Learn to program while creating art and having fun

Processing

Creative Coding and Generative Art in Processing 2

Ira Greenberg
Dianna Xu
Deepak Kumar

Processing

Creative Coding and Generative Art in Processing 2

Ira Greenberg, Dianna Xu, Deepak Kumar

PROCESSING

Copyright © 2013 by **Ira Greenberg, Dianna Xu, Deepak Kumar**

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

ISBN 978-1-4302-4464-6

ISBN 978-1-4302-4465-3 (eBook)

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logos, or image we use the names, logos, or images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please e-mail rights@apress.com or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales. APress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

Any source code or other supplementary materials referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

The front cover image was generated by the Processing sketch noise.pde found in Chapter 9.

Credits

President and Publisher:
Paul Manning

Lead Editor:
Ben Renow-Clarke

Technical Reviewer:
Ryan Rusnak

Editorial Board:
Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Louise Corrigan, Morgan Ertel, Jonathan Gennick, Jonathan Hassell, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Gwenan Spearing, Matt Wade, Tom Welsh

Coordinating Editor:
Anamika Panchoo

Copy Editors:
Michele Bowman, Linda Seifert

Compositor:
SPi Global

Indexer:
SPi Global

Artist:
SPi Global

Cover Image Artist:
Corné van Dooren

Cover Designer:
Anna Ishchenko

To Robin, Ian, and Sophie, and to Hilary, Jerry, and Eric
—Ira Greenberg

To Marcus and Kira
—Dianna Xu

To Debika and Sameer
—Deepak Kumar

Contents at a Glance

Foreword xiii
About the Authors xv
About the Technical Reviewer xvii
About the Cover Image Artist xix
Acknowledgments xxi
Introduction xxiii

Chapter 1: Diving into the Shallow End 1

Chapter 2: Art by Numbers 33

Chapter 3: Processing Boot Camp 65

Chapter 4: Creating Across Time and Curved Space 107

Chapter 5: Expressive Power of Data 149

Chapter 6: Organizing Chaos 187

Chapter 7: Creative Abstraction 233

Chapter 8: Drawing with Recursion 277

Chapter 9: Painting with Bits 311

Chapter 10: Expressive Imaging 369

Chapter 11: Spreading Your Creative Coding Wings 413

Index 437

Contents

Foreword ... xiii
About the Authors ... xv
About the Technical Reviewer ... xvii
About the Cover Image Artist ... xix
Acknowledgments ... xxi
Introduction ... xxiii

Chapter 1: Diving into the Shallow End ... 1

Programming vs. Computer Science ... 2
Art + Science = Creative Coding ... 3
MIT Media Lab ... 4
What Is Processing? ... 4
- Bits and Bytes ... 4
- Mnemonics ... 5
- Java ... 7
- Processing ... 9

Quick Tour ... 12
Processing Menu System ... 16
- Edit Menu ... 17
- Sketch Menu ... 19
- Tools Menu ... 23
- Help Menu ... 27
- Additional Menus ... 29

Summary ... 32

Chapter 2: Art by Numbers ... 33

Algorithms ... 33
- Pseudocode Example ... 34
- Generative Algorithm ... 37

Drawing with Code ... 39
Primitives ... 42
- Code Comments ... 43
- Coordinate Systems ... 44

Algorithmic Face 48
Primitive Variables 50
Face Implementation 53
Quick Math Refresher 58
Summary 63

Chapter 3: Processing Boot Camp 65

Functions 65
Reimplementing rect() 66
Local Variables and Scope 69
Adding Some Logic 72
Switch and Ternary 75
Switch Statement 75
Moving Beyond the Primitives 76
Yet Another Rectangle 77
Expanding the API 81
Polygon Implementation 83
Improved Polygon 85
Perfected Polygon 88
Having Some Polygonal Fun 93
Polygonal Wallpaper 94
Pushing and Popping the Matrix 95
Star Mandala Table 97
Summary 105

Chapter 4: Creating Across Time and Curved Space 107

Keep It Local 107
Examining the Variables 110
Thinking About Memory 110
Returning Value 111
Prime Time 112
Making Things Move 115
Moving and Rotating 117
Adding Simple Collision 118
Introducing Curves 124
Processing's Curve Functions 126
Controlling Curves 128
Summary 147

Chapter 5: Expressive Power of Data 149

Arrays 150
Indexing, Size, and Loops 153
Example: A Simple Bar Graph 155
Array Operations 156
Printing 157
Min, Max, and Sorting 158
Example: A Better, Interactive curveVertex 159
Primitive and Reference Types 161
Arrays As Parameters 164
Time Series Visualization 166
Simple Data Modeling 172
Data Visualization 174
Mapping Numbers 175
Basic Plots 177
Algorithms and Issues of Space and Time 181
Summary 185
References 185

Chapter 6: Organizing Chaos 187

Objects: Attributes and Behavior 188
Classes: Object Factories 189
Object-Oriented Programming in Processing 189
Customizing Instances 194
A Useful Keyword: this 196
Tabs: Organizing Code 198
Defining Additional Behaviors: Motion 201
OOP and Encapsulation in Processing 206
PVector Class in Processing 207
Ball in a Box 210
Composition: Has-a Relationships 214
Simulated Physics: Verlet Motion 215
Two Balls and a Stick: An Application of Verlet Integration 218
Inheritance: Is-a Relationships 223
Interfaces Are Doable 229
Summary 232

Chapter 7: Creative Abstraction 233
Strings 234
String Methods 235
Working with Strings 236
Text Visualization: Creating Word Clouds 238
Acquiring the Data 238
Parsing the Data 239
Filtering the Data 240
Mining the Data: Sorting 250
Learning to Sort 252
Choosing a Visual Representation 256
Making the Visualization Interactive 273
Creative Data Visualization 275
Summary 276
References 276

Chapter 8: Drawing with Recursion 277
Recursive Functions 277
Factorial 278
Recursive Circles 279
Recursive Squares 283
Recursive Tree 285
Lindenmayer Systems 292
Grammar 292
Rendering 293
Koch Snowflake 294
Quadratic Flake 296
Heighway Dragon 297
Plants 299
Implementation 301
Advanced Visualization: Treemap 305
Summary 309
References 309

