

Processing:
a programming
handbook for
visual designers
and artists

Casey Reas
Ben Fry

The MIT Press
Cambridge, Massachusetts
London, England

© 2007 Massachusetts Institute of Technology

All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

MIT Press books may be purchased at special quantity discounts for business or sales promotional use. For information, please email special_sales@mitpress.mit.edu or write to Special Sales Department, The MIT Press, 55 Hayward Street, Cambridge, MA 02142.

Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Reas, Casey.

Processing : a programming handbook for visual designers and artists / Casey Reas & Ben Fry ; foreword by John Maeda.

p. cm.

Includes bibliographical references and index.

ISBN 978-0-262-18262-1 (hardcover : alk. paper)

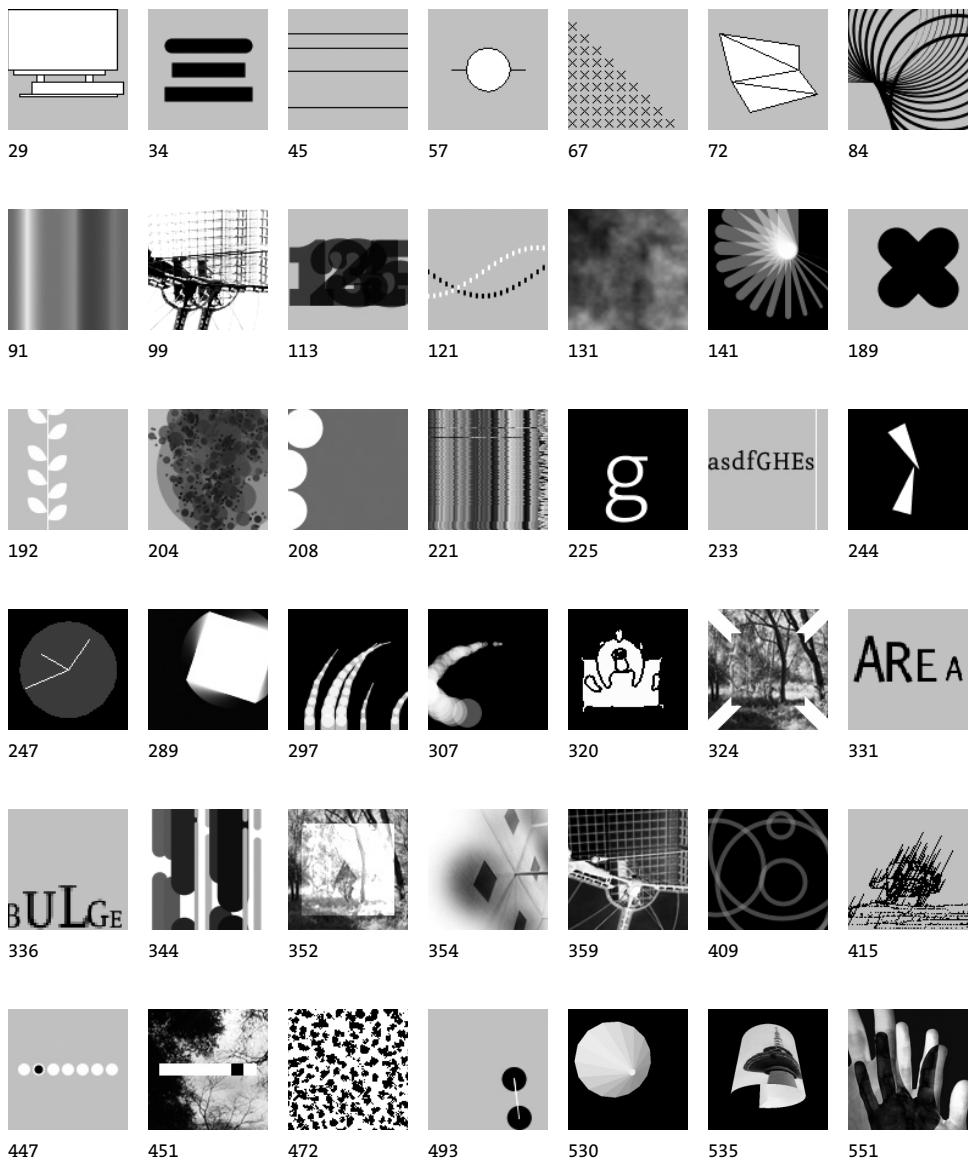
1. Computer programming. 2. Computer graphics—Computer programs. 3. Digital art—Computer programs. 4. Art—Data processing. 5. Art and technology. I. Fry, Ben. II. Title.

QA76.6.R4138 2007

005.1—dc22

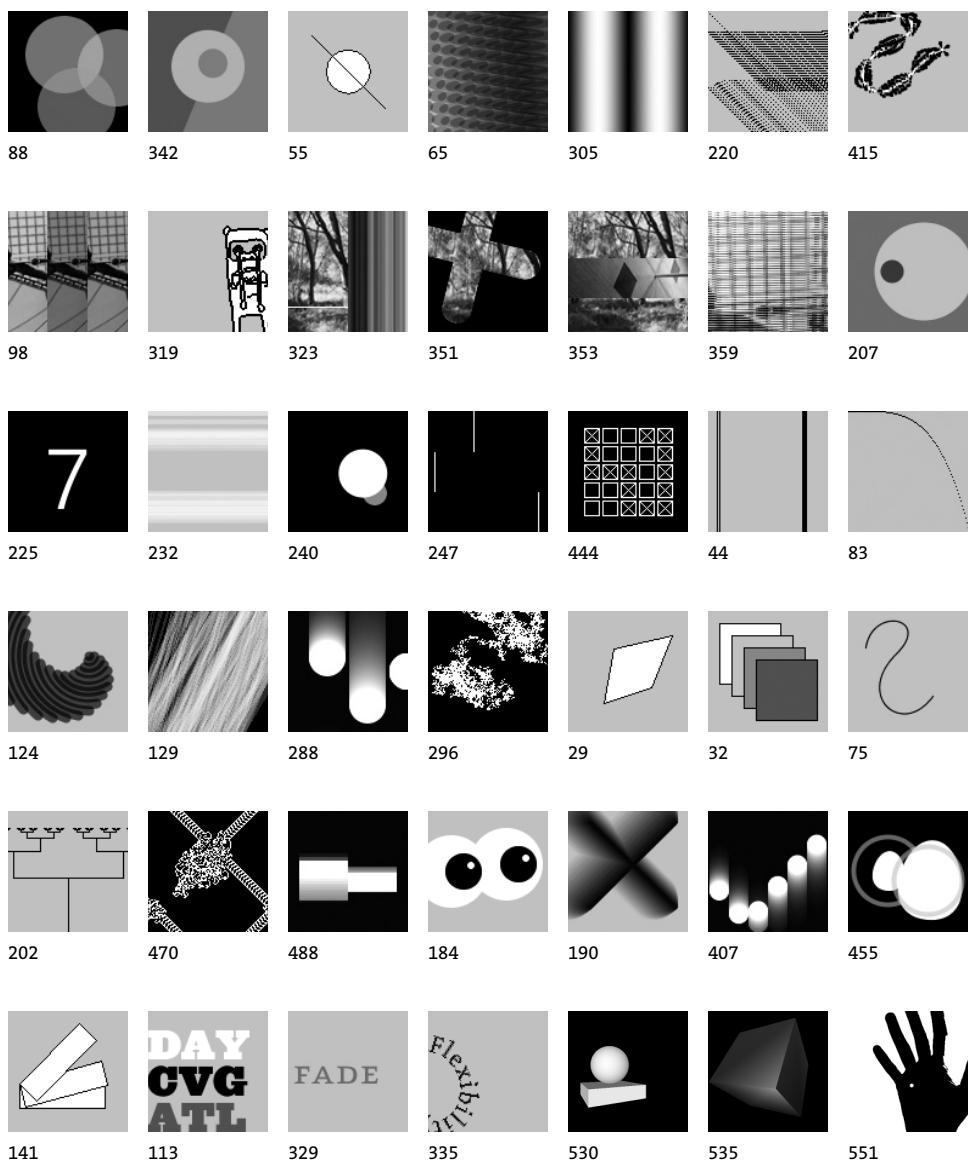
2006034768

10 9 8 7 6 5 4 3 2 1



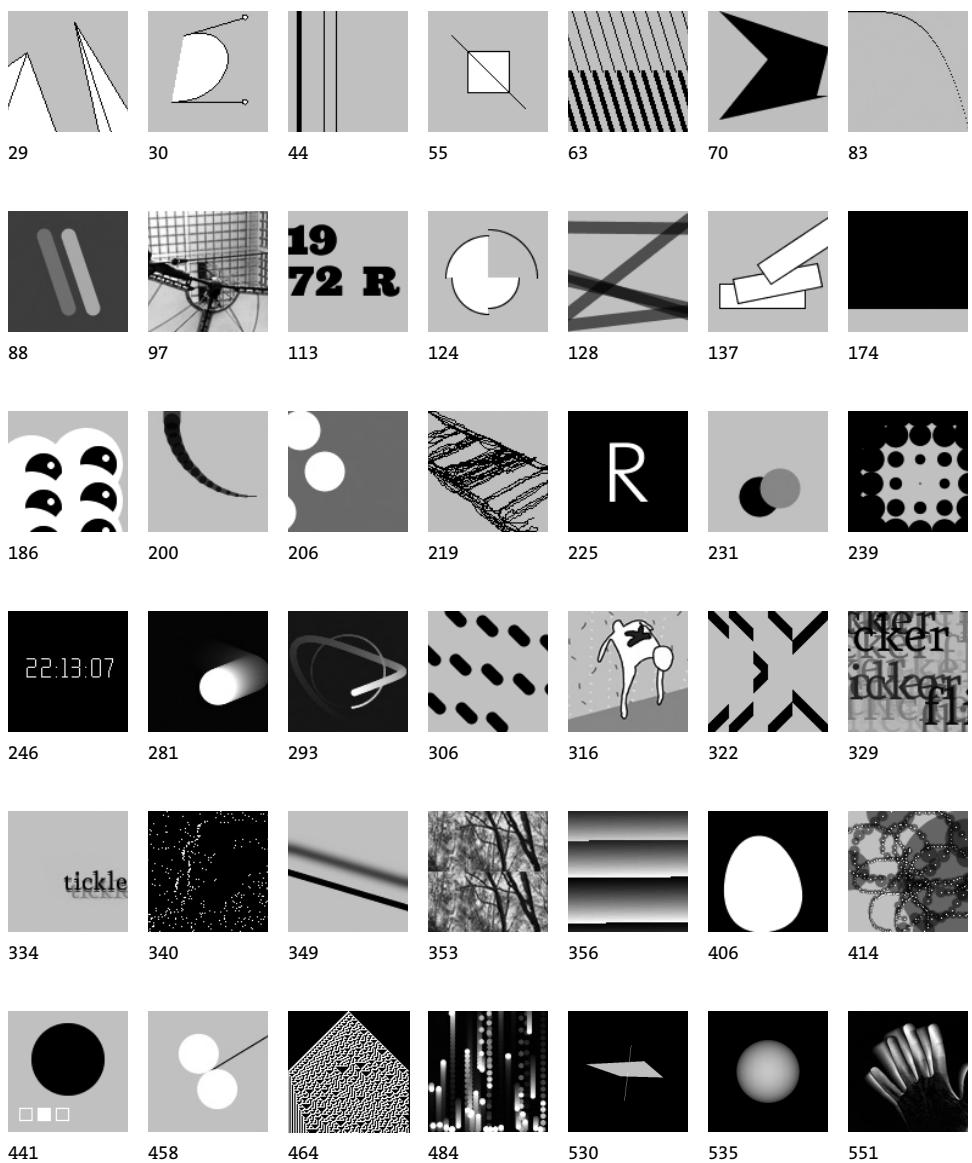
Contents

xix	Foreword	279	Motion 1: Lines, Curves
xxi	Preface	291	Motion 2: Machine, Organism
1	Processing...	301	Data 4: Arrays
9	Using Processing	315	Image 2: Animation
17	Structure 1: Code Elements	321	Image 3: Pixels
23	Shape 1: Coordinates, Primitives	327	Typography 2: Motion
37	Data 1: Variables	333	Typography 3: Response
43	Math 1: Arithmetic, Functions	337	Color 2: Components
51	Control 1: Decisions	347	Image 4: Filter, Blend, Copy, Mask
61	Control 2: Repetition	355	Image 5: Image Processing
69	Shape 2: Vertices	367	Output 1: Images
79	Math 2: Curves	371	Synthesis 3: Motion and Arrays
85	Color 1: Color by Numbers	377	Interviews 3: Animation, Video
95	Image 1: Display, Tint	395	Structure 4: Objects I
101	Data 2: Text	413	Drawing 2: Kinetic Forms
105	Data 3: Conversion, Objects	421	Output 2: File Export
111	Typography 1: Display	427	Input 6: File Import
117	Math 3: Trigonometry	435	Input 7: Interface
127	Math 4: Random	453	Structure 5: Objects II
133	Transform 1: Translate, Matrices	461	Simulate 1: Biology
137	Transform 2: Rotate, Scale	477	Simulate 2: Physics
145	Development 1: Sketching, Techniques	495	Synthesis 4: Structure, Interface
149	Synthesis 1: Form and Code	501	Interviews 4: Performance, Installation
155	Interviews 1: Print	519	Extension 1: Continuing...
173	Structure 2: Continuous	525	Extension 2: 3D
181	Structure 3: Functions	547	Extension 3: Vision
197	Shape 3: Parameters, Recursion	563	Extension 4: Network
205	Input 1: Mouse I	579	Extension 5: Sound
217	Drawing 1: Static Forms	603	Extension 6: Print
223	Input 2: Keyboard	617	Extension 7: Mobile
229	Input 3: Events	633	Extension 8: Electronics
237	Input 4: Mouse II	661	Appendixes
245	Input 5: Time, Date	693	Related Media
251	Development 2: Iteration, Debugging	699	Glossary
255	Synthesis 2: Input and Response	703	Code Index
261	Interviews 2: Software, Web	705	Index



Contents by category

xix	Foreword	23	Shape 1: Coordinates, Primitives
xxi	Preface	69	Shape 2: Vertices
1	Processing...	197	Shape 3: Parameters, Recursion
9	Using Processing	461	Simulate 1: Biology
85	Color 1: Color by Numbers	477	Simulate 2: Physics
337	Color 2: Components	17	Structure 1: Code Elements
51	Control 1: Decisions	173	Structure 2: Continuous
61	Control 2: Repetition	181	Structure 3: Functions
37	Data 1: Variables	395	Structure 4: Objects I
101	Data 2: Text	453	Structure 5: Objects II
105	Data 3: Conversion, Objects	149	Synthesis 1: Form and Code
301	Data 4: Arrays	255	Synthesis 2: Input and Response
145	Development 1: Sketching, Techniques	371	Synthesis 3: Motion and Arrays
251	Development 2: Iteration, Debugging	495	Synthesis 4: Structure, Interface
217	Drawing 1: Static Forms	133	Transform 1: Translate, Matrices
413	Drawing 2: Kinetic Forms	137	Transform 2: Rotate, Scale
95	Image 1: Display, Tint	111	Typography 1: Display
315	Image 2: Animation	327	Typography 2: Motion
321	Image 3: Pixels	333	Typography 3: Response
347	Image 4: Filter, Blend, Copy, Mask	155	Interviews 1: Print
355	Image 5: Image Processing	261	Interviews 2: Software, Web
205	Input 1: Mouse I	377	Interviews 3: Animation, Video
223	Input 2: Keyboard	501	Interviews 4: Performance, Installation
229	Input 3: Events	519	Extension 1: Continuing...
237	Input 4: Mouse II	525	Extension 2: 3D
245	Input 5: Time, Date	547	Extension 3: Vision
427	Input 6: File Import	563	Extension 4: Network
435	Input 7: Interface	579	Extension 5: Sound
43	Math 1: Arithmetic, Functions	603	Extension 6: Print
79	Math 2: Curves	617	Extension 7: Mobile
117	Math 3: Trigonometry	633	Extension 8: Electronics
127	Math 4: Random	661	Appendices
279	Motion 1: Lines, Curves	693	Related Media
291	Motion 2: Machine, Organism	699	Glossary
367	Output 1: Images	703	Code Index
421	Output 2: File Export	705	Index



x

Extended contents

xix	Foreword by John Maeda	23	Shape 1: Coordinates, Primitives
xxi	Preface	23	Coordinates
xxi	Contents	size()	
xxii	How to read this book	25	Primitive shapes
xxiii	Casey's introduction	point(), line(), triangle(), quad(), rect(), ellipse(), bezier()	
xxiv	Ben's introduction	31	Drawing order
xxv	Acknowledgments	31	Gray values
1	Processing...	background(), fill(), stroke(), noFill(), noStroke()	
1	Software	33	Drawing attributes
3	Literacy	smooth(), noSmooth(), strokeWeight(), strokeCap(), strokeJoin()	
4	Open	34	Drawing modes
4	Education	ellipseMode(), rectMode()	
6	Network	37	Data 1: Variables
7	Context	37	Data types
9	Using Processing	int, float, boolean, true, false	
9	Download, Install	38	Variables
9	Environment	=	
10	Export	40	Processing variables
11	Example walk-through	width, height	
16	Reference	43	Math 1: Arithmetic, Functions
17	Structure 1: Code Elements	43	Arithmetic
17	Comments	+, -, *, /, %	
	//, /* */	47	Operator precedence, Grouping
18	Functions	()	
18	Expressions, Statements	48	Shortcuts
	;, “,”	++, --, +=, -=, *=, /=, -	
20	Case sensitivity	49	Constraining numbers
20	Whitespace	ceil(), floor(), round(), min(), max()	
20	Console		
	print(), println()		

51	Control 1: Decisions	101	Data 2: Text
51	Relational expressions	102	Characters
	>, <, >=, <=, ==, !=		char
53	Conditionals	103	Words, Sentences
	if, else, {}		String
57	Logical operators	105	Data 3: Conversion, Objects
	, &&, !	105	Data conversion
61	Control 2: Repetition		boolean(), byte(), char(),
61	Iteration		int(), float(), str()
	for	107	Objects
65	Nested iteration		“.”,
67	Formatting code blocks		PImage.width, PImage.height,
69	Shape 2: Vertices		String.length,
69	Vertex		String.startsWith(),
	beginShape(), endShape(),		String.endsWith();
	vertex()		String.charAt(),
71	Points, Lines		String.toCharArray(),
72	Shapes		String.substring(),
74	Curves		String.toLowerCase(),
	curveVertex(), bezierVertex()		String.toUpperCase(),
			String.equals()
79	Math 2: Curves	111	Typography 1: Display
79	Exponents, Roots	112	Loading fonts, Drawing text
	sq(), sqrt(), pow()		PFont, loadFont(),
80	Normalizing, Mapping		textFont(), text()
	norm(), lerp(), map()	114	Text attributes
83	Simple curves		textSize(), textLeading(),
			textAlign(), textWidth()
85	Color 1: Color by Numbers	117	Math 3: Trigonometry
86	Setting colors	117	Angles, Waves
89	Color data		PI, QUARTER_PI, HALF_PI,
	color, color()		TWO_PI, sin(), cos(),
89	RGB, HSB		radians(), degrees()
	colorMode()	123	Circles, Arcs, Spirals
93	Hexadecimal		arc()
95	Image 1: Display, Tint	127	Math 4: Random
96	Display	127	Unexpected numbers
	PImage, loadImage(), image()		random(), randomSeed()
97	Image color, Transparency	130	Noise
	tint(), noTint()		noise(), noiseSeed()

133	Transform 1: Translate, Matrices	181	Structure 3: Functions
133	Translation	182	Abstraction
	translate()	183	Creating functions
134	Controlling transformations		void
	pushMatrix(), popMatrix()	193	Function overloading
		194	Calculating and returning values
137	Transform 2: Rotate, Scale		return
137	Rotation, Scaling	197	Shape 3: Parameters, Recursion
	rotate(), scale()	197	Parameterized form
139	Combining transformations	201	Recursion
142	New coordinates		
145	Development 1: Sketching, Techniques	205	Input 1: Mouse I
145	Sketching software	205	Mouse data
146	Programming techniques		mouseX, mouseY,
			pmouseX, pmouseY
149	Synthesis 1: Form and Code	212	Mouse buttons
150	Collage Engine		mousePressed, mouseButton
151	Riley Waves	213	Cursor icon
152	Wilson Grids		cursor(), noCursor()
153	Mandelbrot Set		
155	Interviews 1: Print	217	Drawing 1: Static Forms
157	Jared Tarbell.	218	Simple tools
	<i>Fractal.Invaders, Substrate</i>	221	Drawing with images
161	Martin Wattenberg.	223	Input 2: Keyboard
	<i>Shape of Song</i>	224	Keyboard data
165	James Paterson.		keyPressed, key
	<i>The Objectivity Engine</i>	227	Coded keys
169	LettError.		keyCode
	<i>RandomFont Beowulf</i>		
173	Structure 2: Continuous	229	Input 3: Events
173	Continuous evaluation	229	Mouse events
	draw(), frameRate(),		mousePressed(),
	frameCount		mouseReleased(),
177	Controlling the flow		mouseMoved(), mouseDragged()
	setup(), noLoop(),	232	Key events
178	Variable scope		keyPressed(), keyReleased()
		235	Controlling the flow
			loop(), redraw()

237	Input 4: Mouse II	301	Data 4: Arrays
237	Constrain	303	Using arrays
	constrain()		Array, [], new, Array.length
238	Distance	306	Storing mouse data
	dist()	309	Array functions
239	Easing		append(), shorten(), expand(), arraycopy()
	abs()	312	Two-dimensional arrays
242	Speed		
243	Orientation	315	Image 2: Animation
	atan2()	316	Sequential images
245	Input 5: Time, Date	319	Images in motion
245	Seconds, Minutes, Hours	321	Image 3: Pixels
	second(), minute(), hour(), millis()	321	Reading pixels
249	Date		get()
	day(), month(), year()	324	Writing pixels
			set()
251	Development 2: Iteration, Debugging		
251	Iteration	327	Typography 2: Motion
252	Debugging	327	Words in motion
		331	Letters in motion
255	Synthesis 2: Input and Response		
256	Tennis	333	Typography 3: Response
257	Cursor. Peter Cho	333	Responsive words
258	Typing	335	Responsive letters
259	Banded Clock. Golan Levin		
261	Interviews 2: Software, Web	337	Color 2: Components
263	Ed Burton. Sodaconstructor	337	Extracting color
267	Josh On. They Rule		red(), blue(), green(), alpha(), hue(), saturation(), brightness()
271	Jürg Lehni. Hektor and Scriptographer	341	Dynamic color palettes
275	Auriea Harvey and Michaël Samyn. The Endless Forest		
279	Motion 1: Lines, Curves	347	Image 4: Filter, Blend, Copy, Mask
279	Controlling motion	347	Filtering, Blending
284	Moving along curves		filter(), blend(), blendColor()
287	Motion through transformation	353	Copying pixels
			copy()
291	Motion 2: Machine, Organism	354	Masking
291	Mechanical motion		mask()
295	Organic motion		

355	Image 5: Image Processing	421	Output 2: File Export
356	Pixels pixels[], loadPixels(), updatePixels(), createImage()	421	Formatting data nf()
359	Pixel components	422	Exporting files saveStrings(), PrintWriter, createWriter(), PrintWriter.flush(), PrintWriter.close(), exit()
360	Convolution		
364	Image as data		
367	Output 1: Images	427	Input 6: File Import
368	Saving images save()	428	Loading numbers loadStrings(), split(), splitTokens()
369	Saving sequential images saveFrame()	431	Loading characters WHITE SPACE
371	Synthesis 3: Motion and Arrays	435	Input 7: Interface
372	Centipede. Ariel Malka	436	Rollover, Button, Dragging
373	Chronodraw. Andreas Gysin	442	Check boxes, Radio buttons
374	AmoebaAbstract_o3. Marius Watz	448	Scrollbar
375	Mr. Roboto. Leon Hong		
377	Interviews 3: Animation, Video	453	Structure 5: Objects II
379	Motion Theory. R.E.M. "Animal"	453	Multiple constructors
383	Bob Sabiston. <i>Waking Life</i>	454	Composite objects
387	Jennifer Steinkamp. <i>Eye Catching</i>	456	Inheritance extends, super
391	Semiconductor. <i>The Mini-Epoch Series</i>		
395	Structure 4: Objects I	461	Simulate 1: Biology
395	Object-oriented programming	461	Cellular automata
398	Using classes and objects class, Object	469	Autonomous agents
406	Arrays of objects	477	Simulate 2: Physics
409	Multiple files	477	Motion simulation
413	Drawing 2: Kinetic Forms	481	Particle systems
414	Active tools	487	Springs
416	Active drawings	495	Synthesis 4: Structure, Interface
		496	WithoutTitle. Lia
		497	Pond. William Ngan
		498	Swingtree. ART+COM, Andreas Schlegel
		499	SodaProcessing. Ed Burton

501	Interviews 4: Performance, Installation	579	Extension 5: Sound. R. Luke DuBois
503	SUE.C. <i>Mini Movies</i>	579	Music and sound programming in the arts
507	Chris Csikszentmihályi.	582	Sound and musical informatics
	<i>DJI, Robot Sound System</i>	584	Digital representation of sound and music
511	Golan Levin, Zachary Lieberman.	588	Music as information
	<i>Messa di Voce</i>	591	Tools for sound programming
515	Marc Hansen. <i>Listening Post</i>	592	Conclusion
519	Extension 1: Continuing...	593	Code
519	Extending Processing	599	Resources
521	Processing and Java		
522	Other programming languages	603	Extension 6: Print. Casey Reas
525	Extension 2: 3D. Simon Greenwold	603	Print and computers
525	A short history of 3D software	606	High-resolution file export
526	3D form	608	Production
531	Camera	612	Conclusion
532	Material and lights	613	Code
536	Tools for 3D	615	Resources
538	Conclusion		
539	Code	617	Extension 7: Mobile. Francis Li
545	Resources	617	Mobile software applications
		619	The mobile platform
547	Extension 3: Vision. Golan Levin	622	Programming for mobile phones
547	Computer vision in interactive art	624	Mobile programming platforms
549	Elementary computer vision	625	Conclusion
	techniques	626	Code
552	Computer vision in the physical world	631	Resources
554	Tools for computer vision		
555	Conclusion	633	Extension 8: Electronics.
556	Code		Hernando Barragán and Casey Reas
561	Resources	633	Electronics in the arts
		635	Electricity
563	Extension 4: Network. Alexander R. Galloway	637	Components
		638	Circuits
563	The Internet and the arts	639	Microcontrollers and I/O boards
565	Internet protocols and concepts	642	Sensors and communication
569	Network tools	646	Controlling physical media
571	Conclusion	648	Conclusion
572	Code	649	Code
576	Resources	658	Resources

661 Appendix A: Order of Operations
663 Appendix B: Reserved Words
664 Appendix C: ASCII, Unicode
669 Appendix D: Bit, Binary, Hex
673 Appendix E: Optimization
679 Appendix F: Programming Languages
686 Appendix G: Code Comparison

693 Related Media
699 Glossary
703 Code Index
705 Index

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

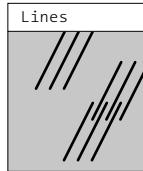
The Processing approach to programming blends with established methods. The core language and additional libraries make use of Java, which also has elements identical to the C programming language. This heritage allows Processing to make use of decades of programming language refinements and makes it 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>.



Display window

The screenshot shows the PDE interface with the following components labeled:

- Menu**: The top horizontal bar with options File, Edit, Sketch, Tools, and Help.
- Toolbar**: A row of icons for running, saving, and exporting sketches.
- Tabs**: A tab bar with "Lines" selected.
- Text editor**: The main area where the sketch code is written.
- Message area**: An empty area below the text editor.
- Console**: An empty area at the bottom of the interface.

```
void setup() {
    size(100, 100);
    noLoop();
}

void draw() {
    diagonals(40, 90);
    diagonals(60, 62);
    diagonals(20, 40);
}

void diagonals(int x, int y) {
    line(x, y, x+20, y-40);
    line(x+10, y, x+30, y-40);
    line(x+20, y, x+40, y-40);
}
```

Menu

Toolbar

Tabs

Text editor

Message area

Console

Processing Development Environment (PDE)

Use the PDE to create programs. Write the code in the text editor and use the buttons in the toolbar to run, save, and export the code.

Using Processing

Download, Install

The Processing software can be downloaded from the Processing website. Using a Web browser, navigate to www.processing.org/download and click on the link for your computer's operating system. The Processing software is available for Linux, Macintosh, and Windows. The most up-to-date installation instructions for your operating system are linked from this page.

Environment

The Processing Development Environment (PDE) consists of a simple text editor for writing code, a message area, a text console, tabs for managing files, a toolbar with buttons for common actions, and a series of menus. When programs are run, they open in a new window called the display window.

Pieces of software written using Processing are called sketches. These sketches are written in the text editor. It has features for cutting/pasting and for searching/replacing text. The message area gives feedback while saving and exporting and also displays errors. The console displays text output by Processing programs including complete error messages and text output from programs with the `print()` and `println()` functions. The toolbar buttons allow you to run and stop programs, create a new sketch, open, save, and export.

Run	Compiles the code, opens a display window, and runs the program inside.
Stop	Terminates a running program, but does not close the display window.
New	Creates a new sketch.
Open	Provides a menu with options to open files from the sketchbook, open an example, or open a sketch from anywhere on your computer or network.
Save	Saves the current sketch to its current location. If you want to give the sketch a different name, select "Save As" from the File menu.
Export	Exports the current sketch as a Java applet embedded in an HTML file. The folder containing the files is opened. Click on the <code>index.html</code> file to load the software in the computer's default Web browser.

The menus provide the same functionality as the toolbar in addition to actions for file management and opening reference materials.

File	Commands to manage and export files
Edit	Controls for the text editor (Undo, Redo, Cut, Copy, Paste, Find, Replace, etc.)

Sketch	Commands to run and stop programs and to add media files and code libraries.
Tools	Tools to assist in using Processing (automated code formatting, creating fonts, etc.)
Help	Reference files for the environment and language

All Processing projects are called sketches. Each sketch has its own folder. The main program file for each sketch has the same name as the folder and is found inside. For example, if the sketch is named *Sketch_123*, the folder for the sketch will be called *Sketch_123* and the main file will be called *Sketch_123.pde*. The PDE file extension stands for the Processing Development Environment.

A sketch folder sometimes contains other folders for media files and code libraries. When a font or image is added to a sketch by selecting “Add File” from the Sketch menu, a *data* folder is created. You can also add files to your Processing sketch by dragging them into the text editor. Image and sound files dragged into the application window will automatically be added to the current sketch’s *data* folder. All images, fonts, sounds, and other data files loaded in the sketch must be in this folder. Sketches are stored in the Processing folder, which will be in different places on your computer or network depending on whether you use PC, Mac, or Linux and on how the preferences are set. To locate this folder, select the “Preferences” option from the File menu (or from the Processing menu on the Mac) and look for the “Sketchbook location.”

