Processing

ビジュアルデザイナーとアーティストのためのプログラミング入門

ベン・フライ、ケイシー・リース 著
中西泰人 監訳
安藤幸央、澤村正樹、杉本達應 翻訳

Processingのバイブル、待望の日本語化

BNN

序文：ジョン・マエダ
日本語版特別インタビュー：
幸村真佐男、杉原 聡、真鍋大度、和田 永

デザイン、アートのためのプログラミング環境／言語「Processing」。
その開発者自身がプログラミング初学者に向けて解説した、
Processing入門の決定版。

Processing 2.0 / 3.0に対応

Processing

ビジュアルデザイナーとアーティストのためのプログラミング入門

ベン・フライ、ケイシー・リース 著
中西泰人 監訳
安藤幸央、澤村正樹、杉本達應 翻訳

BNN
Bug News Network

PROCESSING: A Programming Handbook for Visual Designers and Artists, 2nd edition
by Casey Reas and Ben Fry

Japanese translation published by arrangement with The MIT Press through The English Agency (Japan) Ltd.

The Japanese edition was published in 2015 by BNN, Inc.
1-20-6, Ebisu-minami, Shibuya-ku,
Tokyo 150-0022 JAPAN
www.bnn.co.jp

Printed in Japan

For the ACG

日本語版に寄せて

Processingのプロジェクトは、まさにその始まりから日本にゆかりがあります。ケイシーでもベンでもない様々な人たちに使ってもらおうと、2001年の秋に武蔵野美術大学で開催されたDesign By Numbers（DBN）のワークショップで最初のバージョンがリリースされました。「リビジョン0003」と番号が振られたProcessingの初期のバージョンを学生たちに使ってもらったのです。私たちを指導していたジョン・マエダのDBNは意図的に機能を最低限に絞っていたので、DBNが扱わなかった色や大きな描画領域、3D等の機能を使ってもらいました。

この初めてのワークショップはとても重要でした。私たちがProcessingを通して議論していた自分たちのアイデアを実際に試すことができたからです。そして日本で得た経験をもとにProcessingを改良し、MITでの一学期間の授業として使えるようにすることができました。

そして数年が経った2005年にも、Processingは日本で大きく後押しされました。東京タイポディレクターズクラブから賞をいただくことになり、私たちは日本を訪れたのです。いつも遠距離でコラボレーションをしていた私たちは、この時にまる一週間ずっと一緒にいることができました。それはとても希少で貴重な時間でした。カフェで仕事をしたりおにぎりを頬張りながら東京での時間を活用して、様々な重要な機能を実装したのです。それが、高速な3D描画を実現した1.0のβ版のリリースにつながりました。

1.0のβ版はProcessingにとって新たなターニングポイントでした。これが世界中の人たちに堂々と公開した最初のバージョンだからです。この時からProcessingを使う人の数がかなり増え始め、多くの人がProcessingを使ってコードの書き方を勉強し、先生たちが授業の教材としてとりあげ始めたのです。

最初のProcessingのワークショップから15年が経とうとする今、Processing 3.0のリリースも間近となりました。そして私たちの『Processing：ビジュアルデザイナーとアーティストのためのプログラミング入門』の日本語版を出版できることとなり、とてもワクワクしています。日本語版には、コードによる表現を行う素晴らしい日本人アーティストとデザイナーへのインタビューが追加されました。寄稿していただいた幸村真佐男さん、真鍋大度さん、和田永さん、杉原聡さんに感謝するとともに、彼らの思考と作品を共有できたことをとても嬉しく思います。

私たちは、皆さんがこの本を楽しく読んでくれること願っています。そして、いずれはProcessingで作ったものを世界中に広がっている素晴らしいProcessingコミュニティにシェアすることを願っています！

ベン・フライとケイシー・リース
2015年8月　ボストンとロサンゼルスにて

監訳者のことば

本書はProcessingを作り出したベン・フライとケイシー・リースによるProcessingのバイブルとも言うべき書籍です。第1版は日本語化されておらず、ここに第2版の日本語版を出版できたことをとても嬉しく思います。日本語版には原著に加えて4人の日本人クリエイターのインタビューを追加しました。新しい表現をもたらすアルゴリズムや言語、ツールを自ら開発したり、枯れた技術に新たな意味を見出したりと、テクノロジーを創造的に使いこなしている方ばかりです。こうした思考は、素材に新たな可能性を見出す一流のデザイナーや独自の表現技法を編み出す一流のアーティストに共通するものと言えるでしょう。

ProcessingはJavaをベースに視覚的な表現を作りやすくした言語です。開発環境もシンプルなため、デザイナーやアーティストだけでなくプログラミングの初学者にも適しています。その一方で、グラフィックスライブラリとしてJavaから呼び出したり、関数型言語であるScalaからも呼び出せます。またバージョン毎に描画も高速化され、試作や習作のコードだけでなく実際に展示する作品のコードを書くことも可能です。3.0ではデバッガが正式に組み込まれ、本格的な開発もしやすくなりました。

しかしProcessingは単に便利な道具ではありません。Processingは、入り口は親しみやすく、進むにつれて広がりのあるプログラマー／クリエイターたちの広場でもあります。他の人が書いたコードを調べて変更・拡張し学べるよう、本書や開発環境には豊富なサンプルコードが提供されています。ある程度コードが書けるようになれば、OpenProcessingのサイトでコードをシェアしてお互いに学び合うことができます。さらに上級者なら自分の書いたコードをライブラリやツールとして公開しやすいよう、そのテンプレートが用意されています。「Processing」が指すものは、言語と開発環境だけではなく、学習の方法論、そしてユーザーと開発者のオープンなコミュニテイまでをも含んでいます（ベンとケイシーの「日本語版に寄せて」の最後の文にもこの考え方が現れています）。

さらにベンとケイシーは、他の言語や開発環境も使いこなせるようになることを勧めています。Processingにツールをインストールすれば、JavaScriptやPythonでコードが書けます。またオープンソースのワンボードマイコンArduinoの開発環境はProcessingをベースにしていて、同じような学習の仕組みとコミュニティができています。高速な処理が特徴のopenFrameworksはProcessingに大きな影響を受けて開発され、その枠組みにはProcessingと同じ名前の関数が使われています。Processingは他の広場と有機的に連携するだけではなく、他の広場を生み出した源でもあるのです。Processingがなければ、メディアアート、コンピューテーショナルデザインやデジタルファブリケーションの広まりは、今とは違ったものになっていたでしょう。コンピュータを創造的に使いこなす人々の広場であるProcessingは、「新しい創造のあり方を創造した」と言っても過言ではありません。

ベンとケイシーの信念はこの本の中に散りばめられています。私自身も監訳の作業を進めるなかで、その信念に改めて感動を覚えました。言語と開発環境としてのProcessingの学習が一通り済んだら、ぜひコミュニティとしてのProcessingに参加してください。本書が、新しいあり方の創造性を身につける大きな後押しになることを願っています。

中西泰人

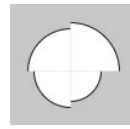
Processing

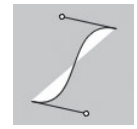
ビジュアルデザイナーと
アーティストのための
プログラミング入門

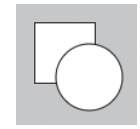
目 次


015

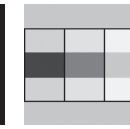

026


029


030


032


033


034


035


037

043


045

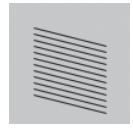

066

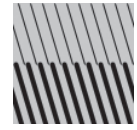

085

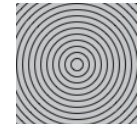
091


092

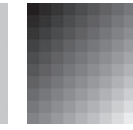

093

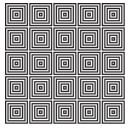

104

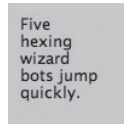

106


108


110

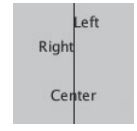

114


114



157

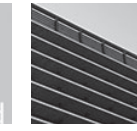
160



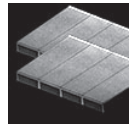
163

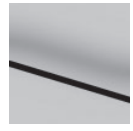
164

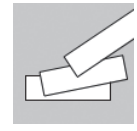
166

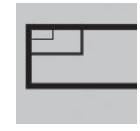

170

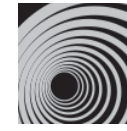

172


174

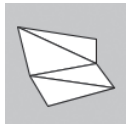

178


184


187

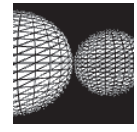

189


195


196


199


203


207


209

217

220


224

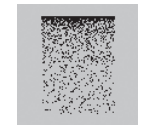

227

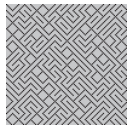

280


282


285


304

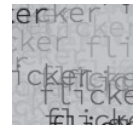

307


309

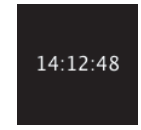

321


329


330


333

335



338

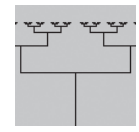

339


348


352


360

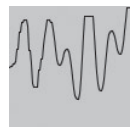

362


364


385

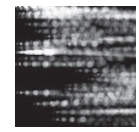

429

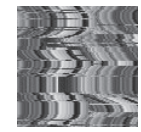

432


433


444


452


453


454

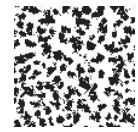

466


471

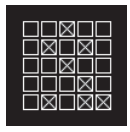

472


476


491


493

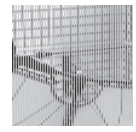

517


529


536

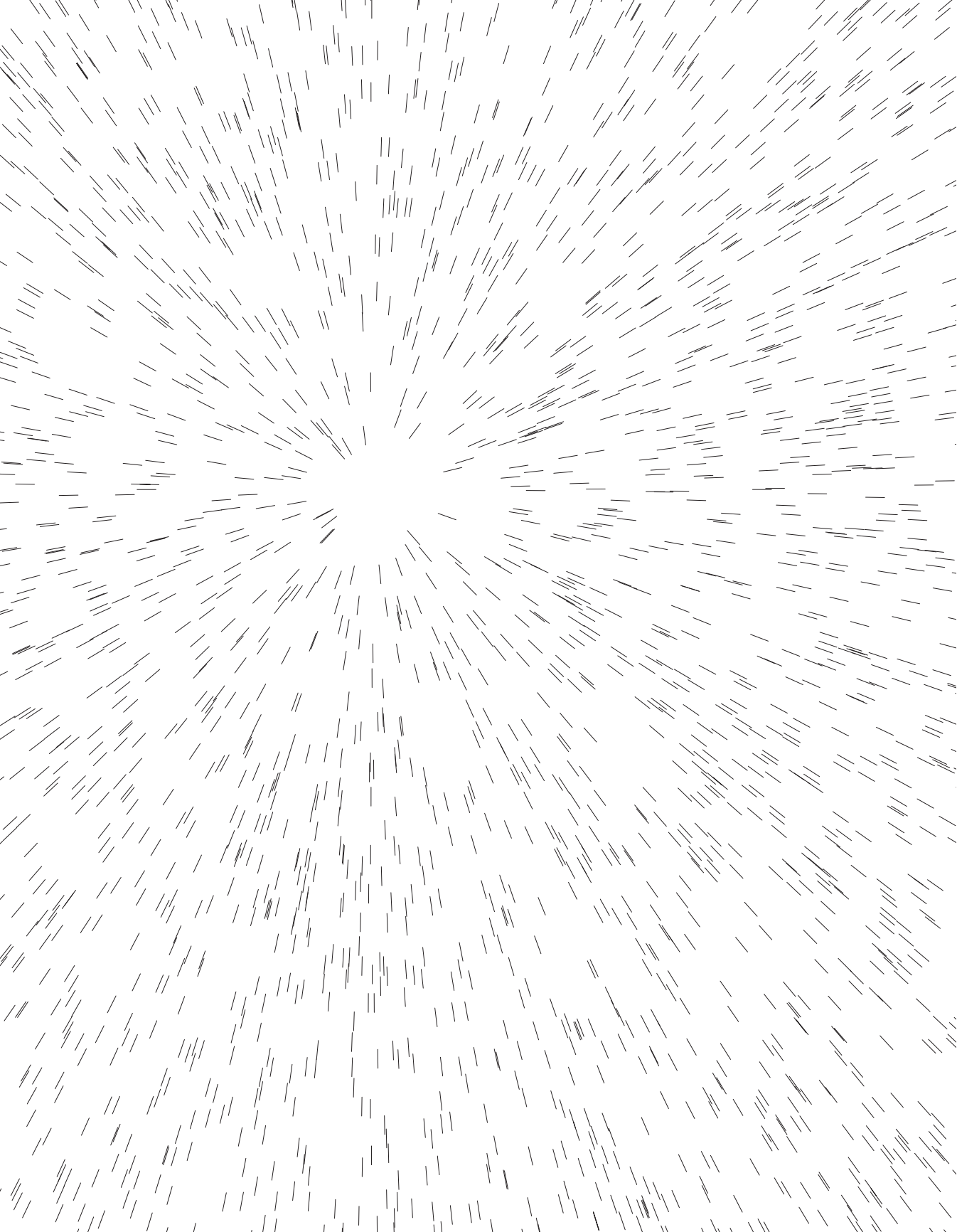

543


548


553

560

564



序文

私がかつてMITで運営していた大学院のスタジオには、ユニークな才能が多く集まりました。彼らは、表現の手段としてコンピュータを扱うという根源的なバランスの課題を抱えていました。プログラミングをすることが、自分のデザイン的・芸術的な欲求の邪魔になることは望まない一方で、新しい視覚的な発見を導く小さな道すじを探る高度なコードをためらいなく書きました。この2つの知性のあり方は絶え間なく対立しますが、彼らの結論はシンプルでした。「どっちもやる」のです。

テクノロジーとアートの間にある深い淵を軽やかに横切っていくハイブリッドな才能を持つ人々は、アカデミックなシステムのなかでは突然変異体のようなものです。伝統的に、大学は技術系の学生や芸術系の学生を育成します。しかし、同じ一人の知性にその2つの知のあり方を混在させないのです。1990年代の間、こうした標準的な考えになんとか従わなかった突然変異体たちは私を探し出し、私自身も彼らを探して接触しようとしました。そうしたユニークな人たちを呼び集め出会わせることに、私は情熱を注いだのです。そのようにして、私はケイシー・リースとベン・フライに出会いました。

教師に最大の賛辞が送られるのは、教え子が教師を超えた時と言われます。彼らと一緒に働き始めてすぐに、その時はやってきました。そして、彼らが作ったProcessingはとどめの一撃でした。2人はインターネットをうまく活用して、最初は10人、100人、そして何万人と、ハイブリッドな才能を持つ世界中の人たちの興味を引きつけ、視覚的な実験への要求を大きく押し上げました。私が出かけるどんな場所でも、若いテクノロジーアーティストたちがProcessingのことをいつも話していて、ケイシーとベンに感謝を伝えるようお願いされます。

