Processing...

Processing relates software concepts to principles of visual form, motion, and interaction. It integrates a programming language, development environment, and teaching methodology into a unified system. Processing was created to teach fundamentals of computer programming within a visual context, to serve as a software sketchbook, and to be used as a production tool. Students, artists, design professionals, and researchers use it for learning, prototyping, and production.

The Processing language is a text programming language specifically designed to generate and modify images. Processing strives to achieve a balance between clarity and advanced features. Beginners can write their own programs after only a few minutes of instruction, but more advanced users can employ and write libraries with additional functions. The system facilitates teaching many computer graphics and interaction techniques including vector/raster drawing, image processing, color models, mouse and keyboard events, network communication, and object-oriented programming. Libraries easily extend Processing's ability to generate sound, send/receive data in diverse formats, and to import/export 2D and 3D file formats.

Software

A group of beliefs about the software medium set the conceptual foundation for Processing and inform decisions related to designing the software and environment.

Software is a unique medium with unique qualities
Concepts and emotions that are not possible to express in other media may be expressed in this medium. Software requires its own terminology and discourse and should not be evaluated in relation to prior media such as film, photography, and painting. History shows that technologies such as oil paint, cameras, and film have changed artistic practice and discourse, and while we do not claim that new technologies improve art, we do feel they enable different forms of communication and expression. Software holds a unique position among artistic media because of its ability to produce dynamic forms, process gestures, define behavior, simulate natural systems, and integrate other media including sound, image, and text.

Every programming language is a distinct material
As with any medium, different materials are appropriate for different tasks. When designing a chair, a designer decides to use steel, wood or other materials based on the intended use and on personal ideas and tastes. This scenario transfers to writing software. The abstract animator and programmer Larry Cuba describes his experience this way: "Each of my films has been made on a different system using a different

programming language. A programming language gives you the power to express some ideas, while limiting your abilities to express others."[1] There are many programming languages available from which to choose, and some are more appropriate than others depending on the project goals. The Processing language utilizes a common computer programming syntax that makes it easy for people to extend the knowledge gained through its use to many diverse programming languages.

Sketching is necessary for the development of ideas
It is necessary to sketch in a medium related to the final medium so the sketch can approximate the finished product. Painters may construct elaborate drawings and sketches before executing the final work. Architects traditionally work first in cardboard and wood to better understand their forms in space. Musicians often work with a piano before scoring a more complex composition. To sketch electronic media, it's important to work with electronic materials. Just as each programming language is a distinct material, some are better for sketching than others, and artists working in software need environments for working through their ideas before writing final code. Processing is built to act as a software sketchbook, making it easy to explore and refine many different ideas within a short period of time.

Programming is not just for engineers
Many people think programming is only for people who are good at math and other technical disciplines. One reason programming remains within the domain of this type of personality is that the technically minded people usually create programming languages. It is possible to create different kinds of programming languages and environments that engage people with visual and spatial minds. Alternative languages such as Processing extend the programming space to people who think differently. An early alternative language was Logo, designed in the late 1960s by Seymour Papert as a language concept for children. Logo made it possible for children to program many different media, including a robotic turtle and graphic images on screen. A more contemporary example is the Max programming environment developed by Miller Puckette in the 1980s. Max is different from typical languages; its programs are created by connecting boxes that represent the program code, rather than lines of text. It has generated enthusiasm from thousands of musicians and visual artists who use it as a base for creating audio and visual software. The same way graphical user interfaces opened up computing for millions of people, alternative programming environments will continue to enable new generations of artists and designers to work directly with software. We hope Processing will encourage many artists and designers to tackle software and that it will stimulate interest in other programming environments built for the arts.

Literacy

Processing does not present a radical departure from the current culture of programming. It repositions programming in a way that is accessible to people who are interested in programming but who may be intimidated by or uninterested in the type taught in computer science departments. The computer originated as a tool for fast calculations and has evolved into a medium for expression.

