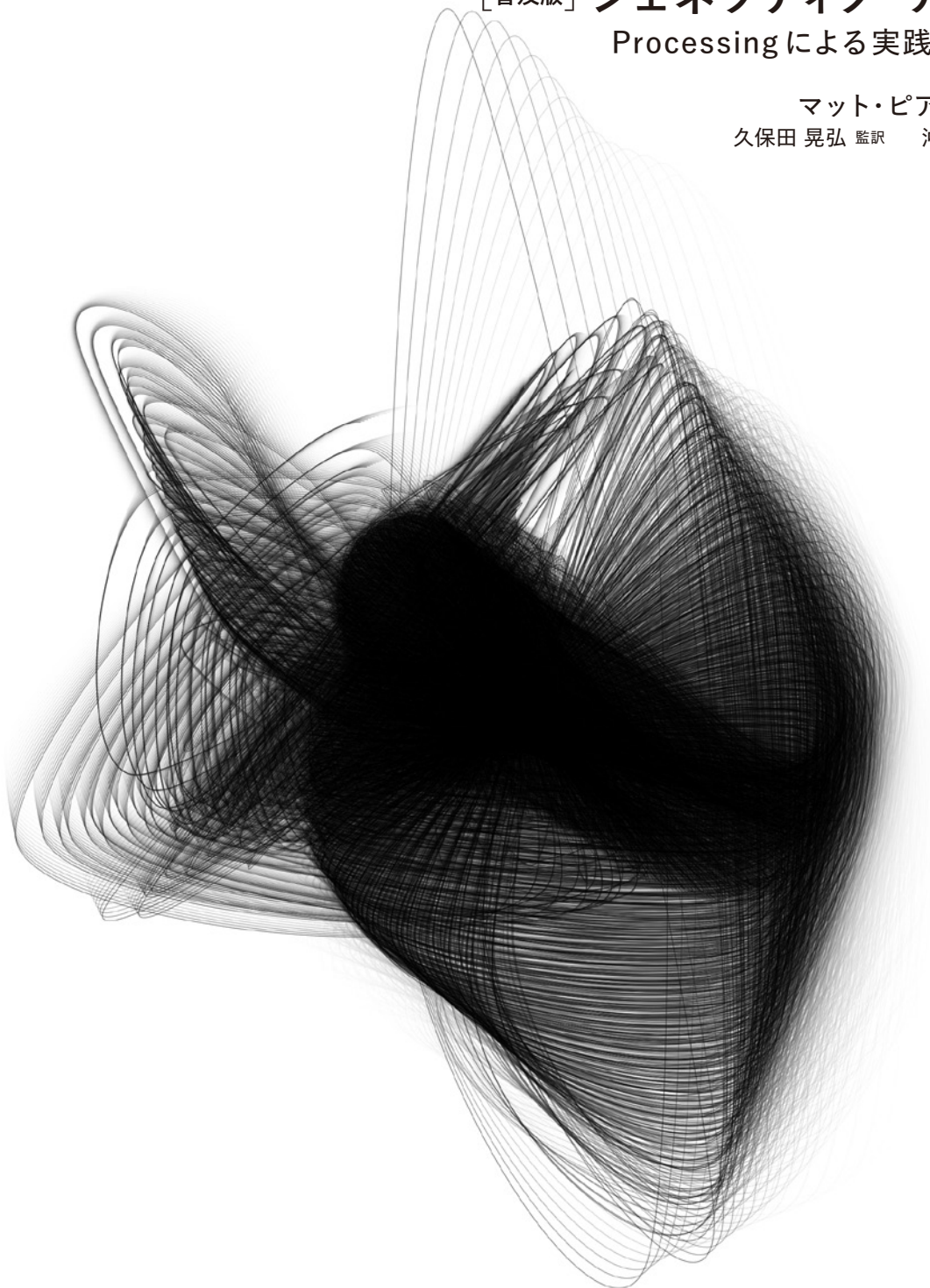
［普及版］ジェネラティブ・アート
Processingによる実践ガイド

マット・ピアソン 著
久保田 晃弘 監訳　沖 啓介 訳



BNN
Bug News Network

ジェネラティブ・アート
Processingによる実践ガイド

マット・ピアソン 著
久保田 晃弘 監訳　沖 啓介 訳

PDF版
2014年11月7日作成

Original English language edition published by Manning Publications, USA.
Copyright © 2012 by Manning Publications Co.
Japanese-language edition copyright © 2014 by BNN, Inc.
All rights reserved.