私は今はKleiner Perkins Caufiled & Byers（カリフォルニア州メンローパークにあるベンチャーキャピタル）のデザインパートナー（共同経営者）ですが、シリコンバレーにいるデザインとエンジニアリングの両方の才能を持つ多くの人たちが経済にインパクトを与える様子を目撃しています。コンピュータを使いこなすデザイナーたちは、Flipboard、Pinterest、Square、AirBnBやNestのような会社の製品に欠かせない技巧と繊細な感性をもたらしています。こうした会社のハイブリッドな才能を持つ人たちと彼らのチームは、そのデザイン過程でProcessingを直接的には使っていないとしても、「Processingの中の人」（最初は2人でしたが、ピクセルの魔術師であるダニエル・シフマンを含めたボーイズバンドに今は拡張されました）の仕事を高く評価しています。

私はここで、ベンとケイシーに感謝の意を表します。Processingが切り拓くコンピュータを使ったアートとデザインの新たな場所へついていこうとする全ての人たちに代わり、私は祈っています。芸術的な知性と計算的な知性をつなぐ架け橋を完璧なものにしようとする絶え間ない追求の中で、あなたたちがたくさんの眠れない夜を過ごすようにと。私は楽しみにしています。あなたたちが世界中のギャラリーや広告代理店やベンチャー企業にインパクトを与え続けることを。私たちは期待しています。コンピュータを使ったアートが、技術的な含蓄を持たない単なる芸術になる道をあなたたちが先導してくれることを。私たち全員が新しい場所にたどり着くには、あなたたちが頼りなのです。

ジョン・マエダ
Kleiner Perkins Caufiled & Byers デザインパートナー

はじめに

本書は、視覚芸術の文脈のなかでプログラミングをする考え方を紹介するために書かれました。コンピュータに詳しい人たちを横で眺めているけれど、ソフトウェアを書いてインタラクティブで視覚的な作品を作ることに関心があり、少ししかあるいは全くそうした経験がないような人を対象にしています。私たちは、コミュニケーションと表現のメディアとしてのソフトウェアの可能性に心を躍らせています。そして本書が、その可能性を多くの人たちに伝えることを願っています。

Processingは、10年以上にわたるソフトウェア開発と教育経験の成果です。教室やコンピュータ室、大学の地下室、美術大学や芸術機関のなかで、そのアイデアは絶えずテストされてきました。著者らはUCLA（カリフォルニア大学ロサンゼルス校）、イブレア・インタラクションデザイン研究所、ハーバード大学、カーネギーメロン大学の関連するコースで教鞭をとり、このトピックについてのワークショップや講演を世界中の会議や学会で行ってきました。本書の内容は、そこでの生徒や同僚である教育者たちからの寛容なフィードバックを通して絶えず改善されてきたものです。この努力の成果を、より大きく、より多様なコミュニティに配布したいと考え、そうして洗練されてきたカリキュラムを本書は提供します。

内容

本書では4種類のコンテンツをとりあげます。本の大部分は、ソフトウェアの特定の要素について述べるチュートリアルの章と、それらがアートにどう関係するかを述べる章に分かれます。変数、関数、オブジェクト指向プログラミングといったソフトウェアの文法や概念を紹介し、ソフトウェアに関連した写真術と絵画のようなトピックをカバーします。関連した画像や説明とともに、たくさんの短い典型的なサンプルコードをとりあげます。アニメーション、パフォーマンス、インスタレーションを含む幅広い領域におけるより先進的でプロフェッショナルなプロジェクトについて、作者にインタビューしています。付録は技術的なトピックについての参照表と、より掘り下げた解説を提供します。参考文献は、関連するトピックについての追加的な書籍やWebサイト等のリストです。本書で扱う専門用語は用語集で定義します。オンライン上（https://www.processing.org/handbook/）に公開した章では、画像処理や音や電子工学を含んだ、より進んだ領域を探索するための簡潔な導入を提供します。

本書には、著者らが開発したProcessingのプログラミング言語を使って書いたサンプルコードがたくさん載っています。Processingはフリーでオープンソースのプログラミング言語／開発環境であり、学生、アーティスト、デザイナー、建築家、研究者、ホビイストが学習そしてプロトタイピングや制作に用います。Processingは、アートやデザインに使われる独占的なソフトウェアにとってかわるものとして、アーティストとデザイナーによって開発されました。このプロジェクトでは、プログラミング言語と開発環境と教育方法を統合し、学習と探究を一体化します。Processingのソフトウェアは、人々を初心者から先進的なプログラマーへとスムースに移行させます。Processingの言語は将来の学習の良い基礎となります。この文章で紹介されている言語やより深いプログラミング

の概念の技術的側面は、他のプログラミング言語、特にアートのなかでよく使われる言語にうまく移行できます。

本書で示すサンプルコードの大部分は、視覚的にミニマルなスタイルをとっています。これはProcessingソフトウェアの限界を示しているのではなく、それぞれのサンプルを簡潔で明快にしようと著者らが意図的に決断したからです。飾り気がなくがらんとしたこれらサンプルの特長が、自分自身の視覚的な言語へと拡張しようとする動機を読者に与えてくれることを望んでいます。

本書の読み方

本書での学習は、単に文字を読むだけではありません。サンプルコードを実行し、修正し、インタラクトすることが大事です。料理をせずに料理を学べないのと同じように、プログラミングをせずにプログラミングを学習することは不可能です。サンプルの多くは、マウスやキーボードを動かして反応させないとしっかり理解することはできません。Processingのソフトウェアと本書のサンプルコードは全てWebからダウンロードでき、これから知的な探求を始めるべく実行させることができます。Processingのソフトウェアはwww.processing.org/downloadから、サンプルはwww.processing.org/handbookからダウンロードしてください。

テキストを補足する本質的な内容を、コード、図、画像が伝えます。本書は、視覚的な志向がある人たちのために書かれているので、図や画像はテキストと同じくらい注意深く読まれることでしょう。タイポグラフィ的および視覚的な約束事は文章を読みやすくします。テキストの中のコードの要素は、区別しやすいよう等幅フォントを使っています。それぞれのコードのサンプルには、参照しやすいよう順番に番号を付けました。最初の行の右のマージンに番号を表示していて、「15-02」という番号は15章の2番目のサンプルを意味しています。サンプルコードの多くは、変数の値を変えると違ったように実行されます。絵の右側に数字が示されているときは（310ページ）、その数字がそのイメージを生成するために使われています。

ケイシーによるイントロダクション

私は子供のときにコンピュータで遊び始めました。私はゲームで遊び、家族のApple IIeでBASICやLogoを使って簡単なプログラムを書きました。そのマシンを探求しテストすることに数年を費やしましたが、私は絵を描く方が好きで、コンピュータへの関心は消えました。1990年代の初めにシンシナティ大学でデザインを学ぶ学生として、私は1年生のときにAdobeのPhotoshopとIllustratorを使い始めましたが、私のデザインスタジオのクラスでは3年生までそれを使うことが許されませんでした。視覚的な形を通してコンポジションと意味を構築できるよう、自分の目と手を訓練することに最初の2年間を費やしました。アイコンと文字を鉛筆で描き、Plaka（ペリカン社の黒いマットなインク）で塗ることに私は自分のエネルギーを集中させました。1つの洗練されたイメージへ向かう制作のなかで、私は何百枚もの紙のスケッチを描きました。この作業では、コンセプトと紙の上の最終的な結果の間の中間的な段階で、ソフトウェアを道具として使いました。

時とともに、私は印刷されたメディアを作ることからソフトウェアへと移行しました。CD-ROMを使うマルチメディア産業が誕生したとき、音と動画と静止画と情報デザインへの自分の興味を統合すべく、私はその領域で働きました。1990代半ばのインターネットの勃興とともに、データベースが

統合された大きなWebサイトを作ることに私は集中しました。私の仕事が紙からスクリーンに移ったので、固定的なグリッドと静的な情報の階層構造は、解像度とコンポジションが可変的な、再構成可能で動的なシステムへと進化しました。素材のディテールと静的なコンポジションに費やされた時間とエネルギーは、運動と反応のディテールへと費やされるようになりました。私は形を生成し、ふるまいを規定し、インタラクションを成立させる、リアルタイムなプロセスを作り上げることに集中しました。こうした興味をより高度なレベルで追求するためには、コンピュータをプログラムすることを学ぶ必要があるとはっきり思いました。コンピュータで遊んだ子供の時期と、コンピュータを使って働いた時期を経て、私は新しい道を歩き始めたのです。

1997年に私はジョン・マエダと出会い、MITのAesthetics and Computation Groupの学生たちの実験的なソフトウェア作品に触れました。彼らは伝統的な美術の知識にコンピュータサイエンスの考えを融合することで、新しいタイプの作品を作っていました。これらの作品と出会うことで私の新しい方向性が浮かび上がり、1998年にコンピュータをプログラムすることを本格的に学び始め、次の年にはMITの大学院での研究を始めました。私がソフトウェアの消費者から生産者へと変わってゆくにしたがい、そこでの私の時間は個人的に別のものに一変していきました。私はテクノロジーについての視野を文化と美術の歴史の観点から広げました。

MITメディアラボの大学院生だったとき、私は複数の分野を勉強して得られた知識を結合する一人の個人という文化に触れました。共通の知識はコンピュータの技術であり、建築、アート、数学、デザイン、音楽といった他の分野のバックグラウンドをみんなが持っていました。その当時は、ほとんどのソフトウェア環境が洗練されたグラフィックスを作る能力と洗練されたプログラミング言語の両方を提供しておらず、MITの先輩や友人たちは自分の要求に見合うソフトウェアを自分で作っていました。こうした新しいソフトウェアのツールは、コンピュータサイエンスの知識と視覚的なカルチャーを統合するユニークなカルチャーを生み出しました。こうしたカルチャーやソフトウェアを技術的な分野や研究所の外側の人々にもアクセスしやすくしたいという願いが、この12年という時間をProcessingの開発に捧げる私の動機でした。私は本書が、芸術のなかにソフトウェアのリテラシーを増やす触媒として働くことを望んでいます。

ベンによるイントロダクション

似たようなキャリアにたどりついた多くの人たちと同じように、私はモノを分解してそれがどう動いているかを理解することに興味を持ってきました。最初は家電製品を分解してその部品たちを比べ、似たような部品を探しました。電話機とラジオを分解し尽くしたのち、私はソフトウェアへ移行しました。コンピュータは、無限に電話番号を持っているかのように果てのない範囲の問題を投げかけました。「IBM Basic by Microsoft」と書かれた黄色く焼けたバインダーを使って、私の父は「forループ」を私に教えてくれました。私は徐々にプログラミングを独学しました。たいていは誰かのコードをじっくりと読み、時には他のことをやらせるよう書き換えました。そのうちに、より簡単にゼロからコードを書けるようになりました。

私はグラフィックデザインにまた別の関心を持っていました。タイポグラフィ、レイアウト、そしてコンポジションに興味がありました。家族の友人がデザイン会社を経営していて、私はそれがこの世で最も面白い仕事のように思っていたのです。その後、私はデザインスクールに応募しました。

ユーザーインターフェイスデザインを勉強するか、インタラクティブなマルチメディアCD-ROMを作ることが、私のこの2つの関心を交わらせる唯一の可能性だと考えたからです。デザインスクールに通うことは、私にとって意義深いものでした。というのも、私のソフトウェアへの関心を含めて、他の領域に応用できる思考とクリエイティブなスキルをそれが与えてくれたからです。

1997年、学部の最終学年のとき、私が在籍していた学科でジョン・マエダの講演がありました。それは私たちのなかの数名にとって圧倒的なものでした。私たちは教室の後ろから見ていたときに隣に座っていた友達は、「おぉ、もっとゆっくり話して…」とつぶやいていました。私がそれまで見たことのないかたちでデザインとコンピュータが交わっている様子を、そのプレゼンテーションで見たのでした。それはツールを作る（それはありきたりに聞こえます）でもインターフェイスを作る（それはまた不満な点が多少ありました）でもない展望でした。そして1年後、幸運なことに私はMITのジョン・マエダと合流する機会を持つことができました。

教育学は、私がMITメディアラボでジョンと過ごした6年間の継続的なテーマでした。ケイシーと私は他の学生とともにDesign By Numbersのプロジェクトに寄与しました。そしてそれは、計算（コンピュテーション）をデザイナーに教えることについて多くのことを私たちに教え、そして、人々が何を望んでいるかを私たちにフィードバックしてくれました。その特徴と、ケイシーと私が「スケッチ」の段階として行っていたこととに類似性を見い出し始め、そののちに「Processing」と呼ばれるもののなかにそれら2つをどう結びつけられるか議論し始めたのです。

私がプログラミングを学んだ方法を振り返り、じっくりと眺めて、修正でき、テストできるたくさんのコードをProcessingが含んでいてほしいと私たちは思いました。しかしもっと大切なことは、メンバーがお互いにコードを共有したいと考え、お互いの質問に答えるられるようなコミュニティがプロジェクトのまわりにあることです。同じように、Processingのソースコードも見ることができますが、それは私にとってコードを共有し私の質問に答えてくれた前の世代の開発者たちの好意にお返しするようなものなのです。

このプロジェクトにおける私の個人的なゴールの1つは、デザイナーが自分自身のツールを使いこなすことを助けることです。1980年代半ばにデスクトップパブリッシングがデザインを再発明してから20年以上が経ちました。そして私たちにはより多くのイノベーションが必要とされています。デザイナーたちが手に入る道具には飽きるようになって、デザイナーの頭の中にあることと彼らが買ったソフトウェアの能力との間の広がり続けるギャップを、コーディングとスクリプティングが埋めるようになりました。Processingの多くのユーザーが自身の作品に使っていると思いますが、企業やコンピュータサイエンティストではなくデザイナーが、Processingを使って自分自身で新しいデザインツールを作ってほしいと願っています。

謝辞

本書は、20年以上にわたる視覚的なデザインとソフトウェアの研究を統合したものです。ジョン・マエダはProcessingと本書の創世に最も重要な役目を果たした人です。MITメディアラボのAesthetics and Computation Group（ACG）のアドバイザーとしての彼の指導と、Design By Numbersプロジェクトにおけるイノベーションが、本書で示される考え方の基礎となっています。Processingは、ACGに1999年から2004年に在籍した大学院生の友人たちとの研究やコラボレー

ションにも強く活気づけられてきました。私たちとコラボレーションしてくれたPeter Cho、Elise Co、Megan Galbraith、Simon Greenwold、Omar Khan、Axel Kilian、Reed Kram、Golan Levin、Justin Manor、Nikita Pashenkov、Jared Schiffman、David Small、Tom Whiteに感謝します。私たちはまた、ACGや視覚言語ワークショップの先輩たちが築いた基盤に感謝します。