It is possible to have multiple files in a single sketch. These can be Processing text files (with the extension *.pde*) or Java files (with the extension *.java*). To create a new file, click on the arrow button to the right of the file tabs. This button enables you to create, delete, and rename the files that comprise the current sketch. You can write functions and classes in new PDE files and you can write any Java code in files with the JAVA extension. Working with multiple files makes it easier to reuse code and to separate programs into small subprograms. This is discussed in more detail in Structure 4 (p. 395).

Export

The export feature packages a sketch to run within a Web browser. When code is exported from Processing it is converted into Java code and then compiled as a Java applet. When a project is exported, a series of files are written to a folder named *applet* that is created within the sketch folder. All files from the sketch folder are exported into a single Java Archive (JAR) file with the same name as the sketch. For example, if the sketch is named *Sketch_123*, the exported file will be called *Sketch_123.jar*. The *applet* folder contains the following:

index.html	HTML file with the applet embedded and a link to the source code and the Processing homepage. Double-click this file to open it in the default Web browser.
Sketch_123.jar	Java Archive containing all necessary files for the sketch to run. Includes the Processing core classes, those written for the sketch, and all included media files from the data folder such as images, fonts, and sounds.

Sketch_123.java	The JAVA file generated by the preprocessor from the PDE file. This is the actual file that is compiled into the applet by the Java compiler used in Processing.
Sketch_123.pde	The original program file. It is linked from the index.html file.
loading.gif	An image file displayed while the program is loading in a Web browser.

Every time a sketch is exported, the contents of the *applet* folder are deleted and the files are written from scratch. Any changes previously made to the *index.html* file are lost. Media files not needed for the applet should be deleted from the *data* folder before it is exported to keep the file size small. For example, if there are unused images in the *data* folder, they will be added to the JAR file, thus needlessly increasing its size.

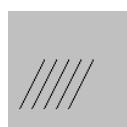
In addition to exporting Java applets for the Web, Processing can also export Java applications for the Linux, Macintosh, and Windows platforms. When “Export Application” is selected from the File menu, folders will be created for each of the operating systems specified in the Preferences. Each folder contains the application, the source code for the sketch, and all required libraries for a specific platform.

Additional and updated information about the Processing environment is available at www.processing.org/reference/environment or by selecting the “Environment” item from the Help menu of the Processing application.

Example walk-through

A Processing program can be as short as one line of code and as long as thousands of lines. This scalability is one of the most important aspects of the language. The following example walk-through presents the modest goal of animating a sequence of diagonal lines as a means to explore some of the basic components of the Processing language. If you are new to programming, some of the terminology and symbols in this section will be unfamiliar. This walk-through is a condensed overview of the entire book, utilizing ideas and techniques that are covered in detail later. Try running these programs inside the Processing application to better understand what the code is doing.

Processing was designed to make it easy to draw graphic elements such as lines, ellipses, and curves in the display window. These shapes are positioned with numbers that define their coordinates. The position of a line is defined by four numbers, two for each endpoint. The parameters used inside the `line()` function determine the position where the line appears. The origin of the coordinate system is in the upper-left corner, and numbers increase right and down. Coordinates and drawing different shapes are discussed on pages 23–30.



```
line(10, 80, 30, 40); // Left line
line(20, 80, 40, 40);
line(30, 80, 50, 40); // Middle line
line(40, 80, 60, 40);
line(50, 80, 70, 40); // Right line
```

0-01

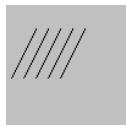
The visual attributes of shapes are controlled with other code elements that set color and gray values, the width of lines, and the quality of the rendering. Drawing attributes are discussed on pages 31–35.



```
background(0);           // Set the black background
stroke(255);            // Set line value to white
strokeWeight(5);         // Set line width to 5 pixels
smooth();                // Smooth line edges
line(10, 80, 30, 40);   // Left line
line(20, 80, 40, 40);
line(30, 80, 50, 40);   // Middle line
line(40, 80, 60, 40);
line(50, 80, 70, 40);   // Right line
```

0-02

A variable, such as `x`, represents a value; this value replaces the symbol `x` when the code is run. One variable can then control many features of the program. Variables are introduced on page 37-41.



```
int x = 5;    // Set the horizontal position
int y = 60;   // Set the vertical position
line(x, y, x+20, y-40); // Line from [5,60] to [25,20]
line(x+10, y, x+30, y-40); // Line from [15,60] to [35,20]
line(x+20, y, x+40, y-40); // Line from [25,60] to [45,20]
line(x+30, y, x+50, y-40); // Line from [35,60] to [55,20]
line(x+40, y, x+60, y-40); // Line from [45,60] to [65,20]
```

0-03

Adding more structure to a program opens further possibilities. The `setup()` and `draw()` functions make it possible for the program to run continuously—this is required to create animation and interactive programs. The code inside `setup()` runs once when the program first starts, and the code inside `draw()` runs continuously. One image frame is drawn to the display window at the end of each loop through `draw()`.

In the following example, the variable `x` is declared as a global variable, meaning it can be assigned and accessed anywhere in the program. The value of `x` increases by 1 each frame, and because the position of the lines is controlled by `x`, they are drawn to a different location each time the value changes. This moves the lines to the right.

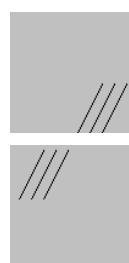
Line 14 in the code is an `if` structure. It contains a relational expression comparing the variable `x` to the value `100`. When the expression is `true`, the code inside the block (the code between the `{` and `}` associated with the `if` structure) runs. When the relational expression is `false`, the code inside the block does not run. When the value of `x` becomes greater than `100`, the line of code inside the block sets the variable `x` to `-40`, causing the lines to jump to the left edge of the window. The details of `draw()` are discussed on pages 173–175, programming animation is discussed on pages 315–320, and the `if` structure is discussed on pages 53–56.



```
int x = 0; // Set the horizontal position
int y = 55; // Set the vertical position
void setup() {
    size(100, 100); // Set the window to 100 x 100 pixels
}
void draw() {
    background(204);
    line(x, y, x+20, y-40); // Left line
    line(x+10, y, x+30, y-40); // Middle line
    line(x+20, y, x+40, y-40); // Right line
    x = x + 1; // Add 1 to x
    if (x > 100) { // If x is greater than 100,
        x = -40; // assign -40 to x
    }
}
```

0-04

When a program is running continuously, Processing stores data from input devices such as the mouse and keyboard. This data can be used to affect what is happening in the display window. Programs that respond to the mouse are discussed on pages 205–244.



```
void setup() {
    size(100, 100);
}
void draw() {
    background(204);
    // Assign the horizontal value of the cursor to x
    float x = mouseX;
    // Assign the vertical value of the cursor to y
    float y = mouseY;
    line(x, y, x+20, y-40);
    line(x+10, y, x+30, y-40);
    line(x+20, y, x+40, y-40);
}
```

0-05

A function is a set of code within a program that performs a specific task. Functions are powerful programming tools that make programs easier to read and change. The `diagonals()` function in the following example was written to draw a sequence of three diagonal lines each time it is run inside `draw()`. Two *parameters*, the numbers in the parentheses after the function name, set the position of the lines. These numbers are passed into the function definition on line 12 and are used as the values for the variables `x` and `y` in lines 13–15. Functions are discussed in more depth on pages 181–196.



```
void setup() {  
    size(100, 100);  
    noLoop();  
}  
  
void draw() {  
    diagonals(40, 90);  
    diagonals(60, 62);  
    diagonals(20, 40);  
}  
  
void diagonals(int x, int y) {  
    line(x, y, x+20, y-40);  
    line(x+10, y, x+30, y-40);  
    line(x+20, y, x+40, y-40);  
}
```

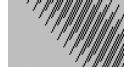
0-06

The variables used in the previous programs each store one data element. If we want to have 20 groups of lines on screen, it will require 40 variables: 20 for the horizontal positions and 20 for the vertical positions. This can make programming tedious and can make programs difficult to read. Instead of using multiple variable names, we can use *arrays*. An array can store a list of data elements as a single name. A *for* structure can be used to cycle through each array element in sequence. Arrays are discussed on pages 301–313, and the *for* structure is discussed on pages 61–68.



```
int num = 20;  
int[] dx = new int[num]; // Declare and create an array  
int[] dy = new int[num]; // Declare and create an array  
  
void setup() {  
    size(100, 100);  
    for (int i = 0; i < num; i++) {  
        dx[i] = i * 5;  
        dy[i] = 12 + (i * 6);  
    }  
}  
  
void draw() {  
    background(204);  
    for (int i = 0; i < num; i++) {  
        dx[i] = dx[i] + 1;  
        if (dx[i] > 100) {  
            dx[i] = -100;  
        }  
    }  
}
```

0-07



```

        diagonals(dx[i], dy[i]);
    }
}

void diagonals(int x, int y) {
    line(x, y, x+20, y-40);
    line(x+10, y, x+30, y-40);
    line(x+20, y, x+40, y-40);
}

```

0-07
cont.

Object-oriented programming is a way of structuring code into *objects*, units of code that contain both data and functions. This style of programming makes a strong connection between groups of data and the functions that act on this data. The `diagonals()` function can be expanded by making it part of a *class* definition. Objects are created using the class as a template. The variables for positioning the lines and setting their drawing attributes then move inside the class definition to be more closely associated with drawing the lines. Object-oriented programming is discussed further on pages 395–411.



```

Diagonals da, db;

void setup() {
    size(100, 100);
    smooth();
    // Inputs: x, y, speed, thick, gray
    da = new Diagonals(0, 80, 1, 2, 0);
    db = new Diagonals(0, 55, 2, 6, 255);
}

void draw() {
    background(204);
    da.update();
    db.update();
}

class Diagonals {
    int x, y, speed, thick, gray;
    Diagonals(int xpos, int ypos, int s, int t, int g) {
        x = xpos;
        y = ypos;
        speed = s;
        thick = t;
        gray = g;
    }
}

```

0-08

```
void update() {  
    strokeWeight(thick);  
    stroke(gray);  
    line(x, y, x+20, y-40);  
    line(x+10, y, x+30, y-40);  
    line(x+20, y, x+40, y-40);  
    x = x + speed;  
    if (x > 100) {  
        x = -100;  
    }  
}  
}
```

0-08
cont.

This short walk-through serves to introduce, but not fully explain, some of the core concepts explored in this text. Many key ideas of working with software were mentioned only briefly and others were omitted. Each topic is covered in depth later in the book.

Reference

The reference for the Processing language complements the text in this book. We advise keeping the reference open and consulting it while programming. The reference can be accessed by selecting the “Reference” option from the Help menu within Processing. It’s also available online at www.processing.org/reference. The reference can also be accessed within the text window. Highlight a word, right-click (or Ctrl-click in Mac OS X), and select “Find in Reference” from the menu that appears. You can also select “Find in Reference” from the Help menu. There are two versions of the Processing reference. The Abridged Reference lists the elements of the Processing language introduced in this book, and the Complete Reference documents additional features.

Structure 1: Code Elements

This unit introduces the most basic elements and vocabulary for writing software.

Syntax introduced:

```
// (comment), /* */ (multiline comment)  
“;” (statement terminator), “,” (comma)  
print(), println()
```

Creating software is an act of *writing*. Before starting to write code, it's important to acknowledge the difference between writing a computer program and writing an Email or an essay. Writing in a human language allows the author to utilize the ambiguity of words and to have great flexibility in constructing phrases. These techniques allow multiple interpretations of a single text and give each author a unique voice. Each computer program also reveals the style of its author, but there is far less room for ambiguity. While people can interpret vague meanings and can usually disregard poor grammar, computers cannot. Some of the linguistic details of writing code are discussed here to prevent early frustration. If you keep these details in mind as you begin to program, they will gradually become habitual. This unit presents variations of a simple program that sets the size and background color of the display window, demonstrating some of the most basic elements of writing code with Processing.

Comments

Comments are ignored by the computer but are important for people. They let you write notes to yourself and to others who read your programs. Because programs use symbols and arcane notation to describe complex procedures, it is often difficult to remember how individual parts of a program work. Good comments serve as reminders when you revisit a program and explain your thoughts to others reading the code. Commented sections appear in a different color than the rest of the code. This program explains how comments work:

```
// Two forward slashes are used to denote a comment.  
// All text on the same line is a part of the comment.  
// There must be no spaces between the slashes. For example,  
// the code "/ /" is not a comment and will cause an error  
  
// If you want to have a comment that is many  
// lines long, you may prefer to use the syntax for a  
// multiline comment
```

1-01

```

/*
A forward slash followed by an asterisk allows the
comment to continue until the opposite
*/
// All letters and symbols that are not comments are translated
// by the compiler. Because the following lines are not comments,
// they are run and draw a display window of 200 x 200 pixels
size(200, 200);
background(102);

```

1-01
cont.

Functions

Functions allow you to draw shapes, set colors, calculate numbers, and to execute many other types of actions. A function's name is usually a lowercase word followed by parentheses. The comma-separated elements between the parentheses are called parameters, and they affect the way the function works. Some functions have no parameters and others have many. This program demonstrates the `size()` and `background()` functions.

```

// The size function has two parameters. The first sets the width
// of the display window and the second sets the height
size(200, 200);

```

1-02

```

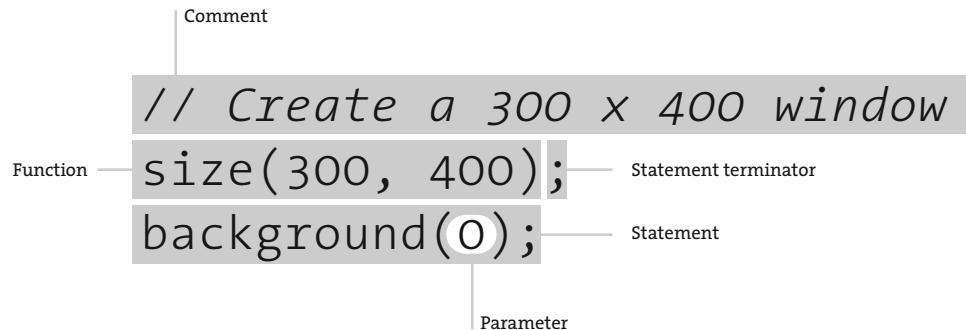
// This version of the background function has one parameter.
// It sets the gray value for the background of the display window
// in the range of 0 (black) to 255 (white)
background(102);

```

Expressions, Statements

Using an analogy to human languages, a software expression is like a phrase. Software expressions are often combinations of operators such as `+`, `*`, and `/` that operate on the values to their left and right. A software expression can be as basic as a single number or can be a long combination of elements. An expression always has a value, determined by evaluating its contents.

<i>Expression</i>	<i>Value</i>
5	5
122.3+3.1	125.4
((3+2)*-10) + 1	-49



Anatomy of a program

Every program is composed of different language elements. These elements work together to describe the intentions of the programmer so they can be interpreted by a computer. The anatomy of a more complicated program is shown on page 176.

Expressions can also compare two values with operators such as `>` (greater than) and `<` (less than). These comparisons are evaluated as `true` or `false`.

Expression	Value
<code>6 > 3</code>	<code>true</code>
<code>54 < 50</code>	<code>false</code>

A set of expressions together create a statement, the programming equivalent of a sentence. It's a complete unit that ends with the statement terminator, the programming equivalent of a period. In the Processing language, the statement terminator is a semicolon.

Just as there are different types of sentences, there are different types of statements. A statement can define a variable, assign a variable, run a function, or construct an object. Each will be explained in more detail later, but examples are shown here:

```

size(200, 200); // Runs the size() function
int x;           // Declares a new variable x
x = 102;         // Assigns the value 102 to the variable x
background(x);  // Runs the background() function

```

1-03

Omitting the semicolon at the end of a statement, a very common mistake, will result in an error message, and the program will not run.

Case sensitivity

In written English, some words are capitalized and others are not. Proper nouns like Ohio and John and the first letter of every sentence are capitalized, while most other words are lowercase. In many programming languages, some parts of the language must be capitalized and others must be lowercase. Processing differentiates between uppercase and lowercase characters; therefore, writing “Size” when you mean to write “size” creates an error. You must be exacting in adhering to the capitalization rules.

```
size(200, 200);  
Background(102); // ERROR! The B in "background" is capitalized
```

1-04

Whitespace

In many programming languages, including Processing, there can be an arbitrary amount of space between the elements of a program. Unlike the rigorous syntax of statement terminators, spacing does not matter. The following two lines of code are a standard way of writing a program:

```
size(200, 200);  
background(102);
```

1-05

However, the whitespace between the code elements can be set to any amount and the program will run exactly the same way:

```
size  
( 200,  
    200) ;  
background ( 102 )  
;
```

1-06

Console

When software runs, the computer performs operations at a rate too fast to perceive with human eyes. Because it is important to understand what is happening inside the machine, the functions `print()` and `println()` can be used to display data while a program is running. These functions don’t send pages to a printer, but instead write text to the console (pp. 8, 9). The console can be used to display a variable, confirm an event, or check incoming data from an external device. Such uses might not seem clear now, but they will reveal themselves over the course of this book. Like comments, `print()` and `println()` can clarify the intentions and execution of computer programs.

```
// To print text to the screen, place the desired output in quotes      1-07
println("Processing..."); // Prints "Processing..." to the console

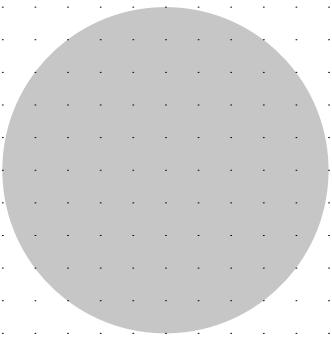
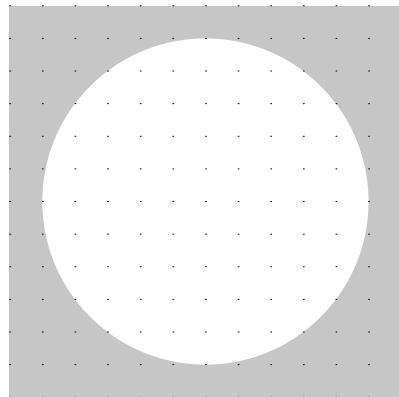
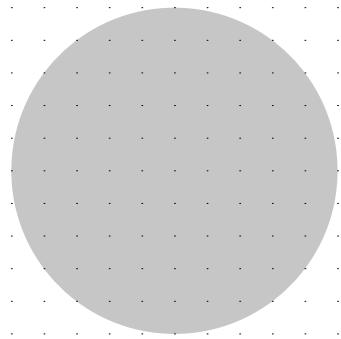
// To print the value of a variable, rather than its name,
// don't put the name of the variable in quotes
int x = 20;
println(x); // Prints "20" to the console

// While println() moves to the next line after the text
// is output, print() does not
print("10");
println("20"); // Prints "1020" to the console
println("30"); // Prints "30" to the console

// The "+" operator can be used for combining multiple text
// elements into one line
int x2 = 20;
int y2 = 80;
println(x2 + " : " + y2); // Prints "20 : 80" to the message window
```

Exercises

1. Write comments in the text area explaining a piece of software you would like to write.
2. Write a program to make a 640×480 pixel display window with a black background.
3. Use print() and println() to write some text to the console.



Shape 1: Coordinates, Primitives

This unit introduces the coordinate system of the display window and a variety of geometric shapes.

Syntax introduced:

```
size(), point(), line(), triangle(), quad(), rect(), ellipse(), bezier()  
background(), fill(), stroke(), noFill(), noStroke()  
strokeWeight(), strokeCap(), strokeJoin()  
smooth(), noSmooth(), ellipseMode(), rectMode()
```

Drawing a shape with code can be difficult because every aspect of its location must be specified with a number. When you're accustomed to drawing with a pencil or moving shapes around on a screen with a mouse, it can take time to start thinking in relation to the screen's strict coordinate grid. The mental gap between seeing a composition on paper or in your mind and translating it into code notation is wide, but easily bridged.

Coordinates

Before making a drawing, it's important to think about the dimensions and qualities of the surface to which you'll be drawing. If you're making a drawing on paper, you can choose from myriad utensils and papers. For quick sketching, newsprint and charcoal are appropriate. For a refined drawing, a smooth handmade paper and range of pencils may be preferred. In contrast, when you are drawing to a computer's screen, the primary options available are the size of the window and the background color.

A computer screen is a grid of small light elements called pixels. Screens come in many sizes and resolutions. We have three different types of computer screens in our studios, and they all have a different number of pixels. The laptops have 1,764,000 pixels (1680 wide × 1050 high), the flat panels have 1,310,720 pixels (1280 wide × 1024 high), and the older monitors have 786,432 pixels (1024 wide × 768 high). Millions of pixels may sound like a vast quantity, but they produce a poor visual resolution compared to physical media such as paper. Contemporary screens have a resolution around 100 dots per inch, while many modern printers provide more than 1000 dots per inch. On the other hand, paper images are fixed, but screens have the advantage of being able to change their image many times per second.

Processing programs can control all or a subset of the screen's pixels. When you click the Run button, a display window opens and allows access to reading and writing the pixels within. It's possible to create images larger than the screen, but in most cases you'll make a window the size of the screen or smaller.

The size of the display window is controlled with the `size()` function:

```
size(width, height)
```

The `size()` function has two parameters: the first sets the width of the window and the second sets its height.



```
// Draw the display window 120 pixels  
// wide and 200 pixels high  
size(120, 200);
```

2-01



```
// Draw the display window 320 pixels  
// wide and 240 pixels high  
size(320, 240);
```

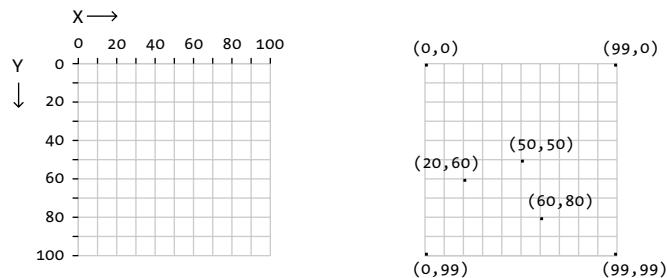
2-02



```
// Draw the display window 200 pixels  
// wide and 200 pixels high  
size(200, 200);
```

2-03

A position on the screen is comprised of an x-coordinate and a y-coordinate. The x-coordinate is the horizontal distance from the origin and the y-coordinate is the vertical distance. In Processing, the origin is the upper-left corner of the display window and coordinate values increase down and to the right. The image on the left shows the coordinate system, and the image on the right shows a few coordinates placed on the grid:



A position is written as the x-coordinate value followed by the y-coordinate, separated with a comma. The notation for the origin is (0,0), the coordinate (50,50) has an x-coordinate of 50 and a y-coordinate of 50, and the coordinate (20,60) is an x-coordinate of 20 and a y-coordinate of 60. If the size of the display window is 100 pixels wide and 100 pixels high, (0,0) is the pixel in the upper-left corner, (99,0) is the pixel in the upper-right corner, (0,99) is the pixel in the lower-left corner, and (99,99) is the pixel in the lower-right corner. This becomes clearer when we look at examples using point().

Primitive shapes

A point is the simplest visual element and is drawn with the point() function:

```
point(x, y)
```

This function has two parameters: the first is the x-coordinate and the second is the y-coordinate. Unless specified otherwise, a point is the size of a single pixel.

```
// Points with the same X and Y parameters  
// form a diagonal line from the  
// upper-left corner to the lower-right corner  
point(20, 20);  
point(30, 30);  
point(40, 40);  
point(50, 50);  
point(60, 60);
```

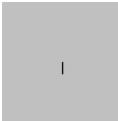
2-04



```
// Points with the same Y parameter have the  
// same distance from the top and bottom  
// edges of the frame  
point(50, 30);  
point(55, 30);  
point(60, 30);  
point(65, 30);  
point(70, 30);
```



```
// Points with the same X parameter have the  
// same distance from the left and right  
// edges of the frame  
point(70, 50);  
point(70, 55);  
point(70, 60);  
point(70, 65);  
point(70, 70);
```



```
// Placing a group of points next to one  
// another creates a line  
point(50, 50);  
point(50, 51);  
point(50, 52);  
point(50, 53);  
point(50, 54);  
point(50, 55);  
point(50, 56);  
point(50, 57);  
point(50, 58);  
point(50, 59);
```



```
// Setting points outside the display  
// area will not cause an error,  
// but the points won't be visible  
point(-500, 100);  
point(400, -600);  
point(140, 2500);  
point(2500, 100);
```

While it's possible to draw any line as a series of points, lines are more simply drawn with the `line()` function. This function has four parameters, two for each endpoint:

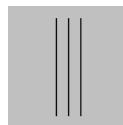
```
line(x1, y1, x2, y2)
```

The first two parameters set the position where the line starts and the last two set the position where the line stops.



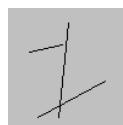
```
// When the y-coordinates for a line are the  
// same, the line is horizontal  
line(10, 30, 90, 30);  
line(10, 40, 90, 40);  
line(10, 50, 90, 50);
```

2-09



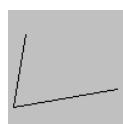
```
// When the x-coordinates for a line are the  
// same, the line is vertical  
line(40, 10, 40, 90);  
line(50, 10, 50, 90);  
line(60, 10, 60, 90);
```

2-10



```
// When all four parameters are different,  
// the lines are diagonal  
line(25, 90, 80, 60);  
line(50, 12, 42, 90);  
line(45, 30, 18, 36);
```

2-11



```
// When two lines share the same point they connect  
line(15, 20, 5, 80);  
line(90, 65, 5, 80);
```

2-12

The `triangle()` function draws triangles. It has six parameters, two for each point:

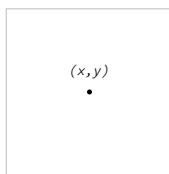
```
triangle(x1, y1, x2, y2, x3, y3)
```

The first pair defines the first point, the middle pair the second point, and the last pair the third point. Any triangle can be drawn by connecting three lines, but the `triangle()` function makes it possible to draw a filled shape. Triangles of all shapes and sizes can be created by changing the parameter values.

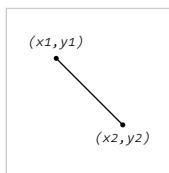


```
triangle(60, 10, 25, 60, 75, 65); // Filled triangle  
line(60, 30, 25, 80); // Outlined triangle edge  
line(25, 80, 75, 85); // Outlined triangle edge  
line(75, 85, 60, 30); // Outlined triangle edge
```

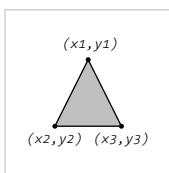
2-13



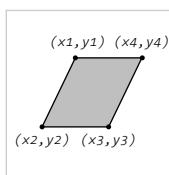
`point(x, y)`



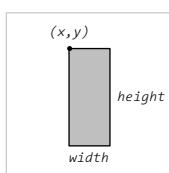
`line(x1, y1, x2, y2)`



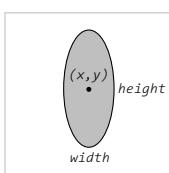
`triangle(x1, y1, x2, y2, x3, y3)`



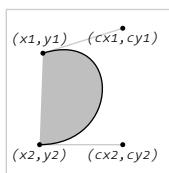
`quad(x1, y1, x2, y2, x3, y3, x4, y4)`



`rect(x, y, width, height)`



`ellipse(x, y, width, height)`



`bezier(x1, y1, cx1, cy1, cx2, cy2, x2, y2)`

Geometry primitives

Processing has seven functions to assist in making simple shapes. These images show the format for each. Replace the parameters with numbers to use them within a program. These functions are demonstrated in codes 2-04 to 2-22.



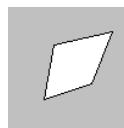
```
triangle(55, 9, 110, 100, 85, 100);
triangle(55, 9, 85, 100, 75, 100);
triangle(-1, 46, 16, 34, -7, 100);
triangle(16, 34, -7, 100, 40, 100);
```

2-14

The `quad()` function draws a quadrilateral, a four-sided polygon. The function has eight parameters, two for each point.

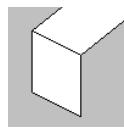
```
quad(x1, y1, x2, y2, x3, y3, x4, y4)
```

Changing the parameter values can yield rectangles, squares, parallelograms, and irregular quadrilaterals.



```
quad(38, 31, 86, 20, 69, 63, 30, 76);
```

2-15



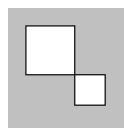
```
quad(20, 20, 20, 70, 60, 90, 60, 40);
quad(20, 20, 70, -20, 110, 0, 60, 40);
```

2-16

Drawing rectangles and ellipses works differently than the shapes previously introduced. Instead of defining each point, the four parameters set the position and the dimensions of the shape. The `rect()` function draws a rectangle:

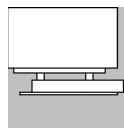
```
rect(x, y, width, height)
```

The first two parameters set the location of the upper-left corner, the third sets the width, and the fourth sets the height. Use the same value for the `width` and `height` parameters to draw a square.



```
rect(15, 15, 40, 40); // Large square
rect(55, 55, 25, 25); // Small square
```

2-17



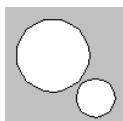
```
rect(0, 0, 90, 50);
rect(5, 50, 75, 4);
rect(24, 54, 6, 6);
rect(64, 54, 6, 6);
rect(20, 60, 75, 10);
rect(10, 70, 80, 2);
```

2-18

The `ellipse()` function draws an ellipse in the display window:

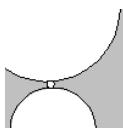
```
ellipse(x, y, width, height)
```

The first two parameters set the location of the center of the ellipse, the third sets the width, and the fourth sets the height. Use the same value for the `width` and `height` parameters to draw a circle.



```
ellipse(40, 40, 60, 60); // Large circle  
ellipse(75, 75, 32, 32); // Small circle
```

2-19



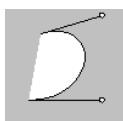
```
ellipse(35, 0, 120, 120);  
ellipse(38, 62, 6, 6);  
ellipse(40, 100, 70, 70);
```

2-20

The `bezier()` function can draw lines that are not straight. A Bézier curve is defined by a series of control points and anchor points. A curve is drawn between the anchor points, and the control points define its shape:

```
bezier(x1, y1, cx1, cy1, cx2, cy2, x2, y2)
```

The function requires eight parameters to set four points. The curve is drawn between the first and fourth points, and the control points are defined by the second and third points. In software that uses Bézier curves, such as Adobe Illustrator, the control points are represented by the tiny handles that protrude from the edge of a curve.



```
bezier(32, 20, 80, 5, 80, 75, 30, 75);  
// Draw the control points  
line(32, 20, 80, 5);  
ellipse(80, 5, 4, 4);  
line(80, 75, 30, 75);  
ellipse(80, 75, 4, 4);
```

2-21

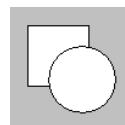


```
bezier(85, 20, 40, 10, 60, 90, 15, 80);  
// Draw the control points  
line(85, 20, 40, 10);  
ellipse(40, 10, 4, 4);  
line(60, 90, 15, 80);  
ellipse(60, 90, 4, 4);
```

2-22

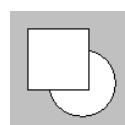
Drawing order

The order in which shapes are drawn in the code defines which shapes appear on top of others in the display window. If a rectangle is drawn in the first line of a program, it is drawn behind an ellipse drawn in the second line of the program. Reversing the order places the rectangle on top.



```
rect(15, 15, 50, 50);      // Bottom  
ellipse(60, 60, 55, 55);  // Top
```

2-23



```
ellipse(60, 60, 55, 55); // Bottom  
rect(15, 15, 50, 50);   // Top
```

2-24

Gray values

The examples so far have used the default light-gray background, black lines, and white shapes. To change these default values, it's necessary to introduce additional syntax. The `background()` function sets the color of the display window with a number between 0 and 255. This range may be awkward if you're not familiar with drawing software on the computer. The value 255 is white and the value 0 is black, with a range of gray values in between. If no background value is defined, the default value 204 (light gray) is used.



```
background(0);
```

2-25



```
background(124);
```

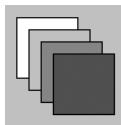
2-26



```
background(230);
```

2-27

The `fill()` function sets the fill value of shapes, and the `stroke()` function sets the outline value of the drawn shapes. If no fill value is defined, the default value of 255 (white) is used. If no stroke value is defined, the default value of 0 (black) is used.