Chapter 9: Painting with Bits 311

Digital Images 311
Raster Processing 312
Pixelation 313
Negative Images 315
Copying Neighbors 316
Multi-dimensional Arrays 319
Two-dimensional Arrays 319
Loops and 2D Arrays 321
Visualizing 2D Arrays 322
Image Animation 325
2D Perlin Noise 326
Ragged Arrays 331
Multi-dimensional Arrays 332
Processing's Pixel Buffer(s) 333
1D and 2D Arrays 333
Gradient Shading 335
Grayscale 336
Sepia and Other Palettes 338
Bitwise Processing and Component Functions 340
Bitwise Operators 340
Retrieving R/G/B/A Values Bitwise 343
Steganography 345
Complex Systems 348
Emergence 348
Cellular Automata 355
Chapter Project: Truchet Tiling 359
Summary 368
References 368

Chapter 10: Expressive Imaging 369

ImageIO 369
Tinting 370
Masking 373
Image Mask 373
Pixel Buffer Mask 375

Blending 378
Filtering 382
GRAY and INVERT 383
OPAQUE 384
THRESHOLD and POSTERIZE 384
BLUR, DILATE, and ERODE 388
Advanced Filte rs 390
Convolution 390
Other Spatial Filters 399
ImagespecialFX 399
Pointillism 399
Confluency 400
Video Processing 401
Chapter Project: Image Mosaic 405
Summary 411

Chapter 11: Spreading Your Creative Coding Wings 413

3D 413
View Frustum 415
It’s All Virtual 416
Some Simple 3D Math 418
P3D and OpenGL 420
Simplicity! 421
Custom Geometry 423
Final Illumination 427
A Very Small Nibble of a Vast and Fascinating Pie 429
Advancing to Java 429
First Java Dip 429
Diving Deeper into Java 432
There *Really* Are No Functions in Processing 433
Modes 434
JavaScript Mode 434
Android, et al 436
Summary 436

Index 437

Foreword

One of the very best things about writing a book is the opportunity it presents to meet new people. After the release of *Processing: Creative Coding and Computational Art*, in June 2007, I received many thoughtful notes from creative coders around the world. Most of these readers, similar to me, came from the arts and were just discovering the exciting creative potential of code. I also received some notes from professional programmers and computer scientists, which at the time was very surprising. As a self-taught coder, I wasn't sure what I could offer trained computing professionals, or even if I would be taken seriously.

Two of the computer scientists I met at the time were Deepak Kumar and Dianna Xu, both professors in the Department of Computer Science at Bryn Mawr College. Deepak and I first communicated (virtually) in the Processing online forum, during a somewhat heated discussion thread about the disconnect between the creative coding and computer science communities, especially within academia. You can read the original thread at: http://processing.org/discourse/beta/num_1212418008.html. Near the end of the thread Deepak graciously extended an open invitation to visit Bryn Mawr, to present Processing and explore the connections between our two communities. What especially motivated me to follow-up on Deepak's invitation was the last sentence of his comment:

"Let's work together and see how we can bring about some radical change!"

During my initial visit to Bryn Mawr I met Dianna and Deepak. Though I had been coding for a fairly long time by now and had regular interactions with computer science faculty from the institution where I was teaching, it was still intimidating sitting across the table from them–*with actual PhDs in Computer Science*–but it was also equally exhilarating! Dianna and Deepak were very seriously exploring new ways to teach Computer Science, including how to make programming more engaging and accessible to the widest possible audience. They had been very successful pioneering an earlier approach utilizing robots in the introductory computing classroom. It was such a thrill to learn about their work and to be able to talk so intensively about coding and computer science pedagogy. Though we started from very different places–the arts and computer science–we somehow ended up in a pretty similar place, with a shared passion for presenting computation as a powerfully creative and fascinating medium, and spreading "the word" to future generations.

Over the next few years, Deepak, Dianna, and I worked closely presenting lectures and workshops on Processing and our emerging "creative coding" approach. We were very fortunate to receive funding from the National Science Foundation enabling us to more formally explore our approach in the Computer Science 1 (CS1) classroom, which we did at our respective schools. During this period we also began discussing the idea for a new CS1 book, based on Processing and creative coding. These discussions, along with the result of our research, directly led to the creation of the book you're holding in your hands.

This book is a departure from the existing Processing and CS1 literature in that it attempts to present the spirit and excitement of creative coding, while rigorously covering the fundamentals taught in the CS1 classroom. We structured the book to be useful to the autodidact, but equally useful to the CS1 instructor, at both the secondary and post-secondary levels. We've each had very positive results utilizing the book's approach within our own classrooms and are very excited to now help others do the same.

Finally, I want to very publically thank Deepak and Dianna for “taking me in” to the Computer Science fold (including tolerating my artistic ways) and also for agreeing to co-author this book. I’ve learned a great deal working with them and know this book is far richer because of their generous and thoughtful collaboration, AND I’m hopeful we will indeed *“…bring about some radical change!*”

Ira Greenberg, 2013

About the Authors

Ira Greenberg directs the Center of Creative Computation and is Associate Professor of Computer Science at Southern Methodist University in Dallas, TX. He is the author of the first major reference on the Processing language, *Processing: Creative Coding and Computational Art* and also wrote *The Essential Guide to Processing for Flash Developers*, both by friends of ED. A formally trained painter (BFA, Cornell University, MFA, University of Pennsylvania) turned computational autodidact; Ira has spent the past 20+ years searching for ways to make code drip. When not programming or hanging out with his family, you can find Ira playing and coaching ice hockey around the Dallas-Fort Worth metroplex.

Dianna Xu is an Associate Professor of Computer Science at Bryn Mawr College, PA. She devotes considerable time and effort rethinking the CS curricula to include contemporary, diverse examples of computing in a modern context, and to attract interdisciplinary and non-conventional students to the field. Her research interests include Computer Graphics, Computational Geometry, Visualization, Creative Computation, and Computer Science Education. She received her B.A. in Computer Science from Smith College and her M.S. and Ph.D. in Computer and Information Science from the University of Pennsylvania.