Processingの起源はMITにありますが、UCLAやthe Interaction Design Institute Ivrea、the Broad Institute、カーネギーメロン大学を含む他の研究機関のなかで大きく育ちました。ロサンゼルスとイブレアでのケイシーの同僚たちは、このテキストのアイデアの多くを進化させる環境を提供してくれました。私たちはUCLAの教員であるRebecca Allen、Mark Hansen、Erkki Huhtamo、Robert Israel、Willem Henri Lucas、Peter Lunenfeld、Rebeca Mendez、Vasa Mihich、Christian Moeller、Jennifer Steinkamp、Eddo Stern、Victoria Vesnaに感謝します。私たちはイブレアの教員と創始者であるGillian Crampton-Smith、Andrew Davidson、Dag Svanaes、Walter Aprile、Michael Kieslinger、Stefano Mirti、Jan-Christoph Zoels、Massimo Banzi、Nathan Shedroff、Bill Moggridge、John Thackara、Bill Verplankに感謝します。ベン・フライによるProcessingを使った可視化の研究に資金を提供してくれたthe Broad InstituteのEric Landerに感謝します。

本書の考え方と構成は、UCLA、CMU、Interaction Design Institute Ivrea、MIT、Harvardで教えてきたこの10年以上の間にわたって洗練されてきました。私たちは特にケイシーの授業（DESMA 28, 152A, and 152B）を受講した学生たちのアイデアと努力とエネルギーに感謝します。ケイシーのUCLAの大学院生の教え子たち：Gottfried Haider、Rhazes Spell、Eric Parren、Lauren McCarthy、David Wicks、Pete Hawkes、Andres Colubri、Michael Kontopoulos、Christo Allegra、Tyler Adams、Aaron Siegel、Tatsuya Saito、Krister Olsson、Aaron Koblin、John Houck、Zai Chang、Andrew Hieronomiは価値あるフィードバックをくれました。

Processingは、まずワークショップを通じて学生たちに紹介されました。2001年と2002年に、私たちの新しいソフトウェアにチャンスを与えてくれた最初の機関たちに感謝します：武蔵野美術大学（東京）、ENSCI–Les Ateliers（パリ）、HyperWerk（バーゼル）、the Royal Conservatory（ハーグ）。たくさんの大学がProcessingをカリキュラムの中に組み入れています。私たちはその機関の先駆的な教員たちと学生たちに感謝します。ここで言及するにはとても多過ぎますが、NYUのInteractive Telecommunication Program（ITP）の学生と教員は特に最初の頃から採用して宣伝をしてくれました。なかでもDan O’ Sullivan、Tom Igoe、Josh Nimoy、Amit Pitaru、Dan Shiffmanには深く感謝します。

Processingのソフトウェアは、コミュニティの努力の賜物です。この10年の間、絶え間ない会話を通して進化してきました。本書とProcessingのソフトウェアのゴールは、かけがえのない提案と活発な議論の結果として拡がったり縮んだりしてきました。ソフトウェアのバージョン2.0のリリースは、Andres Colubri、Dan Shiffman、Florian Jenett、Elie Zananiri、Patrick Hebron、Peter Kalauskas、David Wicks、Scott Murray、Philippe Lhoste、Cedric Kiefer、Filip Visnjic、Jer Thorpの貢献によって可能になりました。

協働したり貢献してくれた全ての人たちのリストを作ることは不可能ですが、公式にソフトウェアに貢献してくれた人たちは、Andreas Schlegel、Jonathan Feinberg、Chris Lonnen、Eric Jordan、Simon Greenwold、Karsten Schmidt、Ariel Malka、Martin Gomez、Mikkel Crone Koser、Koen

Mostert、Timothy Mohn、Dan Mosedale、Jacob Schwartz、Sami Arola、Dan Haskovecです。

このテキストは数えられないぐらい何度も書き直され再設計されました。私たちは、最初の原稿を読んで編集し、最終稿を校正してくれたShannon Huntの恩恵を受けています。Karsten SchmidtとLarry Cubaは初期の章を読んでフィードバックを提供してくれました。Tom IgoeとDavid CuartiellesはWebで公開している「電子工作」のチュートリアル（https://processing.org/tutorials/electronics/）に対して重要なフィードバックを提供してくれました。Rajorshi GhoshとMary Huangはかけがえのない第1版の制作に援助を提供してくれました。Anna Reutinger、Philip Scott、Cindy Chiの援助が第2版を実現しました。Chandler McWilliamsは第1版の最終稿に徹底的なテクニカルレビューをしてくれました。Gottfried Haiderは第2版の原稿を精細に調べて大いに改善しました。第2版に追加された「3Dドローイング」の章は、第1版におけるSimon GreenwoldがWeb上に公開した「3D」の章を改変したものです。

私たちは、MIT Pressの方々と働くことを楽しみました。このプロジェクトへの彼らの献身に感謝します。Doug Seryは出版プロセスのあらゆるステップで私たちを導いてくれました。それがなければこの本は実現できませんでした。第1版については、私たちのミスを注意してくれたKatherine Almeidと編集スタッフ、Terry LamoureuxとJennifer Flintの編集知識に深く感謝します。企画書と草稿を読んだ匿名の査読者は、この本の構成を洗練させるとても価値あるフィードバックを与えてくれました。第2版については、Doug SeryとKatherine Almeidaだけでなく、Susan BuckleyとMary Reillyの継続的な支援に感謝します。

貢献をしてくれた多くのアーティストや著者に感謝します。彼らは時間を惜しみなく割いてくれました。彼らの努力を通じて、本書の質が大きく高められています。

そしていちばん大事なこととして、ケイシーはCait、Ava、Julian、Molly、Bob、Deannaに、ベンはShannon、Augusta、Chief、Rose、Mimi、Jamie、Leif、Erika、Joshに感謝します。

1. Processingについて

Processingは、視覚的な形・動き・インタラクションの原則にソフトウェアの概念を結びつけています。また、プログラミング言語と開発環境、さらに教育の方法論をひとつのシステムに統合しています。Processingは、ビジュアルな世界におけるコンピュータプログラミングの基礎を教えるために、ソフトウェアのスケッチとして役立つよう、そして制作のツールとしても使われるよう、作られています。学生、アーティスト、プロのデザイナー、研究者たちが、学習、プロトタイピング、作品制作のためにProcessingを使っています。

プログラミング言語としてのProcessingは、イメージを生成し処理するために設計されたテキストベースのプログラミング言語です。Processingでは、わかりやすさと高い機能を両立できるようにしています。入門者は数分間の講習を受けるだけで自分のプログラムを書けるようになる一方、上級者は機能を追加するライブラリを使ったり書いたりできるようになります。このシステムを通して、ベクター画像やラスター画像の描画、画像処理、カラーモデル、マウスやキーボードイベント、ネットワーク通信、オブジェクト指向プログラミングなど、数多くのコンピュータグラフィックスとインタラクションの技術の教育を容易にしています。そして様々なライブラリは、サウンド生成、多様な形式のデータの受け渡し、ビデオの取り扱いなど、Processingの能力を拡張します。

ソフトウェア

以下に記したソフトウェアというメディアについての信念が、Processingの基礎を定め、ソフトウェアとコミュニティを設計する際の指針になりました。

「ソフトウェアは独自の性質を持った固有のメディアである」

ほかのメディアでは表現できない概念や感情が、ソフトウェアというメディアでは表現できるかもしれません。ソフトウェアは独自の用語と文法を必要としており、映画、写真、絵画など従来のメディアの延長で評価すべきではありません。油絵、カメラ、フィルムなどの技術は、芸術的な実践と言説を変えたことを歴史が証明しています。新しい技術が芸術を進歩させるとまでは言いませんが、コミュニケーションや表現の新たなかたちを生み出していると思います。ソフトウェアは、芸術的なメディアの中でも独自の位置を保っています。なぜならソフトウェアは、ダイナミックな形を作ったり、身ぶりを解析してふるまいを明らかにしたり、自然の体系をシミュレートしたり、音や画像、文字など他のメディアを統合することができるからです。

「異なるプログラミング言語はそれぞれ別々の素材である」

どんなメディアにも、目的によってふさわしい素材があります。椅子を設計するとき、デザイナー

は使用目的やアイデアや好みにもとづいて、鉄や木といった素材を使用することを判断します。こうしたやり方は、ソフトウェアを書くときにもあてはまります。抽象アニメーションの作家でプログラマーのLarry Cuba（ラリー・キューバ）は、自らの経験を次のように説明しています。「私の作品は全て別々のプログラミング言語を使ったシステムで制作しています。あるプログラミング言語はいくつかのアイデアを表現するのに強力な一方で、それ以外の表現を抑制してしまうのです」[1]。いろいろなプログラミング言語がありますが、プロジェクトのゴールに応じて他よりも適切な言語がいくつかあるものです。Processing言語は、一般的なコンピュータプログラミングの文法を利用しているので、この言語を使って得た知識を多くの様々なプログラミング言語に役立てることができます。

「スケッチすることはアイデアを形にするのに不可欠である」

スケッチをする時には、スケッチから最終形を見積もれるよう、完成品を作る時に使うメディアと関連のあるメディアを使うことが必要です。画家は、最終的な作品を描く前に詳細な下絵やスケッチを描きます。建築家は、空間の中での形を理解しようとして、まず厚紙や木材を使います。音楽家は、複雑な楽曲の楽譜を書く前にたいていピアノを弾きます。つまり、電子的なメディアをスケッチするには、電子的な素材を使うことが大切なのです。プログラミング言語がそれぞれ特有の素材であるように、スケッチに適した言語があります。ソフトウェアで制作するアーティストは、最終的なコードを書く前に自分のアイデアを試す環境を必要とします。Processingはソフトウェアのスケッチブックとしての役割を果たすように作られているので、手軽に短時間で多くのアイデアを探求して改良できます。

「プログラミングはエンジニアだけのものではない」

多くの人々は、プログラミングが数学や理系の科目が得意な人たちのものだと考えています。プログラミングがそうした人たちの世界にとどまっている理由の1つは、技術に関心を持っている人たちがプログラミング言語を作っているからです。一方で、視覚や空間に関心を持っている人たちを引きつける別の種類のプログラミング言語と環境を作ることもできます。Processingのようなオルタナティブな言語は、理系とは別の考え方をする人たちにプログラミングを広げます。初期のオルタナティブな言語にLogoがありました。Logoは1960年代後半にSeymour Papert（シーモア・パパート）が設計した子ども用の言語です。Logoはタートルロボットや画面上のグラフィックなどの様々なメディアを子どもたちがプログラミングできるようにしました。最近の例としては、Maxプログラミング環境があげられます。Maxは、1980年代にMiller Puckette（ミラー・パケット）が開発した一風変わった言語です。Maxのプログラムは、テキストを書かずに、プログラムコードを表すボックスをつないで作成します。多くのミュージシャンとビジュアルアーティストたちが、音や映像を作るソフトのベースとしてMaxを使うことに熱狂しました。GUIが何百万人もの人々のコンピュータ利用を開いたように、オルタナティブなプログラミング環境は新世代のアーティストやデザイナーがソフトウェアを直接使って制作することを可能にするのです。私たちは、Processingによって多くのアーティストやデザイナーがソフトウェアに取り組むようになり、さらにアート用の他のプログラミング環境への興味をかきたてることを願っています。

リテラシー

Processingは、現在のプログラミングの文化から大きく逸脱するわけではありません。プログラミングに興味はあってもコンピュータサイエンスの学科で教えられているものにおじけづいたり関心を持てなかったりした人たちの手に届くよう、プログラミングの位置付けを変えています。コンピュータはもともと高速に計算するための道具でしたが、表現するためのメディアに進化したのです。

一般的なソフトウェアリテラシーの概念は、1970年代初頭から議論されてきました。1974年に、Ted Nelson（テッド・ネルソン）は『コンピュータ・リブ／夢の機械』で当時のミニコンピュータについて書き、次のように述べました。「あなたがコンピュータのことを知れば知るほど……　あなたの想像力は、専門性の合間を流れ、部品とともに滑り込み、コンピュータにさせようとすることの輪郭を浮かび上がらせるのです」[2]。この本でネルソンは、メディアツールとしてのコンピュータの潜在的な未来像を論じ、ハイパーテキスト（Webの基礎となった、リンクしたテキスト）とハイパーグラム（インタラクティブな図）のための概念を述べました。ゼロックス・パロアルト研究所（PARC）の研究開発から、現在のパーソナルコンピュータの原型であるダイナブックが生まれました。ダイナブックの構想はハードウェアだけにとどまりません。子どもたちが物語やお絵描きのプログラムを書いたり、音楽家が作曲のプログラムを書いたりできるような、プログラミング言語が描かれていました。この構想の中では、コンピュータのユーザーとプログラマーの間には明確な区別はなかったのです。

こうした楽天的な考えから40年後、私たちがたどり着いたのはまったく別の場所でした。幅広い利用者に向けたパーソナルコンピュータとインターネットの登場によって、技術的・文化的な革命が起きましたが、人々は自分自身のソフトウェアを作成するよりもプロのプログラマーが作ったソフトウェアツールを圧倒的に使っています。ジョン・マエダは『Creative Code』でこの状況を明快に説明しています。「コンピュータのツールを使うには、マウスで指してクリックするだけで済みます。ツールを作るには、コンピュータプログラミングの神秘的な技を理解しなければいけません」[3]。この現在の状況の負の側面として、ソフトウェアのツールが課している制約があります。ツールが使いやすくなった結果、ツールがコンピュータの可能性を見えづらくしてしまったのです。アートの素材としてのコンピュータを十分に探求するには、この「コンピュータプログラミングの神秘的な技」を理解することが重要です。

Processingは、視覚的なアートに関わる人たちが自分自身のツールを作る方法を学び、ソフトウェアを読み書きできる人にステップアップすることを目指しています。ゼロックスPARCとAppleでの先駆者だったAlan Kay（アラン・ケイ）は、ソフトウェアにおけるリテラシーを次のように説明しています。

> あるメディアを“読む”能力とは、他の誰かが作った素材とツールを使えることを意味します。あるメディアで“書く”能力とは、他の誰かのために素材とツールを作り出せることを意味します。読み書きできる人になるには、両方の能力を獲得しなければなりません。文字で書く場合、作り出すツールは修辞的なものになります。つまり何かを証明し説得するのです。コンピュータで書く場合、作り出すツールはプロセスになります。つまり何かをシミュレートし決定するのです。[4]