Japanese translation rights arranged with Waterside Productions, Inc., as agents for Manning Publications, through Japan UNI Agency, Inc., Tokyo

ISBN: 978-4-86100-963-1
Printed in Japan

dedicated to my boys, *Rudy* and *Oz*

私の息子たち、ルディとオズに捧ぐ。

“Rome misc, Dec 2009 - 17” By Ed Yourdon (CC:BY-SA 2.0)
http://www.flickr.com/photos/yourdon/4196685453/in/photostream/

Generative Art
a practical guide using processing

日本語版に寄せて

『Generative Art』の初版を印刷入稿したすぐ後、私は「HTML5によるジェネラティブ・アート」*1という題の短いチュートリアルを、この本の補遺としてCreativeApplications.netに投稿しました。私がこの本で何か決定的なことをやり残したのではないか、という冷や汗をかくような思いの後、何かしなければならないと感じたからでした。それは、何かやり残しがあったというようなものではなく、あるコンセプトが十分に強調されていないのではないかという実感――ジェネラティブ・アートは普遍的な方法であって、特定のプログラム言語に限られたものではない、という思いです。

Processingは、ジェネラティブ・アートを探求するための素晴らしいツールです。それが、本書でProcessingをじっくりと扱った理由です。しかし私は、Processingがジェネラティブな方法をコードするために使える唯一の言語であると、読者の方々に誤って信じこんで欲しくはありません。本書に見られるコードもアイデアも、容易にHTML5、C++、Ruby、Visual Basic、Python、Scratch、PHP、Objective-C、あるいはあなたが知っているいかなる手続き型の言語にも置き換えることができます。ジェネラティブ・アートは同様に、ペンと紙と電卓で作ることもできますし、棒とサイコロ、さらには目隠しされているハムスターでも大丈夫です。もし私が無人島の上で見捨てられたとしても、救いの手を待つ間、ヤシの実と鳥の糞を使って、何かしらの手続き的な創造方法を思いついているでしょう。けれど私は、計算機械のとてつもない便利さや、超人的なレベルの計算能力で実験を行ってくれる言語を容易に習得できることにも、とても感謝しています。

本に書かれている人間のための言語が、機械語と同じぐらいに翻訳が容易であったらよいのですが。

プログラミング言語を選ぶのと同じように、散文的な文体を選んだことについては、実はあまり深く考えていませんでした。それは単に、決めるのにそれほど選択の幅を持っていなかったというだけの理由です。私はイギリス人の両親のもと、イギリスで生まれ育ちました。私は生かじりのフランス語とドイツ語を話せて、それに少しだけスウェーデン語の悪たれ口を知っています。けれども、私がどうにか熟達した唯一の言語手段は英語です。たくさんのコーディング言語を知っているにもかかわらず、読者にさまざまなアイデアを伝えられるのが英語だけというのは、何か気恥ずかしいことです。

私の仕事がこうして最初に外国語に翻訳されるのは、本当に名誉なことだと思います。まず第一に、私の文章を翻訳に価すると思ってくれたことを嬉しく思います。第二に、この本が文化的な鳥かごから抜け出して新しい世界に飛び立っていくのを見ることには、とても胸躍るものがあります。私自身の文化遺産と結びついているのは、私の言語だけでなく、引用もまた同様です。私はザ・スミスとエコー＆ザ・バニーメンを比喩として使っています［いずれもイギリスのポストパンクバンド］。私は、大好きなイギリスのエレクトロニカ・アーティスト、オウテカの言葉も引用しています。日本から遠く離れた、海のそばの私の故郷ブライトンの海岸を飛びまわるムクドリについて述べています。もしあなたが私のことを理解しようとするなら、翻訳する必要があるのは言語だけではなく、こうした引用の数々についてです。しかし私は、これまでに私が達し遂げたこと以上に、私が書いたものにより大きな広がりがあると信じています。

見事な手腕と行動で、この新しい版をもたらしてくれた沖啓介さんに感謝します。彼は、特異な話し言葉の、ひどくくせのある私の思考表現方法を受け止めて、新しい読者のために読み解いてくれました。私の本は生まれ変わったのです。

