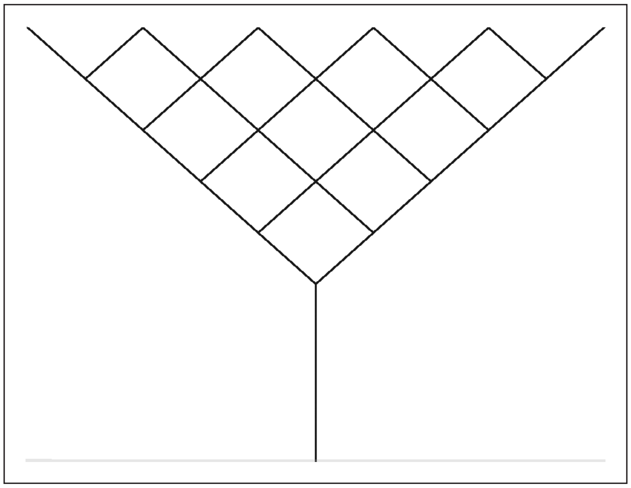


Figure 2-3. Initial code implementation of the branching algorithm

After being somewhat satisfied with the branch algorithm, I started playing with the code. The first thing I did was add some random functions that moved the branches in slightly different places. The random functions contribute to the happy mistakes. Sometimes, I'll put in values that are outside of a reasonable range just to see what happens. Already this started looking better (see Figure 2-4).

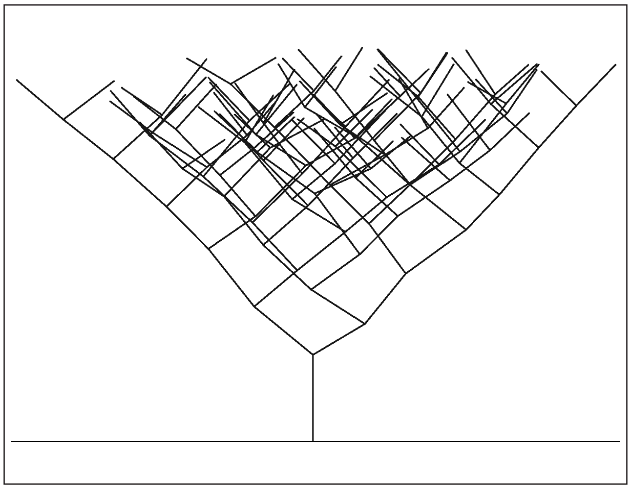


Figure 2-4. Randomness added to the branch structure

Next, I added thickness (using the `radius` variable) to give the tree some form. Beginning with a larger radius value—not really a radius, but more like thickness—for the trunk, I decremented the `radius` variable slowly each time the function ran, letting the branches slowly attenuate. This began looking a little more tree-like, as you can see in Figure 2-5.

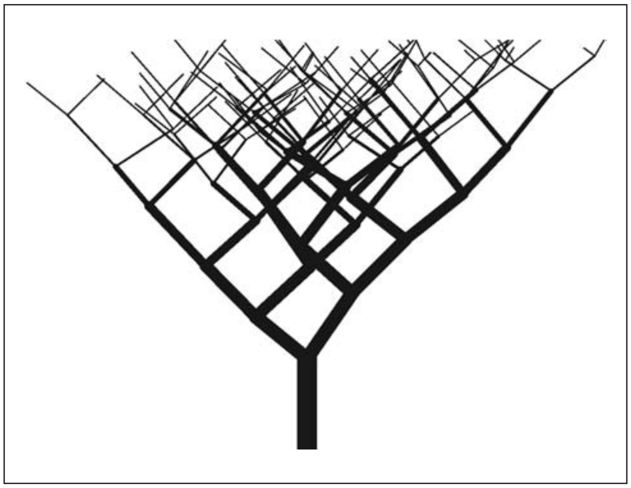


Figure 2-5. Stroke weight variation begins to create a more realistic-looking tree.

The overall movement was good, but the tree seemed too stiff, so I wrote a little redrawing function that added some waviness to the main trunk, and I set this up as a random value. Using randomness is really helpful for generating organic-esque type effects. One of the potential pitfalls of using code as a creative medium is its inherent precision, since it's ultimately based on math. In the real world, "stuff" (friction, gravity, our ineptitude, etc.) intervenes between intention and implementation—the happy accidents. Adding randomness is one (easy) way to introduce some happy chaos into an otherwise highly predictable process (see Figure 2-6). Normally when I use random values, I set a base number that I add to the random part. For example, if I were creating a random number of branches, and I used a random function, such as `random(12)`, I would get a number returned between 0 and 12. However, if I needed a minimum of two branches, I would change the expression to "`2+random(10)`"—this way, I would get a random number between 2 and 12. Processing has a handy random function that allows you to pass two arguments, as in `random(2, 12)`, which does the same thing.