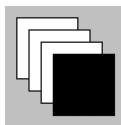
```
rect(10, 10, 50, 50);
fill(204); // Light gray
rect(20, 20, 50, 50);
fill(153); // Middle gray
rect(30, 30, 50, 50);
fill(102); // Dark gray
rect(40, 40, 50, 50);
```

2-28



```
background(0);
rect(10, 10, 50, 50);
stroke(102); // Dark gray
rect(20, 20, 50, 50);
stroke(153); // Middle gray
rect(30, 30, 50, 50);
stroke(204); // Light gray
rect(40, 40, 50, 50);
```

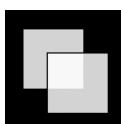
2-29



```
fill(255); // White
rect(10, 10, 50, 50);
rect(20, 20, 50, 50);
rect(30, 30, 50, 50);
fill(0); // Black
rect(40, 40, 50, 50);
```

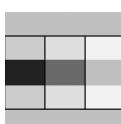
2-30

An optional second parameter to `fill()` and `stroke()` controls transparency. Setting the parameter to 255 makes the shape entirely opaque, and 0 is totally transparent:



```
background(0);
fill(255, 220);
rect(15, 15, 50, 50);
rect(35, 35, 50, 50);
```

2-31



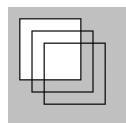
```
fill(0);
rect(0, 40, 100, 20);
fill(255, 51); // Low opacity
rect(0, 20, 33, 60);
fill(255, 127); // Medium opacity
```

2-32

```
rect(33, 20, 33, 60);
fill(255, 204); // High opacity
rect(66, 20, 33, 60);
```

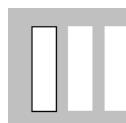
2-32
cont.

The stroke and fill of a shape can be disabled. The `noFill()` function stops Processing from filling shapes, and the `noStroke()` function stops lines from being drawn and shapes from having outlines. If `noFill()` and `noStroke()` are both used, nothing will be drawn to the screen.



```
rect(10, 10, 50, 50);
noFill(); // Disable the fill
rect(20, 20, 50, 50);
rect(30, 30, 50, 50);
```

2-33



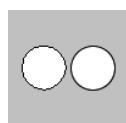
```
rect(20, 15, 20, 70);
noStroke(); // Disable the stroke
rect(50, 15, 20, 70);
rect(80, 15, 20, 70);
```

2-34

Setting color fill and stroke values is introduced in Color 1 (p. 85).

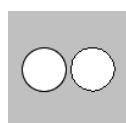
Drawing attributes

In addition to changing the fill and stroke values of shapes, it's also possible to change attributes of the geometry. The `smooth()` and `noSmooth()` functions enable and disable smoothing (also called antialiasing). Once these functions are used, all shapes drawn afterward are affected. If `smooth()` is used first, using `noSmooth()` cancels the setting, and vice versa.



```
ellipse(30, 48, 36, 36);
smooth();
ellipse(70, 48, 36, 36);
```

2-35



```
smooth();
ellipse(30, 48, 36, 36);
noSmooth();
ellipse(70, 48, 36, 36);
```

2-36

Line attributes are controlled by the `strokeWeight()`, `strokeCap()`, and `strokeJoin()` functions. The `strokeWeight()` function has one numeric parameter that sets the thickness of all lines drawn after the function is used. The `strokeCap()` function requires one parameter that can be either `ROUND`, `SQUARE`, or `PROJECT`.

ROUND makes round endpoints, and SQUARE squares them. PROJECT is a mix of the two that extends a SQUARE endpoint by the radius of the line. The strokeJoin() function has one parameter that can be either BEVEL, MITER, or ROUND. These parameters determine the way line segments or the stroke around a shape connects. BEVEL causes lines to join with squared corners, MITER is the default and joins lines with pointed corners, and ROUND creates a curve.



```
smooth();  
line(20, 20, 80, 20); // Default line weight of 1  
strokeWeight(6);  
line(20, 40, 80, 40); // Thicker line  
strokeWeight(18);  
line(20, 70, 80, 70); // Beastly line
```

2-37



```
smooth();  
strokeWeight(12);  
strokeCap(ROUND);  
line(20, 30, 80, 30); // Top line  
strokeCap(SQUARE);  
line(20, 50, 80, 50); // Middle line  
strokeCap(PROJECT);  
line(20, 70, 80, 70); // Bottom line
```

2-38



```
smooth();  
strokeWeight(12);  
strokeJoin(BEVEL);  
rect(12, 33, 15, 33); // Left shape  
strokeJoin(MITER);  
rect(42, 33, 15, 33); // Middle shape  
strokeJoin(ROUND);  
rect(72, 33, 15, 33); // Right shape
```

2-39

Shape 2 (p. 69) and Shape 3 (p. 197) show how to draw shapes with more flexibility.

Drawing modes

By default, the parameters for ellipse() set the x-coordinate of the center, the y-coordinate of the center, the width, and the height. The ellipseMode() function changes the way these parameters are used to draw ellipses. The ellipseMode() function requires one parameter that can be either CENTER, RADIUS, CORNER, or CORNERS. The default mode is CENTER. The RADIUS mode also uses the first and second parameters of ellipse() to set the center, but causes the third parameter to set half of

the width and the fourth parameter to set half of the height. The CORNER mode makes `ellipse()` work similarly to `rect()`. It causes the first and second parameters to position the upper-left corner of the rectangle that circumscribes the ellipse and uses the third and fourth parameters to set the width and height. The CORNERS mode has a similar affect to CORNER, but it causes the third and fourth parameters to `ellipse()` to set the lower-right corner of the rectangle.



```
smooth();
noStroke();
ellipseMode(RADIUS);
fill(126);
ellipse(33, 33, 60, 60); // Gray ellipse
fill(255);
ellipseMode(CORNER);
ellipse(33, 33, 60, 60); // White ellipse
fill(0);
ellipseMode(CORNERS);
ellipse(33, 33, 60, 60); // Black ellipse
```

2-40

In a similar fashion, the `rectMode()` function affects how rectangles are drawn. It requires one parameter that can be either CORNER, CORNERS, or CENTER. The default mode is CORNER, and CORNERS causes the third and fourth parameters of `rect()` to draw the corner opposite the first. The CENTER mode causes the first and second parameters of `rect()` to set the center of the rectangle and uses the third and fourth parameters as the width and height.



```
noStroke();
rectMode(CORNER);
fill(126);
rect(40, 40, 60, 60); // Gray ellipse
rectMode(CENTER);
fill(255);
rect(40, 40, 60, 60); // White ellipse
rectMode(CORNERS);
fill(0);
rect(40, 40, 60, 60); // Black ellipse
```

2-41

Exercises

1. Create a composition by carefully positioning one line and one ellipse.
2. Modify the code for exercise 1 to change the fill, stroke, and background values.
3. Create a visual knot using only Bézier curves.

Control 2: Repetition

This unit focuses on controlling the flow of programs with iterative structures.

Syntax introduced:

for

The early history of computers is the history of automating calculation. A “computer” was originally a person who was paid to calculate math by hand. What we know as a computer today emerged from machines built to automate tedious mathematical calculations. The earliest mechanical computers were calculators developed for speed and accuracy in performing repetitive calculations. Because of this heritage, computers are excellent at executing repetitive tasks accurately and quickly. Modern computers are also logic machines. Building on the work of the logicians Leibniz and Boole, modern computers use logical operations such as AND, OR, and NOT to determine which lines of code are run and which are not.

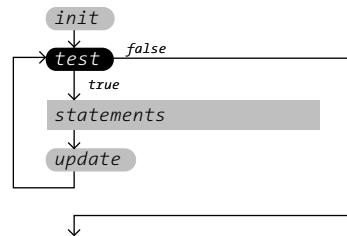
Iteration

Iterative structures are used to compact lengthy lines of repetitive code. Decreasing the length of the code can make programs easier to manage and can also help to reduce errors. The table below shows equivalent programs written without an iterative structure and with a `for` structure. The original 14 lines of code on the left are reduced to the 4 lines on the right:

Original code	Code expressed using a <code>for</code> structure
<code>size(200, 200);</code>	<code>size(200, 200);</code>
<code>line(20, 20, 20, 180);</code>	<code>for (int i = 20; i < 150; i += 10) {</code>
<code>line(30, 20, 30, 180);</code>	<code> line(i, 20, i, 180);</code>
<code>line(40, 20, 40, 180);</code>	<code>}</code>
<code>line(50, 20, 50, 180);</code>	
<code>line(60, 20, 60, 180);</code>	
<code>line(70, 20, 70, 180);</code>	
<code>line(80, 20, 80, 180);</code>	
<code>line(90, 20, 90, 180);</code>	
<code>line(100, 20, 100, 180);</code>	
<code>line(110, 20, 110, 180);</code>	
<code>line(120, 20, 120, 180);</code>	
<code>line(130, 20, 130, 180);</code>	
<code>line(140, 20, 140, 180);</code>	

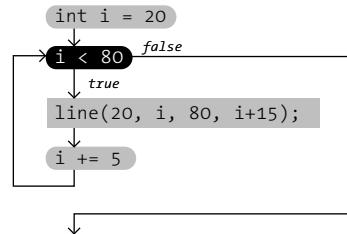
General case for structure

```
for (init; test; update) {  
    statements  
}
```



A specific for structure

```
for (int i = 20; i < 80; i += 5) {  
    line(20, i, 80, i+15);  
}
```



Repetition

The flow of a for structure shown as a diagram. These images show the central importance of the test statement in deciding whether to run the code in the block or to exit. The general case shows the generic format, and the specific case shows one example of how the format can be used within a program.

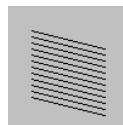
The `for` structure performs repetitive calculations and is structured like this:

```
for (init; test; update) {  
    statements  
}
```

The parentheses associated with the structure enclose three statements: `init`, `test`, and `update`. The statements inside the block are run continuously while the test evaluates to `true`. The `init` portion assigns the initial value of the variable used in the test. The `update` is used to modify the variable after each iteration through the block. A `for` structure runs in the following sequence:

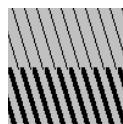
1. The `init` statement is run
2. The `test` is evaluated to `true` or `false`
3. If the `test` is `true`, continue to step 4. If the `test` is `false`, jump to step 6
4. Run the statements within the block
5. Run the `update` statement and jump to step 2
6. Exit the structure and continue running the program

The following examples demonstrate how the `for` structure is used within a program to control the way shapes are drawn to the display window.



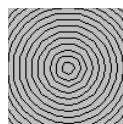
```
// The init is "int i = 20", the test is "i < 80",  
// and the update is "i += 5". Notice the semicolon  
// terminating the first two elements  
for (int i = 20; i < 80; i += 5) {  
    // This line will continue to run until "i"  
    // is greater than or equal to 80  
    line(20, i, 80, i+15);  
}
```

6-01



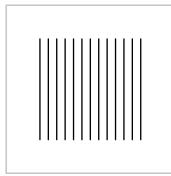
```
for (int x = -16; x < 100; x += 10) {  
    line(x, 0, x+15, 50);  
}  
strokeWeight(4);  
for (int x = -8; x < 100; x += 10) {  
    line(x, 50, x+15, 100);  
}
```

6-02



```
noFill();  
for (int d = 150; d > 0; d -= 10) {  
    ellipse(50, 50, d, d);  
}
```

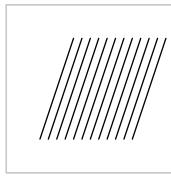
6-03



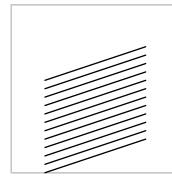
```
for (int x = 20; x <= 80; x += 5) {  
    line(x, 20, x, 80);  
}
```



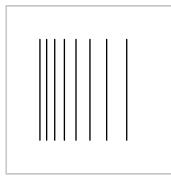
```
for (int x = 20; x <= 80; x += 5) {  
    line(20, x, 80, x);  
}
```



```
for (int x = 20; x < 80; x += 5) {  
    line(x+20, 20, x, 80);  
}
```



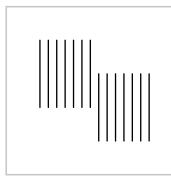
```
for (float x = 80; x > 20; x -= 5) {  
    line(20, x+20, 80, x);  
}
```



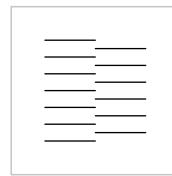
```
for (float x = 20; x < 80; x *= 1.2) {  
    line(x, 20, x, 80);  
}
```



```
for (float x = 80; x > 20; x /= 1.2) {  
    line(20, x, 80, x);  
}
```



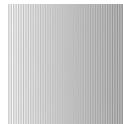
```
for (int x = 20; x <= 85; x += 5) {  
    if (x <= 50) {  
        line(x, 20, x, 60);  
    } else {  
        line(x, 40, x, 80);  
    }  
}
```



```
for (int x = 20; x <= 80; x += 5) {  
    if ((x % 10) == 0) {  
        line(20, x, 50, x);  
    } else {  
        line(50, x, 80, x);  
    }  
}
```

All for one and one for all

The `for` structure is flexible, but it always follows the rules. These examples show how it can be used to generate various patterns.



```
for (int i = 0; i < 100; i += 2) {  
    stroke(255-i);  
    line(i, 0, i, 200);  
}
```

6-04

Nested iteration

The `for` structure produces repetitions in one dimension. Nesting one of these structures into another compounds their effect, creating iteration in two dimensions. Instead of drawing 9 points and then drawing another 9 points, they combine to create 81 points; for each point drawn in the outer structure, 9 points are drawn in the inner structure. The inner structure runs through a complete cycle for each single iteration of the outer structure. In the following examples, the two dimensions are translated into x-coordinates and y-coordinates:



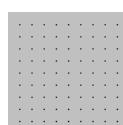
```
for (int y = 10; y < 100; y += 10) {  
    point(10, y);  
}
```

6-05



```
for (int x = 10; x < 100; x += 10) {  
    point(x, 10);  
}
```

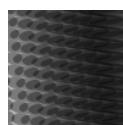
6-06



```
for (int y = 10; y < 100; y += 10) {  
    for (int x = 10; x < 100; x += 10) {  
        point(x, y);  
    }  
}
```

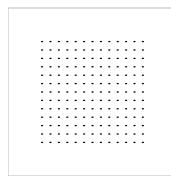
6-07

This technique is useful for creating diverse patterns and effects. The numbers produced by embedding iterative elements can be applied to color, position, size, transparency, and any other visual attribute.

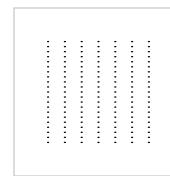


```
fill(0, 76);  
noStroke();  
smooth();  
for (int y = -10; y <= 100; y += 10) {  
    for (int x = -10; x <= 100; x += 10) {  
        ellipse(x + y/8.0, y + x/8.0, 15 + x/2, 10);  
    }  
}
```

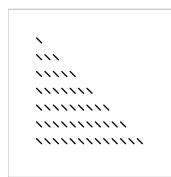
6-08



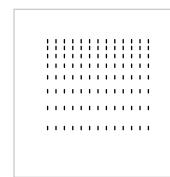
```
for (int y = 20; y <= 80; y += 5) {  
    for (int x = 20; x <= 80; x += 5) {  
        point(x, y);  
    }  
}
```



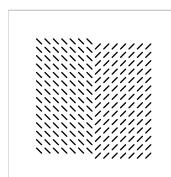
```
for (int y = 20; y <= 80; y += 3) {  
    for (int x = 20; x <= 80; x += 10) {  
        point(x, y);  
    }  
}
```



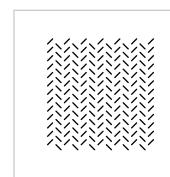
```
for (int y = 20; y <= 80; y += 10) {  
    for (int x = 20; x <= y; x += 5) {  
        line(x, y, x-3, y-3);  
    }  
}
```



```
for (float y = 20; y <= 80; y *= 1.2) {  
    for (int x = 20; x <= 80; x += 5) {  
        line(x, y, x, y-2);  
    }  
}
```



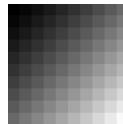
```
for (int y = 20; y <= 85; y += 5) {  
    for (int x = 20; x <= 85; x += 5) {  
        if (x <= 50) {  
            line(x, y, x-3, y-3);  
        } else {  
            line(x, y, x-3, y+3);  
        }  
    }  
}
```



```
for (int y = 20; y <= 80; y += 5) {  
    for (int x = 20; x <= 80; x += 5) {  
        if ((x % 10) == 0) {  
            line(x, y, x+3, y-3);  
        } else {  
            line(x, y, x+3, y+3);  
        }  
    }  
}
```

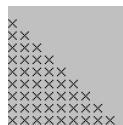
Embedding (nesting)

Embedding one for structure inside another is a highly malleable technique for drawing patterns. These examples show only a few of the possible options.



```
noStroke();
for (int y = 0; y < 100; y += 10) {
    for (int x = 0; x < 100; x += 10) {
        fill((x+y) * 1.4);
        rect(x, y, 10, 10);
    }
}
```

6-09



```
for (int y = 1; y < 100; y += 10) {
    for (int x = 1; x < y; x += 10) {
        line(x, y, x+6, y+6);
        line(x+6, y, x, y+6);
    }
}
```

6-10

Formatting code blocks

It's important to space code so the blocks are clear. The lines inside a block are typically offset to the right with spaces or tabs. When programs become longer, clearly defining the beginning and end of the block reveals the structure of the program and makes it more legible. This is the convention used in this book:

```
int x = 50;

if (x > 100) {
    line(20, 20, 80, 80);
} else {
    line(80, 20, 20, 80);
}
```

6-11

This is an alternative format that is sometimes used elsewhere:

```
int x = 50;

if (x > 100)
{
    line(20, 20, 80, 80);
}
else
{
    line(80, 20, 20, 80);
}
```

6-12

It's essential to use formatting to show the hierarchy of your code. The Processing environment will attempt basic formatting as you type, and you can use the "Auto Format" function from the Tools menu to clean up your code at any time. The `line()` function in the following code fragment is inside the `if` structure, but the spacing does not reveal this at a quick glance. Avoid formatting code like this:

```
int x = 50;
```

6-13

```
if (x > 100) {  
    line(20, 20, 80, 80); // Avoid formatting code like this  
} else { // because it makes it difficult to see  
    line(80, 20, 20, 80); // what is inside the block  
}
```

Exercises

1. Draw a regular pattern with five lines. Rewrite the code using a `for` structure.
2. Draw a dense pattern by embedding two `for` structures.
3. Combine two relational expressions with a logical operator to control the form of a pattern.

Color 1: Color by Numbers

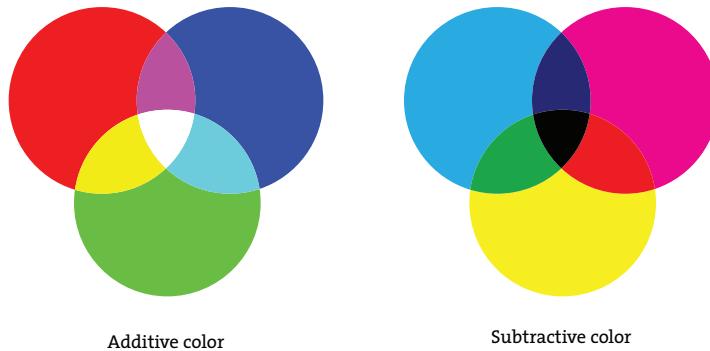
This unit introduces code elements and concepts for working with color in software.

Syntax introduced:

`color, color(), colorMode()`

When Casey and Ben studied color in school, they spent hours carefully mixing paints and applying it to sheets of paper. They cut paper into perfect squares and carefully arranged them into precise gradations from blue to orange, white to yellow, and many other combinations. Over time, they developed an intuition that allowed them to achieve a specific color value by mixing the appropriate components. Through focused labor, they learned how to isolate properties of color, understand the interactions between colors, and discuss qualities of color.

Working with color on screen is different from working with color on paper or canvas. While the same rigor applies, knowledge of pigments for painting (cadmium red, Prussian blue, burnt umber) and from printing (cyan, yellow, magenta) does not translate into the information needed to create colors for digital displays. For example, adding all the colors together on a computer monitor produces white, while adding all the colors together with paint produces black (or a strange brown). A computer monitor mixes colors with light. The screen is a black surface, and colored light is added. This is known as additive color, in contrast to the subtractive color model for inks on paper and canvas. This image presents the difference between these models:



The most common way to specify color on the computer is with RGB values. An RGB value sets the amount of red, green, and blue light in a single pixel of the screen. If you look closely at a computer monitor or television screen, you will see that each pixel is comprised of three separate light elements of the colors red, green, and blue; but because our eyes can see only a limited amount of detail, the three colors mix to create a single color. The intensities of each color element are usually specified with values between 0 and 255 where 0 is the minimum and 255 is the maximum. Many software applications

also use this range. Setting the red, green, and blue components to 0 creates black. Setting these components to 255 creates white. Setting red to 255 and green and blue to 0 creates an intense red.

Selecting colors with convenient numbers can save effort. For example, it's common to see the parameters (0, 0, 255) used for blue and (0, 255, 0) for green. These combinations are often responsible for the garish coloring associated with technical images produced on the computer. They seem extreme and unnatural because they don't account for the human eye's ability to distinguish subtle values. Colors that appeal to our eyes are usually not convenient numbers. Rather than picking numbers like 0 and 255, try using a color selector and choosing colors. Processing's color selector is opened from the Tools menu. Colors are selected by clicking a location on the color field or by entering numbers directly. For example, in the figure on the facing page, the current blue selected is defined by an R value of 35, a G value of 211, and a B value of 229. These numbers can be used to recreate the chosen color in your code.

Setting colors

In Processing, colors are defined by the parameters to the `background()`, `fill()`, and `stroke()` functions:

```
background(value1, value2, value3)
fill(value1, value2, value3)
fill(value1, value2, value3, alpha)
stroke(value1, value2, value3)
stroke(value1, value2, value3, alpha)
```

By default, the `value1` parameter defines the red color component, `value2` the green component, and `value3` the blue. The optional `alpha` parameter to `fill()` or `stroke()` defines the transparency. The `alpha` parameter value 255 means the color is entirely opaque, and the value 0 means it's entirely transparent (it won't be visible).



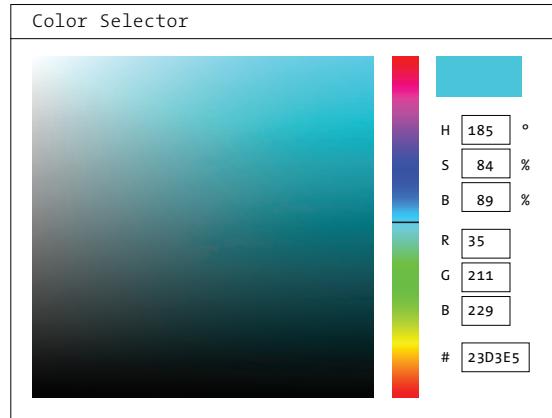
```
background(242, 204, 47);
```

9-01



```
background(174, 221, 60);
```

9-02



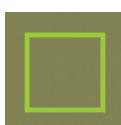
Color Selector

Drag the cursor inside the window or input numbers to select a color. The large square area determines the saturation and brightness, and the thin vertical strip determines the hue. The numeric value of the selected color is displayed in HSB, RGB, and hexadecimal notation.



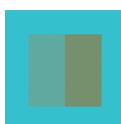
```
background(129, 130, 87);
noStroke();
fill(174, 221, 60);
rect(17, 17, 66, 66);
```

9-03



```
background(129, 130, 87);
noFill();
strokeWeight(4);
stroke(174, 221, 60);
rect(19, 19, 62, 62);
```

9-04



```
background(116, 193, 206);
noStroke();
fill(129, 130, 87, 102); // More transparent
rect(20, 20, 30, 60);
fill(129, 130, 87, 204); // Less transparent
rect(50, 20, 30, 60);
```

9-05



```
background(116, 193, 206);
int x = 0;
noStroke();
for (int i = 51; i <= 255; i += 51) {
  fill(129, 130, 87, i);
  rect(x, 20, 20, 60);
  x += 20;
}
```

9-06



```
background(56, 90, 94);
smooth();
strokeWeight(12);
stroke(242, 204, 47, 102); // More transparency
line(30, 20, 50, 80);
stroke(242, 204, 47, 204); // Less transparency
line(50, 20, 70, 80);
```

9-07



```
background(56, 90, 94);
smooth();
int x = 0;
strokeWeight(12);
for (int i = 51; i <= 255; i += 51) {
    stroke(242, 204, 47, i);
    line(x, 20, x+20, 80);
    x += 20;
}
```

9-08



```
background(0);
noStroke();
smooth();
fill(242, 204, 47, 160); // Yellow
ellipse(47, 36, 64, 64);
fill(174, 221, 60, 160); // Green
ellipse(90, 47, 64, 64);
fill(116, 193, 206, 160); // Blue
ellipse(57, 79, 64, 64);
```

9-09



```
background(255);
noStroke();
smooth();
fill(242, 204, 47, 160); // Yellow
ellipse(47, 36, 64, 64);
fill(174, 221, 60, 160); // Green
ellipse(90, 47, 64, 64);
fill(116, 193, 206, 160); // Blue
ellipse(57, 79, 64, 64);
```

9-10

Color data

The `color` data type is used to store colors in a program, and the `color()` function is used to assign a `color` variable. The `color()` function can create gray values, gray values with transparency, color values, and color values with transparency. Variables of the `color` data type can store all of these configurations:

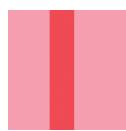
```
color(gray)
color(gray, alpha)
color(value1, value2, value3)
color(value1, value2, value3, alpha)
```

The parameters of the `color()` function define a color. The `gray` parameter used alone or with `alpha` defines tones ranging from white to black. The `alpha` parameter defines transparency with values ranging from 0 (transparent) to 255 (opaque). The `value1`, `value2`, and `value3` parameters define values for the different components. Variables of the `color` data type are defined and assigned in the same way as the `int` and `float` data types discussed in Data 1 (p. 37).

```
color c1 = color(51);           // Creates gray
color c2 = color(51, 204);       // Creates gray with transparency
color c3 = color(51, 102, 153);   // Creates blue
color c4 = color(51, 102, 153, 51); // Creates blue with transparency
```

9-11

After a `color` variable has been defined, it can be used as the parameter to the `background()`, `fill()`, and `stroke()` functions.



```
color ruby = color(211, 24, 24, 160);
color pink = color(237, 159, 176);
background(pink);
noStroke();
fill(ruby);
rect(35, 0, 20, 100);
```

9-12

RGB, HSB

Processing uses the RGB color model as its default for working with color, but the HSB specification can be used instead to define colors in terms of their hue, saturation, and brightness. The hue of a color is what most people normally think of as the color name: yellow, red, blue, orange, green, violet. A pure hue is an undiluted color at its most intense. The saturation is the degree of purity in a color. It is the continuum from the undiluted, pure hue to its most diluted and dull. The brightness of a color is its relation to light and dark.

	RGB			HSB			HEX
	255	0	0	360	100	100	#FF0000
	252	9	45	351	96	99	#FC0A2E
	249	16	85	342	93	98	#F91157
	249	23	126	332	90	98	#F91881
	246	31	160	323	87	97	#F720A4
	244	38	192	314	84	96	#F427C4
	244	45	226	304	81	96	#F42EE7
	226	51	237	295	78	95	#E235F2
	196	58	237	285	75	95	#C43CF2
	171	67	234	276	71	94	#AB45EF
	148	73	232	267	68	93	#944BED
	126	81	232	257	65	93	#7E53ED
	108	87	229	248	62	92	#6C59EA
	95	95	227	239	59	91	#5F61E8
	102	122	227	229	56	91	#667DE8
	107	145	224	220	53	90	#6B94E5
	114	168	224	210	50	90	#72ACE5
	122	186	221	201	46	89	#7ABEE2
	127	200	219	192	43	88	#7FCDE0
	134	216	219	182	40	88	#86DDE0
	139	216	207	173	37	87	#8BDD4
	144	214	195	164	34	86	#90DBC7
	151	214	185	154	31	86	#97DBBD
	156	211	177	145	28	85	#9CD8B5
	162	211	172	135	25	85	#A2D8B0
	169	209	169	126	21	84	#A9D6AD
	175	206	169	117	18	83	#AFD3AD
	185	206	175	107	15	83	#BAD3B3
	192	204	180	98	12	82	#C1D1B8
	197	201	183	89	9	81	#C5CEBB
	202	201	190	79	6	81	#CACEC2
	202	200	193	70	3	80	#CACCC5

Color by numbers

Every color within a program is set by numbers, and there are more than 16 million colors to choose from.

This diagram presents a few colors and their corresponding numbers for the RGB and HSB color models.

The RGB column is in relation to `colorMode(RGB, 255)` and the HSB column is in relation to `colorMode(HSB, 360, 100, 100)`.