シミュレートし決定するプロセスを作るには、プログラミングが必要なのです。

オープン

オープンソースソフトウェア運動は、Linuxを筆頭に、私たちの文化と経済に大きな影響を与えています。しかし、アート系ソフトウェア周辺の文化にはそれほどの影響はありません。オープンソースのプロジェクトはぽつぽつとあるものの、Apple、Adobe、Autodeskといった企業がソフトウェア製品を支配し、アート業界で使われる現代のツールをコントロールしています。アーティストやデザイナーは、集団として独立的なソフトウェア運動を支援する技術的なスキルを持ち合わせていません。Processingは、オープンソースソフトウェアにおけるイノベーションの精神をアートの世界に持ち込もうとしています。私たちは、現状の独占的なソフトウェアの代替品を提供して、アートやデザインのコミュニティの技術を向上させることで、こうした活動への関心を刺激したいのです。そのため、Processingを拡張したり改造しやすくし、できるだけ多くの人が利用できるものにしたいのです。

Processingは、オープンソースソフトウェアとの関係を抜きに存在することはできません。既存のオープンソースプロジェクトを参考にし、重要なソフトウェアの部品として使うことで、より短期間に少人数のプログラマーチームで開発することができました。オープンソースプロジェクトに自分の時間を割いてくれる人たちがいるので、オープンソースソフトウェアは予算がなくても進化します。そのためコストをかけずにソフトウェアを配布でき、高価な市販のソフトウェアを買えない人たちに届けることができるのです。Processingのソースコードがオープンであることにより、そのコードの構造から学習したり、自分のコードでProcessingを拡張することでも学習できます。

Processingを使って書いたプログラムのコードは公開することを勧めています。Webサイト制作技術の急速な普及をもたらしたWebブラウザの「ソースを表示」機能と同じ方法で、他者のProcessingのコードにアクセスすることによりコミュニティのメンバーは互いに学ぶことができ、コミュニティ全体のスキルが向上します。その好例の1つに、ビデオ画像の中の物体を追跡するソフトウェアの制作があります。これにより、マウスやキーボードだけでなく、自分の身体を使ってソフトウェアとダイレクトにインタラクトできるようになりました。オンラインで共有されたあるプロジェクトには、カメラで撮影した最も明るい色の物体を追跡できるコードが入っていましたが、色は追跡していませんでした。より経験豊かなプログラマーが、このコードをもとに同時に複数の色の物体を追跡できる汎用的なコードを開発しました。この改良されたコードを使って、UCLAの大学院生ローラ・エルナンデス・アンドレードさんが「Talking Colors」を制作しました。この作品は、人々が着ている服の色に関する感情的なテキストを投影した画像で人に重ね合わせるインタラクティブなインスタレーションです。コードを共有して改良すると、互いに学びあって、ゼロから作り上げるには複雑なプロジェクトも作ることができるのです。

教育

Processingによってソフトウェアの概念をアートの分野に導入することができ、またアートの概念をより技術的な人たちに開くことを可能にしました。Processingの構文は、広く使われているプログラミング言語から派生しているので、その後の学習の良い基礎となります。Processingで学んだスキルがあれば、Web開発やコンピュータグラフィックス、電子機器など異なる分野に適した他のプログラミング言語を学ぶことができるのです。

コンピュータサイエンスのカリキュラムが数多く確立しているのに対し、メディアアートとコンピュータのコアな概念を統合しようとするような授業はほとんどありませんでした。これまではジョン・マエダが始めた授業をモデルにして、Processingをベースにしたハイブリッドな授業を作ってきました。Processingは1日間から2〜3週間ほどの短期間のワークショップに適しています。Processingはとてもシンプルなので、学生はわずか数分間の説明でプログラミングを始めることができます。Processingの構文は他の一般的な言語とよく似ていて、既になじみのある人も多いので、経験のある学生ならすぐに高度な構文を書き始めることができます。

ソウルの弘益大学校の1週間のワークショップでは、デザインを専攻している学生とコンピュータサイエンスを専攻している学生がまざっていましたが、どちらの学生たちも統合に向かって制作をしました。視覚的によくできた作品もあれば、技術的に高度な作品もありましたが、どれも同じ基準で評価されたのです。まったくプログラミング経験のないスー・ジョン・リーさんのような学生もワークショップに参加していました。Processingという素材にやりがいを感じた彼女は、基本原理を学んで自分のアイデアに活かすことができました。講評会では、彼女の強力な視覚のスキルが、理系寄りの学生のお手本になりました。コンピュータサイエンス学科のタイ・キュン・キムさんのような学生は、Processingの使い方をすぐに理解しましたが、ほかの学生作品のビジュアルに刺激されて美的なセンスが磨かれました。彼のキネティックタイポグラフィ作品は、持っていた技術力と新たに獲得したデザインセンスが融合した例でした。

Processingは学部のもう少し長めの導入的な授業や専門的な大学院レベルの授業にも使われています。小さなアートスクールや私立の単科大学、公立の総合大学で使われてきました。たとえばUCLAでは、大学2年生のデジタルメディアの基礎授業の教材として使用され、大学院生には先端的な領域を探究するためのプラットフォームとして導入されてきました。学部の「インタラクティビティ」の授業では、受講生がインタラクションに関する話題を読んだり議論したりして、Processingを使ってインタラクティブなシステムのサンプルをたくさん作ります。キネティックアートやテレビゲームにおけるファンタジーの役割など新しいトピックが毎週紹介されます。受講生は新しいプログラミング技術を学んで、1つのトピックに焦点をあてた習作を作ります。あるプロジェクトでは、Sherry Turkle（シェリー・タークル）の論文「テレビゲームの魅力」[5]を読み、個人的な脱出願望や変身願望を探究する短いゲームか出来事を制作するという課題が受講生に出されました。レオン・フォンさんは、プレイヤーが水の上を浮くようにして遠い島に向かって進むエレガントな飛行シミュレーションを作りました。マスカン・シュリーヴァスタヴァさんは、テーブルのデザートを10秒以内に全部食べ尽くすことが目的のゲームを制作しました。

プログラミングの技法を教えると同時に基本的な理論を紹介すると、受講生は自分のアイデアをダイレクトに探究し、インタラクティビティとデジタルメディアに関する深い理解と直観力を養うことができます。UCLAの大学院レベルの「インタラクティブ環境」コースでは、コンピュータビジョンを実験するプラットフォームとしてProcessingを使います。サンプルコードを使いながら、ビデオカメラからの画像を入力として身体を使うソフトウェアを学生たちは1週間かけて開発します。ツァイ・チャンさんは、参加者の身体が密集したパーティクル（粒子）としてプロジェクションされる「White Noise」という刺激的なインスタレーションを開発しました。人物のパーティクルの影がそれぞれ別々の色で表示されていて、影が重なりあうとパーティクルが入れ替わるので、物質移動してお互

いの独自のエッセンスを感染させあっているように見えます。カメラから情報を読み出すことは、Processingでは簡単なことです。この機能があることで、授業ではすぐにダイレクトな探究にとりかかれます。もしこの機能がなかったら、同じようなプロジェクトを準備するのに何週間ものプログラミングの指導が必要になったでしょう。

ネットワーク

ProcessingはWebベースのコミュニティの強みを活かし、思わぬ方法で大きくなりました。6つの大陸をまたいだ世界中の何千人もの学生、教育者、実務者たちがProcessingを使い始めたのです。Processingの公式Webサイトはコミュニケーションのハブになっていますが、プロジェクトに貢献するメンバーは世界中の都市に散らばっています。フォーラムのような代表的なWebアプリケーションが、離れた場所にいる人たちどうしで機能やバグ、関連するイベントについて議論する場を提供しています。

過去10年間にわたって、Processingを使用した多くの授業がカリキュラムの全てをWebで公開し、受講生はソフトウェアの課題と他の人がそこから学ぶことができるソースコードを公開してきました。たとえば、ニューヨーク大学のDaniel Shiffman（ダニエル・シフマン）の授業のWebサイトには、オンラインチュートリアルと学生の作品へのリンクが掲載されています。シフマンによる「手続き型ペイント」コースのチュートリアルは、テキストと実行できるソフトウェアのサンプルを組み合わせて、構造化プログラミング、画像処理、3Dグラフィックスなどのトピックをカバーしています。受講生は授業で作成した自分たちのソフトウェアとソースコードの全てを掲載したWebページをメンテナンスしています。こうしたページがあることで、作品の出来映えを率直に振り返ることができ、受講生が互いの作品を見ていくことができます。

Processingのフォーラムは、http://forum.processing.org/にあるオンラインのディスカッションシステムで、人々が自分のプロジェクトについて語ったりアドバイスを共有したりする場です。何千人ものメンバーが、お互いの作品について活発にコメントしあったり、技術的な質問に答えたりしています。ある時、ばねをシミュレートするコードの問題についての投稿がありました。数日後、ルンゲ=クッタ法と比較しながらオイラー法を詳細に解説したメッセージがポストされました。なんだか難しそうな議論に感じられるかもしれませんが、この2つの方法の違いがその作品の成否を左右するものだったのです。このスレッドやそうした多くのスレッドは、詳細な話題に興味のある学生たちにとって簡潔なネット上の資源となっています。

コンテクスト

Processingのプログラミングへのアプローチは、確立された方法をうまく混ぜ合わせるというものです。コアの言語と追加されるライブラリはJavaを利用していますが、これらはC言語と同じ要素を持っています。こうした由来があるため、Processingはプログラミング言語の数十年間にわたる改良を活用することができ、ソフトウェア開発に慣れている多くの人たちにとって理解しやすいものになっています。

Processingはデザインとアートのコンテクストを重視し、そしてそのコンテクストを体現しようとする戦略的な決定において、類いまれな存在です。Processingを使うと、絵やアニメーション、環境に反応するソフトウェアを簡単に書くことができ、オーディオ、ビデオ、電子機器と連携するようにプログラムを拡張することも簡単です。

Processingを利用する人々と教育機関のネットワークはますます広がっています。最初のアイデアが生まれてからの10年、Processingは世界中のプレゼンテーション、ワークショップ、授業、議論を通じて有機的に進化してきました。私たちは、これからもProcessingを改良し育てていきます。プログラミングを実践することで、よりダイナミックなメディアの基盤になる可能性が拓かれることを願いながら。

注

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.

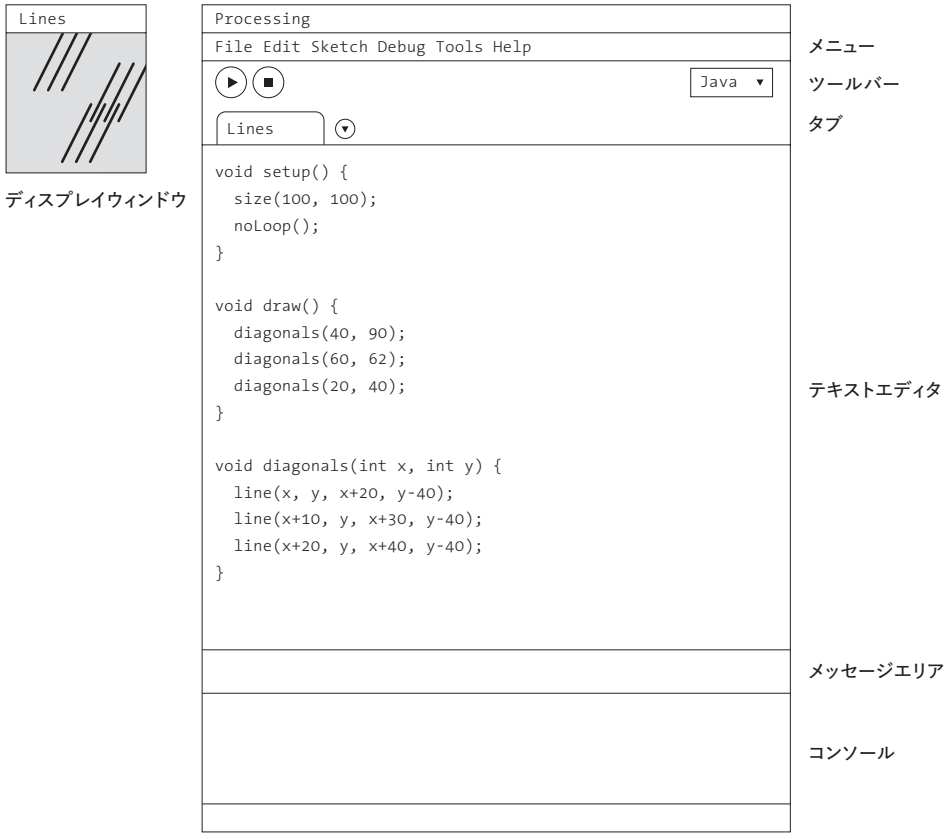


図2-1　Processing開発環境（PDE）

Processing開発環境を使ってスケッチを描きます。テキストエディタにコードを書き、ツールバーのボタンで実行と停止をします。

2. Processingを使う

本章では、Processingの環境と、ソフトウェアを書くうえで最も基本となる要素を紹介します。

紹介する構文：

```
//（コメント）、/* */（複数行にわたるコメント）
;（文の終端記号）、,（カンマ）
print()、println()
```

ダウンロードとインストール

Processingのソフトウェアは、Processingの公式Webサイトからダウンロードできます。Webブラウザでwww.processing.org/downloadを開き、使用するコンピュータのOSのリンクをクリックしてください。Linux版、Mac版、Windows版があります。

環境

Processing開発環境（Processing Development Environment：PDE）は、コードを書くためのシンプルなテキストエディタ、メッセージエリア、テキストコンソール、ファイル管理のタブ、よく使う動作のボタンがあるツールバー、そしてメニューで構成されています。プログラムの実行中は、「ディスプレイウィンドウ」と呼ばれる新規ウィンドウが開きます。

Processingで書かれたソフトウェアを「スケッチ」と呼びます。スケッチはテキストエディタで編集します。エディタには、文字列のカット／ペーストや、検索／置換の機能があります。メッセージエリアは、保存や書き出しの結果やエラーを表示する領域です。コンソールは、Processingのプログラムが出力するテキストを表示します。エラーメッセージや、プログラム中の`print()`関数や`println()`関数が出力する文字列などです。ツールバーのボタンでは、プログラムの実行と停止ができます。

Run	コードをコンパイルし、ディスプレイウィンドウを開いて、プログラムを実行します。
Stop	実行中のプログラミングを停止します。

メニューには、ツールバーと同じ機能に加え、ファイル管理とリファレンスの参照ができます。

File	ファイル管理や書き出しのコマンド。
Edit	テキストエディタの操作（戻す、やり直し、カット、コピー、ペースト、検索など）。
Sketch	プログラムの実行と停止、メディアファイルやライブラリ追加のコマンド。
Tools	Processingの利用を補助するツール（カラー選択、フォント作成など）。

Debug　スケッチのエラー発見を補助する機能。
Help　Processsing の開発環境と言語のリファレンス。

Processing のスケッチは、それぞれ専用のフォルダを持ちます。スケッチのメインのプログラムファイルは、専用のフォルダの中にフォルダと同じ名前であります。「Sketch_123」と名付けたスケッチの場合、スケッチのフォルダは「Sketch_123」という名前になり、メインファイルは「Sketch_123.pde」となります。PDEというファイル拡張子は、Processing Development Environment の頭文字です。