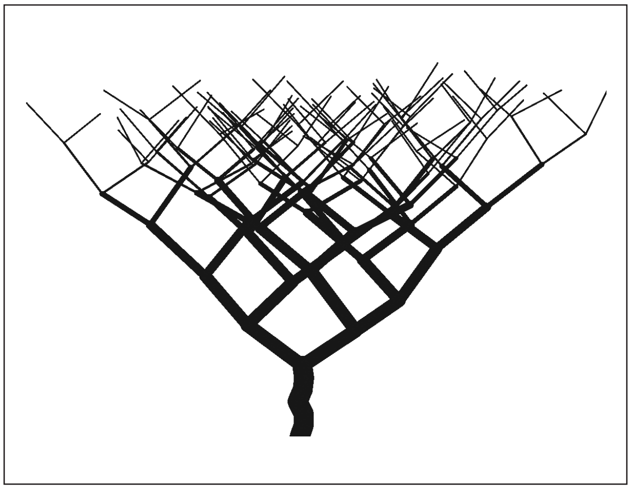


Figure 2-6. Waviness added to the main trunk

Things were beginning to look pretty good, but I still needed to deal with those darn leaves. Then, a really happy coding accident happened. As I began to play with the code, setting weird numbers in the functions, I found a way of generating leaves by really upping the number of iterations and using a couple of conditional statements (`if` and `else`) that would at specific times in the branch-creation cycle decrement or increment certain values. Then I added some color the same way, and finally used a similar wavy function for each branch, like I used on the trunk. Figure 2-7 shows a couple of finished trees.