The `colorMode()` function sets the color space for a program:

```
colorMode(mode)
colorMode(mode, range)
colorMode(mode, range1, range2, range3)
```

The parameters to `colorMode()` change the way Processing interprets color data. The `mode` parameter can be either RGB or HSB. The range parameters allow Processing to use different values than the default of 0 to 255. A range of values frequently used in computer graphics is between 0.0 and 1.0. Either a single range parameter sets the range for all the color components, or the `range1`, `range2`, and `range3` parameters set the range for each—either red, green, blue or hue, saturation, brightness, depending on the value of the `mode` parameter.

```
// Set the range for the red, green, and blue values from 0.0 to 1.0      9-13
colorMode(RGB, 1.0);
```

A useful setting for HSB mode is to set the `range1`, `range2`, and `range3` parameters respectively to 360, 100, and 100. The hue values from 0 to 360 are the degrees around the color wheel, and the saturation and brightness values from 0 to 100 are percentages. This setting matches the values used in many color selectors and therefore makes it easy to transfer color data between other programs and Processing:

```
// Set the range for the hue to values from 0 to 360 and the           9-14
// saturation and brightness to values between 0 and 100
colorMode(HSB, 360, 100, 100);
```

The following examples reveal the differences between hue, saturation, and brightness.



```
// Change the hue, saturation and brightness constant      9-15
colorMode(HSB);
for (int i = 0; i < 100; i++) {
    stroke(i*2.5, 255, 255);
    line(i, 0, i, 100);
}
```



```
// Change the saturation, hue and brightness constant      9-16
colorMode(HSB);
for (int i = 0; i < 100; i++) {
    stroke(132, i*2.5, 204);
    line(i, 0, i, 100);
}
```



```
// Change the brightness, hue and saturation constant  
colorMode(HSB);  
for (int i = 0; i < 100; i++) {  
    stroke(132, 108, i*2.5);  
    line(i, 0, i, 100);  
}
```

9-17



```
// Change the saturation and brightness, hue constant  
colorMode(HSB);  
for (int i = 0; i < 100; i++) {  
    for (int j = 0; j < 100; j++) {  
        stroke(132, j*2.5, i*2.5);  
        point(i, j);  
    }  
}
```

9-18

It's easy to make smooth transitions between colors by changing the values used for `color()`, `fill()`, and `stroke()`. The HSB model has an enormous advantages over the RGB model when working with code because it's more intuitive. Changing the values of the red, green, and blue components often has unexpected results, while estimating the results of changes to hue, saturation, and brightness follows a more logical path. The following examples show a transition from green to blue. The first example makes this transition using the RGB model. It requires calculating all three color values, and the saturation of the color unexpectedly changes in the middle. The second example makes the transition using the HSB model. Only one number needs to be altered, and the hue changes smoothly and independently from the other color properties.



```
// Shift from blue to green in RGB mode  
colorMode(RGB);  
for (int i = 0; i < 100; i++) {  
    float r = 61 + (i*0.92);  
    float g = 156 + (i*0.48);  
    float b = 204 - (i*1.43);  
    stroke(r, g, b);  
    line(i, 0, i, 100);  
}
```

9-19



```
// Shift from blue to green in HSB mode  
colorMode(HSB, 360, 100, 100);  
for (int i = 0; i < 100; i++) {  
    float newHue = 200 - (i*1.2);  
    stroke(newHue, 70, 80);  
    line(i, 0, i, 100);  
}
```

9-20

Hexadecimal

Hexadecimal (hex) notation is an alternative notation for defining color. This method is popular with designers working on the Web because standards such as HyperText Markup Language (HTML) and Cascading Style Sheets (CSS) use this notation. Hex notation for color encodes each of the numbers from 0 to 255 into a two-digit value using the numbers 0 through 9 and the letters A through F. In this way three RGB values from 0 to 255 can be written as a single six-digit hex value. A few sample conversions demonstrate this notation:

RGB	Hex
255, 255, 255	#FFFFFF
0, 0, 0	#000000
102, 153, 204	#6699CC
195, 244, 59	#C3F43B
116, 206, 206	#74CECE

Converting color values from RGB to hex notation is not intuitive. Most often, the value is taken from a color selector. For instance, you can copy and paste a hex value from Processing's color selector into your code. When using color values encoded in hex notation, you must place a # before the value to distinguish it within the code.



```
// Code 9-03 rewritten using hex numbers
background(#818257);
noStroke();
fill(#AEDD3C);
rect(17, 17, 66, 66);
```

9-21

There's more information about hex notation in Appendix D (p. 669).

Exercises

1. Explore a wide range of color combinations within one composition.
2. Use HSB color and a *for* structure to design a gradient between two colors.
3. Redraw your composition from exercise 1 using hexadecimal color values.



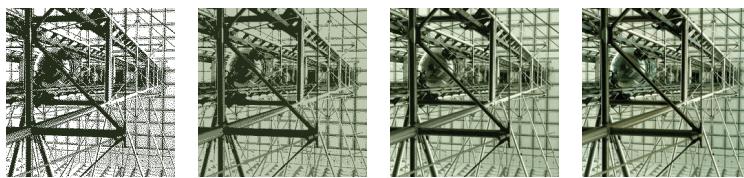
Image 1: Display, Tint

This unit introduces loading and displaying images.

Syntax introduced:

```
PImage, loadImage(), image()  
tint(), notint()
```

Digital photographs are fundamentally different from analog photographs captured on film. Like computer screens, digital photos are rectangular grids of color. The dimensions of digital images are measured in units of pixels. If an image is 320 pixels wide and 240 pixels high, it has 76,800 total pixels. If an image is 1280 pixels wide and 1024 pixels wide, the total number of pixels is an impressive 1,310,720 (1.3 megapixels). Every digital image has a color depth. The color depth refers to the number of bits (p. 669) used to store each pixel. If the color depth of an image is 1, each pixel can be one of two values, for example, black or white. If the color depth is 4, each pixel can be one of 16 values. If the color depth of an image is 8, each pixel can be one of 256 values. Looking at the same image displayed with different color depths reveals how this affects the image's appearance:



1-bit (1 color) 2-bit (4 colors) 4-bit (16 colors) 8-bit (256 colors)

When the Apple Macintosh computer was introduced in 1984, it had a black-and-white screen. Since then, the reproduction of color on screen has rapidly improved. Many contemporary screens have a color depth of 24, which means each pixel can be one of 16,777,216 available colors. This number is typically referred to as “millions of colors.”

Digital images are comprised of numbers representing colors. The file format of an image determines how the numbers are ordered in the file. Some file formats store the color data in mathematically complex arrangements to compress the data and reduce the size of the resulting file. A program that loads an image file must know the file format of the image so it can translate the file’s data into the expected image. Different types of digital image formats serve specific needs. Processing can load GIF, JPEG, and PNG images, along with some other formats as described in the reference. If you don’t already have your images in one of these formats, you can convert other types of digital images to these formats with programs such as GIMP or Adobe Photoshop. Refer to the documentation for these programs if you’re unsure how to convert images.

How do you know which image format to use? They all have obscure names that don't help in making this decision, but each format's advantages becomes clear through comparison:

Format	Extension	Color depth	Transparency
GIF	.gif	1-bit to 8-bit	1-bit
JPEG	.jpg	24-bit	None
PNG	.png	1-bit to 24-bit	8-bit

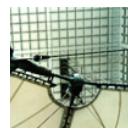
If you are displaying your work on the Internet, image compression becomes an important issue. GIF images are useful for simple graphics with a limited number of colors and transparency. PNG images have similar characteristics but support the full range of colors and transparency. The JPEG format works well for photos, and JPEG files will be smaller than most images saved as PNG. This is because JPEG is a "lossy" format, which means it sacrifices some image quality to reduce file size.

Display

Processing can load images, display them on the screen, and change their size, position, opacity, and tint. There's a data type for images called `PImage`. The same way that integers are stored in variables of the `int` data type and values of `true` and `false` are stored in the `boolean` data type, images are stored in variables of the `PImage` data type. Before displaying an image, it's necessary to first load it with the `loadImage()` function. Be sure to include the file format extension as a part of the name and to put the entire name in quotes (e.g., `"pup.gif"`, `"kat.jpg"`, `"ignatz.png"`). For the image to load, it must be in the data folder of the current program. Add the image by selecting the "Add File" option in the Sketch menu of the Processing environment. Navigate to the image's location on your computer, select the image's icon or name, and click "Open" to add it to the sketch's data folder. As a shortcut, you can also drag and drop an image to the Processing window. To make sure the image was added, select "Show Sketch Folder" from the Sketch menu. The image will be inside the `data` folder. With the image file in the right place, you can load and then display it with the `image()` function:

```
image(name, x, y)
image(name, x, y, width, height)
```

The parameters for `image()` determine the image to draw and its position and size. The `name` parameter must be a `PImage` variable. The `x` and `y` parameters set the position of the upper-left corner of the image. The image will display at its actual size (in units of pixels), but you can change the size by adding the `width` and `height` parameters. Be careful to use the correct capitalization when loading images. If the image is `arch.jpg`, trying to load `Arch.jpg` or `arch.JPG` will create an error. Also, avoid the use of spaces in image names, which can cause problems.



```
PImage img;  
// Image must be in the sketch's "data" folder  
img = loadImage("arch.jpg");  
image(img, 0, 0);
```

10-01



```
PImage img;  
// Image must be in the sketch's "data" folder  
img = loadImage("arch.jpg");  
image(img, 20, 20, 60, 60);
```

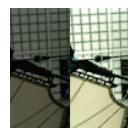
10-02

Image color, Transparency

Images are colored with the `tint()` function. This function is used the same way as `fill()` and `stroke()`, but it affects only images:

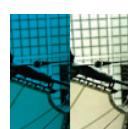
```
tint(gray)  
tint(gray, alpha)  
tint(value1, value2, value3)  
tint(value1, value2, value3, alpha)  
tint(color)
```

All images drawn after running `tint()` will be tinted by the color specified in the parameters. This has no permanent effect on the images, and running the `noTint()` function disables the coloration for all images drawn after it is run.



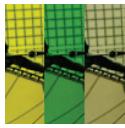
```
PImage img;  
img = loadImage("arch.jpg");  
tint(102); // Tint gray  
image(img, 0, 0);  
noTint(); // Disable tint  
image(img, 50, 0);
```

10-03



```
PImage img;  
img = loadImage("arch.jpg");  
tint(0, 153, 204); // Tint blue  
image(img, 0, 0);  
noTint(); // Disable tint  
image(img, 50, 0);
```

10-04



```
color yellow = color(220, 214, 41);
color green = color(110, 164, 32);
color tan = color(180, 177, 132);
PImage img;
img = loadImage("arch.jpg");
tint(yellow);
image(img, 0, 0);
tint(green);
image(img, 33, 0);
tint(tan);
image(img, 66, 0);
```

10-05

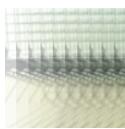
The parameters for `tint()` follow the color space determined by the `colorMode()` function (remember, the default color mode is RGB, with all values ranging from 0 to 255). If the color mode is changed to HSB or a different range, the tint values should be specified relative to that mode.

To make an image transparent without changing its color, set the tint to white. The value will depend on the current color mode, but the default white value is 255.



```
PImage img;
img = loadImage("arch.jpg");
background(255);
tint(255, 102); // Alpha to 102 without changing the tint
image(img, 0, 0, 100, 100);
tint(255, 204, 0, 153); // Tint to yellow, alpha to 153
image(img, 20, 20, 100, 100);
```

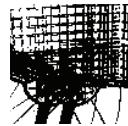
10-06



```
PImage img;
img = loadImage("arch.jpg");
background(255);
tint(255, 51);
// Draw the image 10 times, moving each to the right
for (int i = 0; i < 10; i++) {
    image(img, i*10, 0);
}
```

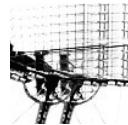
10-07

GIF and PNG images retain their transparency when loaded and displayed in Processing. This allows anything drawn before the image to be visible through the transparent sections of the image. GIF images have only 1-bit transparency, meaning each pixel can only be completely opaque or completely transparent. The PNG format supports 8-bit transparency, meaning there are 256 levels of opacity.



```
// Loads a GIF image with 1-bit transparency  
PImage img;  
img = loadImage("archTrans.gif");  
background(255);  
image(img, 0, 0);  
image(img, -20, 0);
```

10-08



```
// Loads a PNG image with 8-bit transparency  
PImage img;  
img = loadImage("arch.png");  
background(255);  
image(img, 0, 0);  
image(img, -20, 0);
```

10-09

Exercises

1. Draw two images in the display window.
2. Draw three images in the display window, each with a different tint.
3. Load a GIF or PNG image with transparency and create a collage by layering the image.



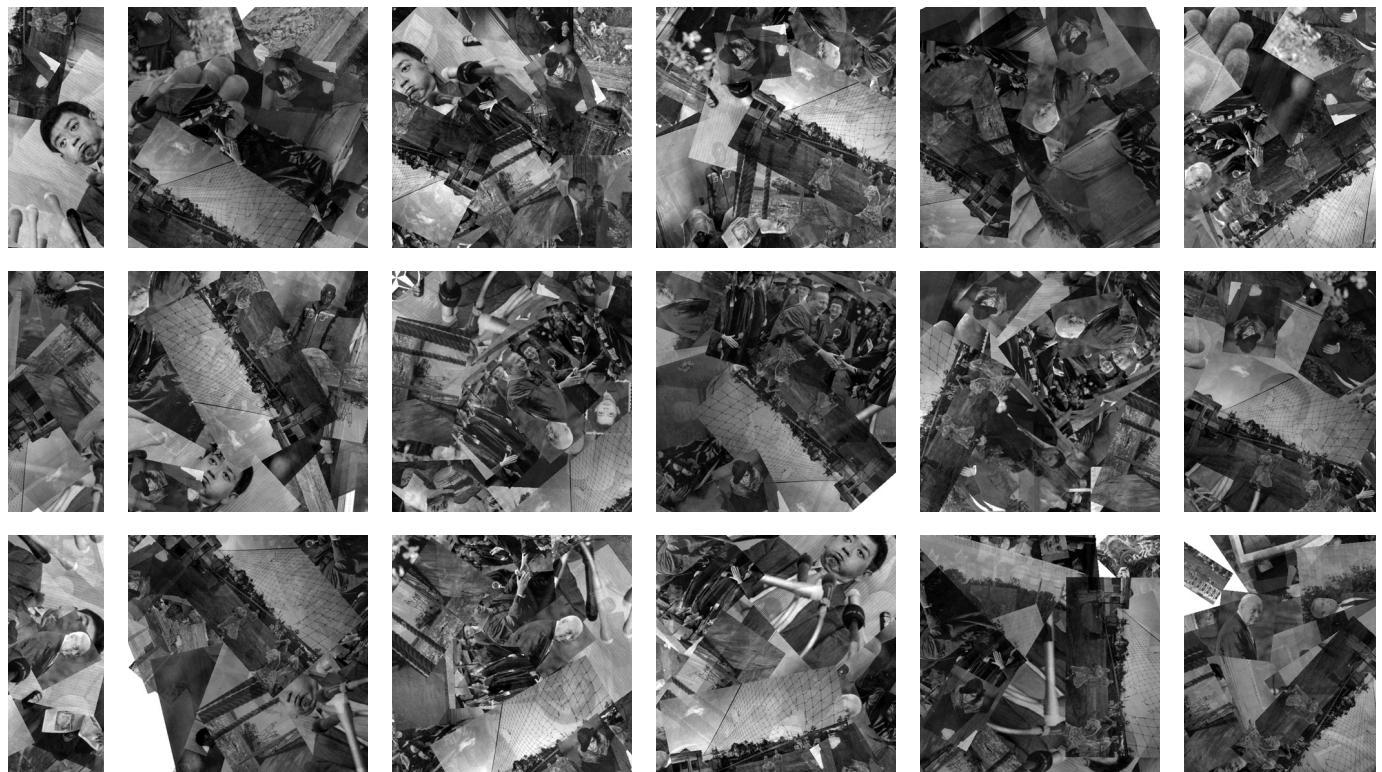
Synthesis 1: Form and Code

This unit presents examples of synthesizing concepts from Structure 1 through Transform 2.

The previous units introduced concepts and techniques including coordinates, drawing with vertices, variables, iteration, conditionals, trigonometry, and transformations. Understanding each of these in isolation is the first step toward learning how to program. Learning how to combine these elements is the second step. There are many ways to combine the components of every programming language for purposes of communication and expression. This programming skill is best acquired through writing more ambitious software and reading more complex programs written by others. This unit introduces four new programs that push beyond those on the previous pages.

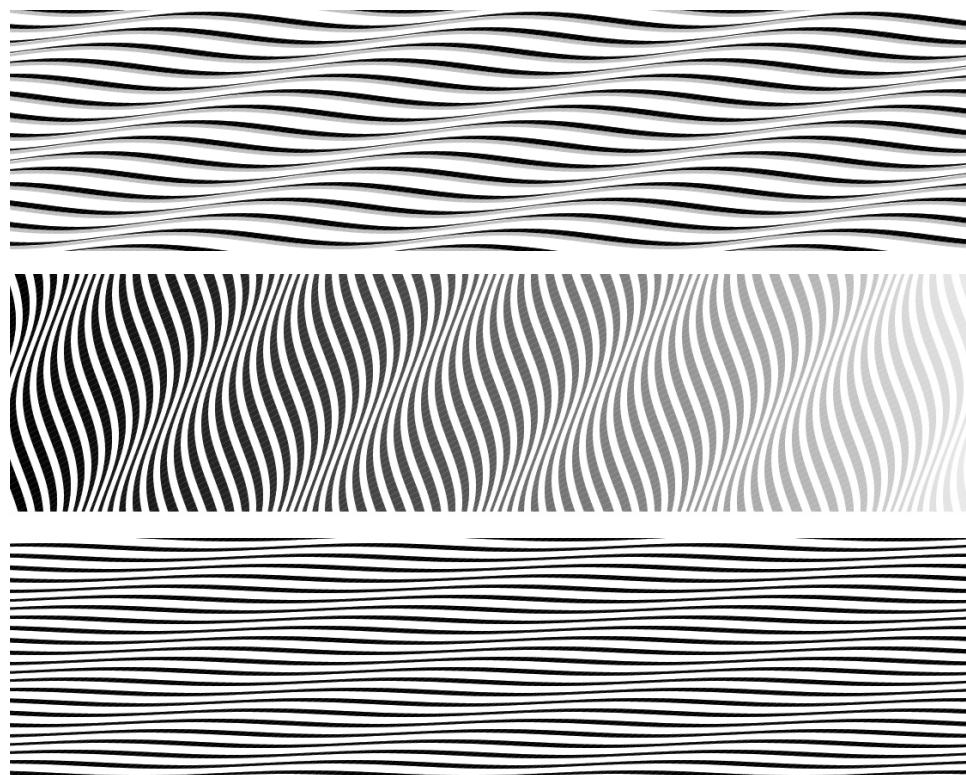
Artists and designers throughout the twentieth century practiced the ideas and visual styles currently associated with software culture, long before personal computers became a common tool. The aesthetic legacies of the Bauhaus, art deco, modernist architecture, and op art movements retain a strong voice in contemporary culture, while new forms have emerged through software explorations within the scientific and artistic communities. The programs in this unit reference images from the last hundred years; sampling from Dadaist collage, optical paintings, a twenty-year-old software program, and mathematics.

The software featured in this unit is longer than the brief examples given in this book. It's not practical to print it on these pages, but the code is included in the Processing code download at www.processing.org/learning.

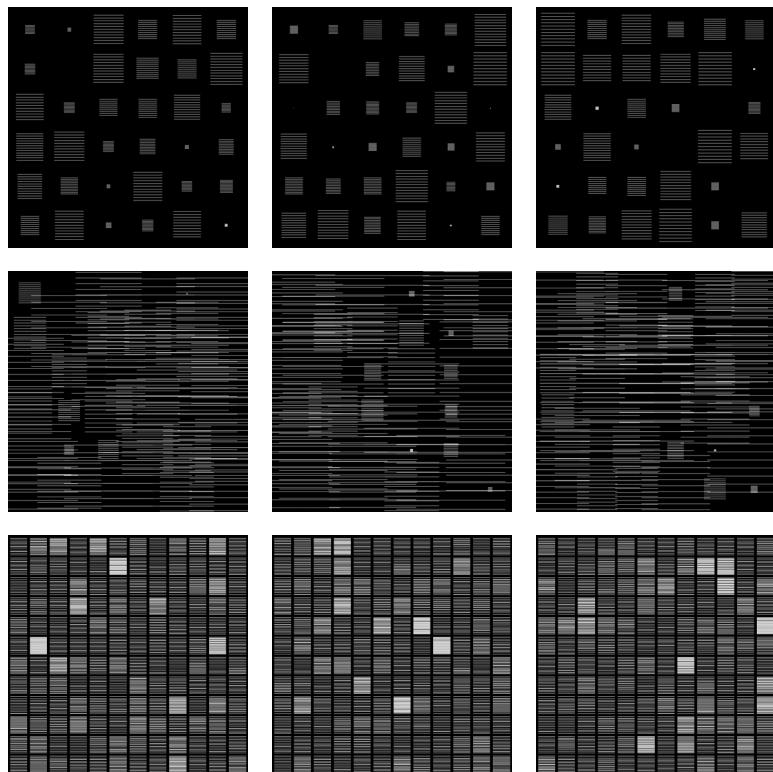


Collage Engine. Reacting to the horror of World War I, European artists and poets within the Dada cultural movement produced works that were deliberately irrational and absurd and that rejected the current standards of art. The poet Tristan Tzara devised a technique for writing that involved taking text from the newspaper, separating the individual words, and putting them back together in random order.

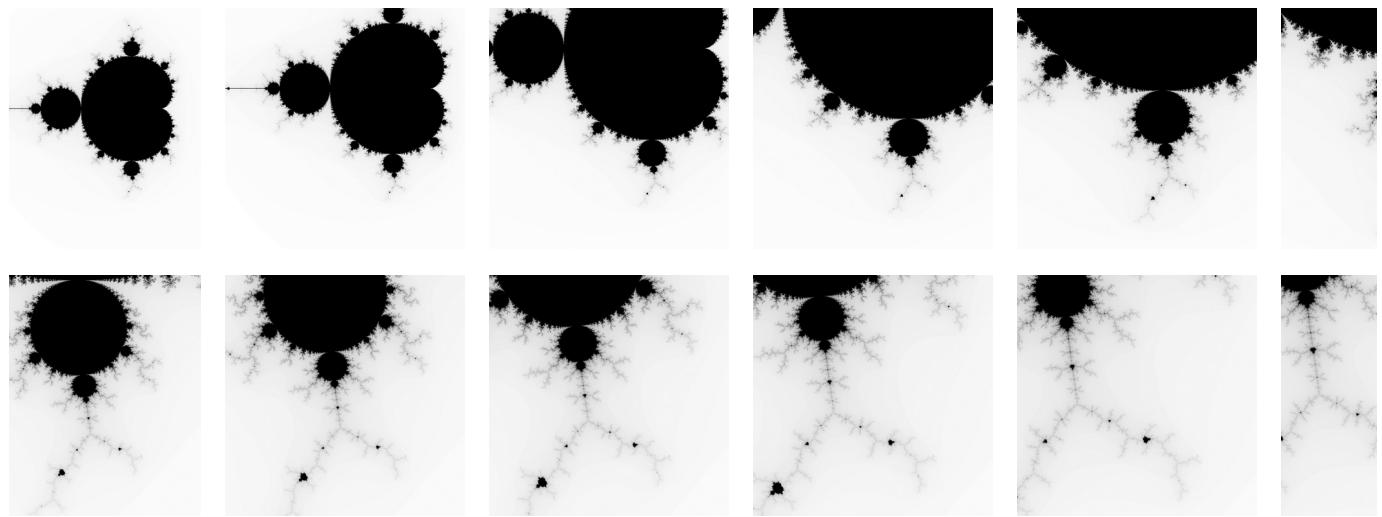
The images shown here were produced using a similar technique with photographs from the first section of *The New York Times* of 9 June 2006. The pictures were cut, scanned, and then repositioned randomly to produce these collages.



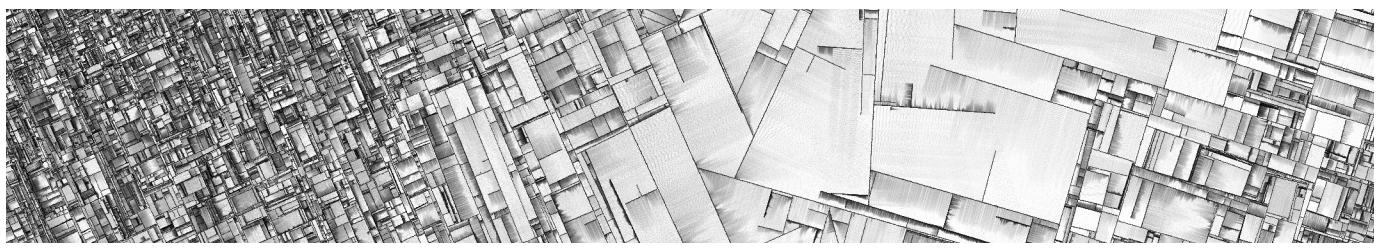
Riley Waves. These images were influenced by the paintings of Bridget Riley, a British artist who has exhibited her work since the mid-1960s. Riley's optically vibrant works often have a strong emotional and visceral effect on the viewer. She works exclusively with simple geometric shapes such as curves and lines and constructs visual vibrations through repetition. Because each of the waves in these images transitions from thick to thin, only the `beginShape()` and `endShape()` functions could create them. Like code 14-09 (p. 122), each wave is comprised of a sequence of triangles drawn using the `TRIANGLE_STRIP` parameter.



Wilson Grids. In his 1985 book *Drawing with Computers*, the artist Mark Wilson surveys the technology of that era and presents many examples of code for drawing to pen plotters and screens. These images were created from code converted from Wilson's programs (written in the BASIC language) to Processing. They utilize the embedded `for` technique introduced in code 6-07 (p. 65).



Mandelbrot Set. A fractal is a shape that appears similar at different scales. Examples of fractals in nature include clouds, mountains, and the network of blood vessels. *Fractal* is a term coined by the mathematician Benoit Mandelbrot, who also devised the Mandelbrot set, an equation that defines a fractal image. The Mandelbrot set left the confines of the mathematics community and entered into popular culture in the 1980s through the popularization of fractal images in books and magazines. These images of the Mandelbrot set were created by changing the scale to render the equation as pixels. The gray value for each pixel of the display window is determined through the equation.



Detail of *Substrate*, 2004. Image courtesy of Jared Tarbell.



Interviews 1: Print

Jared Tarbell. *Fractal.Invaders, Substrate*

Martin Wattenberg. *Shape of Song*

James Paterson. *The Objectivity Engine*

LettError. RandomFont Beowolf



Fractal.Invaders, Substrate (Interview with Jared Tarbell)

Creator	Jared Tarbell
Year	2004
Medium	Software, Prints
Software	Flash, Processing
URL	www.complexification.net

What are *Fractal.Invaders* and *Substrate*?

Fractal.Invaders and *Substrate* are unique programs that both generate space-filling patterns on a two-dimensional surface. Each uses simplified algorithmic processes to render a more complex whole.

Fractal.Invaders begins with a rectangular region and recursively fills it with little "invader" objects. Each invader is a combination of black squares arranged in a 5×5 grid generated at random during runtime. The only rule of construction requires that the left side of the invader be a mirror copy of the right side. This keeps them laterally symmetric, which endows them with a special attractiveness to the human eye.

There are a total of 32,768 (2^{15}) possible invaders. The magnitude of 15 comes from the product of 3 columns and 5 rows (the last 2 columns of the grid are ignored since they are the same as the first 2). The 2 comes from the fact that each space in the grid can be either black or white.

A small bit of interactivity allows each invader to be clicked. Clicking an invader destroys it, although the empty space left behind is quickly filled with smaller invaders. In this way, the user is ultimately doomed.

Substrate begins similarly with an empty rectangular region. It has been compared to crystal formation and the emergent patterns of urban landscapes. A single line (known internally as a "crack" since the algorithm was inspired by sunbaked mud cracks) begins drawing itself from some random point in some random direction. The line continues to draw itself until it either (a) hits the edge of the screen or (b) hits another line, at which point it stops and two more lines begin. The one simple rule used in the creation of new lines is that they begin at tangents to existing lines. This process is repeated until there are too many lines to keep track of or the program is stopped.

Before writing the program, I only had a vague idea of what it might look like. It wasn't until the first couple of bug-free executions that I realized something incredible was happening. The resulting form was much more complex than the originating algorithm. This particular quality of software is what keeps me interested.

Interesting effects can be created by introducing small variations in the way the first couple of lines are drawn. One of my favorite initial conditions is the creation of three lines, each in its own localized space with a direction that varies from the others by about 30 degrees. After growing for a short time into coherent lattices, they eventually crash into each other, creating an affluence of odd shapes and unexpected mazes.

The watercolor quality of the rendering is achieved by placing large numbers of mostly transparent pixels perpendicular to each line's growth. The trick is to deposit precisely the same

Substrate, 2004. Image courtesy of the artist.

number of pixels regardless of the length of the area being filled. This produces an interesting density modulation across an even mass of pixels.

Why did you create this software?

For me, one of the most enjoyable subjects in computer science is combination. I ask myself a question like, "Given some rules and a few simple objects, how many possible ways can they be combined?" Seldom can I answer this using thought alone, mainly because the complexity of even just a few elements is outside the realm of my imagination. Instead, I write computer programs to solve it for me. Fractal.Invaders is definitely one of these questions, and is answered completely with the rendering of every single invader. Substrate asks a similar question but with results that, although beautiful, are a little less complete.

What software tools were used?

For Fractal.Invaders, I used a combination of Flash and custom software to create and capture the invaders, respectively. In Flash, all work was done using ActionScript. A single symbolic element (black square) exists in the library. Code takes this square and duplicates it hundreds of thousands of times. The entire generative process takes about five minutes to complete, depending on the size of the region to be filled and the speed of the execution. Capturing a high-resolution image of the result is accomplished with a program that scales the Shockwave Flash (SWF) file very large and saves the screen image out to a file.

Substrate was created entirely in Processing. Processing was particularly well suited for this as it excels at drawing, especially when dropping millions of deep-color pixels. Processing can also save out extremely large graphic images in an automated fashion. Oftentimes I will run a Processing project overnight. In the morning I awake to a vast collection of unique images, the best of which are archived as print editions.

Why did you use these tools?

I use Flash because I am comfortable working within it. I use Processing because it enables me to do things Flash simply cannot. Both environments allow me to take a program from concept to completion in a number of hours. Complex visual logic can be built up without the bulky overhead required in more traditional graphic programming languages.