スケッチフォルダには、音声や画像のファイルやコードのライブラリを収めたフォルダが入っていることがあります。Sketch メニューから「Add File」を選択して、スケッチにフォントや画像を追加すると、data フォルダが作成されます。こういったファイルはテキストエディタにドラッグすることでも追加できて、たとえば Processing のテキストエディタに画像ファイルをドラッグすると、現在のスケッチの data フォルダ内にそのファイルが自動的にコピーされます。スケッチで読み込む全ての画像、フォント、サウンド、その他のデータファイルは、このフォルダになければいけません。

スケッチは「Processing」という名前のフォルダに保存されます。この Processing フォルダの場所の初期設定は、使用している OS や環境設定で変わりますが、コンピュータかネットワーク上にあります。Processing フォルダの場所を指定するには、File メニュー（Mac の場合は Processing メニュー）から「Preferences」を選択し、「Sketchbook location」で行います。

1 つのスケッチに複数のコードを使うことができます。コードのファイル形式は、Processing のテキスト（拡張子が .pde）か Java（拡張子が .java）です。新規ファイルを作るには、ファイルタブの右側にある矢印ボタンをクリックします。このボタンで、現在のスケッチを構成しているファイルの作成、削除、名称変更ができます。新規の PDE ファイルでは関数やクラスを、JAVA 拡張子のついたファイルでは Java のコードを書くことができます。複数のファイルでコードを書くと、コードを再利用しやすく、プログラムを小さなサブプログラム群へ分割しやすくなります。このことについては、「オブジェクト」の章（368 ページ）で詳しく紹介しています。

書き出し

書き出し（export）とは、スケッチを Processing 開発環境の外部で実行できるようにパッケージ化する機能のことです。Processing からコードを書き出すと、そのコードは Java コードに変換されてから Java アプリケーションとしてコンパイルされます。Linux、Macintosh、Windows の各プラットフォーム用のアプリケーションを書き出すことができます。あるプロジェクトが書き出されると、アプリケーションが入ったフォルダにファイル一式（スケッチのソースコードと、書き出し先のプラットフォームに必要なライブラリ）が出力されます。スケッチを書き出すたびに、アプリケーションフォルダの中身が削除されて、ファイル一式をゼロから出力し直します。そのため、書き出し前にこのファイルに加えた変更は全て消えてしまいます。

アプリケーションに不要なメディアファイルは、ファイルサイズを抑えるために、書き出す前に data フォルダから削除してください。data フォルダ内にある使っていない画像も書き出されるので、ファ

イルサイズが増えてしまいます。

Processingに関する詳しい情報や最新の情報は、www.processing.org/reference/environment/か、Processingアプリケーションの Helpメニューの「Environment」を選択して見ることができます。

サンプルガイドツアー

Processingのプログラムは、1行だけのこともあれば数千行に及ぶこともあります。この小規模なものから大規模なものへと自在に拡張できるスケーラビリティが、この言語の最も大切な特徴の1つです。これから紹介するサンプルは、Processingの言語の基本要素のいくつかを探究する手段として、斜線のアニメーションを制作するという手軽なゴールを示しています。プログラミングが初めての人にとっては、このセクションにはなじみのない用語や記号が出てくるはずです。このガイドツアーは、後の章で詳しく紹介する概念や技法を使いながら、本書全体を短く要約したものになっています。これらのサンプルプログラムをProcessingアプリケーションで実行すれば、そのコードでやっていることをしっかりと理解できます。

Processingは、直線、楕円、曲線といったグラフィックの要素を簡単にディスプレイウィンドウに描けるよう設計されています。こうした図形は、その座標を示した数値で位置が決まります。直線の位置は、始点と終点それぞれ2つの合計4つの数値で指定します。`line()`関数のパラメータで、表示する直線の位置を決めます。座標系の原点は左上の角で、右方向と下方向に数値が増えていきます。座標といろいろな図形の描画については、021–032ページで紹介しています。

2-01

```
line(10, 80, 30, 40); // 左側の直線
line(20, 80, 40, 40);
line(30, 80, 50, 40); // 中央の直線
line(40, 80, 60, 40);
line(50, 80, 70, 40); // 右側の直線
```

図形の視覚的な属性は、他のコードを使って制御します。色やグレーの値、線の幅、レンダリングの品質などを指定するコードがあります。属性の描画については、032–037ページで紹介しています。

2-02

```
background(0);          // 背景を黒に
stroke(255);            // 線の色を白に
strokeWeight(5);        // 線の太さを5ピクセルに
line(10, 80, 30, 40); // 左側の直線
line(20, 80, 40, 40);
line(30, 80, 50, 40); // 中央の直線
line(40, 80, 60, 40);
line(50, 80, 70, 40); // 右側の直線
```

xなどの変数は、ある数値を表しています。コードの実行時に、xの記号がその数値に置き換わります。1つの変数でプログラム内のいろいろな視覚的な特徴を制御することができます。変数については、051–056ページで紹介しています。

2-03

```
int x = 5;  // 水平位置を指定
int y = 60; // 垂直位置を指定
line(x, y, x+20, y-40);    // [5,60]から[25,20]に直線を描く
line(x+10, y, x+30, y-40); // [15,60]から[35,20]に直線を描く
line(x+20, y, x+40, y-40); // [25,60]から[45,20]に直線を描く
line(x+30, y, x+50, y-40); // [35,60]から[55,20]に直線を描く
line(x+40, y, x+60, y-40); // [45,60]から[65,20]に直線を描く
```

プログラムに構造を持たせると可能性が広がります。setup()関数とdraw()関数を使うと、プログラムを繰り返して実行することができます。実行の繰り返しは、アニメーションやインタラクティブなプログラムを制作するのに必要となる機能です。setup()内のコードはプログラムの開始時に1度だけ実行され、draw()内のコードは繰り返して実行されます。毎回のdraw()の実行完了時に、ディスプレイウィンドウにイメージフレームが描画されます。

次のサンプルでは、変数xをグローバル変数として宣言しています。グローバル変数とは、プログラム内のどこでも代入や参照ができる変数のことです。xの値を1フレームごとに1ずつ増やしています。3本の直線の位置をxで制御しているため、値が変化するたびに違う場所に描かれます。これで直線が右方向に動き出します。

次のコードの14行目はif文です。この行に変数xと値100を比較する関係式があります。この式が真（true）のときに、ブロック内のコードが実行されます。if文のブロックとは、「{」と「}」にはさまれたコード部分のことです。この関係式が偽（false）のときは、ブロック内のコードは実行されません。xの値が100よりも大きくなると、ブロック内のコードが変数xを-40にセットし、3本の直線はウィンドウの左端にジャンプします。draw()については065–072ページ、アニメーションのプログラミングについては442–449ページ、if文については072–077ページで、それぞれ詳細について紹介しています。

2-04

```
int x = 0;  // 水平位置を指定
int y = 55; // 垂直位置を指定

void setup() {
  size(100, 100); // ウィンドウを100×100ピクセルに
}

void draw() {
  background(204);
  line(x, y, x+20, y-40);    // 左側の直線
  line(x+10, y, x+30, y-40); // 中央の直線
  line(x+20, y, x+40, y-40); // 右側の直線
```

```
  x = x + 1;       // xに1を足す
  if (x > 100) {   // xが100を超えていたら
    x = -40;       // xに-40を代入
  }
}
```

プログラムが繰り返し実行されているあいだ、Processingはマウスやキーボードなどの入力デバイスからデータを取り込んでいます。こうしたデータは、ディスプレイウィンドウの中で起きていることを変化させるために使うことができます。マウスに反応するプログラムについては、083–101ページで紹介しています。

2-05

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

void draw() {
  background(204);
  // カーソルの水平座標値をxに代入
  float x = mouseX;
  // カーソルの垂直座標値をyに代入
  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);
}
```

「関数（function）」とは、特定の作業を実行するプログラム中のコードのかたまりのことです。関数を使うと、関連するコードをまとめておけるので、プログラムを読みやすく、変更しやすくできます。次のサンプルにあるdiagonals()関数は、draw()内を実行するたびに3本の斜線を描くように書かれたものです。この関数には2つの「パラメータ（parameter）」があり、関数名の後ろにあるかっこ内の数値で、各直線の位置を指定しています。この2つの数値は、12行目の関数の定義に渡されて、13–15行目で変数xと変数yの値として使われます。関数については、343–367ページで紹介しています。

2-06

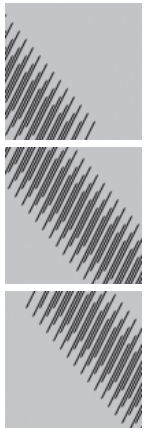
```
void setup() {
  size(100, 100);
}

void draw() {
  background(204);
  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);
}
```

オブジェクト指向プログラミングとは、コードを「オブジェクト（object）」に構造化する手法のことです。オブジェクトとは、変数（データ）や関数を持っているコードの単位です。このプログラミングスタイルを使うと、データ集合とそのデータにもとづいた動作をする機能とをしっかり関連づけることができます。`diagonals()`関数は、「クラス（class）」定義の一部として作成することで拡張することができます。オブジェクトはひな形（テンプレート）となるクラスを使って作成します。各直線の位置や属性を指定する変数をクラス定義の中に移動させると、これらの変数が各直線の描画に関連していることが明確になります。オブジェクト指向プログラミングについては、368–370ページで紹介しています。



2-07

```
int num = 20;
int[] dx = new int[num]; // 配列の宣言と作成
int[] dy = new int[num]; // 配列の宣言と作成

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;
    }
    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);
}
```

前のスケッチで使った変数は、それぞれ1つのデータ要素を格納していました。もしこの直線グループを20個作りたい場合、水平位置に20個、垂直位置に20個の合計40個の変数が必要になります。変数を増やすとプログラミングが面倒になり、プログラムも読みにくくなってしまいます。複数の変数名を使う代わりに、「配列（array）」を使うことができます。配列とは、1つの名前の中にデータ要素のリストを格納するものです。forループは、配列の要素を順番に取り出すのに使われます。配列については426–441ページ、forループについては106–110ページで紹介しています。

2-08

```
Diagonals da, db;

void setup() {
  size(100, 100);
  // 入力: 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;
  }

  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;
        }
    }
}
```

この短いガイドツアーでは十分な説明をしていませんが、本書で探求する中心的な概念をいくつか紹介しました。ソフトウェアを制作するうえでの多くの重要なアイデアについては軽く触れただけで、触れていないものもあります。それぞれの話題は、本書の以降のページで詳しく紹介していきます。

コーディングはライティング

ソフトウェアを作ることは、「書く」という行為の一種です。ただし、コードを書く前に、コンピュータプログラムを書くこととメールやエッセイを書くことの違いを認識しておくことが大切です。自然言語の書き手は、言葉の曖昧さをうまく使って多様な表現を作り出す自由を持っています。そうした文章技法は、1つの文章を多義的に解釈できるようにし、作者特有の文体をもたらします。コンピュータプログラムにも作者のスタイルが表れますが、様々に解釈される余地はほとんどありません。人間は曖昧な表現を解釈し、おかしな文法を読み飛ばすことができますが、コンピュータはそのようなことができません。コーディングを始めたときのイライラを解消するために、ここでコードを書く時の言語的な細かい決まりを紹介します。プログラムを始める時にこうした細かい決まりごとを頭にとどめておけば、少しずつ身に付いていくでしょう。

コメント

「コメント」は、コンピュータが無視する部分ですが、人間にとっては重要です。プログラマーは、個人用のメモや注意書きを、コードを読む他者のためにコメントとして書きます。プログラムは複雑な手続きを表すために記号や人間にはなじみのない記法を使うので、プログラムの各部分がどう動いているかを思い出せないことがよくあります。上手に書かれたコメントは、プログラムを読み返すときに思い出すきっかけになりますし、コードを読む人への説明になります。コメントは他のコードとは別の色で表示されます。次のプログラムでは、コメントの動作を説明しています。

2-09

```
// 2個のスラッシュはコメントを表します。
// この行の全てのテキストがコメント部分になります。
// スラッシュとスラッシュの間にスペースを入れてはいけません。
// たとえば「/ /」はコメントにならず、エラーになります。

// 複数行にわたるコメントを書きたい場合は、
// 複数行コメントの構文を使うと便利です。
```

```
/*
 スラッシュの後にアスタリスクを書くと、
 反対順にしたものを書くまでが全てコメントになります。
*/

// コメントではない文字列や記号は全てコンピュータが実行します。
// 次の2行はコメントではないので、
// 実行されて200×200ピクセルのディスプレイウィンドウが開きます。
size(200, 200);
background(102);
```

関数

関数は、図形の描画、色の指定、数値の計算など、いろいろなことを実行します。関数の名前はふつう小文字の単語で、かっこがつきます。かっこの中でカンマによって区切られている要素を「パラメータ」と呼びます。パラメータは関数が実行する内容を変化させます。パラメータのない関数もあれば、たくさんのパラメータを持つ関数もあります。次のプログラムは`size()`関数と`background()`関数のデモです。

2-10

```
// size関数は2つのパラメータを持ちます。
// 最初のパラメータはディスプレイウィンドウの幅を、
// 2つ目のパラメータはディスプレイウィンドウの高さを指定します。
size(200, 200);

// background関数は1つのパラメータを持ちます。
// このパラメータは、ディスプレイウィンドウの背景色を
// 0（黒色）から255（白色）までのグレー値で指定します。
background(102);
```

式と文

自然言語にたとえると、ソフトウェアの「式」はフレーズのようなものです。ソフトウェアの式は、右側と左側の値を演算する+、*、/といった演算子を組み合わせたかたちをよくとります。ソフトウェアの式は、単一の数値だけの基本的なものや、複数の要素の長い組み合わせになったりします。式はその内容を評価して得られる値を必ず持っています。

式	値
5	5
122.3+3.1	125.4
((3+2)* −10) + 1	-49

式は、>（より大きい）や<（より小さい）といった演算子で2つの値を比較することもできます。こうした比較はtrueかfalseとして評価されます。

式	値
`6 > 3`	`true`
`54 < 50`	`false`

1つまたは複数の式で文を作ります。「文」とは、プログラミングにおいてセンテンスにあたるものです。文は、終端記号で終わる完結した構成要素です。終端記号とは、プログラミングにおいてピリオドにあたるものです。Processing言語では、文の終端記号はセミコロンです。

センテンスにいろいろな種類があるように、文にもいろいろな種類があります。文では、変数を定義したり、変数に代入したり、関数を実行したり、オブジェクトを作成したりします。それぞれ後で詳しく説明しますが、ここでサンプルをお見せします。

2-11