The idea of general software literacy has been discussed since the early 1970s. In 1974, Ted Nelson wrote about the minicomputers of the time in *Computer Lib / Dream Machines*. He explained "the more you know about computers . . . the better your imagination can flow between the technicalities, can slide the parts together, can discern the shapes of what you would have these things do."[2] In his book, Nelson discusses potential futures for the computer as a media tool and clearly outlines ideas for hypertexts (linked text, which set the foundation for the Web) and hypergrams (interactive drawings). Developments at Xerox PARC led to the Dynabook, a prototype for today's personal computers. The Dynabook vision included more than hardware. A programming language was written to enable, for example, children to write storytelling and drawing programs and musicians to write composition programs. In this vision there was no distinction between a computer user and a programmer.

Thirty years after these optimistic ideas, we find ourselves in a different place. A technical and cultural revolution did occur through the introduction of the personal computer and the Internet to a wider audience, but people are overwhelmingly using the software tools created by professional programmers rather than making their own. This situation is described clearly by John Maeda in his book *Creative Code*: "To use a tool on a computer, you need do little more than point and click; to create a tool, you must understand the arcane art of computer programming."[3] The negative aspects of this situation are the constraints imposed by software tools. As a result of being easy to use, these tools obscure some of the computer's potential. To fully explore the computer as an artistic material, it's important to understand this "arcane art of computer programming."

Processing strives to make it possible and advantageous for people within the visual arts to learn how to build their own tools—to become software literate. Alan Kay, a pioneer at Xerox PARC and Apple, explains what literacy means in relation to software:

> The ability to "read" a medium means you can access materials and tools created by others. The ability to "write" in a medium means you can generate materials and tools for others. You must have both to be literate. In print writing, the tools you generate are rhetorical; they demonstrate and convince. In computer writing, the tools you generate are processes; they simulate and decide.[4]

Making processes that simulate and decide requires programming.

Open

The open source software movement is having a major impact on our culture and economy through initiatives such as Linux, but it is having a smaller influence on the culture surrounding software for the arts. There are scattered small projects, but companies such as Adobe and Microsoft dominate software production and therefore control the future of software tools used within the arts. As a group, artists and designers traditionally lack the technical skills to support independent software initiatives. Processing strives to apply the spirit of open source software innovation to the domain of the arts. We want to provide an alternative to available proprietary software and to improve the skills of the arts community, thereby stimulating interest in related initiatives. We want to make Processing easy to extend and adapt and to make it available to as many people as possible.

Processing probably would not exist without its ties to open source software. Using existing open source projects as guidance, and for important software components, has allowed the project to develop in a smaller amount of time and without a large team of programmers. Individuals are more likely to donate their time to an open source project, and therefore the software evolves without a budget. These factors allow the software to be distributed without cost, which enables access to people who cannot afford the high prices of commercial software. The Processing source code allows people to learn from its construction and by extending it with their own code.

People are encouraged to publish the code for programs they've written in Processing. The same way the "view source" function in Web browsers encouraged the rapid proliferation of website-creation skills, access to others' Processing code enables members of the community to learn from each other so that the skills of the community increase as a whole. A good example involves writing software for tracking objects in a video image, thus allowing people to interact directly with the software through their bodies, rather than through a mouse or keyboard. The original submitted code worked well but was limited to tracking only the brightest object in the frame. Karsten Schmidt (a k a toxi), a more experienced programmer, used this code as a foundation for writing more general code that could track multiple colored objects at the same time. Using this improved tracking code as infrastructure enabled Laura Hernandez Andrade, a graduate student at UCLA, to build *Talking Colors*, an interactive installation that superimposes emotive text about the colors people are wearing on top of their projected image. Sharing and improving code allows people to learn from one another and to build projects that would be too complex to accomplish without assistance.

Education

Processing makes it possible to introduce software concepts in the context of the arts and also to open arts concepts to a more technical audience. Because the Processing syntax is derived from widely used programming languages, it's a good base for future learning. Skills learned with Processing enable people to learn other programming

languages suitable for different contexts including Web authoring, networking, electronics, and computer graphics.

There are many established curricula for computer science, but by comparison there have been very few classes that strive to integrate media arts knowledge with core concepts of computation. Using classes initiated by John Maeda as a model, hybrid courses based on Processing are being created. Processing has proved useful for short workshops ranging from one day to a few weeks. Because the environment is so minimal, students are able to begin programming after only a few minutes of instruction. The Processing syntax, similar to other common languages, is already familiar to many people, and so students with more experience can begin writing advanced syntax almost immediately.