ジェネラティブ・アートに共通言語はありません。その言語は、日本語でも英語でもありません。同様にそれは、Processing、HTML5、あるいはC++でもありません。ジェネラティブ・アートは、アルゴリズムを土台にした不可知論的な構築物です。そのルーツは、自然や、世界の働きを表す数学の中にあります。ジェネラティブ・アートは文化の境界に目もくれません。私もまたここに、そうあることを誓います。

ブライトン、2012年11月13日
マット・ピアソン

＊1　http://www.creativeapplications.net/processing/generative-art-in-html5-processing-javascript-tutorial/
［日本語訳　http://www.cbc-net.com/topic/2012/12/generative-art-in-html5/］

監訳者のことば

美術、芸術にとって「生成（generative/generation）」という概念は、とても重要な意味を持っています。ベンヤミンが1936年に発表した有名な論文『複製技術の時代における芸術作品』*1は、写真や映画という複製メディアの出現によって、芸術作品の価値、見方がどう変わったかについて、複製技術と知覚の面から論じていました。その後表現のメディアは機械から情報へと移行し、複製不可能な作品が持つ一回性、唯一性から生まれる「アウラ」が、デジタル技術によって可能になった非破壊ノンリニア編集による多数のバージョン並置によって、分散型、並列型の「アウラ」へと変容していきました。

さらにメディアは、結果を記録するデータから、プロセスを記述できるアルゴリズム、プログラムへと移行していきます。これからの音楽家はCDではなくアプリを配布し、これからの映像作家はDVDやBlu-rayではなくソフトウェアを実行します。この「生成」するメディアによって形成される知覚様式とその価値は、礼拝的な古代の価値でも展示的な現代の価値でもなく、日々変化するウェブ・スペースがそうであるように、それが「今まさに目の前で計算され、生成している」という生成的価値を持ちます。その価値は、従来の複製技術時代を超えて、「生成技術の時代の芸術作品とは何か」という新たな問いを私たちに投げかけることになりました。

本書は、そんな「生成する芸術（Generative Art）」を、コンピュータとProcessingという現代の道具を用いて、自分の手で作ってみようと思っている人のための最良の教科書です。

本書の中にもたびたび登場しますが、80年代以降「非線形」「並列分散」「カオス」「フラクタル」「自己組織」「散逸構造」「オートポイエーシス」「ネットワーク」「複雑系」――こうしたさまざまな言葉が、先を急ぐかのように論客たちにとりあげられ、そそくさと通り過ぎていきました。しかし「生成する」ことを本気で考えようと思ったら、アルゴリズムを実行するための最良の道具であり今日の芸術表現の素材である「プログラム」を書き（コピーし、編集し、でももちろんかまいません）、コンピュータで走らせ、そこで生成されるものと（もちろんうまく生成される前の、たくさんのエラーやバグとも）、時間をかけて全身で対峙していくしかありません。じっくりと見ずに絵画を論じたり、ゆっくりと食事をせずに料理を味わうことは、かなり難しいことだと思います（あえて不可能だとは言いませんが）。

Processingの創始者の一人であるケイシー・リースらによる『FORM＋CODE――デザイン／アート／建築における、かたちとコード』*2、「21世紀のエドワード・タフテ（彼は“データのレ

オナルド・ダ・ヴィンチ”と呼ばれています)」と称されるマニュアル・リマによる『ビジュアル・コンプレキシティ――情報パターンのマッピンク』*3(いずれもビー・エヌ・エヌ新社)、そして本書『ジェネラティブ・アート――Processingによる実践ガイド』と、はからずも監訳／監修者として深く関わることになったこの3冊が、今日の芸術やデザインにおける生成的価値を考えるための「歴史」「概念」「方法」をまとめた3部作のようになりました。訳者の方々のみならず、企画を実現していただいたBNN編集部の村田純一さんには感謝の念に堪えません。それぞれの本のスタンスやアプローチは異なりますが、いずれの本も、今日の膨大なデータと高速な計算力を背景にした「生成技術時代の芸術作品」を考えるための必読書と言っていいでしょう。