```
size(200, 200); // size()関数を実行
int x;          // 新しい変数xを宣言
x = 102;        // 変数xに数値102を代入
background(x);  // background()関数を実行
```

文の最後にセミコロンをつけ忘れるミスはよくやりがちです。セミコロンを忘れると、エラーメッセージが出てプログラムは実行されません。

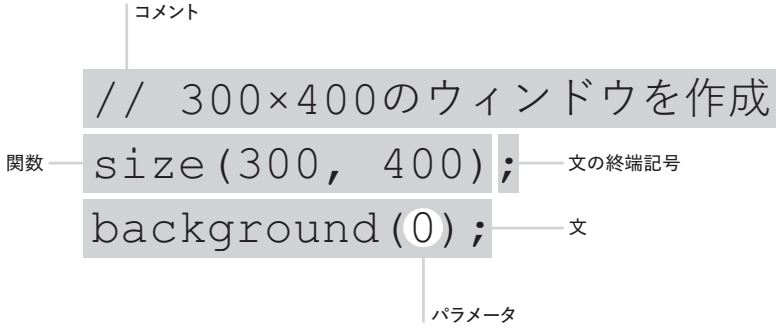


図2-2　Processingスケッチの構造

あらゆるスケッチは、いろいろな言語要素の組み合わせでできています。これらの要素はプログラマーの意図を表すように組み合わさって、コンピュータによって解釈できるものになります。より複雑なスケッチの構造については、068ページで紹介しています。

大文字・小文字

英語の文章では、大文字にする単語としない単語があります。ほとんどの単語は小文字ですが、OhioやJohnのような固有名詞の最初の文字とセンテンスの最初の文字は大文字にします。多くのプログラミング言語では、大文字にする部分と小文字にする部分が決まっています。Processingは大文字と小文字を区別するので、「size」と書こうとして「Size」と書くとエラーになります。大文字と小文字は正確にタイプする必要があります。

```
size(200, 200);                                                    2-12
Background(102); // エラー！ 「background」のbが大文字のBになっています。
```

空白（ホワイトスペース）

Processingをはじめ多くのプログラミング言語では、プログラムの要素の間に好きなだけスペースを入れることができます。文の終端記号や大文字・小文字が厳密なのに対して、スペースは自由です。次のコードの2行は、プログラムの標準的な書き方です。

```
size(200, 200);                                                    2-13
background(102);
```

コード要素間のホワイトスペースはいくらでも入れることができます。プログラムもまったく同じように実行されます。

```
size                                                               2-14

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

コンソール

ソフトウェアの実行中は、人間の目にはとまらない速さでコンピュータが演算をしています。マシンの中で何が起こっているのかを理解することは大切なので、プログラム実行中にデータを表示するために、print()関数とpintln()関数を使います。この2つの関数は、プリンタに印刷するのではなく、コンソールにテキストを出力します。コンソールは、変数を表示したり、イベントを確認したり、外部デバイスからのデータをチェックするのに使います。こうした使い方は今の段階ではピンと来ないかもしれませんが、本書の説明を通じて明らかになっていきます。コメントと同じくprint()とprintln()を使うことで、コンピュータプログラムの意図と実行を明確にすることができるのです。

2-15

```
// コンソールにテキストを出力するには、ダブルクォーテーションで囲みます
println("Processing..."); // コンソールに「Processing...」を出力します

// 変数名ではなく変数の値を出力するには、
// 変数の名前をダブルクォーテーションで囲んではいけません
int x = 20;
println(x); // コンソールに「20」を出力します

// println()はテキストを出力した後に改行しますが、
// print()は改行しません
print("10");
println("20"); // コンソールに「1020」を出力します
println("30"); // コンソールに「30」を出力します

// 複数の値を出力するには、println()のかっこ内でカンマを使ってください
int x2 = 20;
int y2 = 80;
println(x2, y2); // コンソールに「20 80」を出力します

// 複数の変数の間をカスタム文字列でつなぎあわせるには、
// 「+」演算子を使ってください
int x3 = 20;
int y3 = 80;
println(x3 + " and " + y3); // コンソールに「20 and 80」を出力します
```

リファレンス

Processing言語のリファレンスは、本書を補うものです。プログラミング中はリファレンスを開いておいて、しっかり読むことをお勧めします。リファレンスには重要な技術的詳細が載っており、本書はその情報を背景としています。リファレンスを開くには、ProcessingのHelpメニューから「Reference」を選択してください。または以下のアドレスからオンラインで見ることができます。
www.processing.org/reference

3. 基本図形を描く

本章では、ディスプレイウィンドウの座標系と、様々な幾何学的要素を解説します。

紹介する構文：

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

コードで図形を描くには難しい面があります。図形の全ての要素の位置を数値で指定する必要があるからです。ふだん鉛筆でスケッチしたり、マウスで画面上の図形を動かしたりしている人にとっては、画面の厳密な座標格子を考え始めるのに手間取るかもしれません。紙の上や頭のなかで図形の並びを実現できていることと、それをコードの記法へと変換することの間にはギャップがありますが、それは簡単に埋めることができます。

座標

図形を描く時には、まず描こうとする素材のサイズや性質を考えることが大切です。紙にスケッチする場合には、数ある道具や用紙の中から選べます。素早くスケッチするには、新聞紙と木炭が適しています。丁寧に描くには、なめらかな手すき紙といろいろな固さの鉛筆がある方がよいでしょう。一方、紙に描くのとは違って、コンピュータの画面に描くときに最初に選ぶのは、ウィンドウの大きさと背景色です。

コンピュータの画面には、「ピクセル」と呼ばれる小さな光の点が格子状に並んでいます。画面の大きさと解像度は多様です。私たちのスタジオには3つのコンピュータの画面がありますが、それぞれ別々のピクセル数で構成されています。ラップトップは2,304,000ピクセル（幅1920×高さ1200）、液晶ディスプレイは3,686,400ピクセル（幅2560×高さ1440）、旧型のモニターは786,432ピクセル（幅1024×高さ768）です。数百万ピクセルといえば莫大な量のように感じますが、ほとんどの画面はまだ、紙のような物理的メディアに比べると粗い視覚的解像度しかありません。その一方で、画面のイメージは1秒間に何度も変化させることができ、イメージが固定された紙にはない優れた点があります。

Processingで書いたプログラムは、画面のピクセル全体やその一部分を操作できます。Runボタンをクリックすると、ディスプレイウィンドウが開き、ウィンドウのピクセルをProcessingで変化させることができます。画面より大きなイメージを作ることもできますが、一般的にディスプレイウィン

ドウのサイズは画面のサイズ以下にします。

ディスプレイウィンドウのサイズは、size()関数を使って指定します。

```
size(w, h)
```

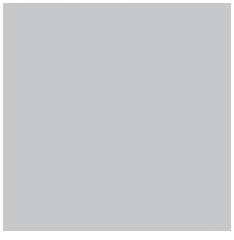
size()関数には、2つのパラメータがあります。1つ目でディスプレイウィンドウの幅を指定し、2つ目でディスプレイウィンドウの高さを指定します。

```
// 幅120ピクセル、高さ200ピクセルの
// ディスプレイウィンドウを開きます
size(120, 200);
```

3-01

```
// 幅320ピクセル、高さ240ピクセルの
// ディスプレイウィンドウを開きます
size(320, 240);
```

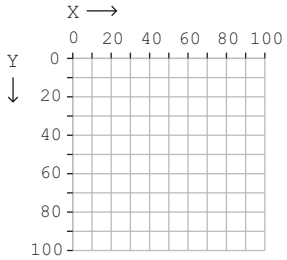
3-02

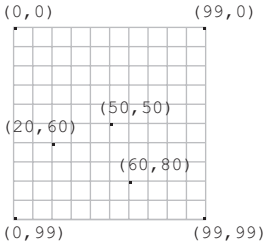


```
// 幅200ピクセル、高さ200ピクセルの
// ディスプレイウィンドウを開きます
size(200, 200);
```

3-03

ディスプレイウィンドウ内の位置は、X座標とY座標で定義します。X座標とは、原点から水平方向の距離、Y座標とは、原点から垂直方向の距離のことです。Processingでは、原点はディスプレイウィンドウの左上にあり、座標値は右方向と下方向に増加します。次ページの図は座標系を示していて、右の図は格子上に置かれた3つの座標値を示しています。





位置は、カンマで区切ったX座標、Y座標の順に記述します。原点は(0,0)と記述し、座標(50,50)はX座標50、Y座標50を示し、座標(20,60)はX座標20、Y座標60を示しています。ディスプレイウィンドウのサイズが、幅100ピクセル、高さ100ピクセルの場合、(0,0)は左上、(99,0)は右上、(0,99)は左下、(99,99)は右下のピクセルにあたります。このことは、point()関数を使ったサンプルを見るとよく理解できます。

基本図形

点は最もシンプルな視覚的要素で、point()関数を使って描きます。

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

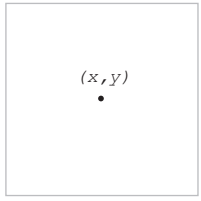
この関数には、2つのパラメータがあります。1つ目はX座標で、2つ目はY座標です。特に指定しない限り、点の大きさは1ピクセルです。

3-04

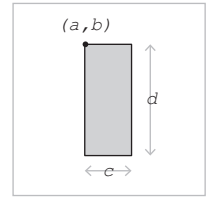
```
// XパラメータとYパラメータが同じ値の点はどれも
// 左上の角から右下の角への対角線上にあらわれます
point(20, 20);
point(30, 30);
point(40, 40);
point(50, 50);
point(60, 60);
```

3-05

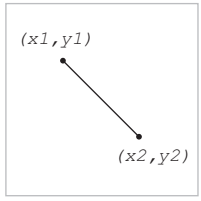
```
// Yパラメータが同じ値の点はどれも
// ディスプレイウィンドウの上端と下端から同じ距離をとります
point(50, 30);
point(55, 30);
point(60, 30);
point(65, 30);
point(70, 30);
```



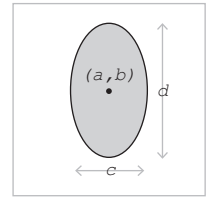
point(x, y)



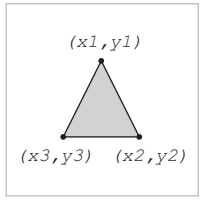
rect(a, b, c, d)



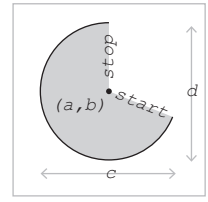
line(x1, y1, x2, y2)



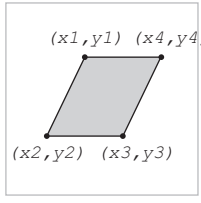
ellipse(a, b, c, d)



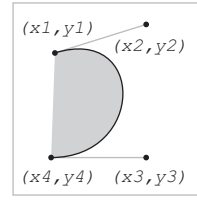
triangle(x1, y1, x2, y2, x3, y3)



arc(a, b, c, d, start, stop)



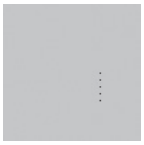
quad(x1, y1, x2, y2, x3, y3, x4, y4)



bezier(x1, y1, x2, y2, x3, y3, x4, y4)

図3-1　基本図形

Processingには、基本図形を描く8種の関数があります。この図はそれぞれの関数の書式を示しています。スケッチで使うときには、各パラメータを数値（または変数）に置き換えてください。



3-06

```
// Xパラメータが同じ値の点はどれも
// ディスプレイウィンドウの左端と右端から同じ距離をとります
point(70, 50);
point(70, 55);
point(70, 60);
point(70, 65);
point(70, 70);
```

3-07

```
// 隣り合う点の集合で直線を作ります
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);
```

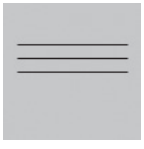
3-08

```
// ウィンドウの外側に点を描いても
// エラーにはなりませんが
// 表示はされません
point(-500, 100);
point(400, -600);
point(140, 2500);
point(2500, 100);
```

直線は、点を連続させて描くこともできますが、line()関数を使って簡単に描けます。この関数には、直線の両端の座標値を指定する4つのパラメータがあります。

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

前半2つのパラメータで直線の始点を指定し、後半2つのパラメータで直線の終点を指定します。



3-09

```
// Y座標が同じ値の直線はどれも水平になります
line(10, 30, 90, 30);
line(10, 40, 90, 40);
line(10, 50, 90, 50);
```

3-10

```
// X座標が同じ値の直線はどれも垂直になります
line(40, 10, 40, 90);
line(50, 10, 50, 90);
line(60, 10, 60, 90);
```

3-11

```
// 4つのパラメータの値が別々の直線はどれも斜めになります
line(25, 90, 80, 60);
line(50, 12, 42, 90);
line(45, 30, 18, 36);
```

3-12

```
// 同じ点を共有している2本の直線はつながります
line(15, 20, 5, 80);
line(90, 65, 5, 80);
```

triangle()関数は三角形を描きます。この関数には、3つの頂点の座標値を指定する6つのパラメータがあります。

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

はじめの2つのパラメータで最初の点、真ん中の2つで2番目の点、終わりの2つで3番目の点を指定します。3本の線をつないで三角形を描くこともできますが、triangle()関数を使えば塗りの色を持った三角形を描くことができます。パラメータ値を変えると、いろいろな形や大きさの三角形を作ることができます。

3-13

```
triangle(60, 10, 25, 60, 75, 65); // 塗りのある三角形
line(60, 30, 25, 80);             // 三角形の輪郭線
line(25, 80, 75, 85);             // 三角形の輪郭線
line(75, 85, 60, 30);             // 三角形の輪郭線
```

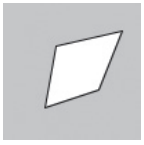
3-14

```
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);
```

quad()関数は、4つの辺を持つ多角形、四辺形を描きます。この関数には、4つの頂点の座標値を指定する8つのパラメータがあります。

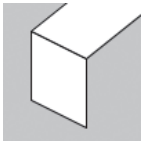
```
quad(x1, y1, x2, y2, x3, y3, x4, y4)
```

パラメータ値を変えると、長方形や正方形、平行四辺形、不規則な四辺形を作ることができます。



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

3-15



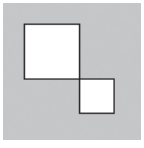
```
quad(20, 20, 20, 70, 60, 90, 60, 40);
quad(20, 20, 70, -20, 110, 0, 60, 40);
```

3-16

長方形や楕円は、これまで紹介した図形とは違った方法で描きます。各頂点を指定するのではなく、4つのパラメータで図形の位置と大きさを指定します。rect()関数は長方形を描きます。