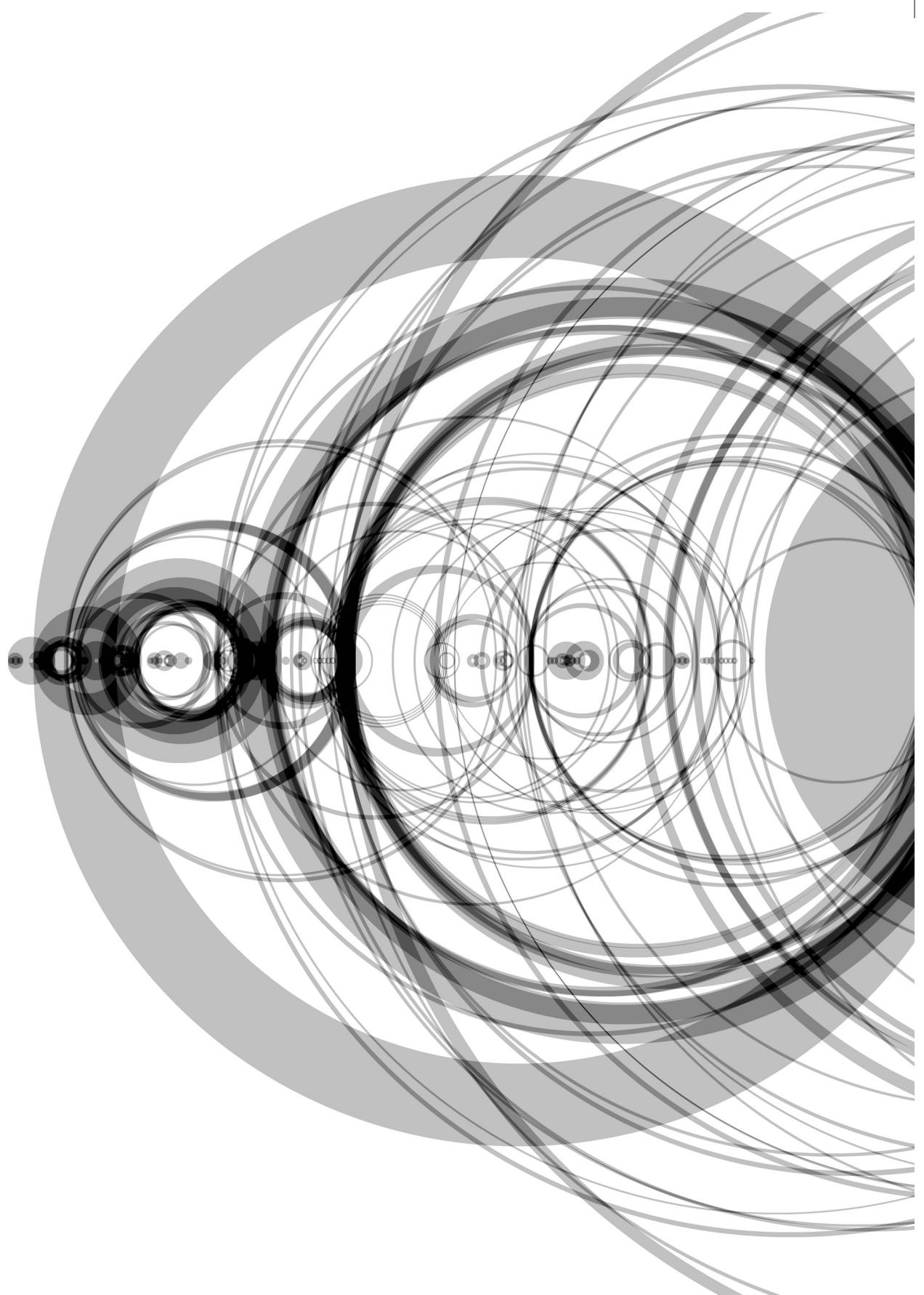
Flash excels at rendering very high resolution images nicely, displaying native vector objects with a high degree of precision and full antialiasing. Processing gives me the computational speed to increase the number of objects in the system by a magnitude of 20 or more. Both programs allow me to produce work that is capable of being viewed by a large number of people worldwide.

Why do you choose to work with software?

With software, anything that can be imagined can be built. Software has a mysterious, undefined border. Programming is truly a process of creating something from nothing. I enjoy most John Maeda's perspective: "While engaged in the deepest trance of coding, all one needs to wish for is any kind of numerical or symbolic resource, and in a flash of lightning it is suddenly there, at your disposal."

Fractal.Invaders, 2004. Image courtesy of the artist.

中南半島的東部，是中國和印度洋之間的一條重要通道。在歷史上，這裏就是中國和印度、東南亞、甚至更遠的西方進行貿易往來的一條重要通道。這裏的民族成分非常複雜，有漢族、壯族、苗族、侗族、瑶族、黎族、哈尼族、白族、彝族、蒙古族、藏族等。這些民族在歷史上都有過各自不同的歷史，並在這裏形成了自己獨特的文化。這裏的風土人情、風俗習慣、宗教信仰、文學藝術等都具有濃厚的地方色彩。這裏的自然環境也十分優美，山清水秀，氣候宜人，四季如春，是理想的旅遊勝地。



Shape of Song (*Interview with Martin Wattenberg*)

Creator	Martin Wattenberg
Year	2002
Medium	Software, Prints
Software	Java
URL	www.turbulence.org/Works/song

What is *Shape of Song*?

The Shape of Song is an attempt to answer the seemingly paradoxical question “What does music look like?” The custom software in this work draws musical patterns in the form of translucent arches, allowing viewers to literally see the shape of a composition.

One of the satisfying aspects of the visualization, to me, is that different musical styles translate to characteristic, distinct visual styles. Folk songs yield simple, repetitive arrangements. Classical pieces have an almost mathematically precise visual structure. Jazz translates to my favorite diagrams, for they often start out with extremely regular patterns and then devolve into something close to chaos.

The work itself has existed in many different forms. It began as a program that I could only run myself. Later I turned it into a Web-based project that let viewers upload their own music. Watching people upload works was fascinating: viewers often tried “extreme” music (the most atonal, the noisiest, the silliest top-40 tunes) to stretch the visualization. Many people are startled when they look at visualizations of “low-culture” music, such as a Led Zeppelin song, because the diagrams are so complex. So the artwork is a good anti-snobbery machine.

Finally, I created prints of some of my favorite diagrams. Most of my work has been purely screen-based, so it was a bit of adventure to work on paper. The level of detail provided by print is wonderful. To see all the different scales of structure at once is beautiful, I think.

I’m currently working on a more dynamic version that will include temporal aspects. My goal is to weave together more closely the spatial rhythms with the actual rhythm of the underlying music.

Why did you create *Shape of Song*?

This project was a personal exploration of the nature of music, balance, and the translation between eye and ear. Music visualization has been a subject of interest for centuries, which is one of the appeals of working on it: you have the sense that you are part of history.

I wanted to understand some of the symmetries found in music. Much of music visualization is aimed at literal translations of notes and rhythms into color and animation. Something more abstract appealed to me, and I pursued a representation of the overall musical form instead.

Although the images created in The Shape of Song are far from a literal translation of the music, the arc-based diagrams came closer than anything else to expressing the mystery and beauty I feel when listening to the underlying compositions.

Why did you write your own software tools?

I had to; they didn't exist yet!

A more nuanced answer includes the fact that I actually used a great deal of existing work when writing the code. The method I used to analyze the music is a standard structure in computer science known as a suffix tree. (Suffix trees work by turning sequences into trees and are traditionally used for rapid searching of text. I am always happy when a piece of computer science, designed for mundane purposes, turns out to be useful in an artwork.) During the course of the project I also used a variety of libraries: some provided by Sun, the developers of the Java programming language, for graphics; and some written by others for writing graphics files and reading scores in the "MIDI" music format.

The use of the MIDI format is a quirk of the piece: more common formats, such as MP3, are harder for my algorithms to handle. (You can think of MIDI as analogous to vector graphics, while MP3s are like JPEGs; if you're trying to find simple patterns like squares or circles, it's far easier with a vector format!) At the time I first started working on the piece, MIDI files were extremely common, but they are becoming more and more rare. That raises some questions about the longevity of the piece, but perhaps by the time MIDI is obsolete someone will develop a reliable algorithm for translating MP3s into musical scores.

To sum up, while the goal of my code was original, most of the computer's time is spent in algorithms or code developed by others.

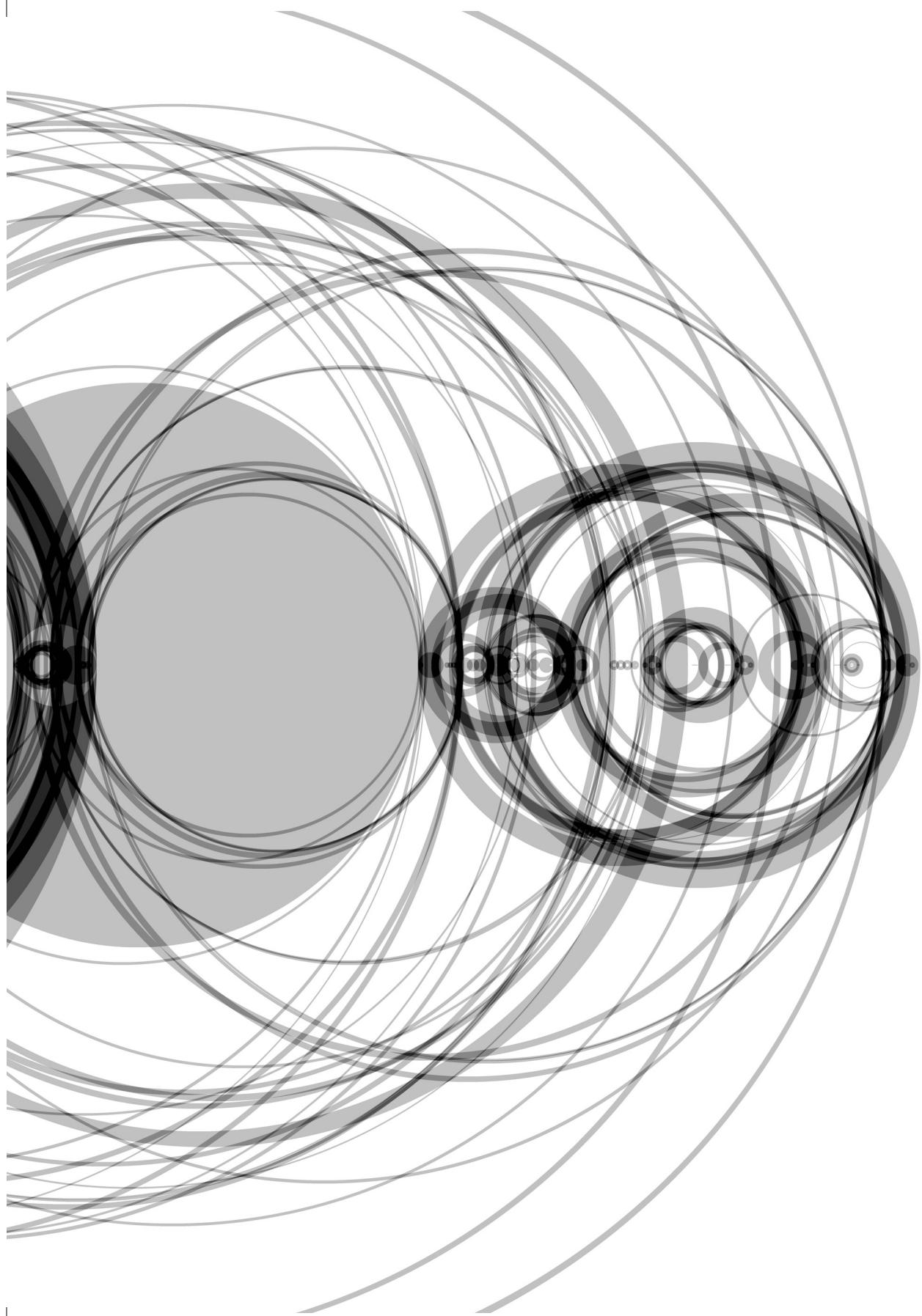
Why do you choose to work with software?

Software is the best way I've found to express myself. When I work in other media, the results somehow always seem worse in reality than in my head. The software I create, however, has a magical quality: it ends up being better than what I originally imagined.

To put it more metaphorically: when I create art, I feel like I am in conversation with the artwork. If I sketch or write, it's like talking to a caustic debater, exposing all the flaws in my thinking. Valuable, perhaps, but also discouraging! When I write programs, I have the opposite feeling: that I am talking with a sympathetic and brilliant partner who helps me organize my thoughts and points out connections I hadn't seen myself.

A second attraction of software art is that it is a new, growing field. There is a kind of energy associated with beginnings that I love. Each new piece seems like it's pointing out new directions, and there's a feeling that you're in a group of settlers on a frontier. As with the American frontier, some people settle down and found new cities, some people keep finding new paths, and some discover gold mines.

Visualization of Vivaldi's Concerto No. 3 in F major, Op. 8, RV 293, "L'autunno." Image courtesy of the artist.





Detail of *Untitled 4*, 2005. Image courtesy of the artist and bitforms gallery, nyc.

The Objectivity Engine (Interview with James Paterson)

Creator	James Paterson
Year	2000–present
Medium	Software, Prints
Software	Flash
URL	www.presstube.com

What is *The Objectivity Engine*?

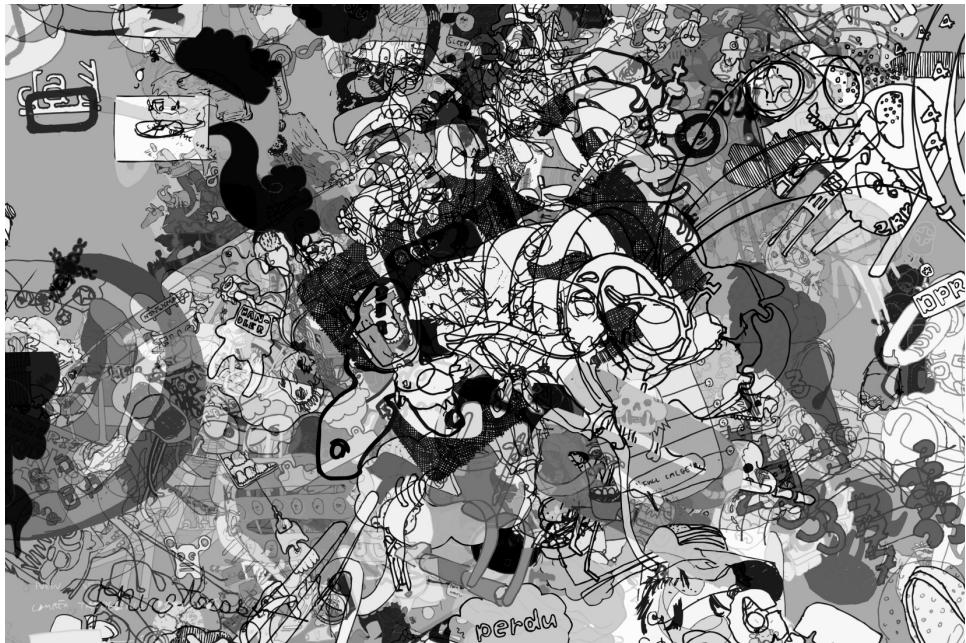
The Objectivity Engine is a system that I developed to help me make composite images out of my sketchbook drawings. The system has two parts: an ever-expanding library of sketchbook drawings, and an arrangement program that pulls drawings from that library and manipulates them to form the composite images. I have been working on the project since 2000.

The process of drawing in a sketchbook is a lot like writing a diary for me: whatever is going on in my life ends up in there one way or another. Sometimes I am creating work that I am proud of and other times I might just be scratching down a reminder to buy more pita bread at the supermarket or trying to figure out how much income tax I'm going to pay that year. The drawing library includes all of it. I am not interested in only entering drawings that I like; I include the good and the bad so that the contents reflect my life more naturally.

The content library consists of about 4,000 sketchbook drawings right now but is always growing as I continue to fill books. The drawings are entered chronologically, and looking at all of them in this organized master library helps me to see where I am coming from and where I am going. It is a bit like looking at the rings on a tree trunk. I can see how I was doing at one point or another. I plan on continuing to expand this library as long as I can. Ideally I would like to keep adding to it for the rest of my life.

The arrangement program is a set of algorithms that are in charge of putting together composites. It controls the amount of images used, the vintage of the images, the coloring system, methods of distribution, the motion control system (when it is outputting animation), the scale, rotation, speed, etc. It's like casting a net into the history of my drawing over the past five years. I never know what strange and potentially embarrassing combinations I will reel in. Sometimes seeing the results gives me ideas for new drawings, sometimes I find a composition that I may want to manipulate manually later, and occasionally the program will spit out a composition that I am really happy with and can just save and use as is. A lot of the time it produces complete crap!

I am always adding new features to the arrangement program as I think of them, and about once a year I rewrite the whole thing from scratch. I do this so that I remember how it all works, and so that new components that I am adding as I go along can be properly integrated. Just as I fancy the idea of continuing to add to the library of drawings for the rest of my life, I also like the idea of adding to the capability of the arrangement system over a long period of time and seeing where it leads me.



Untitled 3, 2005. Digital Lambda print. 40" x 60". Image courtesy of the artist and bitforms gallery, nyc.

Why did you create *The Objectivity Engine*?

I have never been very good at putting disparate ideas and images together to form larger, more coherent finished works, so I decided to create a system that would get me started down this road. It is an assistant that helps me make decisions, and it does a lot of tedious and repetitive work for me very quickly.

What software tools were used?

Photoshop was used to scan and clean up the sketchbook drawings. Streamline was then used to convert the raster scans to vector format. The vector files were brought into Flash. Flash was used to contain the drawing library, write the arrangement software, and export the results. Director was used to export animated results. Photoshop, Illustrator, and After Effects were used to put finishing touches on the raw exports and prep them for print and DVD.

Why did you use these tools?

They just seemed like the best tools I could find for each respective job. I used Flash/ActionScript as my main platform because I have been using it for a long time and I use it to create all of my other work. This way I can mix elements of this project with many other works that have nothing to do with it and vice versa. Also, Flash is a vector-based environment, which lends itself well to the aims of the project.

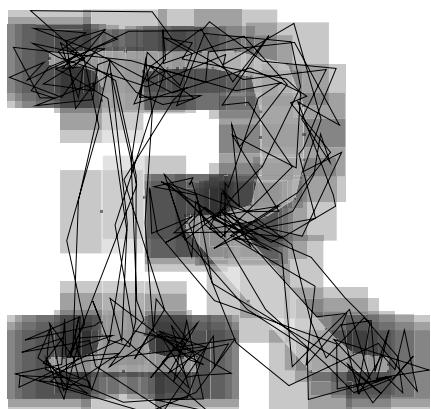
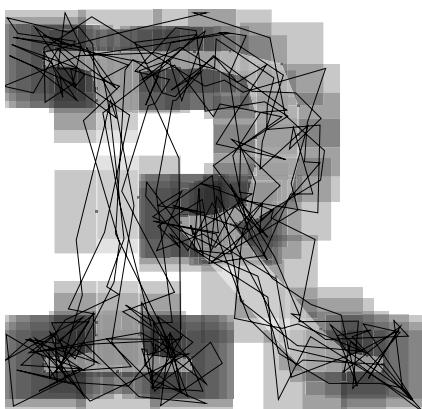
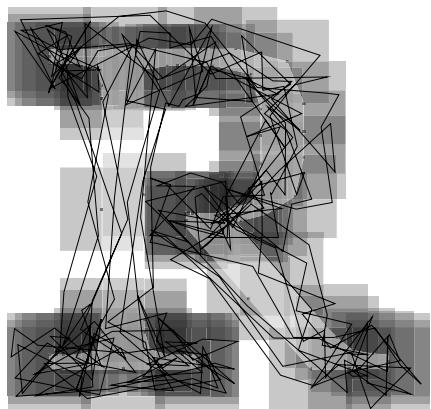
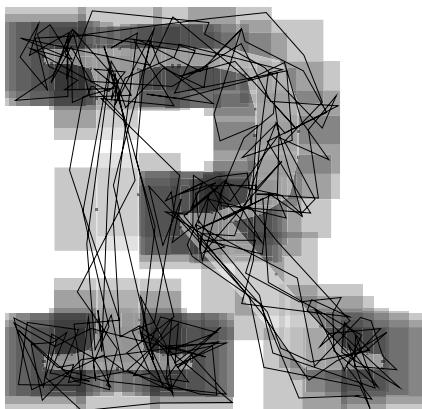
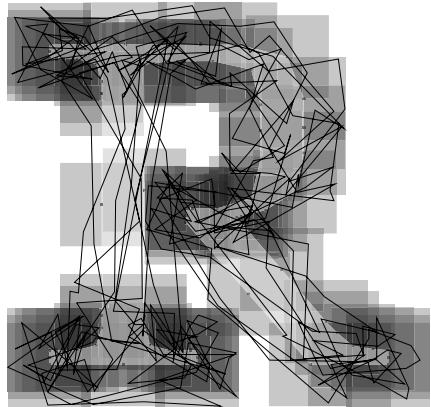
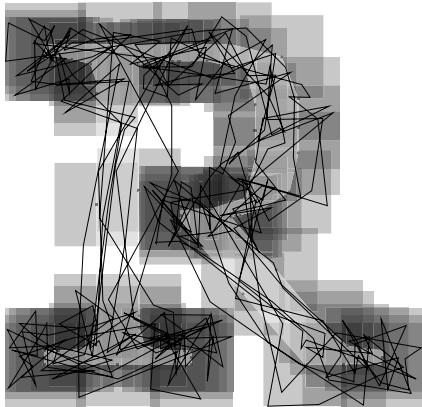
Why do you choose to work with software?

I just stumbled across Flash in 1997 and started playing with it and then never stopped. I had been drawing for quite some time before that, and after a few years my drawing processes merged with my software processes and the two have been pretty much inseparable ever since.



Untitled 5, 2005. Digital Lambda print. 40" × 60". Image courtesy of the artist and bitforms gallery, nyc.

Untitled 10, 2005. Digital Lambda print. 40" × 60". Image courtesy of the artist and bitforms gallery, nyc.



RandomFont Beowolf (*Interview with Erik van Blokland*)

Creators	Just van Rossum and Erik van Blokland (LettError)
Year	1990
Medium	Typeface
Software	PostScript Type 3 font
URL	www.lettererror.com/foundry/beowolf

What is RandomFont Beowolf?

In 1989, after finishing the graphic and typographic design course at the Royal Academy of Arts in The Hague, Netherlands, Just and I were both experimenting with PostScript, a powerful programming language designed for graphics. The only machines we had access to that were capable of executing PostScript programs were laser printers, so we went through a lot of paper. We figured it should be possible to build a font with random functions applied to the letterforms when it prints. We made a test font containing one randomizing square to prove the concept.

Later we used a typeface I had drawn in school, which after some iterations became Beowulf.

A font in PostScript in its most basic form is a dictionary with some standard entries and drawing instructions for each letter. Fonts like this were called “Type 3,” and as long as you managed to get the file in the printer, the letters could use all the functionality of the PostScript language. So that’s what we built. Another PostScript font format, “Type 1,” was more compact and offered faster processing and hinting, but it was proprietary, encrypted and required secret tools to generate. In the summer of 1990 Adobe published the specifications of Type 1 but unfortunately the increased speed of printing with Type 1 fonts came at a price of a very limited instruction set.

Later we built “piggyback fonts,” which incorporated both Type 1 and Type 3 formats. Fonts like this consisted of several Type 1 fonts and a special Type 3 font to contain logic to switch glyphs and fonts while rendering a text. The fonts were all bundled together in a single file, then the Macintosh printer driver would just download the whole thing, and all fonts would make it to the printer. It was a wonderful hack.

The demands of the graphic design workflow made it increasingly difficult to deploy Type 3 formats, and we stopped shipping them. We’ve always made a case for fonts with executable code; typography is a complex field which can benefit from programming on a font level. But we can’t expect the entire digital design industry to accommodate our whims.

The current OpenType font format (developed by Adobe and Microsoft) actually contains ways to define rules for contextual substitution¹ and positioning of glyphs. Though nothing like a fully featured programming language, it’s an improvement and fun to develop for. We have an OpenType font with a decent simulation of a RandomFont, much like our piggyback fonts. Visually it reconstructs the broken, edgy style Beowulf had, but conceptually these OpenType Beowolves have very little to do with the original one.

Why did you create RandomFont Beowulf?

Curiosity. We were both trained as type designers, and we were interested in computers and programming. At that time the fields of design and digital technology didn’t really overlap. After the first versions we started thinking about the context and implications of

randomization. Beowulf became an example of what digital type could be: not that the random aesthetic itself was so appealing, but it was proof that fonts were no longer physical objects but instructions. It also showed us that code and design can merge into a single process, a single object. Code has a major influence on design, and I think it is too important to leave it to anonymous engineers.

What software tools were used?

The first PostScript tests were written in a text editor. Then we took simple PostScript Type 3 fonts generated by Ikarus M (the first version, written by Petr van Blokland) and edited those. These fonts had simple, readable text instructions and absolute coordinates that were easy to modify. Later on with some help from AltSys's Jim Von Ehr we moved on to a more complex variation of Type 3 PostScript, editing in ResEdit. The last incarnation of the RandomFonts was piggybacking on a Type 1 font so that there would be some sort of (nonrandom) preview when used in a layout program; then later in the printer the Type 3 randomizing version would kick in and do the work. The piggyback fonts were also made with ResEdit.

Why did you use these tools?

Type design is a small field, so there aren't many developers interested in writing tools. AltSys wrote Fontographer, at the time the popular choice for editing fonts, but it didn't allow the things we had in mind. Petr van Blokland and David Berlow convinced Jim Von Ehr to give them access to the Fontographer source code and Petr started experimenting with adding a layer using awk, a programming language for processing text data. Just suggested using a new programming language that his brother had invented called Python. Python and Fontographer became RoboFog and gained a small but dedicated group of users. Python is fast to develop in, which allows a regular iterative design process. After the first time you write a program you know how it should have been done and you can afford to start over and do it again, improving the understanding of the problem.

When Mac OS X was released we couldn't port RoboFog because the code base was so old. The FontLab font development software had added Python scripting to their font editor, so we started to work with that instead. We started reshaping their API into ours and this grew into the RoboFab library, an object model and API built on top of FontLab's font and glyph objects. RoboFab now also has an implementation that works independently of FontLab.

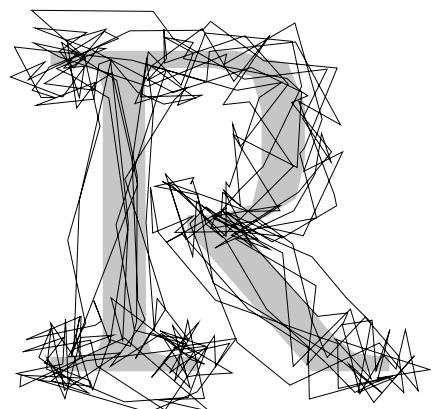
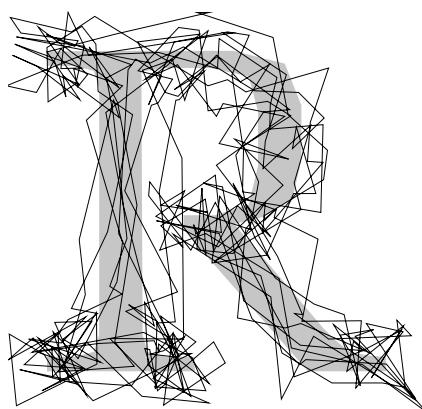
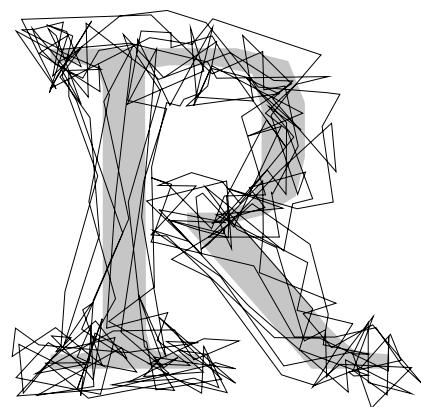
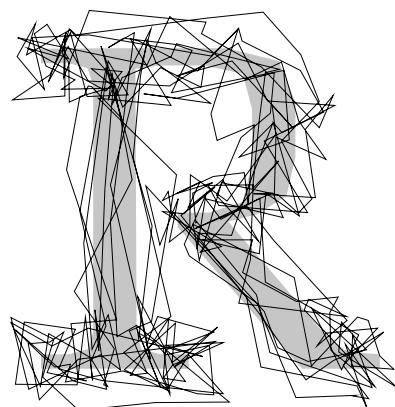
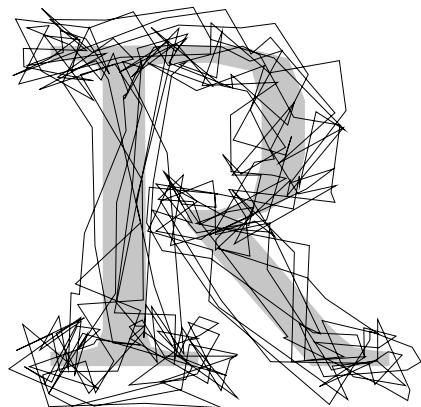
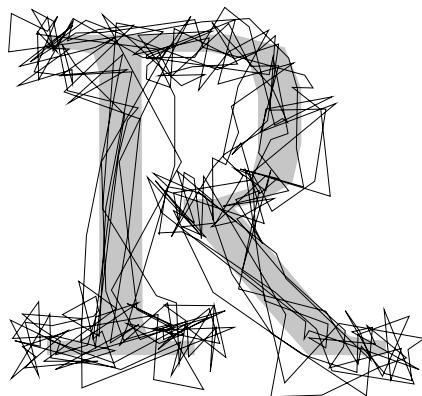
Why do you choose to work with software?

In any creative discipline, the tools influence the process and, indirectly, the results. We try to be aware of this influence, and if it is something we don't like we try to change it. Every application makes certain things easy and others more difficult. This directs the average design project towards the things that are easy, even though other ideas might be more relevant.

Writing your own tools makes the ideas direct the development of software, rather than the other way round. Writing code is also an attractive process in itself. Analyzing problems, breaking them down into ideas that can be coded, and discovering alternative new ways to solve known problems is more universal than just the original design task.

Notes

1. Contextual substitution is the process by which letters are changed based on the letters around them. For instance, an f and i might be joined into a single glyph, a ligature that looks like fi.



Input 1: Mouse I

This unit introduces mouse input as a way to control the position and attributes of shapes on screen. It also explains how to change the cursor icon.

Syntax introduced:

`mouseX, mouseY, pmouseX, pmouseY, mousePressed, mouseButton
cursor(), noCursor()`

The screen forms a bridge between our bodies and the realm of circuits and electricity inside computers. We control elements on screen through a variety of devices such as touch pads, trackballs, and joysticks, but—aside from the keyboard—the most common input device is the mouse. The computer mouse dates back to the late 1960s when Douglas Engelbart presented the device as an element of the oN-Line System (NLS), one of the first computer systems with a video display. The mouse concept was further developed at the Xerox Palo Alto Research Center (PARC), but its introduction with the Apple Macintosh in 1984 was the catalyst for its current ubiquity. The design of the mouse has gone through many revisions in the last thirty years, but its function has remained the same. In Engelbart's original patent application in 1970 he referred to the mouse as an "X-Y position indicator," and this still accurately, but dryly, defines its contemporary use.

The physical mouse object is used to control the position of the cursor on screen and to select interface elements. The cursor position is read by computer programs as two numbers, the x-coordinate and the y-coordinate. These numbers can be used to control attributes of elements on screen. If these coordinates are collected and analyzed, they can be used to extract higher-level information such as the speed and direction of the mouse. This data can in turn be used for gesture and pattern recognition.