とりわけ本書は、理念のみならず実践のための入門書として、アーティストやデザイナーをはじめとするさまざまな分野のクリエイター(志望)の方々に、ぜひ手に取って読んで(実行して)いただければと思います。マット・ピアソンの軽妙な語り口とリベラルな文化的背景の中に込められた芸術を愛する気持ちが、そうした人たちには(不安一杯の創作活動における)励ましの言葉として暖かく届いてくれると思います。創造とエラーが深く関わっているのだとすれば、プログラムを正しく動かすことと間違った方法で表現することの境界はどこにあるのか、予測不可能性を生み出すコンピュータの疑似乱数が決定的であることは一体何を意味しているのか等々、本書が提示したさまざまな問いかけは、いずれもコンピュータで芸術を実践することの根幹を露にする、普遍的で深い設問です。大切なのは、答えを見つけることではなく、作り続けることです。もしその営みから、あなたの中に何かざわざわするような衝動が生成されたのなら、それがきっと、ベンヤミンの言う「芸術の最も重要な課題のひとつは、その時代においてはまだ十分な満足を与えることができないような新しい需要を作り出す(One of the foremost tasks of art has always been the creation of a demand which could be fully satisfied only later)」ことへと繋がっていくのです。

2012年11月

多摩美術大学 メディア芸術コース 教授
久保田晃弘

＊1　http://www.marxists.org/reference/subject/philosophy/works/ge/benjamin.htm
＊2　http://www.bnn.co.jp/support/formandcode/
＊3　http://www.visualcomplexity.com/

日本語版によせて 6
監訳者のことば 8
序 文 13
はじめに 16
謝 辞 19
本書について 20
著者について 22

イントロダクション：
有機的なものと
機械的なもの 24

ジェネラティブ・アートは簡単 28
秩序とカオス 31
詩としてのプログラミング 41
カオス芸術家 47

Part 1 クリエイティブ・コーディング

Chapter 1 ジェネラティブ・アート：
理論と実践 51

1.1 私たちの芸術様式 52
1.2 新しい考え方の歴史 54
1.3 デジタルなツールセット 57
1.3.1 絶え間ないはかなさ 58
1.3.2 最新の稚拙な技術 59
1.4 まとめ 60

Chapter 2 Processing：
アーティストのための
プログラミング言語 61

2.1 Processingとは？ 62
2.1.1 大股歩きとヨチヨチ歩き 63
2.1.2 ハロー・ワールド 65
2.2 プログラムでお絵描き 67
2.2.1 関数、パラメータ、色の値 68
2.2.2 線、スタイル、座標 70
2.2.3 変数 71
2.2.4 塗りつぶし、
アルファ値、描画の順序 72
2.3 構造、論理、アニメーション 75
2.3.1 フレームループ 75
2.3.2 自分の関数を書く 78
2.3.3 演算子 79
2.3.4 条件文 81
2.4 繰り返し 82
2.4.1 whileループ 83
2.4.2 痕跡を残す 85
2.4.3 forループ 87
2.5 作品をセーブし、
パブリッシュし、配信する 88
2.5.1 バージョン管理 90
2.5.2 静止画を作る 91
2.5.3 代替イメージとして静止画を使う 92
2.5.4 ビデオを作る 93
2.5.5 フレームレートと画面サイズ 93
2.5.6 モバイル装置、
iPhone / iPad と Android 94
2.6 まとめ 96