In a one-week workshop at Hongik University in Seoul during the summer of 2003, the students were a mix of design and computer science majors, and both groups worked toward synthesis. Some of the work produced was more visually sophisticated and some more technically advanced, but it was all evaluated with the same criteria. Students like Soo-jeong Lee entered the workshop without any previous programming experience; while she found the material challenging, she was able to learn the basic principles and apply them to her vision. During critiques, her strong visual skills set an example for the students from more technical backgrounds. Students such as Tai-kyung Kim from the computer science department quickly understood how to use the Processing software, but he was encouraged by the visuals in other students' work to increase his aesthetic sensibility. His work with kinetic typography is a good example of a synthesis between his technical skills and emerging design sensitivity.

Processing is also used to teach longer introductory classes for undergraduates and for topical graduate-level classes. It has been used at small art schools, private colleges, and public universities. At UCLA, for example, it is used to teach a foundation class in digital media to second-year undergraduates and has been introduced to the graduate students as a platform for explorations into more advanced domains. In the undergraduate Introduction to Interactivity class, students read and discuss the topic of interaction and make many examples of interactive systems using the Processing language. Each week new topics such as kinetic art and the role of fantasy in video games are introduced. The students learn new programming skills, and they produce an example of work addressing a topic. For one of their projects, the students read Sherry Turkle's "Video Games and Computer Holding Power"[5] and were given the assignment to write a short game or event exploring their personal desire for escape or transformation. Leon Hong created an elegant flying simulation in which the player floats above a body of water and moves toward a distant island. Muskan Srivastava wrote a game in which the objective was to consume an entire table of desserts within ten seconds.

Teaching basic programming techniques while simultaneously introducing basic theory allows the students to explore their ideas directly and to develop a deep understanding and intuition about interactivity and digital media. In the graduate-level Interactive Environments course at UCLA, Processing is used as a platform for experimentation with computer vision. Using sample code, each student has one week to develop software that uses the body as an input via images from a video camera.

Zai Chang developed a provocative installation called White Noise where participants' bodies are projected as a dense series of colored particles. The shadow of each person is displayed with a different color, and when they overlap, the particles exchange, thus appearing to transfer matter and infect each other with their unique essence. Reading information from a camera is an extremely simple action within the Processing environment, and this facility fosters quick and direct exploration within courses that might otherwise require weeks of programming tutorials to lead up to a similar project.

Network

Processing takes advantage of the strengths of Web-based communities, and this has allowed the project to grow in unexpected ways. Thousands of students, educators, and practitioners across five continents are involved in using the software. The project website serves as the communication hub, but contributors are found remotely in cities around the world. Typical Web applications such as bulletin boards host discussions between people in remote locations about features, bugs, and related events.

Processing programs are easily exported to the Web, which supports networked collaboration and individuals sharing their work. Many talented people have been learning rapidly and publishing their work, thus inspiring others. Websites such as Jared Tarbell's *Complexification.net* and Robert Hodgin's *Flight404.com* present explorations into form, motion, and interaction created in Processing. Tarbell creates images from known algorithms such as Henon Phase diagrams and invents his own algorithms for image creation, such as those from *Substrate*, which are reminiscent of urban patterns (p. 157). On sharing his code from his website, Tarbell writes, "Opening one's code is a beneficial practice for both the programmer and the community. I appreciate modifications and extensions of these algorithms."[6] Hodgin is a self-trained programmer who uses Processing to explore the software medium. It has allowed him to move deeper into the topic of simulating natural forms and motion than he could in other programming environments, while still providing the ability to upload his software to the Internet. His highly trafficked website documents these explorations by displaying the running software as well as providing supplemental text, images, and movies. Websites such as those developed by Jared and Robert are popular destinations for younger artists and designers and other interested individuals. By publishing their work on the Web in this manner they gain recognition within the community.

Many classes taught using Processing publish the complete curriculum on the Web, and students publish their software assignments and source code from which others can learn. The websites for Daniel Shiffman's classes at New York University, for example, include online tutorials and links to the students' work. The tutorials for his Procedural Painting course cover topics including modular programming, image processing, and 3D graphics by combining text with running software examples. Each student maintains a web page containing all of their software and source code created for the class. These pages provide a straightforward way to review performance and make it easy for members of the class to access each others's work.