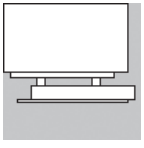
```
rect(a, b, c, d)
```

デフォルトでは、はじめの2つのパラメータで左上の位置を指定し、3つ目に幅、4つ目に高さを指定します。正方形を描くには、3つ目と4つ目に同じ値を指定してください。rect()関数は、パラメータの意味が違う別の使い方もあります。詳しくは、037ページのrectMode()関数を参照してください。



```
rect(15, 15, 40, 40); // 大きい正方形
rect(55, 55, 25, 25); // 小さい正方形
```

3-17



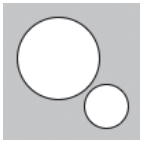
```
rect(0, 0, 90, 50);
rect(5, 50, 75, 4);
rect(24, 54, 6, 6);
rect(64, 54, 6, 6);
rect(20, 60, 75, 10);
rect(10, 70, 80, 2);
```

3-18

ellipse()関数は、楕円をディスプレイウィンドウに描きます。

```
ellipse(a, b, c, d)
```

はじめの2つのパラメータで楕円の中心位置を指定し、3つ目に幅を、4つ目に高さを指定します。正円を描くには、3つ目と4つ目のパラメータに同じ値を指定してください。



```
ellipse(40, 40, 60, 60); // 大きい円
ellipse(75, 75, 32, 32); // 小さい円
```

3-19

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

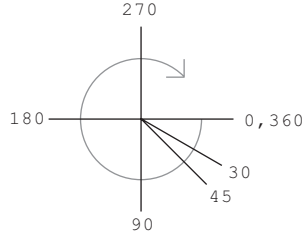
3-20

曲線

円弧は最も基本的な曲線形状で、楕円の一部分です。Processingには、円弧を簡単に描けるarc()関数があります。

```
arc(a, b, c, d, start, stop)
```

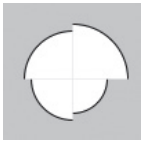
円弧は、楕円の円周上に沿って描かれます。楕円はellipse()関数のように、はじめの4つのパラメータで定義します。startパラメータとstopパラメータで円弧の開始角度と終了角度を定めます。この2つのパラメータの角度は「ラジアン」です。ラジアンとは、π（円周率）に関連して定められている角度のことです。ラジアンについては289ページで解説しているので、以下のサンプルでは一般的な「度数」を使っています。角度の値0は、楕円の右端を指し、楕円の周りを時計回り方向に角度が増加します。



radians()関数を使って、角度をarc()関数で使うラジアンに変換できます。ある関数を別の関数のパラメータとして使うと、次のサンプルのように関数の計算が先に実行されます。つまり、はじめにradians()関数で計算されたラジアンの値が、arc()関数のパラメータとして使われます。

```
arc(50, 50, 75, 75, radians(40), radians(320));
```

3-21



```
arc(50, 55, 50, 50, radians(0), radians(90));
arc(50, 55, 60, 60, radians(90), radians(180));
arc(50, 55, 70, 70, radians(180), radians(270));
arc(50, 55, 80, 80, radians(270), radians(360));
```

3-22

arc()にはもう1つのバージョンがあり、円弧を別の方法で描くことができます。

```
arc(a, b, c, d, start, stop, mode)
```

modeパラメータは、OPEN、CHORD、PIEの3つの中から1つを選びます。図3-2のように、OPENがデフォルトのモードで、CHORDとPIEでは別の図形になります。次のサンプルでは、fill()とstroke()を使った描画の選択肢を全て示しています。線がない場合、OPENとCHORDの見た目に違いがないことに気をつけてください。

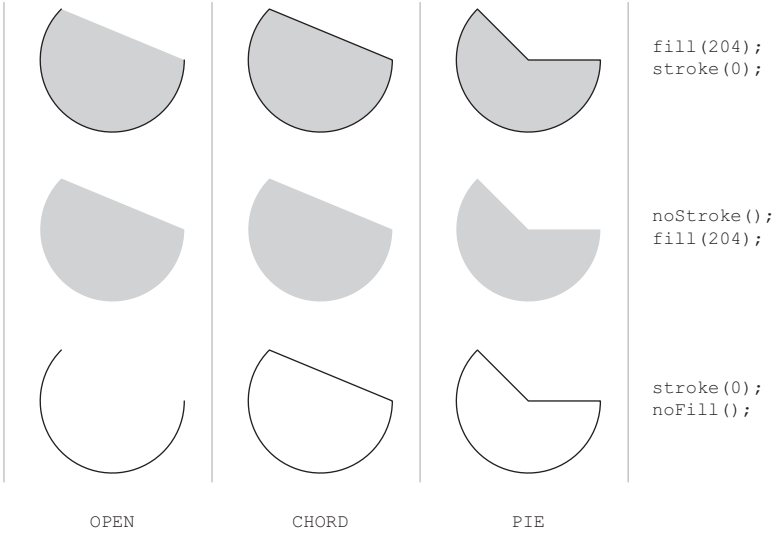


図3-2　円弧

円弧の位置、サイズ、角度は、6つのパラメータで指定します。7つ目のパラメータで、円弧の始点と終点のつなぎ方を指定できます。この図では、パラメータOPEN、CHORD、PIEの使い方をあらわしています。

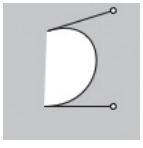
3-23

```
// 上段、塗りと線あり
arc(20, 20, 28, 28, radians(0), radians(225), OPEN);
arc(50, 20, 28, 28, radians(0), radians(225), CHORD);
arc(80, 20, 28, 28, radians(0), radians(225), PIE);
// 中段、線なし
noStroke();
arc(20, 50, 28, 28, radians(0), radians(225), OPEN);
arc(50, 50, 28, 28, radians(0), radians(225), CHORD);
arc(80, 50, 28, 28, radians(0), radians(225), PIE);
// 下段、塗りなし
stroke(0);
noFill();
arc(20, 80, 28, 28, radians(0), radians(225), OPEN);
arc(50, 80, 28, 28, radians(0), radians(225), CHORD);
arc(80, 80, 28, 28, radians(0), radians(225), PIE);
```

bezier()関数では、円弧よりも複雑な曲線を描くことができます。「ベジェ曲線」は、コントロールポイントとアンカーポイントによって定義される曲線です。曲線はアンカーポイント間に描かれ、コントロールポイントによって曲線の形状が決まります。

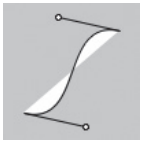
bezier(x1, y1, x2, y2, x3, y3, x4, y4)

bezier()関数には、4つの座標値を指定する8つのパラメータが必要です。はじめの座標と4つ目の座標の間に曲線が描かれ、2つ目の座標と3つ目の座標でコントロールポイントが定義されます。ベジェ曲線を使うAdobe Illustratorのようなソフトウェアでは、コントロールポイントは、曲線の端から引き出された小さなハンドルで表示されています。



3-24

```
bezier(32, 20, 80, 5, 80, 75, 30, 75);
// コントロールポイントを描きます
line(32, 20, 80, 5);
ellipse(80, 5, 4, 4);
line(80, 75, 30, 75);
ellipse(80, 75, 4, 4);
```



3-25

```
bezier(85, 20, 40, 10, 60, 90, 15, 80);
// コントロールポイントを描きます
line(85, 20, 40, 10);
ellipse(40, 10, 4, 4);
line(60, 90, 15, 80);
ellipse(60, 90, 4, 4);
```

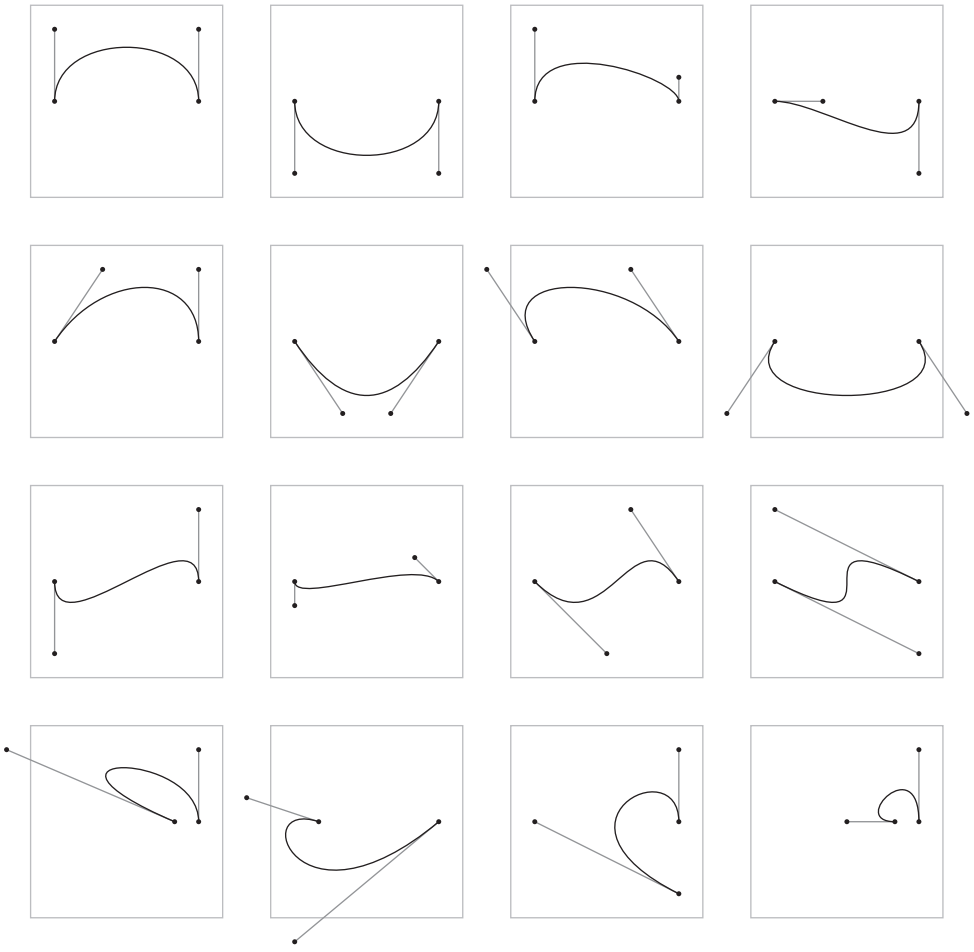


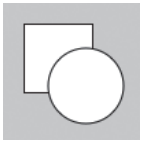
図3-3　ベジェ曲線

ベジェ曲線の形状は、アンカーポイントとコントロールポイントによって決まります。曲線は、2つのアンカーポイント間に描かれますが、その形状はコントロールポイントの場所によって決まります。

長いベジェ曲線は、bezierVertex()関数を使い、vertex()とbeginShape()を組み合わせて作ります。その使い方は199ページで説明しています。その他のタイプの曲線は、198ページで説明するcurveVertex()で定義します。

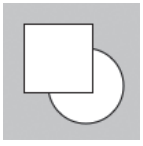
描画の順序

コード中の図形を描く順序によって、ディスプレイウィンドウでどの図形が前面に出るかが決まります。プログラムの1行目に描いた長方形は、2行目で描いた楕円に覆われます。コードの順序をひっくり返すと、ビジュアルの結果も逆になります。



```
rect(15, 15, 50, 50);     // 背面
ellipse(60, 60, 55, 55); // 前面
```

3-26



```
ellipse(60, 60, 55, 55); // 背面
rect(15, 15, 50, 50);     // 前面
```

3-27

グレーの値

これまでのサンプルでは、デフォルトの明るいグレーの背景色、黒色の線、白色の塗りを使ってきました。このデフォルトの値は、background()関数、fill()関数、stroke()関数で変更できます。background()関数はディスプレイウィンドウのグレー値を0から255の間の数値で指定します。コンピュータ上の色選択になじみのない人は、この数値の範囲を中途半端に感じるかもしれません。値255は白色で、値0は黒色です。様々な濃さのグレーはその中間の値をとります。背景の値を指定しない場合、デフォルト値の204（明るいグレー）が使われます。

```
background(0);
```

3-28

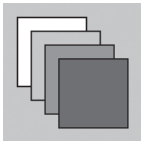
```
background(124);
```

3-29

```
background(230);
```

3-30

`fill()`関数は図形の塗りの値、`stroke()`関数は図形の輪郭線の値を指定します。塗りの値を指定しないと、デフォルトの値255（白色）が使われます。線の値を指定しないと、デフォルトの値0（黒色）が使われます。



```
rect(10, 10, 50, 50);
fill(204); // 明るいグレー
rect(20, 20, 50, 50);
fill(153); // 中間のグレー
rect(30, 30, 50, 50);
fill(102); // 濃いグレー
rect(40, 40, 50, 50);
```

3-31

```
background(0);
rect(10, 10, 50, 50);
stroke(102); // 濃いグレー
rect(20, 20, 50, 50);
stroke(153); // 中間のグレー
rect(30, 30, 50, 50);
stroke(204); // 明るいグレー
rect(40, 40, 50, 50);
```

3-32

いったん塗りや線の値を設定すると、設定した後に描かれる全ての図形にその値が適用されます。塗りや線の値を変更したいときは、`fill()`関数や`stroke()`関数をもう一度使ってください。

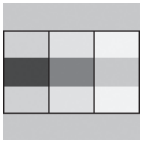
```
fill(255); // 白色
rect(10, 10, 50, 50);
rect(20, 20, 50, 50);
rect(30, 30, 50, 50);
fill(0);   // 黒色
rect(40, 40, 50, 50);
```

3-33

`fill()`と`stroke()`には、透明度を設定する2つ目のパラメータを追加して指定できます。このパラメータに255を指定すると図形が完全な不透明になり、0を指定すると完全な透明になります。

3-34

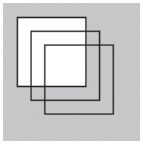
```
background(0);
fill(255, 220); // 高い不透明度
rect(15, 15, 50, 50);
rect(35, 35, 50, 50);
```



3-35

```
fill(0);
rect(0, 40, 100, 20);
fill(255, 51);  // 低い不透明度
rect(0, 20, 33, 60);
fill(255, 127); // 中間の不透明度
rect(33, 20, 33, 60);
fill(255, 204); // 高い不透明度
rect(66, 20, 33, 60);
```

図形の線と塗りをなくすこともできます。noFill()関数は、Processingに図形の塗りをやめさせます。noStroke()関数は、線を描くのをやめて図形の輪郭線を取り除きます。noFill()とnoStroke()の両方を使うと、画面には何も描かれません。



3-36

```
rect(10, 10, 50, 50);
noFill(); // 塗りを無しにします
rect(20, 20, 50, 50);
rect(30, 30, 50, 50);
```

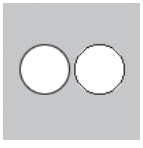
3-37

```
rect(20, 15, 20, 70);
noStroke(); // 線を無しにします
rect(50, 15, 20, 70);
rect(80, 15, 20, 70);
```