Deepak Kumar is a Professor of Computer Science at Bryn Mawr College and the associate director for education and diversity of the Center for Science of Information, a National Science Foundation Science & Technology Center (`www.soihub.org`). His research interests include artificial intelligence, science of information, data visualization, creative computing, and computer science education. He received a MS in Instrumentation from Birla Institute of Technology & Science, and a MS and PhD in computer science from the University at Buffalo.

About the Technical Reviewer

Ryan Rusnak is a software developer with a passion for creative coding. He holds a master's degree from Carnegie Mellon University in Human-Computer Interaction. His projects have been featured in *WIRED*, *Popular Science*, and on the Science Channel and the *Graham Norton Show*. He currently resides in Arlington, VA with his wife Kirby Rusnak.

About the Cover Image Artist

Corné van Dooren designed the front cover image for this book. After taking a break from friends of ED to create a new design for the Foundation series, he worked at combining technological and organic forms, with the results now appearing on the cover of this and other books.

Corné spent his childhood drawing on everything at hand and then began exploring the infinite world of multimedia—and his journey of discovery hasn't stopped since. His mantra has always been "the only limit to multimedia is the imagination," a saying that keeps him moving forward constantly.

Corné works for many international clients, writes features for multimedia magazines, reviews and tests software, authors multimedia studies, and works on many other friends of ED books. If you like Corné's work, be sure to check out his chapter in *New Masters of Photoshop: Volume 2* (friends of ED, 2004). You can see more of his work (and contact him) at his website, `www.cornevandooren.com`.

Acknowledgments

The authors would like to thank Aaron Cadle, Eric Eaton, Mark Russo, and Paul Ruvolo for their willingness to try this new approach of teaching Introduction to Computer Science with us; and for their support, encouragement, and assistance on this project in general. We would also like to thank our numerous students who took a course from us as we were developing these materials, with special thanks to Amanda Guadalupe, a student, whose project work on visualizing her Twitter tweets is included in Chapter 5.

We would like to express gratitude to Ben Fry and Casey Reas for their responsiveness to questions on the Processing 2.0 transition.

In addition, we would like to thank our families, who supported and encouraged us in spite of the time it took us away from them.

This work was partially supported by funds from Bryn Mawr College, Southern Methodist University, the Mellon Foundation, and the National Science Foundation under Grant No. 0942626, No. 0942628, No. 1140519, and CCF-0939370. We sincerely thank them for their support.

Introduction

Creative Coding grew primarily out of the digital art and design community, as an approach to programming based on intuitive, expressive, and organic algorithmic development, with an iterative leap-before-you-look style. Related, but broader, is the idea of Creative Computation, which explores computation as a universal, generative, and primary creative medium. We find these paradigms well suited for introducing computing to a new generation of students, who respond well to creative tasks and visual feedback. This book attempts to introduce programming and the fundamentals of computing and computer science by engaging you, the reader, in Creative Coding and Creative Computation contexts.

This book is designed for independent learning as well as a primary text for an introductory computing class (also known as CS1, or CS Principles, in the computing education community). A lot of the material grew out of our very successful NSF TUES funded project to develop a complete CS1 curriculum using Processing and Creative Coding principles. The central goal of the project was to strengthen formative/introductory computer science education by catalyzing excitement, creativity, and innovation. The digital representation of data, access to authentic sources of big data, and creative visualization techniques are revolutionizing intellectual inquiry in many disciplines, including the arts, humanities, and social sciences. We strongly believe that the introductory computing curriculum should be updated with contemporary, diverse examples of computing in a modern context.

We developed our introductory computing curriculum based on the philosophy that it should cover the same set of core CS1 topics as any conventional Java-based CS1, but show applications in the visual arts and interactive media, as well as through clean, concise, intuitive examples of advanced areas not typically accessible to CS1 students, including physics-based simulations, fractals and L-systems, image processing, emergent systems, cellular automata, aspects of data science and data visualization.

While it is entirely possible to learn to program on your own with this book, teaching with this book will require additional organization and initiative on the part of the instructor. This is an unconventional CS1 text in that our priority was to demonstrate the Creative Computation way of thinking (or way of teaching) and how it connects to the introductory computing curriculum, rather than to provide systematic/detailed lesson plans. Many standard, but basic programming constructs have not received the typical amount of attention because we trust a certain level of instructor experience and comfort to fill the gaps in the classrooms. It is our hope and belief that once shown the creative possibilities, following the same philosophy and adapting the material to your own classrooms will be an enjoyable and motivating task.

Resources

The home of the Processing project is the website:

`http://processing.org`

First and foremost, install Processing from their Download section. At the time of this writing, Processing 2.0 is still in later stages of Beta (the most current version is 2.0b8). The stable release remains to be 1.5.1. The Processing installation comes with an extensive collection of examples directly accessible from the IDE through the File➤Examples pull-down menu that should not be overlooked. This book includes coverage of most of the key features included in Processing 2.0. Those using Processing 1.5.1 release should bear this in mind.

Besides the Download section, the Processing website maintains a complete API Reference. We refer to it as the Processing Reference in this book and would encourage the reader to become familiar with navigating their browsers to it as well as to learn to read the rich documentation and examples it provides. Much learning can and will take place with increased usage of the Processing Reference. Equally important for the beginner (as well as the instructor) is the Learning section of the `Processing.org` website. The set of tutorials provided there will be valuable for anyone starting to learn Processing from this book. These comprehensive tutorials are good starting points for the basics, as well as additional information. In fact, these should be treated as ancillary materials for the earlier chapters. Both reference and tutorials are also conveniently accessible from the Processing IDE from the Help pull-down menu (just access the Help➤Reference and Help➤Getting Started options).

We have taught using the approach presented here well over a dozen times in the past few years. Consequently, we have accumulated a wealth of curricular material, much more than there is room for in this book. Instructors interested in gaining access to complete syllabi, lecture notes, class examples, assignments, and problem sets, please write to us.