The Processing website, *www.processing.org*, is a place for people to discuss their projects and share advice. The Processing Discourse section of the website, an online bulletin board, has thousands of members, with a subset actively commenting on each others' work and helping with technical questions. For example, a recent post focused on a problem with code to simulate springs. Over the course of a few days, messages were posted discussing the details of Euler integration in comparison to the Runge-Kutta method. While this may sound like an arcane discussion, the differences between the two methods can be the reason a project works well or fails. This thread and many others like it are becoming concise Internet resources for students interested in detailed topics.

Context

The Processing approach to programming blends with established methods. The core language and additional libraries make use of Java, which also has elements identical to the C programming language. This heritage allows Processing to make use of decades of programming language refinements and makes it understandable to many people who are already familiar with writing software.

Processing is unique in its emphasis and in the tactical decisions it embodies with respect to its context within design and the arts. Processing makes it easy to write software for drawing, animation, and reacting to the environment, and programs are easily extended to integrate with additional media types including audio, video, and electronics. Modified versions of the Processing environment have been built by community members to enable programs to run on mobile phones (p. 617) and to program microcontrollers (p. 633).

The network of people and schools using the software continues to grow. In the five years since the origin on the idea for the software, it has evolved organically through presentations, workshops, classes, and discussions around the globe. We plan to continually improve the software and foster its growth, with the hope that the practice of programming will reveal its potential as the foundation for a more dynamic media.

Notes

1. Larry Cuba, "Calculated Movements," in *Prix Ars Electronica Edition '87: Meisterwerke der Computerkunst* (H. S. Sauer, 1987), p. 111.
2. Theodore Nelson, "Computer Lib / Dream Machines," in *The New Media Reader*, edited by Noah Wardrip-Fruin and Nick Montfort (MIT Press, 2003), p. 306.
3. John Maeda, *Creative Code* (Thames & Hudson, 2004), p. 113.
4. Alan Kay, "User Interface: A Personal View," in *The Art of Human-Computer Interface Design*, edited by Brenda Laurel (Addison-Wesley, 1989), p. 193.
5. Chapter 2 in Sherry Turkle, *The Second Self: Computers and the Human Spirit* (Simon & Schuster, 1984), pp. 64–92.
6. Jared Tarbell, Complexification.net (2004), http://www.complexification.net/medium.html.

Development 1: Sketching, Techniques

This unit discusses the idea of sketching code and the iterative development process.

There are similarities between learning a programming language and learning a new spoken language. Initially, one learns basic elements of a spoken language—such as simple words and grammar rules—and mimics short phrases. Learning to communicate ideas and express emotions within the language takes more time. Similarly, the first step in computer programming is to understand the basic elements such as comments, variables, and functions. The next step is to learn to read and modify simple example programs. Later, one begins to write programs from scratch. The most interesting and difficult stage of learning to program comes later, as one gains the ability to put the language elements together to express ideas about form, motion, and behavior. Like learning a foreign language, becoming fluent in a programming language can take years.

Sketching software

Sketching ranges from informal exploration to focused refinement. It is used to create many variations within a short period of time, or to develop a specific idea. Sketching forces the definition of vague ideas by making them physical. Sketches are powerful communication tools—they can get ideas out of one's head and into a format that can be better understood by others.

It is important to work out ideas on paper before investing time in writing code. Paper and pencil allow for fast iteration in the early stages of a project. The most important aspect of programming is figuring out what will be created and how it will function, so working out these ideas away from a computer keeps the focus on an idea, rather than on its implementation.

A good paper sketch for software will include a series of images that demonstrate how the narrative structure of the piece works, much like an animator's storyboard. In addition to images that will appear on screen, sketches often contain diagrams of the program's flow, data elements, and notation for showing how forms will move and interact. Programs can also be planned using combinations of image mock-ups, formal schematics, and text descriptions.

After refined ideas reach a point where working on paper is no longer useful, code can continue the development. The first step in creating code is a continuation of the sketching process. Write short pieces of code independently before worrying about the structure of the larger program. Writing small, focused programs makes a developer better at writing code when it matters most: when working on a more refined implementation.