Mouse data

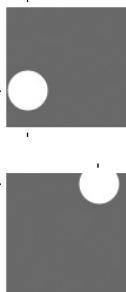
The Processing variables `mouseX` and `mouseY` (note the capital X and Y) store the x-coordinate and y-coordinate of the cursor relative to the origin in the upper-left corner of the display window. To see the actual values produced while moving the mouse, run this program to print the values to the console:

```
void draw() {  
    frameRate(12);  
    println(mouseX + " : " + mouseY);  
}
```

23-01

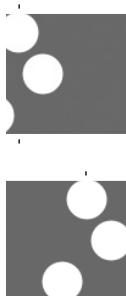
When a program starts, `mouseX` and `mouseY` values are 0. If the cursor moves into the display window, the values are set to the current position of the cursor. If the cursor is at the left, the `mouseX` value is 0 and the value increases as the cursor moves to the right. If the cursor is at the top, the `mouseY` value is 0 and the value increases as the cursor moves down. If `mouseX` and `mouseY` are used in programs without a `draw()` or if `noLoop()` is run in `setup()`, the values will always be 0.

The mouse position is most commonly used to control the location of visual elements on screen. More interesting relations are created when the visual elements relate differently to the mouse values, rather than simply mimicking the current position. Adding and subtracting values from the mouse position creates relationships that remain constant, while multiplying and dividing these values creates changing visual relationships between the mouse position and the elements on the screen. To invert the value of the mouse, simply subtract the `mouseX` value from the width of the window and subtract the `mouseY` value from the height of the screen.



```
// Circle follows the cursor (the cursor position is  
// implied by the crosshairs around the illustration)  
  
void setup() {  
    size(100, 100);  
    smooth();  
    noStroke();  
}  
  
void draw() {  
    background(126);  
    ellipse(mouseX, mouseY, 33, 33);  
}
```

23-02



```
// Add and subtract to create offsets  
  
void setup() {  
    size(100, 100);  
    smooth();  
    noStroke();  
}  
  
void draw() {  
    background(126);  
    ellipse(mouseX, 16, 33, 33); // Top circle  
    ellipse(mouseX+20, 50, 33, 33); // Middle circle  
    ellipse(mouseX-20, 84, 33, 33); // Bottom circle  
}
```

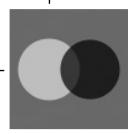
23-03



// Multiply and divide to creates scaling offsets

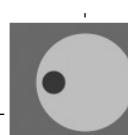
23-04

```
void setup() {  
    size(100, 100);  
    smooth();  
    noStroke();  
}  
  
void draw() {  
    background(126);  
    ellipse(mouseX, 16, 33, 33); // Top circle  
    ellipse(mouseX/2, 50, 33, 33); // Middle circle  
    ellipse(mouseX*2, 84, 33, 33); // Bottom circle  
}
```

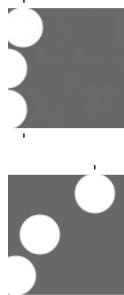


// Invert cursor position to create a secondary response

23-05



```
void setup() {  
    size(100, 100);  
    noStroke();  
    smooth();  
}  
  
void draw() {  
    float x = mouseX;  
    float y = mouseY;  
    float ix = width - mouseX; // Inverse X  
    float iy = mouseY - height; // Inverse Y  
    background(126);  
    fill(255, 150);  
    ellipse(x, height/2, y, y);  
    fill(0, 159);  
    ellipse(ix, height/2, iy, iy);  
}
```



```
// Exponential functions can create nonlinear relations      23-06
// between the mouse and shapes affected by the mouse

void setup() {
    size(100, 100);
    smooth();
    noStroke();
}

void draw() {
    background(126);
    float normX = mouseX / float(width);
    ellipse(mouseX, 16, 33, 33);                      // Top
    ellipse(pow(normX, 4) * width, 50, 33, 33);        // Middle
    ellipse(pow(normX, 8) * width, 84, 33, 33);        // Bottom
}
```

The Processing variables `pmouseX` and `pmouseY` store the mouse values from the previous frame. If the mouse does not move, the values will be the same, but if the mouse is moving quickly there can be large differences between the values. To see the difference, run the following program and alternate moving the mouse slowly and quickly. Watch the values print to the console.

```
void draw() {                                              23-07
    frameRate(12);
    println(pmouseX - mouseX);
}
```

Drawing a line from the previous mouse position to the current position shows the changing position in one frame, revealing the speed and direction of the mouse. When the mouse is not moving, a point is drawn, but quick mouse movements create long lines.



```
// Draw a line between the current and previous positions 23-08

void setup() {
    size(100, 100);
    strokeWeight(8);
    smooth();
}

void draw() {
    background(204);
    line(mouseX, mouseY, pmouseX, pmouseY);
}
```

The `mouseX` and `mouseY` values can control translation, rotation, and scale by using them as parameters in the transformation functions. You can move a circle around the screen by changing the parameters to `translate()` rather than by changing the `x` and `y` parameters of `ellipse()`.



```
// Use translate() to move a shape  
  
void setup() {  
    size(100, 100);  
    smooth();  
    noStroke();  
}  
  
void draw() {  
    background(126);  
    translate(mouseX, mouseY);  
    ellipse(0, 0, 33, 33);  
}
```

23-09

Before using `mouseX` and `mouseY` as parameters to transformation functions, it's important to think first about how they relate to the expected parameters. For example, the `rotate()` function expects its parameters in units of radians (p. 117). To make a shape rotate 360 degrees as the cursor moves from the left edge to the right edge of the window, the values of `mouseX` must be converted to values from 0.0 to 2π . In the following example, the `map()` function is used to make this conversion. The resulting value is used as the parameter to `rotate()` to turn the line as the mouse moves back and forth between the left and right edge of the display window.



```
// Use rotate() to move a shape  
  
void setup() {  
    size(100, 100);  
    strokeWeight(8);  
    smooth();  
}  
  
void draw() {  
    background(204);  
    float angle = map(mouseX, 0, width, 0, TWO_PI);  
    translate(50, 50);  
    rotate(angle);  
    line(0, 0, 40, 0);  
}
```

23-10

Using the `mouseX` and `mouseY` variables with an `if` structure allows the cursor to select regions of the screen. The following examples demonstrate the cursor making a selection between different areas of the display window.



*// Cursor position selects the left or right half
// of the display window*

23-11

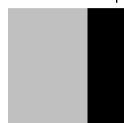


```
void setup() {  
    size(100, 100);  
    noStroke();  
    fill(0);  
}  
  
void draw() {  
    background(204);  
    if (mouseX < 50) {  
        rect(0, 0, 50, 100); // Left  
    } else {  
        rect(50, 0, 50, 100); // Right  
    }  
}
```

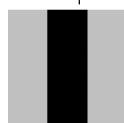


*// Cursor position selects the left, middle,
// or right third of the display window*

23-12



```
void setup() {  
    size(100, 100);  
    noStroke();  
    fill(0);  
}  
  
void draw() {  
    background(204);  
    if (mouseX < 33) {  
        rect(0, 0, 33, 100); // Left  
    } else if ((mouseX >= 33) && (mouseX <= 66)) {  
        rect(33, 0, 33, 100); // Middle  
    } else {  
        rect(66, 0, 33, 100); // Right  
    }  
}
```





// Cursor position selects a quadrant of
// the display window

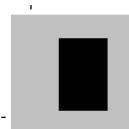
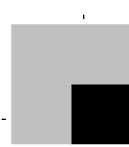
23-13



```
void setup() {  
    size(100, 100);  
    noStroke();  
    fill(0);  
}
```

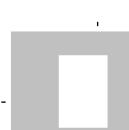


```
void draw() {  
    background(204);  
    if ((mouseX <= 50) && (mouseY <= 50)) {  
        rect(0, 0, 50, 50); // Upper-left  
    } else if ((mouseX <= 50) && (mouseY > 50)) {  
        rect(0, 50, 50, 50); // Lower-left  
    } else if ((mouseX > 50) && (mouseY < 50)) {  
        rect(50, 0, 50, 50); // Upper-right  
    } else {  
        rect(50, 50, 50, 50); // Lower-right  
    }  
}
```



// Cursor position selects a rectangular area to
// change the fill color

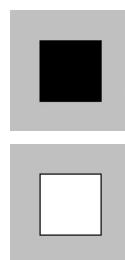
23-14



```
void setup() {  
    size(100, 100);  
    noStroke();  
    fill(0);  
}  
  
void draw() {  
    background(204);  
    if ((mouseX > 40) && (mouseX < 80) &&  
        (mouseY > 20) && (mouseY < 80)) {  
        fill(255);  
    } else {  
        fill(0);  
    }  
    rect(40, 20, 40, 60);  
}
```

Mouse buttons

Computer mice and other similar input devices typically have between one and three buttons, and Processing can detect when these buttons are pressed. The button status and the cursor position together allow the mouse to perform different actions. For example, pressing a button when the mouse is over an icon can select it, so the icon can be moved to a different location on screen. The `mousePressed` variable is `true` if any mouse button is pressed and `false` if no mouse button is pressed. The variable `mouseButton` is `LEFT`, `CENTER`, or `RIGHT` depending on the mouse button most recently pressed. The `mousePressed` variable reverts to `false` as soon as the button is released, but the `mouseButton` variable retains its value until a different button is pressed. These variables can be used independently or in combination to control your software. Run these programs to see how the software responds to your fingers.



// Set the square to white when a mouse button is pressed 23-15

```
void setup() {  
    size(100, 100);  
}  
  
void draw() {  
    background(204);  
    if (mousePressed == true) {  
        fill(255); // White  
    } else {  
        fill(0); // Black  
    }  
    rect(25, 25, 50, 50);  
}
```

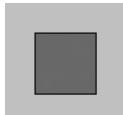


// Set the square to black when the left mouse button 23-16
// is pressed and white when the right button is pressed

```
void setup() {  
    size(100, 100);  
}  
  
void draw() {  
    if (mouseButton == LEFT) {  
        fill(0); // Black  
    } else if (mouseButton == RIGHT) {  
        fill(255); // White  
    } else {
```

```
    fill(126); // Gray  
}  
rect(25, 25, 50, 50);  
}
```

23-16
cont.



```
// Set the square to black when the left mouse button  
// is pressed, white when the right button is pressed,  
// and gray when a button is not pressed
```

23-17



```
void setup() {  
    size(100, 100);  
}  
  
void draw() {  
    if (mousePressed == true) {  
        if (mouseButton == LEFT) {  
            fill(0); // Black  
        } else if (mouseButton == RIGHT) {  
            fill(255); // White  
        }  
    } else {  
        fill(126); // Gray  
    }  
    rect(25, 25, 50, 50);  
}
```

Not all mice have multiple buttons, and if software is distributed widely, the interaction should not rely on detecting which button is pressed. For example, if you are posting your work on the Web, don't rely on the middle or right button for using the software because many users won't have a two- or three-button mouse.

Cursor icon

The cursor can be hidden with the `noCursor()` function and can be set to appear as a different icon with the `cursor()` function. When the `noCursor()` function is run, the cursor icon disappears as it moves into the display window. To give feedback about the location of the cursor within the software, a custom cursor can be drawn and controlled with the `mouseX` and `mouseY` variables.

```
// Draw an ellipse to show the position of the hidden cursor
```

23-18

```
void setup() {  
    size(100, 100);  
    strokeWeight(7);  
    smooth();  
    noCursor();  
}  
  
void draw() {  
    background(204);  
    ellipse(mouseX, mouseY, 10, 10);  
}
```

If `noCursor()` is run, the cursor will be hidden while the program is running until the `cursor()` function is run to reveal it.

```
// Hides the cursor until a mouse button is pressed
```

23-19

```
void setup() {  
    size(100, 100);  
    noCursor();  
}  
  
void draw() {  
    background(204);  
    if (mousePressed == true) {  
        cursor();  
    }  
}
```

Adding a parameter to the `cursor()` function allows it to be changed to another icon. The self-descriptive options for the `MODE` parameter are ARROW, CROSS, HAND, MOVE, TEXT, and WAIT.

```
// Draws the cursor as a hand when a mouse button is pressed
```

23-20

```
void setup() {  
    size(100, 100);  
    smooth();  
}  
  
void draw() {  
    background(204);
```

```
if (mousePressed == true) {  
    cursor(HAND);  
} else {  
    cursor(MOVE);  
}  
line(mouseX, 0, mouseX, height);  
line(0, mouseY, height, mouseY);  
}
```

23-20
cont.

These cursor images are part of your computer's operating system and will appear differently on different machines.

Exercises

1. *Control the position of a shape with the mouse. Strive to create a more interesting relation than one directly mimicking the position of the cursor.*
2. *Invent three unique shapes that behave differently in relation to the mouse. Each shape's behavior should change when the mouse is pressed. Relate the form of each shape to its behavior.*
3. *Create a custom cursor that changes as the mouse moves through the display window.*

Motion 2: Machine, Organism

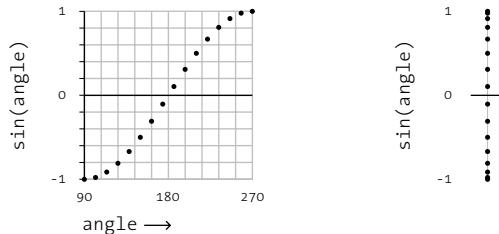
This unit introduces the communicative aspects of motion—how different qualities of motion create diverse moods and meanings.

Programmers determine how elements in their programs will move. These decisions influence how the forms are interpreted by viewers. Over the course of evolution, humans have developed instincts and attitudes toward the motion in the world around us. We classify and categorize elements in the world by how they move or do not move. We make distinctions between animate and inanimate things. Motion in software can take advantage of humans' innate understanding of movement or ignore it. Software makes possible countless types of motion, but two major categories of interest are mechanical and organic motion. This unit focuses on the qualities that define these types of motion.

Mechanical motion

In the catalog for The Museum of Modern Art's 1934 Machine Art exhibition, Alfred Barr described characteristics of the machines of that time: "Machines are, visually speaking, a practical application of geometry. Forces which act in straight lines are changed in direction and degree by machines which are themselves formed of straight lines and curves. The lever is geometrically a straight line resting on a point. The wheel and axle is composed of concentric circles and radiating straight lines." He further explained that motion "increases their aesthetic interest, principally through the addition of temporal rhythms." While machines and society's ideas about machines have changed dramatically in the last seventy years, Barr's insights remain relevant in defining the characteristics of mechanical motion. Prototypical examples of machine motion include the clock, the metronome, and the piston. These mechanisms' movements are all characterized by regular rhythm, repetition, and efficiency. Writing code to produce these qualities of motion communicates the essence of the machines through software.

The `sin()` function is often used to produce elegant motion. It can generate an accelerating and decelerating speed as a shape moves from one frame to another. The images below show how the speed along a sine wave is consistent if the angle is increased at a constant rate, but the speed along the y-axis increases and decreases at the top and bottom of the curve. The image on the left displays the changing sine value as the angle grows. The image on the right displays only the sine values but does not draw the x-axis, to emphasize the changes in speed along the y-axis:



The values from `sin()` are used to create the motion for a shape in the following example. The angle variable increases continually to produce changing values from `sin()` in the range of -1 to 1. These values are multiplied by the radius variable to magnify the values. The result is assigned to the `yoffset` variable and is then used to determine the y-coordinate of the ellipse on the following line. Notice how the circle slows down at the top and bottom of the screen and accelerates in the middle.

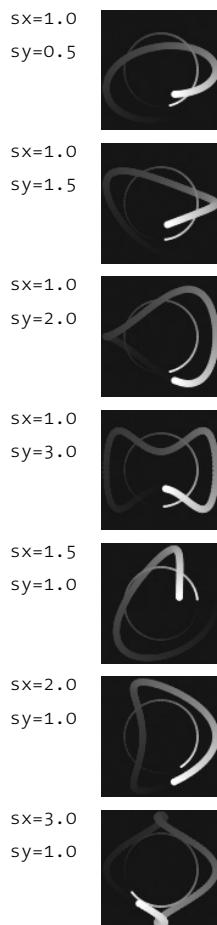


```
float angle = 0.0;      // Current angle           32-01
float speed = 0.1;      // Speed of motion
float radius = 40.0;    // Range of motion

void setup() {
  size(100, 100);
  noStroke();
  smooth();
}

void draw() {
  fill(0, 12);
  rect(0, 0, width, height);
  fill(255);
  angle += speed;
  float sinval = sin(angle);
  float yoffset = sinval * radius;
  ellipse(50, 50 + yoffset, 80, 80);
}
```

Adding values from `sin()` and `cos()` can produce more complex movement that remains periodic. In this example, a small dot moves in a circular pattern using values from `sin()` and `cos()`. A larger dot uses the same values for its base position but adds additional `sin()` and `cos()` calculations to produce an offset. You can easily see the difference between the two movements by looking at the positions of each point as the program runs.



```

sx=1.0      float angle = 0.0;      // Current angle           32-02
sy=0.5      float speed = 0.05;    // Speed of motion
            float radius = 30.0;   // Range of motion
            float sx = 2.0;
            float sy = 2.0;

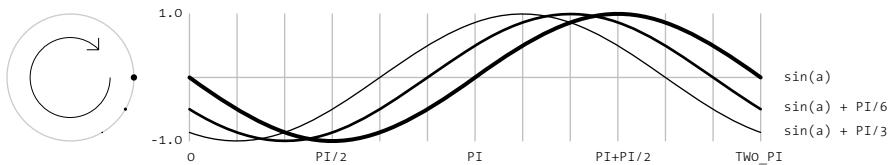
void setup() {
  size(100, 100);
  noStroke();
  smooth();
}

void draw() {
  fill(0, 4);
  rect(0, 0, width, height);
  angle += speed; // Update the angle
  float sinval = sin(angle);
  float cosval = cos(angle);
  // Set the position of the small circle based on new
  // values from sine and cosine
  float x = 50 + (cosval * radius);
  float y = 50 + (sinval * radius);
  fill(255);
  ellipse(x, y, 2, 2); // Draw smaller circle
  // Set the position of the large circles based on the
  // new position of the small circle
  float x2 = x + cos(angle * sx) * radius/2;
  float y2 = y + sin(angle * sy) * radius/2;
  ellipse(x2, y2, 6, 6); // Draw larger circle
}

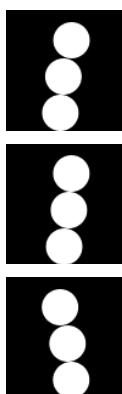
```

The *phase* of a function is one iteration through its possible values—for example, a single rise and fall sequence of a sine curve. *Phase shifting* occurs when the function is offset to start at a different point within the phase, such as the downward part of a sine curve rather than the top. Musical, visual, and numeric sequences all have phases, as do physical phenomena such as light waves. In the sound domain, Steve Reich's *Piano Phases* is a composition in which two pianos play the same sequence of notes, but one musician plays faster, creating a continuous phase shift as notes move in and out of synchrony. The player with the faster tempo periodically moves back into phase with the slower player, but is first offset by one note, then two notes, then three, etc. The performance ends when both pianos are back in phase and are playing the notes at the same time. In the visual domain, phase shifting creates the complex moiré patterns that result when two identical patterns are superimposed and shifted. In trigonometry, a cosine wave is a sine wave offset by 90° (p. 121). Shifting the angle used to generate

values from `sin()` provides the same sequence of numbers, but shifted across frames of animation:



The following two examples change the x-coordinate and diameter of circles to demonstrate phase shifting. In the first example, each circle has the same horizontal motion, but the motion is offset in time. In the second example, each circle has the same position and rate of growth cycle, but the growth cycle is offset.



```
float angle = 0.0;                                32-03
float speed = 0.1;

void setup() {
    size(100, 100);
    noStroke();
    smooth();
}

void draw() {
    background(0);
    angle = angle + speed;
    ellipse(50 + (sin(angle + PI) * 5), 25, 30, 30);
    ellipse(50 + (sin(angle + HALF_PI) * 5), 55, 30, 30);
    ellipse(50 + (sin(angle + QUARTER_PI) * 5), 85, 30, 30);
}
```



```
float angle = 0.0; // Changing angle          32-04
float speed = 0.05; // Speed of growth

void setup() {
    size(100, 100);
    noStroke();
    smooth();
    fill(255, 180);
}

void draw() {
    background(0);
```

```

    circlePhase(0.0);
    circlePhase(QUARTER_PI);
    circlePhase(HALF_PI);
    angle += speed;
}

void circlePhase(float phase) {
    float diameter = 65 + (sin(angle + phase) * 45);
    ellipse(50, 50, diameter, diameter);
}

```

32-04
cont.

Organic motion

Examples of organic movement include a leaf falling, an insect walking, a bird flying, a person breathing, a river flowing, and smoke rising. This type of motion is often considered idiosyncratic and stochastic.

Explorations in photography have led to a new understanding of human and animal motion. Étienne-Jules Marey and Eadweard Muybridge focused the lenses of their cameras on bodies in motion. In the 1880s Marey famously captured birds in flight, revealing the changing shapes of wings in image montages. Muybridge's sensational photographs of horses in motion used fifty cameras in sequence along a track to capture still images of a running horse. These photographs provided visual evidence regarding the horse's stride that had previously been impossible to collect. In the 1930s, Harold Edgerton invented technologies for capturing unique movements such as the beating of a hummingbird's wings and the motion of a starfish across the sea floor. Such images and films have changed the way the world is understood and can inform the way organic motion is approached using software.

Software explorations within the last twenty years have also provided a new understanding of organic motion. The Boids (pp. 473–475) software created by Craig Reynolds in 1986 simulates the flocking behavior found in birds and fish and has led to a new understanding of these emergent behaviors of animals. Karl Sims's Evolved Virtual Creatures from 1994 presents an artificial evolution, focusing on locomotion, where virtual block creatures engage in walking, jumping, and swimming competitions. The simple blocks demonstrate extraordinary emotive qualities as they twist and turn, appearing to struggle in their pursuit of locomotion.

Brownian motion, named in honor of the botanist Robert Brown, is jittery, stochastic motion that was originally ascribed to the movements of minute particles within fluids or the air; it appears entirely random. This motion can be simulated in software by setting a new position for a particle each frame, without preference as to the direction of motion. Leaving a trail of the previous positions of an element is a good technique for tracing its path through space.



```
float x = 50.0;           // X-coordinate
float y = 80.0;           // Y-coordinate

void setup() {
    size(100, 100);
    randomSeed(0);          // Force the same random values
    background(0);
    stroke(255);
}

void draw() {
    x += random(-2, 2);   // Assign new x-coordinate
    y += random(-2, 2);   // Assign new y-coordinate
    point(x, y);
}
```

32-05

The `sin()` and `cos()` functions can be used to create unpredictable motion when employed with the `random()` function. The following example presents a line with a position and direction, and every frame the direction is changed by a small value between -0.3 and 0.3. The position of the line at each frame is based on its current position and the slight variation to its direction. The `cos()` function uses the angle to calculate the next x-coordinate for the line, and the `sin()` function uses the same angle to calculate the next y-coordinate.



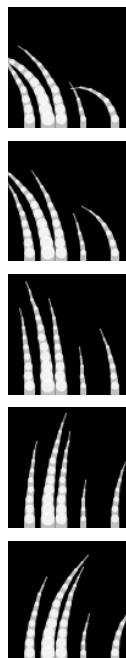
```
float x = 0.0;           // X-coordinate
float y = 50.0;           // Y-coordinate
float angle = 0.0;         // Direction of motion
float speed = 0.5;         // Speed of motion

void setup() {
    size(100, 100);
    background(0);
    stroke(255, 130);
    randomSeed(121);        // Force the same random values
}

void draw() {
    angle += random(-0.3, 0.3);
    x += cos(angle) * speed; // Update x-coordinate
    y += sin(angle) * speed; // Update y-coordinate
    translate(x, y);
    rotate(angle);
    line(0, -10, 0, 10);
}
```

32-06

The following example is an animated extension of code 22-04 (p. 200). Here the angle variable for the tail() function is continually changing to produce a swaying motion. Because the angles for each shape accumulate with each unit, the longest shapes with the most units swing from side to side with a greater curvature.



```
float inc = 0.0;                                         32-07

void setup() {
    size(100, 100);
    stroke(255, 204);
    smooth();
}

void draw() {
    background(0);
    inc += 0.01;
    float angle = sin(inc)/10.0 + sin(inc*1.2)/20.0;
    tail(18, 9, angle/1.3);
    tail(33, 12, angle);
    tail(44, 10, angle/1.3);
    tail(62, 5, angle);
    tail(88, 7, angle*2);
}

void tail(int x, int units, float angle) {
    pushMatrix();
    translate(x, 100);
    for (int i = units; i > 0; i--) {
        strokeWeight(i);
        line(0, 0, 0, -8);
        translate(0, -8);
        rotate(angle);
    }
    popMatrix();
}
```

The noise() function introduced in Math 4 (p. 127) is another resource for producing organic motion. Because the numbers returned from noise() are easy to control, they are a good way to add subtle irregularity to movement. The following example draws two lines to the screen and sets their endpoints based on numbers returned from noise().



```
float inc1 = 0.1;
float n1 = 0.0;
float inc2 = 0.09;
float n2 = 0.0;

void setup() {
    size(100, 100);
    stroke(255);
    strokeWeight(20);
    smooth();
}

void draw() {
    background(0);
    float y1 = (noise(n1) - 0.5) * 30.0; // Values -15 to 15
    float y2 = (noise(n2) - 0.5) * 30.0; // Values -15 to 15
    line(0, 50, 40, 50 + y1);
    line(100, 50, 60, 50 + y2);
    n1 += inc1;
    n2 += inc2;
}
```

32-08

The `noise()` function can also be used to generate dynamic textures. In the following example, the first two parameters are used to produce a two-dimensional texture and the third parameter increases its value each frame to vary the texture. Changing the `density` parameter increases the image resolution, and changing the `inc` parameter changes the texture resolution.



```
float inc = 0.06;
int density = 4;
float znoise = 0.0;

void setup() {
    size(100, 100);
    noStroke();
}

void draw() {
    float xnoise = 0.0;
    float ynoise = 0.0;
    for (int y = 0; y < height; y += density) {
        for (int x = 0; x < width; x += density) {
            float n = noise(xnoise, ynoise, znoise) * 256;
            fill(n);
```

32-09

```
    rect(y, x, density, density);
    xnoise += inc;
}
xnoise = 0;
ynoise += inc;
}
znoise += inc;
}
```

32-09
cont.

Exercises

1. Make a shape move with numbers returned from `sin()` and `cos()`.
2. Develop a kinetic composition using the concept of phase shifting.
3. Use code 32-06 as a base for creating a more advanced organism.



Extension 3: Vision

Text by Golan Levin

A well-known anecdote relates how, sometime in 1966, the legendary artificial intelligence pioneer Marvin Minsky directed an undergraduate student to solve “the problem of computer vision” as a summer project.¹ This anecdote is often resuscitated to illustrate how egregiously the difficulty of computational vision has been underestimated. Indeed, nearly forty years later the discipline continues to confront numerous unsolved (and perhaps unsolvable) challenges, particularly with respect to high-level “image understanding” issues such as pattern recognition and feature recognition. Nevertheless, the intervening decades of research have yielded a great wealth of well-understood, low-level techniques that are able, under controlled circumstances, to extract meaningful information from a camera scene. These techniques are indeed elementary enough to be implemented by novice programmers at the undergraduate or even high-school level.

Computer vision in interactive art

The first interactive artwork to incorporate computer vision was, interestingly enough, also one of the first interactive artworks. Myron Krueger’s legendary *Videoplace*, developed between 1969 and 1975, was motivated by his deeply felt belief that the entire human body ought to have a role in our interactions with computers. In the *Videoplace* installation, a participant stands in front of a backlit wall and faces a video projection screen. The participant’s silhouette is then digitized and its posture, shape, and gestural movements analyzed. In response, *Videoplace* synthesizes graphics such as small “critters” that climb up the participant’s projected silhouette, or colored loops drawn between the participant’s fingers. Krueger also allowed participants to paint lines with their fingers, and, indeed, entire shapes with their bodies; eventually, *Videoplace* offered more than fifty compositions and interactions. *Videoplace* is notable for many “firsts” in the history of human-computer interaction. Some of its interaction modules allowed two participants in mutually remote locations to participate in the same shared video space, connected across the network—an implementation of the first multiperson virtual reality, or, as Krueger termed it, an “artificial reality.” *Videoplace*, it should be noted, was developed before the mouse became the ubiquitous desktop device it is today, and was (in part) created to demonstrate interface alternatives to the keyboard terminals that dominated computing so completely in the early 1970s.

Messa di Voce (p. 511), created by this text’s author in collaboration with Zachary Lieberman, uses whole-body vision-based interactions similar to Krueger’s, but combines them with speech analysis and situates them within a kind of projection-based

augmented reality. In this audiovisual performance, the speech, shouts, and songs produced by two abstract vocalists are visualized and augmented in real time by synthetic graphics. To accomplish this, a computer uses a set of vision algorithms to track the locations of the performers' heads; this computer also analyzes the audio signals coming from the performers' microphones. In response, the system displays various kinds of visualizations on a projection screen located just behind the performers; these visualizations are synthesized in ways that are tightly coupled to the sounds being spoken and sung. With the help of the head-tracking system, moreover, these visualizations are projected such that they appear to emerge directly from the performers' mouths.

Rafael Lozano-Hemmer's installation *Standards and Double Standards* (2004) incorporates full-body input in a less direct, more metaphorical context. This work consists of fifty leather belts, suspended at waist height from robotic servomotors mounted on the ceiling of the exhibition room. Controlled by a computer vision-based tracking system, the belts rotate automatically to follow the public, turning their buckles slowly to face passers-by. Lozano-Hemmer's piece "turns a condition of pure surveillance into an 'absent crowd' using a fetish of paternal authority: the belt."²

The theme of surveillance plays a foreground role in David Rokeby's *Sorting Daemon* (2003). Motivated by the artist's concerns about the increasing use of automated systems for profiling people as part of the "war on terrorism," this site-specific installation works toward the automatic construction of a diagnostic portrait of its social (and racial) environment. Rokeby writes: "The system looks out onto the street, panning, tilting and zooming, looking for moving things that might be people. When it finds what it thinks might be a person, it removes the person's image from the background. The extracted person is then divided up according to areas of similar colour. The resulting swatches of colour are then organized [by hue, saturation and size] within the arbitrary context of the composite image" projected onsite at the installation's host location.³

Another project themed around issues of surveillance is *Suicide Box*, by the Bureau of Inverse Technology (Natalie Jeremijenko and Kate Rich). Presented as a device for measuring the hypothetical "despondency index" of a given locale, the *Suicide Box* nevertheless records very real data regarding suicide jumpers from the Golden Gate Bridge. According to the artists, "The *Suicide Box* is a motion-detection video system, positioned in range of the Golden Gate Bridge, San Francisco, in 1996. It watched the bridge constantly and when it recognized vertical motion, captured it to a video record. The resulting footage displays as a continuous stream the trickle of people who jump off the bridge. The Golden Gate Bridge is the premiere suicide destination in the United States; a 100-day initial deployment period of the *Suicide Box* recorded 17 suicides. During the same time period the Port Authority counted only 13."⁴ Elsewhere, Jeremijenko has explained that "the idea was to track a tragic social phenomenon which was not being counted—that is, doesn't count."⁵ The *Suicide Box* has met with considerable controversy, ranging from ethical questions about recording the suicides to disbelief that the recordings could be real. Jeremijenko, whose aim is to address the hidden politics of technology, has pointed out that such attitudes express a recurrent theme—"the inherent suspicion of artists working with material evidence"—evidence

obtained, in this case, with the help of machine vision-based surveillance.