Part 2 ランダム性とノイズ

Chapter 3 線を引く間違った方法 …… 99

3.1 デタラメさとそうでもないこと …… 100

3.2 変化の繰り返し …… 102

3.3 自然な変動 …… 105

3.3.1 Processingでのパーリンノイズ …… 105

3.3.2 自分のノイズを作る …… 108

3.3.3 自分だけのランダム関数 …… 111

3.4 まとめ …… 112

Chapter 4 円を描く間違った方法 …… 113

4.1 回転するドローイング …… 113

4.1.1 初めて円を描く …… 114

4.1.2 円をらせんに変える …… 117

4.1.3 ノイズの多いらせん …… 117

4.1.4 自分のノイズを作る。ふたたび …… 121

4.2 ケーススタディ：Wave Clock …… 124

4.3 まとめ …… 130

Chapter 5 次元を加える …… 131

5.1 2次元のノイズ …… 132

5.1.1 ノイズグリッドを作る …… 132

5.1.2 ノイズの視覚化 …… 133

5.2 ノイズ・アニメーション …… 137

5.3 3次元 …… 140

5.3.1 3次元空間で描く …… 141

5.3.2 3次元のノイズ …… 143

5.3.3 球を描く間違った方法 …… 148

5.4 まとめ …… 151

Part 3 複雑性

Chapter 6 創 発 …… 155

6.1 創発を定義する …… 155

6.1.1 アリの群れと群集アルゴリズム …… 156

6.1.2 ローカルに考え、ローカルに行動する …… 158

6.2 オブジェクト指向プログラミング …… 160

6.2.1 クラスとインスタンス …… 161

6.2.2 ローカルな知識（衝突判定） …… 167

6.2.3 インタラクション・パターン …… 169

6.3 まとめ …… 174

Chapter 7 自 律 性 …… 175

7.1 セル・オートマトン …… 176

7.1.1 枠組みをセットアップ …… 178

7.1.2 ゲーム・オブ・ライフ …… 182

7.1.3 ヴィシュニアク・ヴォート …… 185

7.1.4 ブライアンの脳 …… 186

7.1.5 波（平均化） …… 191

7.2 シミュレーションと
ビジュアライゼーション …… 194

7.2.1 ソフトウェア・エージェント …… 194

7.2.2 人間エージェント …… 196

7.3 まとめ …… 201

Chapter 8 フラクタル …… 203

8.1 無限の再帰 …… 205

8.2 自己相似性のコーディング …… 207

8.2.1 幹と枝 …… 207

8.2.2 あなたの木を動かす …… 211

8.3 指数的成長 …… 213

8.4 ケーススタディ：
サトクリフ五角形 …… 218

8.4.1 組み立て …… 219

8.4.2 探求 …… 228

8.5 まとめ …… 235

訳者あとがき …… 238

アートワークリスト …… 239　　索 引 …… 240

■ 凡例

- 本書は、2012年12月刊行書籍『ジェネラティブ・アート――Processingによる実践ガイド』の［普及版］です。構成内容に変更はありません。
- 本書は、Matt Pearson “Generative Art: A Practical Guide Using Processing”（Manning Publications Co. , 2011）の翻訳版です。
- 日本語版制作にともないデザインを変更しました。
- 訳注・編注は［ ］で括っています。
- 外国人名はカタカナで示しました。ただし図版などの出典表示は原書の表記に従い欧文で示しています。

■ Processingのバージョンとサンプルコードについて

- 本書は、Processingのバージョン1.5.1をベースに執筆されています。動作確認はバージョン2.2.1で行っています。
- Processingは以下のURLからダウンロードできます。
  http://processing.org/download/
- 本書のサンプルコードは以下のURLからダウンロードできます。
  http://www.bnn.co.jp/dl/GenArtPE/

■ PDFのダウンロードについて

- 本書購入者は、本書と同内容の、カラーのPDFファイルをダウンロードすることができます。
- 以下のURLにアクセスして「PDFダウンロード」をクリックし、「ユーザー名」および「パスワード」に、巻末の袋とじの中のシリアルキーを入力して「ログイン」ボタンを押してください。
  http://www.bnn.co.jp/dl/GenArtPE/
- ダウンロードは何回でも可能です。
- シリアルキーは書籍ごとに異なります。
- PDFファイルの無断複製、無断配布はご遠慮ください。

■ ご注意

- 本書に記載されたURL、バージョン等は予告なく変更される場合があります。
- サンプルコードを実行した結果については著者や出版社のいずれも一切の責任を負いかねます。ご自身の責任においてご利用ください。
- 本書に記載されている会社名、商品名などは、それぞれ各社の商標または登録商標です。本書中ではTMおよび®マークは省略させていただいております。

序 文 foreword

ここ10年、私たちのデジタルな道具に対する考え方に大きな変化が見られました。今ではそれらを当たり前のものとして利用するだけでなく、私たち自身が、普通の人々からはかけ離れた神経インプラントや仮想現実などなしに、1990年代の千年理論で予言された以上のサイボーグ生物になりつつあります。

その代わりに、私たちはポケットにスマートフォンを入れて、検索エンジン、Wikipedia、ソーシャルネットワーク、高度な地図情報サービスを装備して世界に出て行きます。あまり深く考えなくても、既に私たちは、リアルかつヴァーチャルな世界に同時に生きる、拡張された存在に変容しています。