The book is sprinkled with a number of "Try This" suggestion boxes, which we have chosen to place wherever they are relevant. Some merely test a grasp of the current discussion; others require more substantial thought and even full-fledged programming work. The complete code examples in the book can be downloaded from the book publisher's resource website:

`http://apress.com/9781430244646`

While it would be trivial to copy the code and run it, we strongly urge everyone to actually type the programs yourselves. This is a book about learning after all, and there is no substitute for experiencing the coding process first hand. Entering your own code from scratch, even if you are copying from these examples, is essential to learning the concepts that are being communicated. Question every word in every line of the code as you type, and don't be afraid to experiment. You will learn more!

We also recommend visiting the online community OpenProcessing:

`http://www.openprocessing.org`

This website is devoted to sharing open-source Processing sketches. There is much to learn from and get inspired by from the sketches shared there by creative coders, students, and teachers alike from across the globe. We hope that, as you learn and evolve your own creative style, you will contribute your own sketches.

Finally, the Processing.org website itself has its own discussion forum where you can post queries and get involved in discussions involving all things Processing. We urge you to support these efforts in any way you can. Buy a Processing t-shirt!

In the Book

In line with the philosophy of leap-before-you-look we begin by literally diving into the shallow end (Chapter 1) where we provide background and introduction to creative coding, Processing, its origins, and its relationship with Java. We offer a 30,000 feet overview of creative coding, the distinction between programming and the discipline of computer science. We also provide a detailed tour of the Processing IDE, which could be skimmed in the initial encounter but returned to later as the need arises. We follow this, in Chapter 2 (Art by Numbers) with a whirlwind tour of pseudocode, algorithms, and basic programming constructs: variables, simple types, expressions, and comments. Drawing with these coding primitives is presented by introducing Processing's

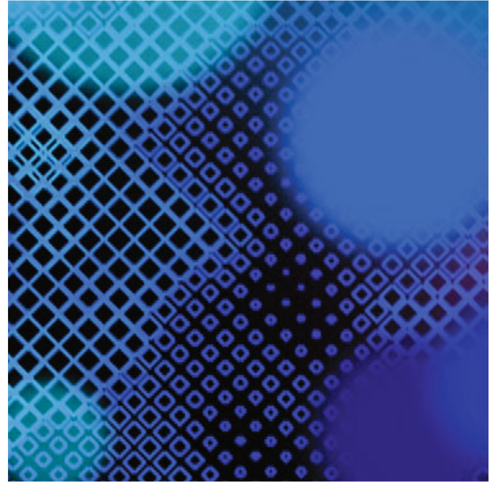
coordinate system and commands for drawing 2D shapes. This sets the stage for a deep immersion in Processing and creative coding. By taking a boot camp approach (Chapter 3), we introduce functions, all the basic control structures (conditionals and loops), and additional drawing concepts like transformations, and drawing contexts. Instructors of introductory computing courses may want to take a more deliberate approach here especially with the key programming constructs. These are further developed in Chapter 4 to enable a deeper understanding of structuring programs. At the same time, we introduce the dynamic mode in Processing, including handling mouse and keyboard events, as well as a deeper treatment of drawing curves. By this time, most of the basics of programming as well as Processing have been introduced. We would encourage the readers to take some time to reflect, take stock, and review before moving on into the book.

Beyond the basics, we next delve into some of the core CS1 topics: arrays, object-oriented programming, and recursion. We use the backdrop of learning about arrays to introduce concepts in data science and visualization. Arrays are introduced as mechanisms for storing, processing, and visualizing datasets (Chapter 5). Also, one of the core computer science ideas: algorithms and space-time complexity are given an informal treatment. Object-oriented programming and its manifestation in Processing is introduced in Chapter 6 as a mechanism for organizing the chaos of large programs and also as a way of modeling “live” or animated entities in a sketch by way of particles and physical motion models. We introduce Processing’s PVector class and go beyond basic physics by providing a detailed example of Verlet Integration for natural looking motion and behaviors in sketches, thereby truly enhancing a creative coder’s conceptual toolbox. We go deeper into creative data visualization (Chapter 7) by deconstructing (and reconstructing) a popular visualization application: word clouds. In the process, strings, ArrayLists, sorting algorithms, and font metrics are introduced. In the context of a fairly non-trivial example, we have attempted to illustrate program design and redesign principles and how object-oriented programming facilitates well-designed, maintainable, clean code. More creative fun follows with recursion (Chapter 8) as a computational medium for creating and experimenting with models of biological systems, fractal geometry, and advanced data visualization.

Our journey continues deeper both into computing and creative computing where in Chapter 9 we explore painting with bits in the context of image processing and manipulation. Two dimensional arrays and pixel buffers as underlying computing concepts come together in manipulating images to create stunning visual effects. We go beyond, by using bitwise operations and Processing’s color component functions to see examples of steganography. Further, iterative solutions are presented as solutions for modeling emergent systems, cellular automata, and glorious tiling patterns. We build further on image manipulation in Chapter 10 where we try to impress upon the idea that image processing is a creative medium with wider range of applications that go beyond traditional “photo manipulation.” This is also where the fact that Processing is designed by artists and creative coders really pays off: We introduce several built-in image manipulation functions that are typically not accessible to students in an introductory computing course.

If you consider this book a (big) meal, you can think of the last chapter as dessert–a hard earned and guilt free dessert! It is truly a buffet, with a glimpse into the three-dimensional world, developing sketches for Android devices, and going beyond Processing into Java, OpenGL, C++, and other programming environments. In today’s ubiquitous world of computation, the boundaries between languages, environments, and devices are blurring. Processing gives you a handle on the myriad of techniques and tools that serve as building blocks at first and then takes you to the very cusp of exciting new frontiers of creative computing. We hope you will enjoy this gradual, yet deliberate computational, intellectual, and visual feast. If we have not been successful at any aspect of the book, please write to us, we want to know.

Bon appétit and happy creative coding!



Chapter 1

Diving into the Shallow End

Imagine enrolling in a class to learn a new sport, perhaps Irish Hurling. (This type of hurling involves a ball and bat.) You arrive at class on the first day excited to dive into this exotic new hands-on activity, only to be confronted by a long lecture on the theoretical foundation of hurling. Over the next fourteen weeks, the course proceeds in a similar fashion, with maybe a couple of contrived on-the-field exercises to reinforce the lectures. We don't know about you, but we're not so sure we'd make it through this class long enough to get to the on-the-field part (the part that got us excited about learning hurling to begin with.) This is what many students experience when they take their first computer science class.