Processing programs are called sketches to emphasize this method of working. The Processing sketchbook is a way of storing and organizing programs. Code sketches can be reviewed and developed incrementally like drawings in a paper sketchbook. Ideas that flash by while walking or just after waking up can be quickly made into code and stored for future use. The Processing environment encourages this type of writing because one need only press the New button in the toolbar to start a new sketch.

Some programming languages encourage a sketching approach, and others make it difficult. Scripting languages, such as Perl or Python, are designed to encourage rapid development at the expense of running speed and control. Processing is not a scripting language, but is designed to "feel" like a scripting language while providing the same capabilities as a more complete language like Java. This topic is discussed in more depth in Appendix F (p. 679).

Programming techniques

There are as many ways to write programs as there are people who write software. Some common strategies for creating programs include modification, augmentation, collage, and writing code from scratch. People learning to code often expect most programs to be written from scratch, but that's rarely the case, particularly for the style of work built with Processing. Learning to read and modify code helps programmers increase their skills. Even advanced programmers work from others' examples when learning new techniques.

Modification
Changing the values of variables in existing programs is a good way to explore code. Programs can be modified by trial and error or more deliberately. One way to start understanding a program is to change slightly the value of one variable and then run the code to see the result. If there is no obvious difference, change the value again. Making the correlation between a variable and a change in the way a program runs is a good first step to understanding how it works. Disabling lines of code by placing them inside a `/*` and `*/` comment block (called "commenting out") is another way to decrypt a program. A little understanding of the way a program is structured can facilitate logical guesses about what different lines of code are doing. These modification techniques aid in learning new skills or parsing an example. Making small changes to an existing program encourages exploration and getting a feel for the code.

Augmentation
Augmentation uses existing code as a base for further exploration. It is similar to but more ambitious than modification. Generic program examples can serve as a foundation for longer, more specific programs. An example that draws a Bézier curve can be used as a base for drawing a series of curves (as shown in Shape 2, p. 69). A program that displays a photograph can form the basis of a photo montage application. Sample programs

provide a concise reminder of syntax. The spartan programs presented in this book provide a broad base for making enhancements.

Collage
The collage technique involves cutting and pasting elements of different programs together to create a new program. It's analogous to creating music by sampling or making a visual collage from newspaper and magazine clippings. In order to avoid errors, combine code carefully by copying a few lines of code at a time and running the program to make sure it's always working. Copying large portions of code can introduce a number of simultaneous errors. Mindlessly copying and pasting code can create "Frankenstein" code that's difficult to debug. As an individual's knowledge of programming increases, using this technique becomes easier, and common problems can be avoided, such as adding multiple copies of the same method, like `draw()` or `setup()`, to the code.

Coding from scratch
Rarely do programmers write a complete program entirely from scratch. At the minimum, most people start with a template. A template is an outline with code infrastructure common to many programs. Sometimes it's not possible to find a related example or appropriate template and it's necessary to start with a blank page. In this case, comments are a great way to start building a program. Comments can be used to build an outline of the program's intention, logic, and flow. After this structure has been defined, lines of code can be slowly added and run in an attempt to realize those decisions.

Regardless of the technique used for programming, writing a few lines of code or making only a few changes at a time is a good tactic. Entering many lines of untested code before running the program increases the potential for multiple errors. The more errors in a program, the more difficult they become to find. Running a growing program piece by piece reduces the chance for multiple errors. As your comfort with code and your skills increase, it becomes possible to make more modifications between tests.

Development 2: Iteration, Debugging

This unit discusses the iterative software development process and the activity of debugging code.

The programs included up to this point in the book have been short. Programs of this length can be written without much forethought, but planning becomes important when writing longer programs. The extent of the planning will be up to the programmer, but one aspect of programming is always the same: large, complex programs must be divided into series of short, simpler programs. Learning how to divide programs into manageable parts takes time and experience. As the scope of a program grows, the number of decisions involved in writing it multiplies. Making changes to a program, evaluating the result, and then making additional changes is an iterative process. Like a project in any medium, software improves through many cycles of changes and evaluations.

Longer programs also present a higher likelihood of mistakes. The flow of logic and data becomes less obvious in a larger program, and the errors—known as bugs—introduced are more subtle. Learning to track down and fix errors is an important skill in writing software.