この革命は、文化的人工物としてのソフトウェアへの新たな理解なくして不可能です。かつてはワードプロセッサをタイプライターの後継物と見なしていましたが、今は大衆娯楽のようにアプリケーションをダウンロードしたり交換しています。かつては商用の情報ばかりで閉じられていたWebサイトでも、今では専門家や愛好家が自分たちのデータをもとに人気の高い「マッシュアップ」をいつでも作ることができるように、APIを公開しています。

クリエイティブな分野での最も重要な発展は、ソフトウェアが単に道具ではなく作品の素材にもなることがわかったことです。インタラクション・デザインやインフォメーション・ビジュアライゼーションのような新しいデザイン分野では、コンピュータの解をデザイン問題に応用する一方で、ジェネラティブ・アートはコードで表現された芸術作品を語る日常的な用語になっています。新世代のエレクトロニック・アーティストは、コードを概念的で形式的な実験のための肥沃な土壌に変え、同時にそれを、計算によるクリエイティビティと制作された作品の理論的な文脈のための実用的な枠組みとしています。

こうした傾向の起源は、クリエイターたちが1990年代半ばに、HTML、Shockwave、Flash、Javaアプレットを、創造的なメディアとして実験に使い始めたところまで遡れるでしょう。iPhoneやAndroidよりも10年以上も前から、World Wide Webはコンピュータ科学者ではなく、クリエイターが計算から生まれたコンテンツを配信するための最初のメディア・プラットフォームでした。しかしWebの「黄金時代」になされた重要な業績にもかかわらず、本当の革

命はProcessing*1のようなオープンソースのツールの導入によってもたらされました。

アーティストたちのためにアーティストによって書かれ、もともとは教育ツールであったProcessingは、リアルタイムグラフィックスやインタラクティブなシステムのようなクリエイティブなアプリケーションのために、Javaの上でシンプルに作られた言語です。それは、初心者でもすぐにコードを使ってスケッチできるよう、通常のプログラミングツールにありがちなつまらない作業を省いています。しかしその単純さにもかかわらず、Processingは最も高度なデジタル・メディア・アプリケーションをサポートすることができる強力なプラットフォームです。ユーザーが提供するライブラリによって、コアとなる枠組みに機能を追加することができ、ポピュラーなマイクロコントローラのArduinoのような他のシステムとも容易に統合できます。最近では、GoogleのAndroid OSのためのアプリケーションを開発できるようになり、Processingはまるで創造的な計算のためのスイス・アーミーナイフのようになっています。

ジェネラティブ・アートの評判が高まる一方で、まだそれは何かしら神秘的であったり、漠然とした数学的な魔法の世界にとどまっています。これらの作品は、どのように作られたのでしょう？　私たちはどのようにコードを使ってスケッチするのでしょう？　コードを書くことについての技術的な側面、言うまでもなく美的な原理を担えるようにアルゴリズムを操ることは、かなりの手際を要します。

視覚的に考える人は、論理的に組み立てられたワークフロー（写真を撮って、操作して、グラフィックな要素と組み合わせ、タイポグラフィを加えるというような）を考えます。コーディングは、最初は視覚的に構成する目的からはまったくかけ離れたもののように思えるかもしれません。わかりにくいステップ（データセットを見い出して、パーサを書いて、境界値を分析し、視覚的に置き換えるアルゴリズムを書いて、パラメータを微調整し、そしてリライトする…というような）を伴っていたりします。コードの要素間の論理的なつながりを確認し、そして無関係に思われるかもしれないルールによって、振る舞いを記述することを要求し、そしてとてもシンプルなアニメーションにさえ三角関数のようなものが入り込むと、初心者はすぐに挫折してしまいます。

幸いにして、ジェネラティブ・アーティストのレパートリーの中でも必須とされる手段の多くは、単純な原則の集まりだけです。マット・ピアソンが言うように、ジェネラティブ・アートは簡単です（少なくとも、時には）。Processingを道具として、マットは基本のドローイングからインタラクティブなアニメーション、セル・オートマトンのような現象のシミュレーションなどのより複雑な話題まで、どのように進んでいけばいいのかを示してくれます。ただ構文を説明するのではなく、むしろ彼は、パラメータの扱い方や、アルゴリズムの調整の仕方によって、劇的に異なる結果が生まれることを見せることで、ジェネラティブなシステムをデザインしていくための創造的な過程を語っていきます。