Of course, learning the theory behind Irish Hurling might provide you with pretty interesting and ultimately valuable information, especially if your goal is to become a world-class Hurler. However, for most of us, diving directly into the theoretical aspects of an activity such as hurling, or computer science, before getting a handle on why the theory might actually be useful can be intimidating and off-putting (and very few of us are destined for the Hurling Hall of Fame.) Worst of all, this approach can lead to wider societal misconceptions, such as: *computer science is obscure, difficult, and even boring*. These misconceptions can also become self-fulfilling prophecies, ultimately attracting only the *types* of students who buy into the misconception, leading to a population of students and practitioners lacking in diversity and varied perspective.

In the last few years, some computer scientists and other computing professionals, the authors of this book included, have begun challenging the entrenched and narrow approach to teaching computer science. This book champions a new way–*a creative coding approach*, in which you'll learn by doing. Building creative code sketches, you'll learn the principles behind computer science, but in the context of creating and discovery. Returning to the hurling analogy, first you'll learn how to whack the hurling ball (*the silotar*) with the hurling bat (the *hurley)* wicked hard; then you'll learn the physics behind it. Or, to use some computing lingo, first

you'll learn to code a cool app, then you'll learn about the fundamental principles behind it. Not only will this make coding easier and more fun to learn, but it will make the theory part much more relevant and hopefully even fascinating.

This chapter provides just a little context and background for the rest of the book. You'll learn about the history of Processing, including its origins at the famous MIT Media Lab. We'll discuss the creative coding approach in a bit more detail, including some relevant research into its effectiveness. Finally, you'll have a detailed tour of the Processing language and development environment.

Programming vs. Computer Science

If you want to tick off a computer scientist, tell him that you *know* computer science because you can write some code. (Of course, many computer scientists don't show a lot of emotion so you may not even be sure you've succeeded.) Kidding aside, programming is not computer science. BUT, from our perspective, it is often the most fun part of it. Yet, there are computer scientists who don't actually program. These are theoreticians who see computation more as applied mathematics than as hands-on implementation. Such a theoretician might be interested in proving something about computing, using mathematical proofs. However, to the average end-user, programming is often equated with computer science.

According to Dictionary.com, computer science is defined as:

> *the science that deals with the theory and methods of processing information in digital computers, the design of computer hardware and software, and the applications of computers.*

The first part of the definition, *the theory and methods of processing information* is concerned with more fundamental mathematical principles behind computing. This is perhaps the most pure scientific part of computer science. Research in this area affects things like the speed, efficiency, and reliability of computers. Arguably, this area of research provides the bedrock for all other aspects and applications of computing. Though programming is a part of this branch of computer science, its role is primarily for testing and verifying theory.

A company like Apple spends a great deal of time and resources researching how its hardware and software should look, feel, and function in the hands of users. This area of computer science research, the second part of the dictionary.com definition, the *design of computer hardware and software,* is where science gives way to engineering and design–where theory is applied, creating tangible systems. Another way of describing this area might be: the interface between the mathematical and theoretical aspects of computing and the incredible things we can do with it. Programming is a huge part of this area and is commonly referred to as software engineering.

The last part of the definition*, applications of computers* (not to be confused with computer apps) is about how computers (really computation) can be applied in the world. This part of the definition may be too general, as computers impact nearly all aspects of life, and it's extremely likely this impact will only increase in the future. It's not such a leap to imagine our cars driving themselves, our walls and countertops acting like smart touch screens, and our communication devices shrinking and getting even further integrated into perhaps even our

physical bodies. Programming is very relevant to this part of computer science as well, mostly in the development of specialized software and hardware targeting specific application domains.

Google developed and released the Android Software Development Kit, which includes libraries of code and application software for creating custom Android apps. Apple has its own similar development platform, as do many other companies. These development environments enable people to efficiently program applications, without the need of years of formal computer science training. Clearly, this evolution in software development is challenging long held notions of required technical expertise. There are high school students writing highly successful mobile applications, artists programming interactive artworks, and many other “non-experts” creating small software businesses overnight. So no, programming is not computer science, but apparently computer science is not necessarily required for programming either.

Art + Science = Creative Coding

At Southern Methodist University in Dallas there is a Center of Creative Computation (C^3) that explores computation as a fundamental creative medium. C^3 considers computer code (as well as other aspects of computation) the same way a painter thinks about paint, or a musician sound or even how a dancer thinks about gesture. C^3 is less concerned with why computation solves a specific problem and more interested in how it is solved, and most importantly, how it can be solved in a more interesting and novel way. Yet in spite of this creative approach, C^3 requires students to take very challenging courses in computer science, math, and physics. It also requires an equal amount of rigorous creative courses. This integration of quantitative material with creative practice can be a daunting challenge for some students, especially those who were labeled at an early age: “the artist” or “the geek,” but probably not both.

C^3 has been successful (as has a similar interdisciplinary approach at Bryn Mawr College) integrating difficult quantitative material with creative practice in the classroom, and research lab, utilizing a “Creative Coding” approach. This approach was originally developed at the Massachusetts Institute of Technology (MIT) Media Lab, by past lab director John Maeda, who you’ll hear more about shortly. Creative coding combines approaches from the arts classroom, such as critiques, portfolio development and emphasis on aesthetics and personal expression, with fundamental principles from computer science. Creative coding uses computer code as the creative medium by which students develop a body of art, while developing core competency in programming.

In 2010, researchers from Bryn Mawr College and C^3 at Southern Methodist University received a National Science Foundation grant to explore the use of creative coding in the introductory computer science classroom. Based on early research results, it is very promising that students learning the creative coding approach develop significantly greater personal interest in programming as compared to students in a more traditional computer science class.

To help facilitate this integration in the classroom, the creative coding approach relies on some innovative programming languages and development environments, especially Processing, which grew directly out of work done at the MIT Media Lab.