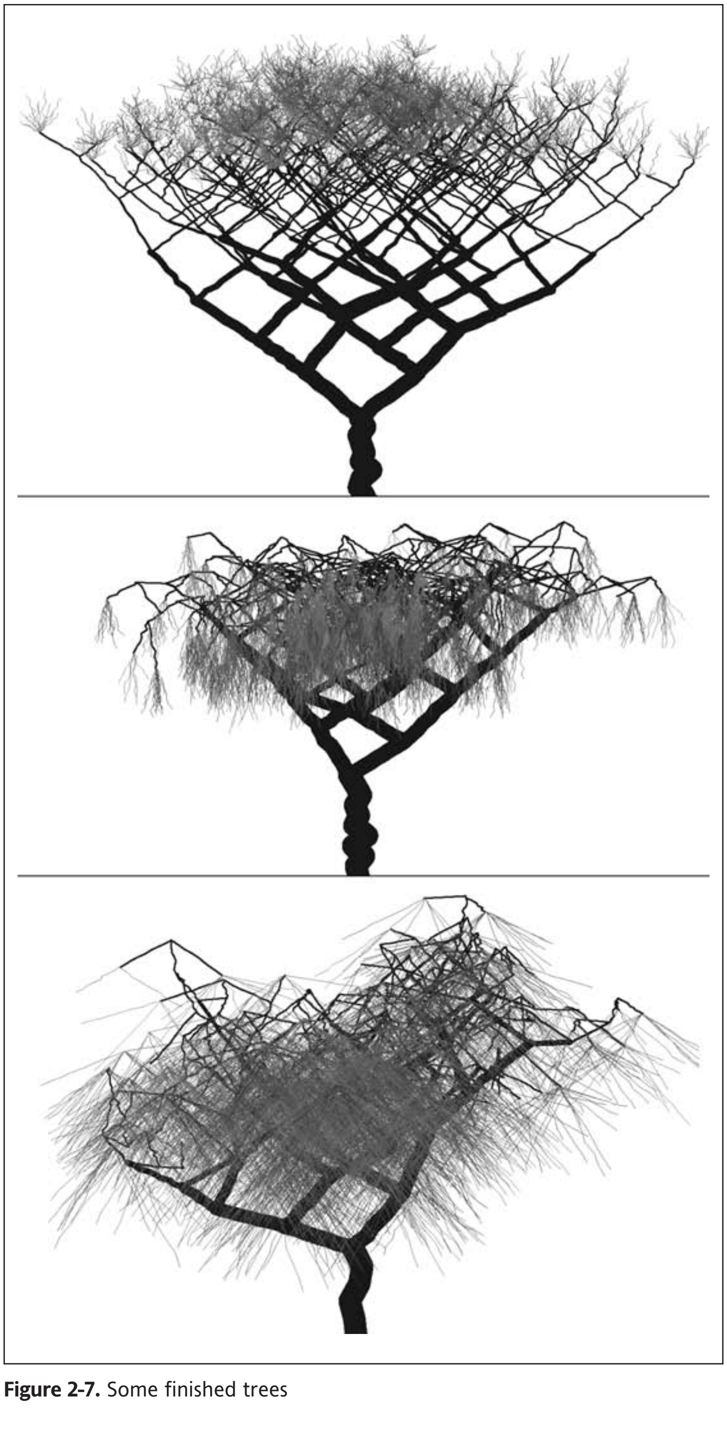


Figure 2-7. Some finished trees

Obviously, there are a lot of things I could still do with the tree code, but hopefully one of you will eventually take the code and improve upon it. I hope I was able to illustrate in this example how you can play with the code and even find stuff in the process of coding. Yes, there is some thinking involved, but there is also a lot of play and discovery. I've included the entire tree program following, with code comments for your hacking pleasure. If you want to try running this, launch Processing and type (or paste) the code into the Processing text editor.

```
/*
 algorithmic tree sketch
 Ira greenberg, August, 2005
*/

/* import Handy Java class with
   public x and y float props */
import java.awt.geom.Point2D;

// declare/define global variables
int counter;
int counter2;
float xg = 58;
float yg = 46;
int trunkSegments = int(random(7))+5;
int nodeLimit = 20000;
Point2D.Float[]pts = new Point2D.Float[nodeLimit];
int branchLimit = 620;
float trunkLength = int(random(50))+130;
float[]lean2 = new float[trunkSegments+1];
float radius = 26;

// initialize sketch
void setup(){
  size(900, 600);
  background(255);
  stroke(30, 10, 5);

  // create tree turnk
  trunk();
}

void trunk(){
  //draw trunk
  for (int i=0; i<trunkSegments; i++){
    float lean = myRand(22);
    strokeWeight(radius+12);
    line(width/2+lean2[i], height-(trunkLength/trunkSegments)*i, ➥
      width/2+lean, height-(trunkLength/trunkSegments)*(i+1));
    lean2[i+1] = lean;
  }
  // set inital branch point from top of trunk
  pts[0] = new Point2D.Float(width/2+lean2[trunkSegments], ➥
    height-trunkLength);

  //create branches
  branch(pts);
}

//main function that draws branches and leaves
void branch(Point2D.Float[]pts){
  int stemCount=2;

  // global variable branchLimit controls complexity of tree
  if (counter2<branchLimit){
      //set branch thickness
      strokeWeight(radius);

     // some conditionals change branches as
     // they get further away from the trunk
     if(counter2<200){
       yg-=random(.354);
       xg-=random(.625);

       if (radius>2) {
         radius*=.931;
       }

     } else if(counter2>=200){
         // at top of tree branches get thinner and more numerous
          stemCount = 2+(int)(Math.random()*15);
          // leave color
          stroke(random(60), 50+random(90), random(20), 230);

          yg-=myRand(.65);
          xg+=random(1.5);

          radius*=(.91);
      }
      for (int j=0; j<stemCount; j++){
        // randomize branch positions
        float xx= myRand(30);
        float yy= myRand(40);

        /* commented out line below generates straight branches
        line(pts[counter2].x, pts[counter2].y, pts[counter2].x+xg+xx,➥
          pts[counter2].y-yg+yy);
        */
        // generates organic looking branches
        orgLine(pts[counter2].x, pts[counter2].y, ➥
          pts[counter2].x+xg+xx, pts[counter2].y-yg+yy);

        /* fill up pts array to be passed back
           recursively to branch function */
         pts[counter+1] = new Point2D.Float(pts[counter2].x+xg+xx, ➥
           pts[counter2].y-yg+yy);

        // alternate branches left and right
        xg*=-1;

        // keep track of nodes
        counter++;
      }

    // keeps track of branches
    counter2++;

    //recursive call
    branch(pts);
    }
 }

// generates organic-looking branches
void  orgLine (float x1, float y1, float x2, float y2){

  int sections = 8;

  float xd = x2-x1;
  float yd = y2-y1;

  float twist;
  float[]twist2 = new float[sections+1];

 for (int i =0; i<sections; i++){
   twist = myRand(5);
   line(x1+xd/sections*i+twist2[i], y1+yd/sections*i, ➥
     x1+xd/sections*(i+1)+twist, y1+yd/sections*(i+1));
   twist2[i+1] = twist;
 }
}

//generate a random val between (-n, n)
float myRand(float val){
  return random(val)+random(-val);
}
```

Summary

In this chapter, I introduced creative coding as an approach to programming that integrates both analytical and creative processes. Using my friend Mark as an example, I described some of the stereotypes and biases that can make this cross-brain integration so challenging—a problem often (and sadly) reinforced in the classroom, which helped motivate me to teach creative coding and eventually write this book.

I gave a very top-view description about how Processing works, built on top of the Java programming language, and I described some ways of structuring your code, including procedural and object-oriented approaches—both utilized within Processing. I tried to demystify the term "algorithm" and show the relationship between an algorithm and its implementation in code. I also discussed the challenge of learning to think like a "dumb" machine.

Finally, I covered the important role of happy accidents in the creative coding process, and I illustrated the point with an algorithmic tree example. In the next chapter, you'll begin to explore the actual nuts and bolts of coding.