塗りと線のカラーの設定は、次の章（038ページ）で紹介しています。

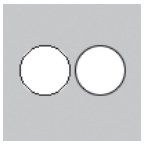
属性

塗りと線の値の変更のほかに、幾何学的な属性を変更することもできます。noSmooth()関数とsmooth()関数は、スムージング（アンチエイリアシングとも呼ばれます）を無効にしたり有効にしたりします。スムージングはデフォルトで有効になっているので、noSmooth()を使うと無効になります。一度スムージングを無効にすると、smooth()を再び使うまでは有効にはなりません。



3-38

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



3-39

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

線の属性は、strokeWeight()関数、strokeCap()関数、strokeJoin()関数で設定します。strokeWeight()関数は1つの数値パラメータを持っていて、この関数を使った後に描かれる全ての線の太さを指定します。strokeCap()関数は、ROUND、SQUARE、PROJECTのうちどれか1つをパラメータとして指定します。

ROUNDは線の端を丸くし、SQUAREは四角くします。PROJECTはこの2つを組み合わせたもので、SQUAREの線の端を線の半径分だけ伸ばします。strokeJoin()関数は、BEVEL、MITER、ROUNDのうちどれか1つをパラメータとして指定します。このパラメータは、線の部分や図形の輪郭線の接合方法を定めます。BEVELは四角い角、デフォルトのMITERは鋭い角、ROUNDは曲線を作って線をつなぎます。

3-40

```
line(20, 20, 80, 20); // デフォルトの線幅1ピクセル
strokeWeight(6);
line(20, 40, 80, 40); // 太い線
strokeWeight(18);
line(20, 70, 80, 70); // さらに太い線
```

3-41

```
strokeWeight(12);
str1okeCap(ROUND);
line(20, 30, 80, 30); // 上の線
strokeCap(SQUARE);
line(20, 50, 80, 50); // 中央の線
strokeCap(PROJECT);
line(20, 70, 80, 70); // 下の線
```

3-42

```
strokeWeight(12);
strokeJoin(BEVEL);
rect(12, 33, 15, 33); // 左側の図形
strokeJoin(MITER);
rect(42, 33, 15, 33); // 中央の図形
strokeJoin(ROUND);
rect(72, 33, 15, 33); // 右側の図形
```

モード

デフォルトでは、ellipse()のパラメータは、中心点のX座標、中心点のY座標、幅、高さを指定しています。ellipseMode()関数は、楕円を描くときのパラメータの指定方法を変えます。ellipseMode()関数は、CENTER、RADIUS、CORNER、CORNERSのうちどれか1つをパラメータとして指定します。デフォルトのモードはCENTERです。RADIUSモードも、ellipse()のはじめの2つのパラメータで中心点を指定しますが、3つ目のパラメータには幅の半分を、4つ目のパラメータには高さの半分を指定します。CORNERモードは、ellipse()をrect()と同じように指定できるようにします。つまり、はじめの2つのパラメータで楕円に外接する長方形の左上角の位置を指定し、3つ目と4つ目のパラメータで長方形の幅と高さを指定します。CORNERSモードはCORNERとほぼ同じですが、ellipse()の3つ目と4つ目のパラメータで外接する長方形の右下角を指定します。

3-43

```
noStroke();
ellipseMode(RADIUS);
fill(126);
ellipse(33, 33, 60, 60); // グレーの円
fill(255);
ellipseMode(CORNER);
ellipse(33, 33, 60, 60); // 白色の円
fill(0);
ellipseMode(CORNERS);
ellipse(33, 33, 60, 60); // 黒色の円
```

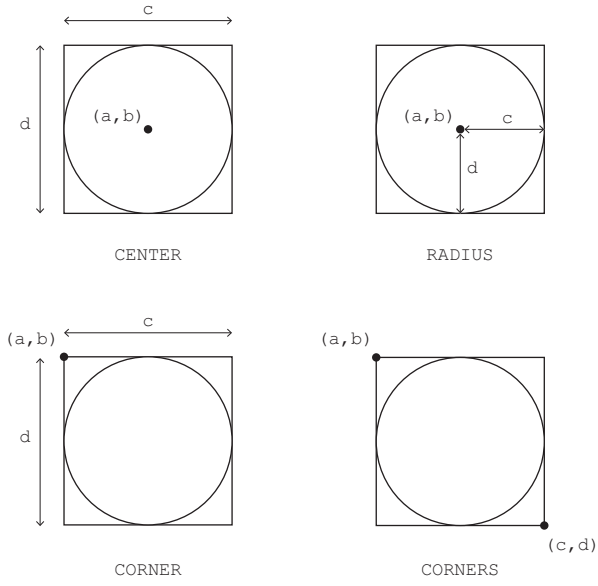


図3-4　描画モード

ellipseMode()関数とrectMode()関数は、楕円と長方形を画面に描く方法を変更します。この図であらわしている通り、それぞれのパラメータ（たとえばCENTER）が図形の位置を調整しています。モードの動作はellipse()とrect()で共通ですが、ellipse()のデフォルトのモードはCENTERで、rect()のデフォルトのモードはCORNERです。

同じやり方で、rectMode()関数は長方形の描画方法を変更します。rectMode()関数は、CORNER、CORNERS、CENTERのうちどれか1つをパラメータとして指定します。デフォルトのモードはCORNERで、CORNERSモードはrect()の3つ目と4つ目のパラメータに最初に指定した角と反対側の角を指定します。CENTERモードは、rect()のはじめの2つのパラメータに長方形の中心点を指定し、3つ目と4つ目のパラメータに幅と高さを指定します。

3-44

```
noStroke();
rectMode(CORNER);
fill(126);
rect(40, 40, 60, 60); // グレーの正方形
rectMode(CENTER);
fill(255);
rect(40, 40, 60, 60); // 白色の正方形
rectMode(CORNERS);
fill(0);
rect(40, 40, 60, 60); // 黒色の正方形
```

練習問題

1. 紙に4×4の正方形のグリッドを描いてみよう。正方形の中に複数の線と円からなる構図をそれぞれ描いてみよう。
2. 練習問題1から、1つの構図を選んでコードにしてみよう。
3. 練習問題2のコードを編集して、塗り、線、背景の値を変えてみよう。
4. 3つの楕円を構成して、ディスプレイウィンドウ内に疑似的な深さを表現してみよう。大きさと描画の順序、透明度を考えながら。
5. ベジェ曲線だけを使って、視覚に訴える結び目を作ってみよう。

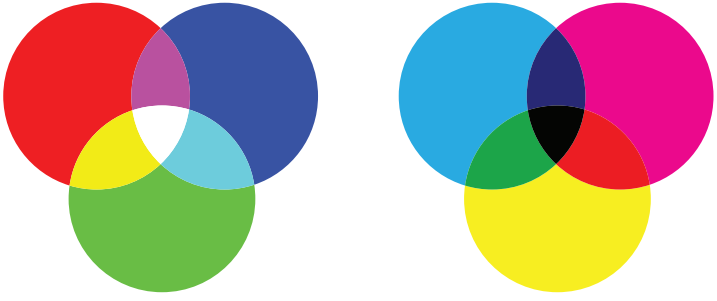
4. 色

本章では、ソフトウェアで色を扱うコードと概念を解説します。

紹介する構文：

`blendMode()`、`colorMode()`

画面上の色の働きは、紙やキャンバスの色の働きとは違います。絵画と印刷物の色には同じ原則があてはまります。絵画の基本的な顔料はカドミウムレッド、プルシャンブルー、バーントアンバーで、印刷物の基本的なインクの色はシアン、イエロー、マゼンタです。ところがこうした知識をデジタルディスプレイの色づくりに必要な情報へと移し替えることはできません。絵画の基本色を全部混ぜると黒色か濃い茶色になりますが、コンピュータのモニタで基本色を全部混ぜると白色になります。モニタは、光を使って色を混ぜ合わせています。画面は黒い面で、そこに色のついた光が足されているのです。これは「加法混色」として知られていて、紙やキャンバス上のインクに適用される「減法混色」とは対照的です。この図は、2つのカラーモデルの違いをあらわしています。



加法混色　減法混色

コンピュータでは、色をRGB値で指定するのが最も一般的な方法です。RGB値では、画面の1ピクセル中の赤、緑、青の光の量を設定します。モニタやテレビの画面に近づいてよく見てみると、どのピクセルも赤、緑、青で3分割された光の要素で構成されていることがわかります。しかし、私たちは各色の量を細かく見分けることができないので、3色が混ぜ合わさってできた1つの色を見ているのです。各色の要素の量は、通常0が最小値で255が最大値の、0から255までの値で指定されます。他の多くのソフトウェアもこの値の範囲を利用しています。赤、緑、青の各要素を0に指定すると黒色ができます。各要素を255に指定すると白色ができます。赤を255、緑と青を0に指定すると、鮮やかな赤色ができます。

切りの良い数値で色を選ぶのは簡単です。たとえば青色の(0, 0, 255)や、緑色の(0, 255, 0)といったパラメータをよく見かけます。こうした組み合わせは、コンピュータで作られた技術系の図にありがちな、けばけばしい色使いを引き起こしてしまいます。このような色使いは、微妙な色を見分けることのできる人間の感覚を無視したもので、強烈すぎて不自然です。私たちの目にほどよい色は、通常このような切りの良い数値になることはありません。0や255といった数値は使わず、カラーセレクターを使って丁寧に色を選ぶようにしてください。Processingのカラーセレクターは、Toolsメニューから開きます。カラーフィールド内をクリックするか数値を直接入力して、色を選択します。たとえば図4-1で選択している青色は、R値が35、G値が211、B値が229です。この数値を使って、選択した色をコード内で再利用できます。

数で色を作る

Processingでは、`background()`関数、`fill()`関数、`stroke()`関数の数値パラメータで色を指定します。デフォルトでは、1つ目のパラメータに赤、2つ目に緑、3つ目に青の色要素を指定します。`fill()`関数と`stroke()`関数では、4つ目のパラメータを追加して透明度を指定できます。透明度のパラメータの値は、255が完全な不透明色で、0が完全な透明（表示されない）です。

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

4-01

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

4-02

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

4-03

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

4-04

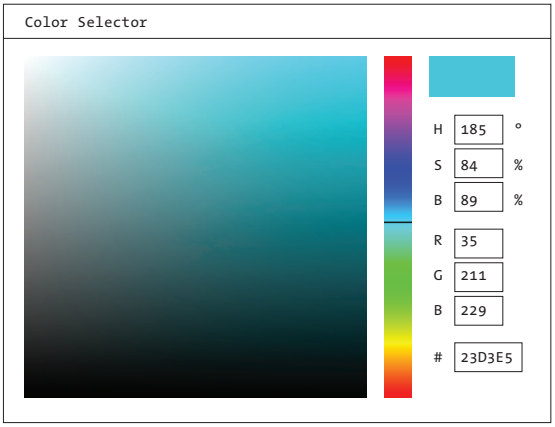


図4-1　カラーセレクター

色を選択するには、ウィンドウ内でカーソルをドラッグするか数値を入力してください。大きな正方形のエリアで彩度と明度を決め、縦に細長い帯で色相を決めます。選択した色の数値は、HSB、RGB、16進数記法で表示されます。

4-05

```
background(116, 193, 206);
noStroke();
fill(129, 130, 87, 102); // 高い透明度
rect(20, 20, 30, 60);
fill(129, 130, 87, 204); // 低い透明度
rect(50, 20, 30, 60);
```

4-06

```
background(116, 193, 206);
noStroke();
fill(129, 130, 87, 51);
rect(0, 20, 20, 60);
fill(129, 130, 87, 102);
rect(20, 20, 20, 60);
fill(129, 130, 87, 153);
rect(40, 20, 20, 60);
fill(129, 130, 87, 204);
rect(60, 20, 20, 60);
fill(129, 130, 87, 255);
rect(80, 20, 20, 60);
```

4-07

```
background(56, 90, 94);
strokeWeight(12);
stroke(242, 204, 47, 102); // 高い透明度
line(30, 20, 50, 80);
stroke(242, 204, 47, 204); // 低い透明度
line(50, 20, 70, 80);
```

4-08

```
background(56, 90, 94);
strokeWeight(12);
stroke(242, 204, 47, 51);
line(0, 20, 20, 80);
stroke(242, 204, 47, 102);
line(20, 20, 40, 80);
stroke(242, 204, 47, 153);
line(40, 20, 60, 80);
stroke(242, 204, 47, 204);
line(60, 20, 80, 80);
stroke(242, 204, 47, 255);
line(80, 20, 100, 80);
```

透明度を使うと、図形を重ね合わせて新しい色を作ることができます。重なり合った部分の色は、図形を描く順序によって変わります。

4-09

```
background(0);
noStroke();
fill(242, 204, 47, 160);  // 黄色
ellipse(47, 36, 64, 64);
fill(174, 221, 60, 160);  // 緑色
ellipse(90, 47, 64, 64);
fill(116, 193, 206, 160); // 青色
ellipse(57, 79, 64, 64);
```

4-10

```
background(255);
noStroke();
fill(242, 204, 47, 160);  // 黄色
ellipse(47, 36, 64, 64);
fill(174, 221, 60, 160);  // 緑色
ellipse(90, 47, 64, 64);
fill(116, 193, 206, 160); // 青色
ellipse(57, 79, 64, 64);
```

ブレンド

画面の色は、スケッチの現在のブレンドモードにしたがって混ぜ合わされます。デフォルトでは、Processingは色を置き換えるか、透明度が設定されていればブレンドします。blendMode()関数は、ピクセルをミックスするいろいろな方法を1つのパラメータで指定します。図形や画像が画面に描画されると、各ピクセルは、それぞれのブレンドモードのルールにしたがってミックスします。モードの選択肢には、BLEND、ADD、SUBTRACT、DARKEST、LIGHTEST、DIFFERENCE、EXCLUSION、MULTIPLY、SCREEN、REPLACEがあります。塗りや線の設定と同じく、ブレンドモードを設定すると、モードが再設定するまでは引き続き適用されます。モードの選択については図4-2で説明しています。サンプルコードはこちらです。

4-11

```
size(100, 100);
stroke(153, 204);
strokeWeight(12);
background(0);
line(20, 20, 40, 80);
line(40, 20, 20, 80);
blendMode(ADD); // ブレンドモードを変更します
line(60, 20, 80, 80);
line(80, 20, 60, 80);
```

デフォルトのモードはBLENDなので、モードを変更した後に元に戻す場合は、BLENDに設定しなおします。次のサンプルは、前のサンプルとよく似ていますが、はじめにブレンドモードをADDに変更していて、後でデフォルトに戻しています。

4-12

```
size(100, 100);
stroke(153, 204);
strokeWeight(12);
background(0);
blendMode(ADD);   // ブレンドモードを変更します
line(20, 20, 40, 80);
line(40, 20, 20, 80);
blendMode(BLEND); // デフォルトに戻します
line(60, 20, 80, 80);
line(80, 20, 60, 80);
```