MIT Media Lab

The MIT Media Lab was founded by MIT professor Nicholas Negroponte and then-MIT President Jerome Wiesner in 1985. Its mission, as stated on the Media Lab site (http://www.media.mit.edu/about), is to:

> *envision the impact of emerging technologies on everyday life—technologies that promise to fundamentally transform our most basic notions of human capabilities.*

Though an academic lab at MIT within the School of Architecture and Planning, the Media Lab has always radically crossed disciplines and blurred distinctions between theory and implementation, academia and industry, and science and art. It has been involved in fundamental breakthroughs of the digital age since its founding, including the World Wide Web and wireless networks. The lab has also pioneered innovative research and development in radically new areas, such as smart toys, ubiquitous computing, and aesthetics and computation.

The Aesthetics + Computation Group (ACG) at MIT was created in 1996 by John Maeda, a formally trained computer scientist and graphic designer. Maeda and ACG explored novel approaches to software tools and language development, as well as computational artistic practice. One of the projects developed at the Media Lab was a new programming language and programming environment named "Design By Numbers" (DBN). DBN is a very simplified programming language built on top of the Java programming language (explained a bit later in this chapter). DBN greatly simplified the process of graphics programming using Java by creating a simplified language syntax (the commands and rules used to program) and a development environment that enabled fast prototyping of simple graphics patterns, code art, and designs. DBN was never intended as a full-featured programming language, but rather a proof of concept for a radically new approach to language design; it was tested primarily in the design arts classroom to teach programming to beginners.

DBN as a proof of concept was a big success, though as a usable language, it wasn't much more than an academic exercise. Two of Maeda's students in the Media Lab, Ben Fry and Casey Reas, worked on DBN. After finishing their studies at the Media Lab, Fry and Reas decided to take the lessons learned developing DBN and build a more full-featured language. They named their new project Processing, which they kicked off in 2001.

What Is Processing?

In the very simplest sense, Processing is a software application that allows you to write, edit, compile (which will be explained shortly), and run Java code. However, before discussing Processing further, it will help you to understand a little bit about Java, but even before we talk about Java, we need to talk briefly about computing in general. (Please note, this will be one of the only places in the book where we throw some theory at you without a fun, hands-on activity.)

Bits and Bytes

You probably have some sense that computers and 1's and 0's go together. But it may not be so clear to you how a bunch of 1's and 0's can lead to Shrek running through a field of blowing grass, your computer's operating system, or Facebook. It really is truly remarkable what has been done with a bunch of 1's and 0's. Though really, it's not about 1's or 0's, but instead a state of being true or false, or more accurately, something being

open or closed. If you've ever looked inside your computer, at the beautiful and mysterious boards, cards, chips, wires, etc., it should be obvious that everything in there is fundamentally reliant on electricity. However, electricity is a pretty mysterious force unto itself, and it will be simpler for this discussion to think of electricity in a much more general form, as a flowing source, something akin to water. (Though we can't recommend filling the inside of your computer up with a hose.)

Using the water metaphor, you can think of the guts of your computer as a series of incredibly complex canals with controllable dams. If a dam is down or closed, water doesn't flow past it; if it's up or open, water does pass through. As a complex interconnected system, some dams control the availability of water for thousands of other dams. By systematically controlling dams, you can control how, when, and where water flows through the system. Perhaps some of the dams are controlled by water pressure in parts of the system; when water starts flowing they open up and remain open. You can think of this like a water loop that keeps flowing. Other dams might be open by default and water pressure closes them. Some of the dams can even be constructed in a series where one dam's state (open or closed) controls another's state. As the dam master, you can design complex systems to precisely control how water (and ultimately ships) move through the system, based on certain conditions. For example, if dam A and dam B are both open then perhaps a ship can safely pass through canal C, but if either dam is shut it can't. So even though we're just describing dams and canals, you can see how simple logic–if a certain condition is true, something occurs–can be constructed in the system. By simply opening or closing dams, we can control specific outcomes to the larger system.

Of course, computers use flowing electricity instead of water, but the system works in a similar way. Gates, the computer's version of dams, control the passage of electrons. Gates in modern transistors–the fundamental electronic components in devices such as our computers–can be open or closed, just like the dams. We can use a 1 or 0 to represent the two discrete states, which we refer to technically as binary digits, or more commonly as "bits."

A binary number system is based on only 2 unique digits (0 or 1), as opposed to our more familiar decimal system that uses 10 unique digits (0–9). We can design number systems with any number of unique characters. For example, another commonly used number system in computing is hexadecimal, with 16 unique characters (0–9 and A–F). Since computing is fundamentally based on those previously mentioned open or closed gates, it's efficient for computers (not us) to utilize a binary system.

A bit, at any one point in time, can either be 1 or 0, but never both. When we group eight of these bits together, we refer to this as a byte: one thousand bytes is a kilobyte, one thousand kilobytes is a megabyte, and a thousand of these is a gigabyte, and it keeps going. Hopefully, the common "buzz" terms people throw around when comparing their mobiles devices (*"Dude, my phone has 20 gigs of memory…"*) have a little more context now. If you think back to the dams and canals analogy, imagine the complex ship movement you could get with billions of individual dams. You can probably now imagine how, as seen from a plane above, millions of different boats moving through such a complex system of dams could create organized patterns–even approximating a running Shrek perhaps, or forming the basis for complex logic determining the rules about how to friend someone on Facebook.

Mnemonics

Manipulating individual bits to represent everything a computer does, though theoretically possible, is extremely impractical. The computer's language is purely mathematical–it breathes 0's and 1's. We humans, however, are

not quite as numerate as our machines, and we rely on more descriptive, symbolic systems to communicate, such as our natural spoken and written languages. While a computer might be happy with the binary code 0110011 to signify an operation such as adding two numbers together, humans prefer something more along the lines of the word "add." Though a series of 0's and 1's is efficient, it's difficult for most of us to efficiently decipher binary patterns and then remember what each unique pattern means. This divide between how computers *process* information as compared to how we *comprehend* it has led to the development of programming languages.

At a fundamental information processing level, our brains work quite similarly to our computers. Instead of a complex array of transistors, we have an interconnected network of neurons. The individual neurons can be thought of as analogous to individual transistors. Though instead of having gates, neurons utilize something called an action potential. The action potential, like a transistor's gate, is controlled by an electrical impulse, determining when the neuron transmits information, or fires. It's an all or nothing response, like an open or closed gate.