Considerably less macabre is Christian Möller's clever *Cheese* installation (2003), which the artist developed in collaboration with the California Institute of Technology and the Machine Perception Laboratories of the University of California, San Diego.

Motivated, perhaps, by the culture shock of his relocation to Hollywood, the German-born Möller directed "six actresses to hold a smile for as long as they could, up to one and a half hours. Each ongoing smile is scrutinized by an emotion recognition system, and whenever the display of happiness fell below a certain threshold, an alarm alerted them to show more sincerity."⁶ The installation replays recordings of the analyzed video on six flat-panel monitors, with the addition of a fluctuating graphic level-meter to indicate the strength of each actress' smile. The technical implementation of this artwork's vision-based emotion recognition system is quite sophisticated.

As can be seen from these examples, artworks employing computer vision range from the highly formal and abstract to the humorous and sociopolitical. They concern themselves with the activities of willing participants, paid volunteers, or unaware strangers. They track people of interest at a wide variety of spatial scales, from extremely intimate studies of their facial expressions, to the gestures of their limbs, to the movements of entire bodies. The examples above represent just a small selection of notable works in the field and of the ways in which people (and objects) have been tracked and dissected by video analysis. Other noteworthy artworks that use machine vision include Marie Sester's *Access*; Joachim Sauter and Dirk Lüsebrink's *Zerseher* and *Bodymover*; Scott Snibbe's *Boundary Functions* and *Screen Series*; Camille Utterback and Romy Achituv's *TextRain*; Jim Campbell's *Solstice*; Christa Sommerer and Laurent Mignonneau's *A-Volve*; Danny Rozin's *Wooden Mirror*; Chico MacMurtrie's *Skeletal Reflection*, and various works by Simon Penny, Toshio Iwai, and numerous others. No doubt many more vision-based artworks remain to be created, especially as these techniques gradually become incorporated into developing fields like physical computing and robotics.

Elementary computer vision techniques

To understand how novel forms of interactive media can take advantage of computer vision techniques, it is helpful to begin with an understanding of the kinds of problems that vision algorithms have been developed to address, and of their basic mechanisms of operation. The fundamental challenge presented by digital video is that it is computationally "opaque." Unlike text, digital video data in its basic form—stored solely as a stream of rectangular pixel buffers—contains no intrinsic semantic or symbolic information. There is no widely agreed upon standard for representing the content of video, in a manner analogous to HTML, XML, or even ASCII for text (though some new initiatives, notably the MPEG-7 description language, may evolve into such a standard in the future). As a result, a computer, without additional programming, is unable to answer even the most elementary questions about whether a video stream contains a person or object, or whether an outdoor video scene shows daytime or nighttime, et

cetera. The discipline of computer vision has developed to address this need.

Many low-level computer vision algorithms are geared to the task of distinguishing which pixels, if any, belong to people or other objects of interest in the scene. Three elementary techniques for accomplishing this are frame differencing, which attempts to locate features by detecting their movements; background subtraction, which locates visitor pixels according to their difference from a known background scene; and brightness thresholding, which uses hoped-for differences in luminosity between foreground people and their background environment. These algorithms, described in the following examples, are extremely simple to implement and help constitute a base of detection schemes from which sophisticated interactive systems may be built.

Example 1: Detecting motion (p. 556)

The movements of people (or other objects) within the video frame can be detected and quantified using a straightforward method called frame differencing. In this technique, each pixel in a video frame F₁ is compared with its corresponding pixel in the subsequent frame F₂. The difference in color and/or brightness between these two pixels is a measure of the amount of movement in that particular location. These differences can be summed across all of the pixels' locations to provide a single measurement of the aggregate movement within the video frame. In some motion detection implementations, the video frame is spatially subdivided into a grid of cells, and the values derived from frame differencing are reported for each of the individual cells. For accuracy, the frame differencing algorithm depends on relatively stable environmental lighting, and on having a stationary camera (unless it is the motion of the camera that is being measured).

Example 2: Detecting presence (p. 557)

A technique called background subtraction makes it possible to detect the presence of people or other objects in a scene, and to distinguish the pixels that belong to them from those that do not. The technique operates by comparing each frame of video with a stored image of the scene's background, captured at a point in time when the scene was known to be empty. For every pixel in the frame, the absolute difference is computed between its color and that of its corresponding pixel in the stored background image; areas that are very different from the background are likely to represent objects of interest. Background subtraction works well in heterogeneous environments, but it is very sensitive to changes in lighting conditions and depends on objects of interest having sufficient contrast against the background scene.

Example 3: Detection through brightness thresholding (p. 559)

With the aid of controlled illumination (such as backlighting) and/or surface treatments (such as high-contrast paints), it is possible to ensure that objects are considerably darker or lighter than their surroundings. In such cases objects of interest can be distinguished based on their brightness alone. To do this, each video pixel's brightness is compared to a threshold value and tagged accordingly as foreground or background.



Example 1. Detects motion by comparing each video frame to the previous frame. The change is visualized and is calculated as a number.



Example 2. Detects the presence of someone or something in front of the camera by comparing each video frame with a previously saved frame. The change is visualized and is calculated as a number.



Example 3. Distinguishes the silhouette of people or objects in each video frame by comparing each pixel to a threshold value. The circle is filled with white when it is within the silhouette.



Example 4. Tracks the brightest object in each video frame by calculating the brightest pixel. The light from the flashlight is the brightest element in the frame; therefore, the circle follows it.

Example 4: Brightness tracking (p. 560)

A rudimentary scheme for object tracking, ideal for tracking the location of a single illuminated point (such as a flashlight), finds the location of the single brightest pixel in every fresh frame of video. In this algorithm, the brightness of each pixel in the incoming video frame is compared with the brightest value yet encountered in that frame; if a pixel is brighter than the brightest value yet encountered, then the location and brightness of that pixel are stored. After all of the pixels have been examined, then the brightest location in the video frame is known. This technique relies on an operational assumption that there is only one such object of interest. With trivial modifications, it can equivalently locate and track the darkest pixel in the scene, or track multiple and differently colored objects.

Of course, many more software techniques exist, at every level of sophistication, for detecting, recognizing, and interacting with people and other objects of interest. Each of the tracking algorithms described above, for example, can be found in elaborated versions that amend its various limitations. Other easy-to-implement algorithms can compute specific features of a tracked object, such as its area, center of mass, angular orientation, compactness, edge pixels, and contour features such as corners and cavities. On the other hand, some of the most difficult to implement algorithms, representing the cutting edge of computer vision research today, are able (within limits) to recognize unique people, track the orientation of a person's gaze, or correctly identify facial expressions. Pseudocodes, source codes, or ready-to-use implementations of all of these techniques can be found on the Internet in excellent resources like Daniel Huber's Computer Vision Homepage, Robert Fisher's HIPR (Hypermedia Image Processing Reference), or in the software toolkits discussed on pages 554-555.

Computer vision in the physical world

Unlike the human eye and brain, no computer vision algorithm is completely general, which is to say, able to perform its intended function given any possible video input. Instead, each software tracking or detection algorithm is critically dependent on certain unique assumptions about the real-world video scene it is expected to analyze. If any of these expectations are not met, then the algorithm can produce poor or ambiguous results or even fail altogether. For this reason, it is essential to design physical conditions in tandem with the development of computer vision code, and to select the software techniques that are most compatible with the available physical conditions.

Background subtraction and brightness thresholding, for example, can fail if the people in the scene are too close in color or brightness to their surroundings. For these algorithms to work well, it is greatly beneficial to prepare physical circumstances that naturally emphasize the contrast between people and their environments. This can be achieved with lighting situations that silhouette the people, or through the use of specially colored costumes. The frame-differencing technique, likewise, fails to detect people if they are stationary. It will therefore have very different degrees of success

detecting people in videos of office waiting rooms compared with videos of the Tour de France bicycle race.

A wealth of other methods exist for optimizing physical conditions in order to enhance the robustness, accuracy, and effectiveness of computer vision software. Most are geared toward ensuring a high-contrast, low-noise input image. Under low-light conditions, for example, one of the most helpful such techniques is the use of infrared (IR) illumination. Infrared, which is invisible to the human eye, can supplement the light detected by conventional black-and-white security cameras. Using IR significantly improves the signal-to-noise ratio of video captured in low-light circumstances and can even permit vision systems to operate in (apparently) complete darkness. Another physical optimization technique is the use of retroreflective marking materials, such as those manufactured by 3M Corporation for safety uniforms. These materials are remarkably efficient at reflecting light back toward their source of illumination and are ideal aids for ensuring high-contrast video of tracked objects. If a small light is placed coincident with the camera's axis, objects with retroreflective markers will be detected with tremendous reliability.

Finally, some of the most powerful physical optimizations for machine vision can be made without intervening in the observed environment at all, through well-informed selections of the imaging system's camera, lens, and frame-grabber components. To take one example, the use of a "telecentric" lens can significantly improve the performance of certain kinds of shape-based or size-based object recognition algorithms. For this type of lens, which has an effectively infinite focal length, magnification is nearly independent of object distance. As one manufacturer describes it, "an object moved from far away to near the lens goes into and out of sharp focus, but its image size is constant. This property is very important for gauging three-dimensional objects, or objects whose distance from the lens is not known precisely."⁷ Likewise, polarizing filters offer a simple, nonintrusive solution to another common problem in video systems, namely glare from reflective surfaces. And a wide range of video cameras are available, optimized for conditions like high-resolution capture, high-frame-rate capture, short exposure times, dim light, ultraviolet light, and thermal imaging. It pays to research imaging components carefully.

As we have seen, computer vision algorithms can be selected to negotiate best the physical conditions presented by the world, and physical conditions can be modified to be more easily legible to vision algorithms. But even the most sophisticated algorithms and the highest-quality hardware cannot help us find meaning where there is none, or track an object that cannot be described in code. It is therefore worth emphasizing that some visual features contain more information about the world, and are also more easily detected by the computer, than others. In designing systems to "see for us," we must not only become freshly awakened to the many things about the world that make it visually intelligible to us, but also develop a keen intuition about their ease of computability. The sun is the brightest point in the sky, and by its height also indicates the time of day. The mouth cavity is easily segmentable as a dark region, and the circularity of its shape is also closely linked to vowel sound. The pupils of the eyes emit an easy-to-track infrared retroreflection, and they also indicate a person's direction of

gaze. Simple frame differencing makes it easy to track motion in a video. The *Suicide Box* (p. 548) uses this technique to dramatic effect.

Tools for computer vision

It can be a rewarding experience to implement machine vision techniques from scratch using code such as the examples provided in this section. To make this possible, the only requirement of one's software development environment is that it should provide direct read-access to the array of video pixels obtained by the computer's frame-grabber.

Hopefully, the example algorithms discussed earlier illustrate that creating low-level vision algorithms from first principles isn't so hard. Of course, a vast range of functionality can also be obtained immediately from readily available solutions. Some of the most popular machine vision toolkits take the form of plug-ins or extension libraries for commercial authoring environments geared toward the creation of interactive media. Such plug-ins simplify the developer's problem of connecting the results of the vision-based analysis to the audio, visual, and textual affordances generally provided by such authoring systems.

Many vision plug-ins have been developed for Max/MSP/Jitter, a visual programming environment that is widely used by electronic musicians and VJs. Originally developed at the Parisian IRCAM research center in the mid-1980s and now marketed commercially by the California-based Cycling'74 company, this extensible environment offers powerful control of (and connectivity between) MIDI devices, real-time sound synthesis and analysis, OpenGL-based 3D graphics, video filtering, network communications, and serial control of hardware devices. The various computer vision plug-ins for Max/MSP/Jitter, such as David Rokeby's SoftVNS, Eric Singer's Cyclops, and Jean-Marc Pelletier's CV.Jit, can be used to trigger any Max processes or control any system parameters. Pelletier's toolkit, which is the most feature-rich of the three, is also the only one that is freeware. CV.Jit provides abstractions to assist users in tasks such as image segmentation, shape and gesture recognition, and motion tracking, as well as educational tools that outline the basics of computer vision techniques.

Some computer vision toolkits take the form of stand-alone applications and are designed to communicate the results of their analyses to other environments (such as Processing, Director, or Max) through protocols like MIDI, serial RS-232, UDP, or TCP/IP networks. BigEye, developed by the STEIM (Studio for Electro-Instrumental Music) group in Holland, is a simple and inexpensive example. BigEye can track up to 16 objects of interest simultaneously, according to their brightness, color, and size. The software allows for a simple mode of operation in which the user can quickly link MIDI messages to many object parameters, such as position, speed, and size. Another example is the powerful EyesWeb open platform, a free system developed at the University of Genoa. Designed with a special focus on the analysis and processing of expressive gesture, EyesWeb includes a collection of modules for real-time motion tracking and extraction of movement cues from human full-body movement; a collection of modules for analysis of occupation of 2D space; and a collection of modules for extraction of features from

trajectories in 2D space. EyesWeb's extensive vision affordances make it highly recommended for students.

The most sophisticated toolkits for computer vision generally demand greater familiarity with digital signal processing, and they require developers to program in compiled languages like C++ rather than languages like Java, Lingo, or Max. The Intel Integrated Performance Primitives (IPP) library, for example, is among the most general commercial solutions available for computers with Intel-based CPUs. The OpenCV library, by contrast, is a free, open source toolkit with nearly similar capabilities and a tighter focus on commonplace computer vision tasks. The capabilities of these tools, as well as all of those mentioned above, are continually evolving.

Processing includes a basic video library that handles getting pixel information from a camera or movie file as demonstrated in the examples included with this text. The computer vision capabilities of Processing are extended by libraries like Myron, which handles video input and has basic image processing capabilities. Other libraries connect Processing to EyesWeb and OpenCV. They can be found on the libraries page of the Processing website: www.processing.org/reference/libraries.

Conclusion

Computer vision algorithms are increasingly used in interactive and other computer-based artworks to track people's activities. Techniques exist that can create real-time reports about people's identities, locations, gestural movements, facial expressions, gait characteristics, gaze directions, and other attributes. Although the implementation of some vision algorithms requires advanced understanding of image processing and statistics, a number of widely used and highly effective techniques can be implemented by novice programmers in as little as an afternoon. For artists and designers who are familiar with popular multimedia authoring systems like Macromedia Director and Max/MSP/Jitter, a wide range of free and commercial toolkits are also available that provide ready access to more advanced vision functionalities.

Since the reliability of computer vision algorithms is limited according to the quality of the incoming video scene and the definition of a scene's quality is determined by the specific algorithms that are used to analyze it, students approaching computer vision for the first time are encouraged to apply as much effort to optimizing their physical scenario as they do their software code. In many cases, a cleverly designed physical environment can permit the tracking of phenomena that might otherwise require much more sophisticated software. As computers and video hardware become more available, and software-authoring tools continue to improve, we can expect to see the use of computer vision techniques increasingly incorporated into media-art education and into the creation of games, artworks, and many other applications.

Notes

1. <http://mechanism.ucsd.edu/~bill/research/mercier/2ndlecture.pdf>.
2. <http://www.fundacion.telefonica.com/at/rjh/eproyecto.html>.

3. <http://homepage.mac.com/davidrokeby/sorting.html>.
4. <http://www.bureauit.org/sbox>.
5. <http://www.wired.com/news/culture/0,1284,64720,00.html>.
6. <http://www.christian-moeller.com>.
7. <http://www.mellesgriot.com/pdf/pg11-19.pdf>.

Code

Video can be captured into Processing from USB cameras, IEEE 1394 cameras, or video cards with composite or S-video input devices. The examples that follow assume you already have a camera working with Processing. Before trying these examples, first get the examples included with the Processing software to work. Sometimes you can plug a camera into your computer and it will work immediately. Other times it's a difficult process involving trial-and-error changes. It depends on the operating system, the camera, and how the computer is configured. For the most up-to-date information, refer to the Video reference on the Processing website: www.processing.org/reference/libraries.

Example 1: Detecting motion

```
// Quantify the amount of movement in the video frame using frame-differencing

import processing.video.*;

int numPixels;
int[] previousFrame;
Capture video;

void setup(){
    size(640, 480); // Change size to 320 x 240 if too slow at 640 x 480
    video = new Capture(this, width, height, 24);
    numPixels = video.width * video.height;
    // Create an array to store the previously captured frame
    previousFrame = new int[numPixels];
}

void draw() {
    if (video.available()) {
        // When using video to manipulate the screen, use video.available() and
        // video.read() inside the draw() method so that it's safe to draw to the screen
        video.read(); // Read the new frame from the camera
        video.loadPixels(); // Make its pixels[] array available

        int movementSum = 0; // Amount of movement in the frame
        loadPixels();

        for (int i = 0; i < numPixels; i++) { // For each pixel in the video frame...

```

```

color currColor = video.pixels[i];
color prevColor = previousFrame[i];

// Extract the red, green, and blue components from current pixel
int currR = (currColor >> 16) & 0xFF; // Like red(), but faster (see p. 673)
int currG = (currColor >> 8) & 0xFF;
int currB = currColor & 0xFF;

// Extract red, green, and blue components from previous pixel
int prevR = (prevColor >> 16) & 0xFF;
int prevG = (prevColor >> 8) & 0xFF;
int prevB = prevColor & 0xFF;

// Compute the difference of the red, green, and blue values
int diffR = abs(currR - prevR);
int diffG = abs(currG - prevG);
int diffB = abs(currB - prevB);

// Add these differences to the running tally
movementSum += diffR + diffG + diffB;
// Render the difference image to the screen
pixels[i] = color(diffR, diffG, diffB);
// The following line is much faster, but more confusing to read
//pixels[i] = 0xff000000 | (diffR << 16) | (diffG << 8) | diffB;
// Save the current color into the 'previous' buffer
previousFrame[i] = currColor;
}

// To prevent flicker from frames that are all black (no movement),
// only update the screen if the image has changed.
if (movementSum > 0) {
    updatePixels();
    println(movementSum); // Print the total amount of movement to the console
}
}
}

```

Example 2: Detecting presence

```

// Detect the presence of people and objects in the frame using a simple
// background-subtraction technique. To initialize the background, press a key.

import processing.video.*;

int numPixels;
int[] backgroundPixels;
Capture video;

void setup() {
    size(640, 480); // Change size to 320 x 240 if too slow at 640 x 480
    video = new Capture(this, width, height, 24);
    numPixels = video.width * video.height;
}

```

```

// Create array to store the background image
backgroundPixels = new int[numPixels];
// Make the pixels[] array available for direct manipulation
loadPixels();
}

void draw() {
    if (video.available()) {
        video.read();           // Read a new video frame
        video.loadPixels();    // Make the pixels of video available

        // Difference between the current frame and the stored background
        int presenceSum = 0;

        for (int i = 0; i < numPixels; i++) { // For each pixel in the video frame...
            // Fetch the current color in that location, and also the color
            // of the background in that spot
            color currColor = video.pixels[i];
            color bkgdColor = backgroundPixels[i];

            // Extract the red, green, and blue components of the current pixel's color
            int currR = (currColor >> 16) & 0xFF;
            int currG = (currColor >> 8) & 0xFF;
            int currB = currColor & 0xFF;

            // Extract the red, green, and blue components of the background pixel's color
            int bkgdR = (bkgdColor >> 16) & 0xFF;
            int bkgdG = (bkgdColor >> 8) & 0xFF;
            int bkgdB = bkgdColor & 0xFF;

            // Compute the difference of the red, green, and blue values
            int diffR = abs(currR - bkgdR);
            int diffG = abs(currG - bkgdG);
            int diffB = abs(currB - bkgdB);

            // Add these differences to the running tally
            presenceSum += diffR + diffG + diffB;
            // Render the difference image to the screen
            pixels[i] = color(diffR, diffG, diffB);
            // The following line does the same thing much faster, but is more technical
            //pixels[i] = 0xFF000000 | (diffR << 16) | (diffG << 8) | diffB;
        }
        updatePixels();          // Notify that the pixels[] array has changed
        println(presenceSum);   // Print out the total amount of movement
    }
}

// When a key is pressed, capture the background image into the backgroundPixels
// buffer by copying each of the current frame's pixels into it.
void keyPressed() {
    video.loadPixels();
    arraycopy(video.pixels, backgroundPixels);
}

```

Example 3: Detection through brightness thresholding

```
// Determines whether a test location (such as the cursor) is contained within
// the silhouette of a dark object

import processing.video.*;

color black = color(0);
color white = color(255);
int numPixels;
Capture video;

void setup() {
    size(640, 480); // Change size to 320 x 240 if too slow at 640 x 480
    strokeWeight(5);
    video = new Capture(this, width, height, 24);
    numPixels = video.width * video.height;
    noCursor();
    smooth();
}

void draw() {
    if (video.available()) {
        video.read();
        video.loadPixels();
        int threshold = 127; // Set the threshold value
        float pixelBrightness; // Declare variable to store a pixel's color

        // Turn each pixel in the video frame black or white depending on its brightness
        loadPixels();
        for (int i = 0; i < numPixels; i++) {
            pixelBrightness = brightness(video.pixels[i]);
            if (pixelBrightness > threshold) { // If the pixel is brighter than the
                pixels[i] = white; // threshold value, make it white
            } else { // Otherwise,
                pixels[i] = black; // make it black
            }
        }
        updatePixels();

        // Test a location to see where it is contained. Fetch the pixel at the test
        // location (the cursor), and compute its brightness
        int testValue = get(mouseX, mouseY);
        float testBrightness = brightness(testValue);
        if (testBrightness > threshold) { // If the test location is brighter than
            fill(black); // the threshold set the fill to black
        } else { // Otherwise,
            fill(white); // set the fill to white
        }
        ellipse(mouseX, mouseY, 20, 20);
    }
}
```

Example 4: Brightness tracking

```
// Tracks the brightest pixel in a live video signal

import processing.video.*;

Capture video;

void setup(){
    size(640, 480); // Change size to 320 x 240 if too slow at 640 x 480
    video = new Capture(this, width, height, 30);
    noStroke();
    smooth();
}

void draw() {
    if (video.available()) {
        video.read();
        image(video, 0, 0, width, height); // Draw the webcam video onto the screen

        int brightestX = 0; // X-coordinate of the brightest video pixel
        int brightestY = 0; // Y-coordinate of the brightest video pixel
        float brightestValue = 0; // Brightness of the brightest video pixel

        // Search for the brightest pixel: For each row of pixels in the video image and
        // for each pixel in the yth row, compute each pixel's index in the video
        video.loadPixels();
        int index = 0;
        for (int y = 0; y < video.height; y++) {
            for (int x = 0; x < video.width; x++) {
                // Get the color stored in the pixel
                int pixelValue = video.pixels[index];
                // Determine the brightness of the pixel
                float pixelBrightness = brightness(pixelValue);
                // If that value is brighter than any previous, then store the
                // brightness of that pixel, as well as its (x,y) location
                if (pixelBrightness > brightestValue){
                    brightestValue = pixelBrightness;
                    brightestY = y;
                    brightestX = x;
                }
                index++;
            }
        }

        // Draw a large, yellow circle at the brightest pixel
        fill(255, 204, 0, 128);
        ellipse(brightestX, brightestY, 200, 200);
    }
}
```

Resources

Computer vision software toolkits

Camurri, Antonio, et al. Eyesweb. Vision-oriented software development environment. <http://www.eyesweb.org>.
Cycling'74 Inc. Max/MSP/Jitter. Graphic software development environment. <http://www.cycling74.com>.
Davies, Bob, et al. OpenCV. Open source computer vision library. <http://sourceforge.net/projects/opencvlibrary>.
Nimoy, Joshua. Myron (WebCamXtra). Library (plug-in) for Macromedia Director and Processing.
<http://webcamxtra.sourceforge.net>.
Pelletier, Jean-Marc. CVJit. Extension library for Max/MSP/Jitter. <http://www.iamas.ac.jp/~jovanoz/cv>.
Rokeby, David. SoftVNS. Extension library for Max/MSP/Jitter.
<http://homepage.mac.com/davidrokeby/softVNS.html>.
Singer, Eric. Cyclops. Extension library for Max/MSP/Jitter. <http://www.cycling74.com/products/cyclops.html>.
STEIM (Studio for Electro-Instrumental Music). BigEye. Video analysis software. <http://www.steim.org>

Texts and artworks

Bureau of Inverse Technology. *Suicide Box*. <http://www.bureauit.org/sbox>.
Bechtel, William. The Cardinal Mercier Lectures at the Catholic University of Louvain. Lecture 2, An Exemplar.
Neural Mechanism: The Brain's Visual Processing System. 2003, p.1.
<http://mechanism.ucsd.edu/~bill/research/mercier/2ndlecture.pdf>.
Fisher, Robert, et. al. HIPR (The Hypermedia Image Processing Reference).
<http://homepages.inf.ed.ac.uk/rbf/HIPR2/index.htm>.
Fisher, Robert, et al. CVonline: The Evolving, Distributed, Non-Proprietary, On-Line Compendium of
Computer Vision. <http://homepages.inf.ed.ac.uk/rbf/CVonline>.
Huber, Daniel, et al. The Computer Vision Homepage. <http://www-2.cs.cmu.edu/~cil/vision.html>.
Krueger, Myron. *Artificial Reality II*. Addison-Wesley Professional, 1991.
Levin, Golan and Lieberman, Zachary. *Messa di Voce*. Interactive installation, 2003.
<http://www.tmem.org/messa>.
Levin, Golan, and Zachary Lieberman. "In-Situ Speech Visualization in Real-Time Interactive Installation
and Performance." *Proceedings of the Third International Symposium on Non-Photorealistic Animation and
Rendering*. Annecy, France, June 7-9, 2004. http://www.flong.com/writings/pdf/messa_NPAR_2004_15odpi.pdf.
Lozano-Hemmer, Rafael. *Standards and Double Standards*. Interactive installation.
<http://www.fundacion.telefonica.com/at/rlh/eproyecto.html>.
Melles Griot Corporation. Machine Vision Lens Fundamentals. <http://www.mellesgriot.com/pdf/pg11-19.pdf>.
Möller, Christian. *Cheese*. Installation artwork, 2003. <http://www.christian-moeller.com>.
Rokeby, David. *Sorting Daemon*. Computer-based installation, 2003.
<http://homepage.mac.com/davidrokeby/sorting.html>.
Shachtman, Noah. "Tech and Art Mix at RNC Protest." *Wired News*, 27 August 2004.
<http://www.wired.com/news/culture/0,1284,64720,00.html>.
Sparacino, Flavia. "(Some) computer vision based interfaces for interactive art and entertainment installations."
INTER_FACE Body Boundaries, issue edited by Emanuele Quinz. *Anomalie* no. 2. Anomos, 2001.
http://www.sensingplaces.com/papers/Flavia_isea2000.pdf.

Code Index

This index contains all of the Processing language elements introduced within this book. The page numbers refer to the first use.

! (logical NOT), 57
!= (inequality), 52
% (modulo), 45
&& (logical AND), 57
() (parentheses)
 for functions, 18
 for precedence, 47
*= (multiply), 44
*+= (multiply assign), 49
+ (addition), 43
++ (increment), 48
+= (add assign), 48
,, (comma), 18
- (minus), 44
-- (decrement), 48
-= (subtract assign), 48
. (dot), 107
/ (divide), 44
/= (divide assign), 49
/* */ (comment), 18
// (comment), 17
; (semicolon), 19
< (less than), 51
<= (less than or
 equal to), 52
= (assign), 38
== (equality), 52
 for String objects, 109
> (greater than), 51
>= (greater than
 or equal to), 52
[] (array access), 301
 2D arrays, 312
 arrays of objects, 406
{ } (braces), 53
 and variable scope, 178
|| (logical OR), 57
(hex color), 93

abs(), 241
alpha(), 338
ambient(), 533
ambientLight(), 533
append(), 309
arc(), 124
arraycopy, 310
Array, 301
 length, 304
atan2(), 243

background(), 31
beginRaw(), 531
beginRecord(), 607
beginShape(), 69
bezier(), 30
bezierVertex(), 75
blend(), 351
blendColor(), 352
blue(), 337
boolean, 38
boolean(), 106
brightness(), 338
byte, 38
byte(), 106

camera(), 531
Capture, 556
ceil(), 49
char, 38, 102
char(), 106
class, 395
Client, 567
color, 38, 89
color(), 89
colorMode(), 91
constrain(), 237
copy(), 353
cos(), 118
createGraphics(), 614
createImage(), 362
createWriter(), 423
cursor(), 213
curveVertex(), 74

day(), 249
degrees(), 117
directionalLight(), 536
dist(), 238
draw(), 173

ellipse(), 30
ellipseMode(), 34
else, 55
else if, 56
endRaw(), 531
endRecord(), 607
endShape(), 69
exit(), 422
expand(), 309
extends, 456

false, 38
fill(), 32
filter(), 347
float, 37
float(), 106
floor(), 49
for, 61
frameCount, 173
frameRate(), 173

get(), 321
green(), 337

HALF_PI, 117
height, 40
hour(), 245
HSB, 89
hue(), 338

if, 53
image(), 96
int, 37
int(), 107
key, 225
keyCode, 227
keyPressed, 224
keyPressed(), 232
keyReleased(), 232

lerp(), 81
lightSpecular(), 536
line(), 27
loadFont(), 112
loadImage(), 96
loadPixels(), 356
loadStrings(), 428
loop(), 235

map(), 81
mask(), 354
max(), 50
millis(), 248
min(), 50
minute(), 245
month(), 249
mouseButton, 212
mouseDragged(), 229
mouseMoved(), 229
mousePressed, 212
mousePressed(), 229
mouseReleased(), 229
mouseX, 205
mouseY, 205

new
 for arrays, 303
 for objects, 399
nf(), 422
noCursor(), 213
noFill(), 33
noise(), 130
noiseSeed(), 131
noLoop(), 178
norm(), 80
noSmooth(), 33
noStroke(), 33
noTint(), 97

Object, 107, 395

PFont, 112
PI, 117
PImage, 96
pixels[], 356
pmouseX, 208
pmouseY, 208
point(), 25
pointLight(), 536
popMatrix(), 134
pow(), 80
print(), 20
println(), 20
PrintWriter, 423
 close(), 423
 flush(), 423
 println(), 424
pushMatrix(), 134

quad(), 29
QUARTER_PI, 117

radians(), 117
random(), 127
randomSeed(), 129
rect(), 29
rectMode(), 34
red(), 337
redraw(), 235
return, 194
RGB, 89
rotate(), 137
round(), 50

saturation(), 338
save(), 368
saveFrame(), 369
saveStrings(), 422
scale(), 138
second(), 245
Server, 567
set(), 324
setup(), 177
shorten(), 309
sin(), 118
size(), 24
 with P3D, 528
 with OPENGL, 528
 with PDF, 607
smooth(), 33
specular(), 536
split(), 429
splitTokens(), 430
spotLight(), 536
sq(), 79
sqrt(), 79
str(), 107
String, 103
 length(), 108
 endsWith(), 108
 equals(), 109
 startsWith(), 108
 substring(), 109
 toCharArray(), 108
 toLowerCase(), 109
 toUpperCase(), 109
stroke(), 32
strokeCap(), 33
strokeJoin(), 33
strokeWeight(), 33
super, 456

Index

This index contains mostly people, software, artwork, and programming languages. For topics, see the table of contents (pp. vii–xvii); for code, see the Code Index.