Iteration

There are many different models for software development, but they all contain elements of analysis, synthesis, and evaluation. A continuous cycle of synthesis and evaluation is the core of the iterative process. Every project demands variations on each of these stages, but the purpose of each remains consistent. A more detailed description of each stage illuminates how they interact:

Analysis
Analysis leads to an understanding of the software—its function, audience, and purpose. This stage can involve months of research or mere seconds of consideration, resulting in a proposal, project description, or other means of communicating the project to others.

Synthesis
The goals and concepts that emerge from the analysis are realized through synthesis. Early steps in synthesis often include paper sketches, followed by software sketches, and then refinement of the finished software. The results of this stage are evaluated, edited, and augmented with additional synthesis until the software is finished.

Evaluation
The results of the synthesis phase are evaluated in relation to the analysis to determine what remains to be done. Is the project complete or is another round of synthesis needed? What improvements can be made? What is working and what needs to be fixed? Depending on the nature of the project, the evaluation sometimes returns to analysis and the goals of the project are modified.

Programs change quickly, and sometimes the programmer prefers an earlier version. Save multiple versions of the sketch while working so it's always possible to return to a previous iteration of the code. Simply select "Save As" from the File menu to save a new version of the program with a different name. The "Archive Sketch" option from the Tools menu saves the code and all additional media for the current sketch inside a ZIP archive with the name of the current sketch and the date. Saving multiple versions of a sketch ensures that older, working examples of the code remain intact.

Debugging

When a person first starts programming, errors (bugs) occur frequently; learning how to find and fix (debug) them is an important part of learning to program. In *The Practice of Programming*, the authors explain: "Good programmers know that they spend as much time debugging as writing so they try to learn from their mistakes. Every bug you find can teach you how to prevent a similar bug from happening again or to recognize it if it does." [1]

Some bugs reveal themselves when the Run button is pressed, as they prevent the program from starting. Other bugs appear while the program is running, causing the program to stop. The message area (p. 8) turns red and reveals a summary of the problem. Sometimes the bug message text is too long to fit in this area, but the full message always appears in the console. The Processing environment always tries to highlight the line where the bug occurs, but since the bug may be the result of something that happened earlier in the program, the error does not always appear on the highlighted line. The highlighted line is usually related to the error, but perhaps not in an obvious way.

Not all bugs stop a program from running. Errors in logic or problems with equations are sometimes more difficult to find because they don't stop the program.

Fixing bugs is one of the more difficult and less satisfying aspects of programming. Sometimes they are obvious and quick to fix, but sometimes it can take hours. Finding a bug is like solving a mystery. It's necessary to search through the code to find clues in pursuit of the culprit. Try the following:

Scrutinize the newest code
If the program is constructed step by step, the bug is often in the newest code or is linked to it. Check these areas for bugs first.

Check related code
Sometimes a bug may linger within a program for a long time because the line containing the bug is not run. When code is introduced that runs a line with a bug, or when the value of a variable changes so that code within a previously unused `if` or `for` structure is run, the bug will reveal itself.

Display output
Displaying the data produced by a program while it's running can expose problems and lead to a better understanding of the code in general. The `println()` function can be used to display data as text to the console. This technique can answer questions about the status of a variable and can be used to check whether a specific line or block of code is running. The data can also be represented as positions or colors in the program's display.

Isolate the problem
It's often difficult to find a bug within a large program. If possible, try to reduce the problem to its essence. Is it possible to reproduce the bug by running only a few lines of code or a much simpler program?

Learn from previous bugs
All programmers—new and experienced—inadvertently introduce bugs into their code. The hardest time to find a particular bug is usually during its first occurrence. Learn from previous mistakes to avoid the same bug in the future.

Take a break
Sometimes the best way to fix a bug is to take a break. After hours of programming, the perspective gained from a diversion or rest can bring clarity.

As with all software, there are bugs in Processing, and some are added and removed with each release of the software. For the most current information about bugs in the Processing software, read the Frequently Asked Questions (FAQ), accessible from the Help menu. A complete list can be found at http://dev.processing.org/bugs. This website can be searched for known bugs and used to report new ones.

Notes

1. Brian W. Kernighan and Bob Pike, *The Practice of Programming* (Addison-Wesley, 1999), p. 117.