Information processing–whether in the brain or computer–is quite distinct from human comprehension. The computer is sort of a silicon brain in a shiny box. As mentioned earlier it groks on 1's and 0's, or what is more technically called machine language. When computers were first developed if you actually wanted to do something with them, you needed to learn to speak their native machine language. This was a very difficult, tedious, and slow process. Computer scientists quickly realized they needed to simplify the programming process and began to develop higher-level languages. By higher-level, we mean languages less directly mapped to how computers process information and more closely aligned with how we understand it. One of the first such languages developed was Assembly language.

Assembly language by today's standards is still a very low-level language–*pretty darn close to the 1's and 0's*–but it was a huge step forward in simplifying programming. Assembly language converts machine language commands from pure numbers to statements, including familiar words such as: `set`, `store`, `load`, and `jump`. In the context of the machine's native language, we can refer to these more natural language terms as mnemonics, or devices to help us understand and remember the underlying machine commands.

Though Assembly was a big step forward in simplifying programming, it's still a dense and complex approach. Because Assembly maps individual machine language commands with mnemonics, it still takes a lot of code to do relatively simple things. For example, the following is Assembly code to output the phrase: "Happy Creative Coding!"

```
; code based on example: http://michaux.ca/articles/assembly-hello-world-for-os-x
; A "Happy Creative Coding!" program using NASM
section .text
global c3Start
c3Start:
    push dword msglen
    push dword mymsg
    push dword 1
    mov eax, 0x4
    sub esp, 4
    int 0x80
    add esp, 20
    push dword 0
```

```
    mov eax, 0x1
    sub esp, 4
    int 0x80
section .data
  mymsg db "Happy Creative Coding!", 0xa
  msglen equ $-mymsg
```

By comparison, here is code to do the same thing in Java:

```
// A "Happy Creative Coding!" program using Java
public class Happy {
    public static void main(String[] args){
        System.out.println("Happy Creative Coding!");
    }
}
```

And finally, here's the same program in Processing

```
// A "Happy Creative Coding!" program using Processing
println("Happy Creative Coding!");
```

If it wasn't obvious, Java, and especially Processing, greatly reduced the number of lines in code. Also, if you read through the example code, we suspect you were able to understand much more of the Java and Processing code than the Assembly.

The first language assemblers, the software that converts Assembly code to machine language, emerged in around 1950. Java was released in the mid-1990s. In the forty or so years between Assembly and Java, many other programming languages were developed. One of the most important languages that emerged, which strongly influenced Java and ultimately Processing, was the C programming language. For our discussion, it's not necessary to say too much about C, other than that it was considerably more high-level than Assembly, and it became very widely adopted. Compared to Assembly, C greatly reduced the lines of code necessary to program the computer, allowing single programming calls to internally map to many lines of Assembly code; it was no longer a 1 to 1 translation with just added mnemonics. The grammar of the Java programming language, more commonly referred to as the syntax of the language, is heavily based on C, but as you'll learn, Java is an even more high-level language approach than C.

Java

The development of Java was, by some standards, a failure. Java was initially developed for interactive television and ultimately to connect "smart" devices, which didn't really catch on until about fifteen years after Java's release. We take it for granted now that our newer flat screen TVs are Internet ready, allowing us to surf the Net while we watch shows on demand and check our email. Back in 1995, this was a pipedream held by a few technology zealots, certainly not by the mainstream public. What arguably saved Java was the proliferation of the Internet, which we'll say more about in a moment. From near failure to ultimate success, Java is today one of the most popular programming language in the world, according to the TIOBE Programming Community Index (http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html).

Java was designed as a full-featured programming language, like C, but with one very big difference–universal portability. Java's original slogan was "Write once, run everywhere." The idea was that a programmer could write a Java program on any machine, and the program would run consistently on any other machine. This may not seem like such a big deal at first glance, but computers are not simply just the buzz words we've reduced them to: Mac, Windows, Linux. Computers are composed of lots of complex parts, such as central processing units (CPUs), graphical processing units (GPUs), and random access memory (RAM), just to name a few. These parts, especially the CPU, the main brain of the computer, rely on specific instructions, the machine language, to do their magic. Unfortunately, these machine instructions vary widely across not only computer brands, but also specific hardware components like CPUs.

Thinking back on our discussion about the Assembly programming language, you learned that Assembly wraps machine language with mnemonics–adding somewhat more normal-sounding language commands to the underlying binary math. The C language takes the process a step further, in a sense wrapping the Assembly language with a much higher-level language construct, greatly simplifying programming. For example, one line of C code might replace ten lines of Assembly. The problem with the way this approach works is that the code (regardless if it's Assembly or C) all still reduces down to machine language, and as previously mentioned, machine language code is specific to the machine hardware you're working on. To run a C program, you need to explicitly convert your C code to machine language for your specific CPU. We refer to this process as compilation or compiling. A C language compiler is software that makes the conversion from the C source code you write to the machine's native language, the binary code (1's and 0's).

So how did Java improve this situation? Java incorporates an additional layer of software, called a language interpreter, or to use Java speak, a Java Virtual Machine (commonly shortened to JVM). Java code, like C code, is compiled. However, the Java code is not compiled down to the native machine level, which again would be hardware specific. Rather, Java code is compiled to a higher universal form, called bytecode. The Java bytecode is universal, in that it should be able to run on any machine, regardless of the underlying hardware architecture, as long as the machine includes a Java Virtual Machine. If you think back to our earlier discussion, we wrote that Java was an initial failure saved by the proliferation of the Internet. The Internet is a vast network of computers, with widely varied hardware configurations, running various operating systems. The Java environment, including its compiler (to bytecode) and its interpreter (JVM), became the perfect solution to connect all these disparate devices. In addition, some Web browsers include their own Java Virtual Machine, allowing Java programs (referred to as applets in this context) to also run on the Web. To learn more about Java's interesting history, see: http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html.

In addition to the Internet, Java is also having a dramatic impact on mobile, and more generally, ubiquitous computing, with Java Virtual Machines widely available for many of our portable and handheld devices. For example, Google's Android operating system is based on Java.