1:1 (Jevbratt), 566
3M Corporation, 553
3 Stoppages Étalon (Duchamp), 127
7–11 Email list, 563

AARON, 218
Aesthetics and Computation Group (ACG), xxiii, 682
Achituv, Romy, 549
ActionScript, 158, 166, 522–523, 565, 680–681, 686–687, 689, 691
Adair, Sandra, 384
Adobe, 4, 169, 683
Adobe After Effects, 166, 327, 379, 387
Adobe Flash, 157–158, 165–166, 267–268, 275, 278, 327, 436, 564–565, 624, 629, 642, 680–681, 683, 686, 701
Adobe Flash Lite, 624, 681
Adobe Garamond (font), 112
Adobe Illustrator, xxiii, 30, 77, 166, 143, 217, 271, 273, 607–608, 683
Adobe Photoshop, xxiii, 95, 166, 268, 276, 347, 355, 360, 384, 387–388, 391–392, 607–608, 611, 683
Adobe Premiere, 391–392
Adobe Streamline, 166
AAC (Advanced Audio Coding), 585
AIFF (Audio Interchange File Format), 585–586, 699
Aldus PageMaker, 605
Alexander, Ryan, 380
Alias Maya, 379, 387–388, 537, 680
AltSys, 170
Andrade, Laura Hernandez, 4
Apple IIe, xxiii
Apple Audio Units (AU), 591

Apple Computer, 3, 111, 537, 585, 699
Apple Logic Audio, 503, 591
Apple Mac G3, 383
Apple Mac G4, 383
Apple Macintosh (Mac), 9–11, 95, 111–112, 169, 205, 227, 367, 383, 521, 568–569, 574, 604, 639, 665, 682, 685
Apple Mac Mini, 639
Apple Mac OS, 264, 435, 665–666, 681
Apple Mac OS X, 16, 170, 435, 645, 649, 684
Apple QuickTime, 367, 383–384, 387–388
AppleScript, 681
Arduino, 521, 633, 640, 641, 645–646, 648–649, 681, 685
Arp, Jean, 127
Ars Electronica Festival, 618
ART+COM, 498
ASCII (American Standard Code for Information Interchange), 102–103, 226–227, 549, 565, 664–668, 670, 691, 699
Athena, 387
ATI, 537
AT&T/Bell, 564
Audacity, 591
AutoCAD, 217, 529, 537
Autodesk 3ds Max, 268, 276, 391–392, 537
AutoDesq Revit, 537
AutoLISP, 522, 681
Autonomedia, 564
Avid/Digidesign Pro Tools, 591
AVR (Atmel), 640
awk, 517, 684
Babbitt, Milton, 580–581
Bach, J. S., 581
Bailey, Chris, 581
Balkin, Amy, 267
Baran, Paul, 564
Barr, Alfred, 291
Barragán, Hernando, 633
BASIC, xxiii, xxiv, 152, 264, 522, 604–605, 640, 642, 681
BASIC Stamp 2 (Parallax), 640
BasicX-24 (NetMedia), 642
Bass, Saul, 327
Baumgärtel, Tilman, 564
Bauhaus, 149
BBC Acorn Archimedes, 264
Beach Culture, 605
Beethoven, Ludwig van, 581
BEFLIX, 315, 681
Bell Laboratories, 315, 580–581, 604
Bentley Systems
GenerativeComponents, 537
Berliner, Emile, 579
Berlow, David, 170
Bernard (a k a Flip 1), 508
BIAS Peak, 591
BigEye, 554
Binary Runtime Environment for Wireless (BREW), 625
Binary space partition (BSP), 527
Binder, Maurice, 327
bitforms gallery, 164, 166–167, 525, 547, 603, 633
BitTorrent, 571
Blackwell, Lewis, 605
Blender, 276, 576
Blinkenlights (Chaos Computer Club), 618
Blonk, Jaap, 511
Bluetooth, 619, 621–622, 624, 641, 645, 683
Blyth, Steven, 512
Boids (Reynolds), 295, 473, 475, 497
Boole, George, 38, 61, 669
Boolean algebra, 38
Boulez, Pierre, 581
Braitenberg, Valentino, 473–474
Brakhage, Stan, 413
Brecht, Bertolt, 564
Brooklyn Academy of Music (BAM), 515–516
Brown, Robert, 295
Brownian motion, 295
Brunelleschi, Filippo, 525
Bunting, Heath, 563–564
Bureau of Inverse Technology, 548, 634
Burke, Phil, 592
Burton, Ed, 263–264, 413, 499
Byrne, David, 581

- C, 7, 264, 515–517, 522–523, 592, 640, 642, 682–685, 693, 697
- C++, 264, 271, 383, 507–508, 511–512, 515–516, 522–523, 555, 592, 599, 640, 679, 681–682
- CAD (computer-aided drawing software), 217, 526, 537–538
- Cage, John, 127, 579
- CalArts School of Art, 564
- California Institute of Technology (Caltech), 388, 549
- Cameron, Dan, 387
- Campbell, Jim, 549
- Carmack, John, 525
- Carnegie Mellon University, xxi
- Carnivore, 566, 568–569
- Carson, David, 605
- Cascading Style Sheets (CSS), 93
- CCRMA Synthesis ToolKit (STK), 592
- Chang, Zai, 6
- Cheese (Möller), 549
- Cho, Peter, 257, 327
- CIA World Fact Book, 267
- Citron, Jack, 315
- CityPoems*, 617, 624
- ChucK, 592, 682
- Cloaca* (Delvoye), 461
- Clash of the Titans*, 387
- Close, Chuck, 606
- CODE (Petzold), 648
- Cohen, Harold, 218
- Columbia–Princeton Electronic Music Center, 580
- Commodore C–64, 272
- Commodore VC–20, 272
- Common Lisp, 592
- Complexification.net*, 6, 157
- Computational Beauty of Nature, The* (Flake), 469
- Computers and Automation*, 603
- Computer Clubhouse, 680
- Computer Lib / Dream Machines (Nelson), 3
- Computer Vision Homepage (Huber), 552
- Coniglio, Mark, 512
- “Constituents for a Theory of the Media” (Enzensberger), 564
- Conway, John, 461, 463, 467–468, 475
- Cook, Perry, 592
- Cooper, Muriel, 327
- Cope, David, 581
- CorelDRAW, 608
- Cosic, Vic, 563–564
- Costabile, Sue (SUE.C), 503–504
- Craighead, Alison, 618
- Crawford, David, 316
- Crystal Castle, 525
- Csikszentmihályi, Chris, 507–508, 634
- CSIRAC, 580
- Csuri, Charles, 217
- Cuba, Larry, 1, 315
- Cullen, Mathew, 379–380
- CV.Jit, 554
- Cybernetic Serendipity, 101, 603
- Cycling '74, 554, 592
- Cyclops, 554
- Dada, 149–150
- Davies, Char, 526
- Davis, Joshua, 564–565
- Deck, Barry, 112
- Deleuze and Guattari, 564
- Delvoye, Wim, 461
- De Mol, Gerry, 275
- Design By Numbers (DBN), xxiv, 552–523, 682
- Designers Republic, The, 605
- Dextro, 316
- Dialtones* (Levin et al.), 617–618
- Digidesign, 587, 591
- Dine, Jim, 606
- DJ I, Robot Sound System*, 506–509
- Dodgeball, 617, 624
- Domain Name System (DNS), 566
- DrawBot, 169, 682, 684
- Drawing with Computers* (Wilson), 152, 217, 604
- Drawn* (Lieberman), 413
- DuBois, R. Luke, 579
- Duchamp, Marcel, 127, 633
- Dunne, Tony, 634
- Dürer, Albrecht, 525, 612
- DXF, 520, 529–531
- Dynabook, 3
- Eagle, 272
- écal (école cantonale d'art de Lausanne), 271
- Eclipse, 571, 625
- ECMAScript, 681, 683
- Edelweiss Series* (Maywa Denki), 634
- Edgerton, Harold, 295
- Edison, Thomas, 579
- Eighth Istanbul Biennial, 387
- Eimart, Herbert, 580
- Electronic Arts, 585
- ELIZA, 101
- Emacs, 516
- Emigre, 605
- End of Print, The* (Blackwell), 605
- Endless Forest, The* (Tale of Tales), 274–277
- Engelbart, Douglas, 205
- Eno, Brian, 581
- Enron, 268
- Enzensberger, Hans Magnus, 564
- EPS, 606
- Euler's method, 7, 494
- Every Icon* (Simon), 565
- Evolved Virtual Creatures* (Sims), 295
- Experiments in Art and Technology (E.A.T.), 633
- Extend Script, 683
- Eye magazine, 605
- Eye Catching* (Steinkamp), 386–389
- EyesWeb, 554–555
- EZIO (NIQ), 642
- Feingold, Ken, 633
- Ferro, Pablo, 327
- Final Cut Pro (FCP), 383, 503
- Final Scratch, 507
- Fischinger, Oskar, 413
- Fisher, Robert, 552
- Flake, Gary William, 469
- Flight404.com*, 6
- Flight Simulator, 525
- Foldes, Peter, 315
- FontLab, 170
- Fontographer, 170
- Fortran, 522
- Fractal.Invaders* (Tarbell), 156–159
- Franceschini, Amy, 267
- Franke, Uli, 260, 271
- Free Radicals*, 413
- Friendster, 617
- Fourier, Jean-Baptiste-Joseph, 584
- Fourier transform, 585, 588, 590
- Futurist, 279, 579
- Gabo, Nam, 633
- Galloway, Alexander R., 563
- Game of Life, 461, 463, 465–466, 468, 475
- Gardner, Martin, 461, 463
- Garton, Brad, 581
- Gerhardt, Joseph, 391–392
- Gestalt psychology, 584

- GIF, 95–96, 98–99, 421, 700–701
 Girroir, Jonathan, 506–509
 Google, 568, 617
 GPS (Global positioning system), 619, 621
 Graffiti, 223
 GRASS, 681
 Groeneveld, Dirk, 333
 GNU Image Manipulation Program (GIMP), 95, 347, 355, 607–608
 GNU Public License (GPL), 271
 Gnutella, 566, 571
 GPU (graphics processing unit), 536–537
 Graphomat Z64 (Zuse), 603
 Greenwald, Simon, 525
 Greie, Antye (AGF), 503–504
 Grzinic, Marina, 563
 GUI (Graphical user interface), 435–436, 448, 450, 499, 604, 634, 679–680, 683, 685, 700
 Gutenberg, Johannes, 111
 Gutenberg archive, 433
 Guttmann, Newmann, 580
 Gysin, Andreas, 373
 Hall, Grady, 379
 Handel, George Frideric, 581
 Hansen, Mark, 515–516, 634
 Harmon, Leon, 604
 Harvard University, xxi
 Harvey, Auriea, 275
 Hewlett-Packard (HP), 604, 610
 Hawkinson, Tim, 633
 Hawtin, Richie, 507
 Hébert, Jean-Pierre, 217, 606
Hektor (Lehni, Franke), 260, 270–273
 Henry, John, 507
 Henry, Pierre, 580
 Hiller, Lejaren, 581
 Hoefer, Jonathan, 112
 Hodgin, Robert, 6, 692
 Hokusai, 612
 Hongik University, 5
 Hong, Leon, 5, 375
 Hooke's law, 263, 487
 Howard Wise gallery, 603
 HTML (HyperText Markup Language), 9–11, 93, 268, 427, 549, 564–565, 568–569, 621, 624, 665–666, 684
 HTTP (Hypertext Transfer Protocol), 567–569, 623
 Huber, Daniel, 552
 Huff, Kenneth A., 606
 Hypermedia Image Processing Reference (HIPR), 552
 HyperTalk, 522
 IANA, 569
 IBM, 315, 537, 580, 585, 604, 620, 702
 IC (integrated circuit), 639, 647
 I-Cube X (Infusion Systems), 642
 IEEE 1394 camera, 556
If/Then (Feingold), 633
 Igarashi, Takeo, 538
 Igoe, Tom, 635, 648
 Ikarus M, 170
Incredibles, The, 315
 Internet Explorer, 565
 Internet Protocol (IP), 566–567, 569, 589, 645
 Impressionist, 279
Inaudible Cities: Part One (Semiconductor), 392
 InDesign, 683
 Infrared, 553, 621
 Inge, Leif, 581
 Inkscape, 77, 607–608
Installation (Greenwald), 526
 Institute of Contemporary Arts (ICA), 101, 522
 Intel Integrated Performance Primitives (IPP), 512, 555
 Interaction Design Institute Ivrea (IDII), xxi, 634
 i/o 360°, 565
I/O/D 4 ("The Webstalker"), 566
 IRCAM, 554, 581, 592
 Ishii, Hiroshi, 634
 Ishizaki, Suguru, 327
 ISO 216 standard, 611
 Iwai, Toshio, 512, 549
 James, Richard (Aphex Twin), 582
 Jarman, Ruth, 391–392
 Java, 7, 9–11, 146, 161–162, 263–264, 271, 499, 521–523, 528, 555, 564–565, 571, 574, 592, 622, 625–626, 642, 663, 673, 677, 679–683, 686–690, 699–700
 Java 2 Micro Edition (J2ME), 625
 Java applet, 9–11, 264, 521, 656, 657, 675, 699
 Java Archive (JAR), 10–11, 700
 Java Core API, 271
 JavaScript, 268, 271, 522, 624, 680, 681, 683
 Java Virtual Machine (JVM), 680
 Jeremijenko, Natalie, 548
 Jevbratt, Lisa, 566
 jMax, 592
 Jodi, 563–566
 Jones, Crispin, 634
 Jones, Ronald, 275
 Jonzun Crew, 508
 JPEG, 95–96, 162, 421, 606, 611, 620, 701
 JSyn (Java Synthesis), 592
 Julesz, Bela, 603
 Kay, Alan, 3
 Kim, Tai-kyung, 5
 Kimura, Mari, 582
 King's Quest, 525
 Klee, Paul, 217
 Knowlton, Kenneth C., 315, 604
 Krueger, Myron, 255, 512, 547
 Kusaite, Lina, 275
 Kuwakubo, Ryota, 634
 La Barbara, Joan, 511
 Langton, Chris, 469, 471
Putto8 2.2.2.2 (Rees), 524, 526
 LaserWriter, 111, 604
 Lee, Soo-jeong, 5
 Led Zeppelin, 161
Legible City, The (Shaw, Groeneveld), 333
 Lehni, Jürg, 260, 271–273
 Leibniz, Gottfried Wilhelm, 61
Letterscapes (Cho), 327
 LettError, 111, 168–170, 605
 Levin, Golan, 259, 333, 511–512, 547, 617–618
 Lewis, George, 582
 LeWitt, Sol, 217
 Li, Francis, 617
 Lia, 316, 496
 Lialina, Olia, 563–564
 Licko, Zuzana, 112, 605
 Lieberman, Zachary, 413, 512–512, 547
 Lifecode, 425–426
 Limewire, 571
 Ling, 522–523, 555, 565, 683, 686–687, 689, 691
 Linklater, Richard, 383
 Linotype, 111
 Linux, 4, 9–11, 508, 521, 568–569, 625, 645, 649

- Listening Post* (Rubin, Hansen), 514–517
 LISP, 101
 LiveScript, 683
 Local area network (LAN), 568–569
 Logo, xxii, 2, 217, 522, 681
 Lovink, Geert, 564
 Lozano-Hemmer, Rafael, 546, 548
 Lucent Technologies, 515
 Lucier, Alvin, 590
 Luening, Otto, 580
 Lüsebrink, Dirk, 549
 Lye, Len, 413
- Machine Art exhibition, 291, 633
 Machine Perception Laboratories, 549
 MacMurtrie, Chico, 549
 Macromedia Director, 166, 387–388, 554–555, 642, 683, 686
 Maeda, John, xix, xxiii, xxiv, 3, 5, 158, 333, 564, 606, 682
 Malka, Ariel, 372
 Makela, P. Scott, 605
 Mandelbrot, Benoit, 153
 Manovich, Lev, 565
 Marble Madness, 525
 Marconi, Guglielmo, 579
 Marey, Étienne-Jules, 295
 Mark of the Unicorn Digital Performer, 591
 Markov chain, 581
 Marx, Karl, 267–268
 Massachusetts Institute of Technology (MIT), xix, xxiii, xxiv, 327, 634, 680, 682, 693, 695
 Masterman, Margaret, 101
 Mathews, Max, 580, 586, 591, 683
 MATLAB, 522
 Max/MSP/Jitter, 2, 503–504, 515–517, 522, 554–555, 571, 580, 592, 642, 683–685
 Maya Embedded Language (MEL), 680, 683
 Maywa Denki, 634
 McCarthy, John, 101
 McCartney, James, 592
 McCay, Winsor, 315
 McLaren, Norman, 413
 Medusa, 387
 MEL, 680, 683
 Mendel, Lucy, 507
Messa di Voce (Tmema et al.), 510–513, 547
- Metrowerks Codewarrior, 512
 Microsoft, 4, 111, 169, 436, 508, 525, 537, 585, 702
 Microsoft Direct3D, 537
 Microsoft Visual Basic, 436
 Microsoft Windows, 9, 11, 264, 367, 421, 435–436, 511, 521, 568, 625, 645, 649, 665–666, 685
 MIDI (Musical Instrument Digital Interface) 162, 554, 588–589, 591–592, 618, 621, 623, 642, 645, 683, 685
 Mignonneau, Laurent, 549
 MIME, 623
 Mims, Forest M., III, 648
Mini-Epoch Series, The (Semiconductor), 390–393
Mini Movies (AGF+SUE.C), 500, 502–505
Minitasking (Schoenerwissen/OfCD), 562, 566
 Minsky, Marvin, 547
 MIT Media Laboratory, xxiii, 327, 634, 680, 682, 702
 MixViews, 591
 MP3, 162, 421, 585, 621, 623
 MPEG-7, 549
 Mobile Processing, 521, 622–626, 683
 Mohr, Manfred, 217, 602, 606
 Möller, Christian, 549
 Moore, F. Richard, 592
 Mophun, 625
 Morisawa, 605
 Motion Theory, 378–381
 MTV, 384
[murmur], 618
 Museum of Modern Art, The (MOMA), 291, 633
 MUSIC, 580, 591
 Musique concrète, 580–581
 Muybridge, Eadweard, 295, 373
 Myron, 555
 MySQL, 267–268
 Myst, 525
- Nakamura, Yugo, 565
 Nake, Frieder, 217, 603
 Napier, Mark, 566
 Napster, 507, 571
 Nees, Georg, 217, 603
 Nelson, Ted, 3
 “net.art”, 563–564
net.art (Baumgärtel), 564
net.art 2.0 (Baumgärtel), 564
- NetBeans, 625
 Netscape Navigator, 565, 683
 Newton, Isaac, 477, 488
 New York University (NYU), 6, 634
New York Times, The, 150
 Ngan, William, 497
 Nimoy, Josh, 512
 Noll, A. Michael, 217, 603
 Nokia, 517, 618–619, 625
 Nmap, 569
 NSA (National Security Agency), 268
NTNTNT (Cal Arts), 564
 NTSC, 367
 NTT DoCoMo’s i-Mode, 624
 Nuendo, Steinberg, 591
 null, 40, 701
 NURBS (Non-uniform Rational B-splines), 526
 nVidia, 537
 Nyquist theorem, 585
- OBJ, 529–531
Objectivity Engine, The (Paterson), 164–167
 Oliveros, Pauline, 582
 Olsson, Krister, 589
Once-Upon-A-Forest (Davis), 564
 On, Josh, 267–268
 oN-Line System (NLS), 205
 OpenCV, 512, 555
 OpenGL, 512, 520, 528, 531, 537, 554, 684
 Open source, 4, 268, 271, 512, 521, 555, 591, 625–626, 640, 684
 OpenType, 111, 169
 Oracle database, 264
 OSC (Open Sound Control), 516–517, 571, 589
 oscP5 (Schlegel), 571
Osmose (Davies), 526
 O’Sullivan, Dan, 635, 648
 Oswald, John, 581
 Owens, Matt, 565
- Pad, 435
 Paik, Nam June, 633
 PAL, 367
 Palm Pilot, 223, 625
 Palm OS, 625
 Panasonic, 625
 Papert, Seymour, 2, 217
 Parallax, 640
 Parallel Development, 516
 Pascal, 522

- Paterson, James, 165–166, 316, 565, 606
 Paul, Les, 580
 PBASIC, 642, 681
 PC, 10, 227, 388, 625, 665, 682
 PCB (printed circuit board), 639, 640
 PCM (pulse-code modulation), 585–586, 699, 702
 PDF, 520, 606–608, 682
 Pelletier, Jean-Marc, 554
 Penny, Simon, 549
 Perl, 146, 515–517, 522–523, 565, 571, 681, 684
 Perlin, Ken, 130
 Personal area network (PAN), 621–622
 Petzold, Charles, 648
 Phidgets, 642
 Philips, 634
 PHP, 267–268, 522–523, 565, 682, 684
 PHPMyAdmin, 268
Physical Computing (O’Sullivan, Igoe), 648
Piano Phases (Reich), 293
 PIC (Microchip), 272, 640
 PIC Assembler, 271–272
 PIC BASIC, 681
 Pickard, Galen, 507
 Pickering, Will, 516
 Pixar, 315
Pixillation (Schwartz), 315
 PNG (Portable Network Graphics), 95–96, 98–99, 606, 622, 701
 Pocket PC, 625
 PoemPoints, 617
 Pong, 256, 590, 618
 PortAudio, 512
 PostScript, 111, 143, 169–170, 522, 604–605, 681
 Poyntor, Rick, 605
Practical Electronics for Inventors (Scherz), 648
Practice of Programming, The (Kernighan, Pike), 252
Playstation (Davis), 564
 Public Enemy, 581
 Puckette, Miller, 2, 592, 684
 Pulse-code modulation (PCM), 585–586, 699, 702
 Pure Data (Pd), 592, 684–685
 Python, 146, 170, 517, 522–523, 681–682, 684
 Q*bert, 525
 Quartz Composer, 684
 Qualcomm, 625
 Quest3D, 275–276
 R, 515, 517
 Rabby, Fiona, 634
 Radial, 503–504
 RAM, 701
 RandomFont Beowulf (*LettError*), 111, 168–170, 605
 Rauschenberg, Robert, 606
Ray Gun, 605
 Razorfish, 565
 RCA Mark II Sound Synthesizer, 580
 Readme!, 563
 Real-Time Cmix, 592
 Rees, Michael, 526
 Reeves, Alec, 585
 Reich, Steve, 293
 Reichardt, Jasia, 522
 Reiniger, Lotte, 315
 RenderMan, 315
 R.E.M. “Animal” (Motion Theory), 378–381
 ResEdit, 170
 Resnick, Mitchel, 471, 680
 Reynolds, Craig, 295, 473, 497
 Rhino, 271, 537
 Rich, Kate, 548
 Riley, Bridget, 151
 Ringtail Studios, 275
 Risset, Jean-Claude, 581
 RoboFog, 170
 Rokeby, David, 548, 554
 Rotoshop, 383–384, 413
 Royal Academy of Arts, 169
 Royal College of Art, 634
 Rozin, Danny, 549
 RS-232, 639, 554, 640, 645
 Rubin, Ben, 515, 634
 Ruby, 681, 684
 Ruby on Rails, 684
 Runge-Kutta method 7, 494
 Russolo, Luigi, 579
 Sabiston, Bob, 383–384, 413
 Saito, Tatsuya, 198, 529, 568
 Samyn, Michaël, 275
 Sauter, Joachim, 549
 Schaeffer, Pierre, 580
 Scheme, 522
 Scherz, Paul, 648
 Schiele, Egon, 217
 Schlegel, Andreas, 498, 571
 Schmidt, Karsten (a.k.a. tox1), 4, 518
 Schoenerwissen/OfCD, 562
 Schöffer, Nicolas, 633
 Schumacher, Michael, 582
 Schwartz, Lillian, 315
 Scientific American, 461, 463
 Scratch, 680
 Screen Series (Snibbe), 549
Scriptographer (Lehni, Franke), 270–273, 683
 Seawright, James, 633
 sed, 684
 Semiconductor, 390–393, 646
 Sessions, Roger, 580
 Sester, Marie, 549
 Shannon, Claude, 669
Shape of Song (Wattenberg), 160–163
 Shaw, Jeffrey, 333
 Shiffman, Daniel, 6
 Shockwave Flash (SWF), 158, 565
 Short Messaging Service (SMS), 617, 619, 621
 SHRDLU, 101
 sh/tcsh, 515, 684
 Shulgin, Alexi, 563–564
 Silicon Graphics, 529, 537
 Simon, John F. Jr., 413, 565
SimpleTEXT, 618
 Sims, Karl, 295
 Sinclair Spectrum, 264
 Singer, Eric, 554
 Sketchpad, 217
 SketchUp, 538
Slacker, 383
 Slimbach, Robert, 112
 Smalltalk, 685
 Smith, Laura, 275
 Snake, 618
 Snibbe, Scott, 413, 549
 Social Mobiles (SoMo), 634
Sodaconstructor (Burton), 262–265, 413, 499
 Soda Creative Ltd., 263–264
 SoftVNS, 554
 Solidworks, 537
 Sommerer, Christa, 549
 Sonami, Laetitia, 582
 Sonic Inc., 392
 Sony, 634
 Sony Ericsson, 625
 Sorenson, 388
 Sorting Daemon (Rokeby), 548, 554
 Sound Films, 392

- Spark Fun Electronics, 640
 SQL (Structured Query Language), 685
 Srivastava, Muskan, 5
Standards and Double Standards (Lozano-Hemmer), 547–548
 Star Wars, 315
 Strausfeld, Lisa, 327
 Stedelijk Museum, 218
 Stehura, John, 315
 STEIM (Studio for Electro-Instrumental Music), 554
 Steinkamp, Jennifer, 387–388
 Stipe, Michael, 379–380
 Stockhausen, Karlheinz, 580
 Stone, Carl, 582
Stop Motion Studies (Crawford), 316
 Studies in Perception I, (Knowlton Harmon), 604
Substrate (Tarbell), 6, 154, 156–159
 Sudol, Jeremi, 507
Suicide Box, 548, 554
 Sun Java Wireless Toolkit, 625
 Sun Microsystems, 521, 537, 625, 682
 SuperCollider, 571, 592, 685
 Sutherland, Ivan, 217
 SVG (Scalable Vector Graphics), 77, 520, 606, 624
 Symbian, 625
 Synergenix, 625
- Tale of Tales, 274–277
Talmud Project (Small), 327
Takeluma (Cho), 327
 Takis, 633
 Tarbell, Jared, 6, 155–156, 606
 Tangible Media Group (TMG), 634
 TARGA, 368, 606, 702
 Tate Gallery, 218
 T|C Electronics Powercore, 587
 tcpdump, 568–569
 TCP/IP, 554, 569, 589
 Technics, 507
 Teddy (Igarashi), 538
Telephony (Thompson, Craighead), 618
 TeleNav, 619
 Teleo (Making Things), 642
 Tesla, Nikola, 579
 Text-to-speech (TTS), 516–517
They Rule (On et al.), 266–269
 Thomson, Jon, 618
 TIFF, 368, 507, 606, 608, 611, 702
- Toy Story*, 315
 Tmema, 510–513
 Transmission Control Protocol (TCP), 569
Tron, 315
 Truax, Barry, 581
 TrueType, 111
 Tsai, Wen-Ying, 633
 TurboGears, 684
 Turkle, Sherry, 5
Turux (Lia, Dextro), 316
Type, Tap, Write (Maeda), 333
 Tzara, Tristan, 150
- Überorgan* (Hawkinson), 633
 Unicode, 432, 665–668
 University of California Berkeley, 589
 Los Angeles (UCLA), xxi, 4, 5, 574
 San Diego (UCSD), 549
 University of Cincinnati (UC), xxiii
 University of Genoa, 554
 UNIX, 227, 435, 517, 569, 645
 U.S. Army Ballistic Missile Research Laboratories, 603
 USB, 556, 640–645, 701
 User Datagram Protocol (UDP), 554, 569, 589
 Ussachevsky, Vladimir, 580
 UTF-8, 665
 Utterback, Camille, 549
- Valicenti, Rick, 605
 van Blokland, Erik, 169–170
 van Blokland, Petr, 170
 VanDerBeek, Stan, 315
 Vanderlands, Rudy, 605
 van Rossum, Just, 169
 Vaucanson's Duck, 461
Vehicles: Experiments in Synthetic Psychology (Braitenberg), 473
 Venice Biennale, 391
 Verschoren, Jan, 275
 “Video Games and Computer Holding Power” (Turkle), 5
Videoplace (Krueger), 547
 Visual Language Workshop (VLW), 327, 702
 Visual programming languages (VPL or VL), 679–680
 Vitiello, Stephen, 582
 VLW font format, 112, 702
 Vogel, Peter, 632, 633
 Von Ehr, Jim, 170
 Vonnegut, Kurt, 507
- von Neumann, John, 461
 Vorbis codec, 585
 Voxel, 527
 vvvv, 685
- Wacom, 383
Waking Life, 382–385, 413
 Walt Disney, 315, 379
 Wang, Ge, 592
 Warner Bros. Records, 379
 Wattenberg, Martin, 161–162, 606
 Watz, Marius, 374
 WAV, 585–586, 621, 623, 702
 Wayfinder Systems, 619
 Wegman, William 606
 Weizenbaum, Joseph, 101
 Whitney, James, 315
 Whitney, John, 315
 Whitney Museum of American Art, 516
 Wilhelm Imaging Research, 610
 Wilson, Mark, 152, 217, 604
 Winograd, Terry, 101
 Wiring, 521, 633, 640, 641, 645–646, 648–649, 685
 Wright, Frank Lloyd, 333
 Wrongbrowser (Jodi), 566
 Wolfram, Steven, 461, 463–464, 467, 475
 Wolfenstein 3D, 525
 Wong, Andy, 507
 Wong, Yin Yin, 327
- Xenakis, Iannis, 581
 Xerox Palo Alto Research Center (PARC), 3, 205
 Xerox, 507
 xHTML Mobile, 624
 XML, 421, 427–428, 520, 549, 621, 624, 702
- Yamaha Digital Mixing Engine (DME), 516
Yellow Arrow, 618
 Youngblood, Gene, 388
- Ziggurat (font), 112
 Zooming user interface (ZUI), 435