So in the end, it seems Java, like other revolutionary ideas, was not so much a failure as ahead of its time. That's not to say that Java is without its critics. Java is still a fairly complex language and environment to work with and also challenging to teach with in the introductory computing classroom. Some critics also fault Java for being slower than a purely compiled language like C, which doesn't need to be run through a language interpreter. (It's quite a hot topic as to how much slower Java actually is compared to a purely compiled language like C.) Since Java's release, many other new languages have been developed that aim to further reduce the complexity of programming. One of these is Processing, which has been steadily growing in popularity since its release in 2001. Though Processing is indeed an independent programming environment, with its own language, you'll learn next that it is also inextricably linked to Java.

Processing

As we mentioned earlier, Processing emerged out of the MIT Media Lab, inspired by the simple DBN language. While DBN was developed as a proof of concept, a showcase to demonstrate an approach to programming, Processing was created as a full-featured programming environment for "real" creative development (*creative coding!*) However, DBN provided important lessons for Processing's initial developers, Reas and Fry:

- **Keep it simple!**
  - The Processing interface, which you can see in Figure 1-1, is incredibly minimal, *by design*. Reas and Fry conceived of Processing as a sketchbook of a sort, with essentially a blank page to begin creating on. Though Processing greatly simplifies the programming process, it was never intended to reduce the complexity of the creative process. The language is devoid of most slick filters and effects you might find in a software application like Adobe PhotoShop, again by design.

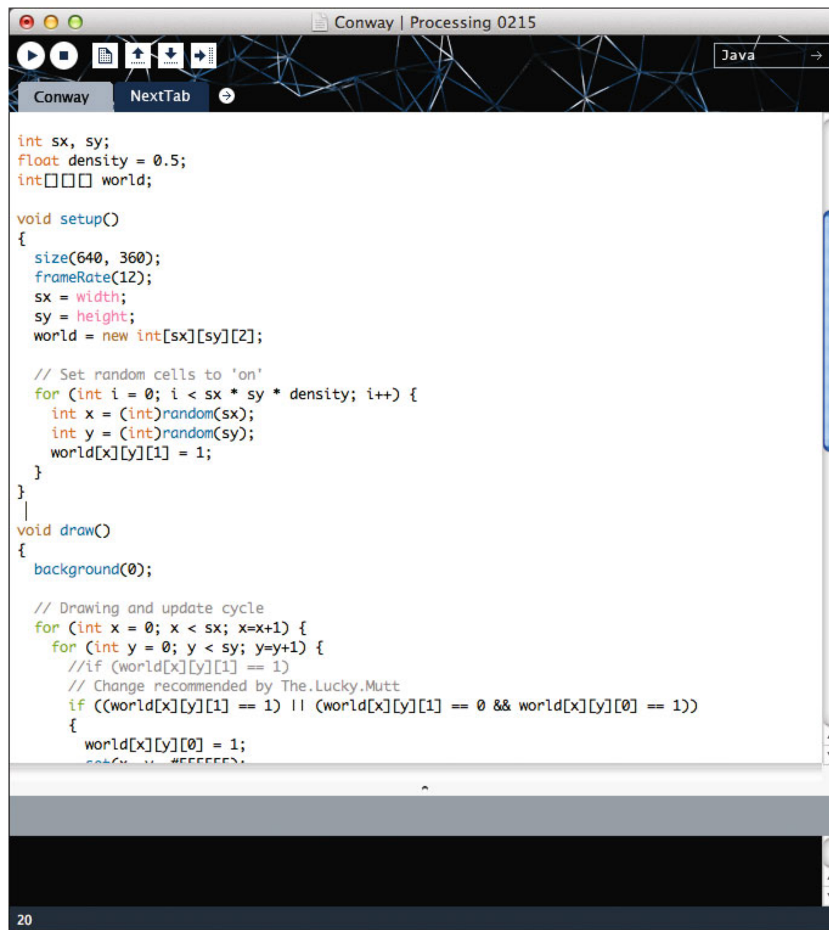


Figure 1-1. Main Processing interface

- **Create an easy-to-use environment to write, test, and run your code, also referred to as an integrated development environment, or IDE**
  - Processing is a completely self-contained executable application. You simply launch it by double-clicking and begin coding. Most other programming environments require a fair amount of fussing with system settings and preferences to get working. In addition, many of these other environments are temperamental and can easily *break*, as files get accidentally moved or saved. Also, the programming environments themselves (not even the programming languages you use within them) can be extremely complex to master. The Processing environment by contrast is simple and intuitive to use and doesn't add to the complexity of coding.
- **Create a zero-entry approach to coding**
  - On the first day of the introductory Computer Science class we have students who have never programmed before coding interesting creative work with Processing. This is nearly impossible with most other programming languages and development environments. Some languages will allow you to relatively easily output some text–the old school tradition is for CS 1 students to output "Hello World." Using processing, students create an original design or pattern as their first project. You can view examples of this work at our classroom site `http://openprocessing.org/classroom/1262`. As you might imagine, it's much more interesting and fun to create an image of your own design, than to simply output the words "Hello World."
- **Give the software away for free and release the source code**
  - Reas and Fry may have lost millions of dollars based on their "give it away for free" strategy, but this model also allowed Processing to be adopted worldwide. In addition, by releasing the Processing source-code, the actual code that created Processing, they attracted a devoted group of developers to help push the language along. Processing 2.0 benefited greatly from the extended team of passionate Processing developers.
- **Focus on graphics and multimedia development**
  - Like John Maeda, Reas and Fry have graphic design backgrounds, and that spirit has influenced many aspects of Processing. We sometimes joke with our students that Processing includes all the fun parts of Java and hides all the boring and annoying stuff; it's not quite that simple. However, Processing is a programming language that is focused on creative programming. In addition to the core Processing language, which we'll look at shortly, Processing includes extensive libraries of code contributed by the Processing community. These libraries extend Processing's capabilities all over the place, from vision detection, to the Microsoft Kinect, to physics engines, to network and database connectivity and many, many other creative and intriguing domains. And these libraries are free to download and quite easy to integrate within your projects.