シンクレアZX Spectrum（1982年に販売されたホームコンピュータ）と共に育ち、そしてプログラマーとして長く働きながらアートに関わってきたことで、マットはまさにコードとクリエイティブな考え方の交点にいます。この本で彼は、読者のみなさんにお勧めの練習課題の道具箱を提供すると同時に、作品に対するもっと深い文化的な状況を紹介していきます。それは読者に、単純なコードのコツの先にある、形の計算モデルの中に込められた、より複雑な思想について考えさせてくれます。私が個人的に気に入っているのは「間違った」やり方でやることについての部分で、線や円のような単純な形ですら、それがどのように作られているのかをクリエイティブに考えることで、複雑なシステムに変化していけることを見せてくれます。

ハッピーコーディング！

マリウス・ワッツ

マリウス・ワッツは、コードを素材として制作を行っているアーティストで、世界中で作品を発表しています。彼は、ジェネラティブ・アートとデザインのためのプラットフォームGenerator.x（http://www.generatorx.no/）の創設者であり、またオスロ建築デザイン大学でインタラクション・デザインの講師をつとめています。彼は現在オスロとニューヨークに本拠地を置いています。
http://mariuswatz.com/

＊1　Processingは、アーティストのために用意された多くのフリーな開発ツールの1つにすぎません。Pure Data（Pd）とvvvvはビデオとサウンド操作が得意な、いわゆるビジュアル・パッチング・ツールです。NodeBoxやScriptographerはグラフィックプログラミングに特化していて、Structure SynthやContext Freeのようなシステムは再帰的な図形文法にもとづいています。

はじめに　preface

先に進む前に、私はここに断り書きをします。私は正しい道を歩もうとして、「本当の」プログラマーになろうとしましたが、結局は失敗してしまいました。私はコンピュータサイエンスの学位を取得しようとしたのですが、わずか1年半でドロップアウトしてしまいました。残念に思ってはいますが、コンピュータサイエンスは私をひどく退屈させたのです。

若者だったので、このキャリアの変化によって私の若い脳の左右の半球のバランスを回復させる必要はまったくありませんでした。むしろそれは、アラン・チューリングとC++について多くを知ることが、ガールフレンドをつかまえる最良の方法ではないんじゃないか、という懸念に関係があったのかもしれません。

1990年代初期のコンピューティングについてのアイデアに関する私の研究は、すっかり私をうんざりさせたので、次の10年間はコンピュータからなるべく遠ざかっていようと思いました。90年代のほとんどの間、私はコンピュータを持ちさえしませんでした。その代わりに、ギター、悪態、無分別なヘアスタイルでいました。Webが始まり、多くのクリエイティブな人々がビデオカメラ、写真、ハイパーテキストを使ってやっていたことが、今やニューメディアと呼ばれて、誰もがやるようになったことに突然気づいた時、私は10年前に引き戻されました。そして、今あなたが「コンピュータを使う」ということが呼吸するのと同じぐらいの意味になるまで、コンピューティングに対するリハビリを続けました。

私がドロップアウトした時には、論理とケーブルの無常な獣たちと日々を過ごすこと以上におぞましいものを想像できませんでした。けれども大人になって、コンピュータ（計算機）とコンピューティング（計算）は同じものではないという当たり前のことに気づいて以来、私はコンピューティングに対する新たな情熱を発見しました。大学の日々の生活の中では、「コンピューティング」はいたるところにあって、それは美しいものになりうるということには気づかなかったのです。

コンピューティングは、水が流れていく道筋を見つけて、やがて大海に流れ込むようなものです。それは惑星が軌道を動いていくようなものです。私たちの身体がまっすぐ立つために、バランスをとるようなものです。私たちのDNAのらせんがほどけていくようなものです。コン

ピューティングは、アイデアを処理してテキストとしてアウトプットするという、今私がやっているようなことです。そして、あなたがこの文章を読んで、結果として自分の考えを生み出すという、あなたの脳が今やっているようなことでもあります。

コンピュータにはいまだにうんざりするけれど、コンピューティングはかっこいいと矛盾なく言うことができます。コンピュータがほんとうに使いものになるのは、自然に匹敵するような計算をシミュレートしたり、何か新しいものを作り出そうとする時です。「ジェネラティブ・アート」が、そこに私を導いたのです。

職業的なプログラマーとして、私はいつもできる限りジェネラティブなアイデアでやってみました。何か新しいキットを手に入れた時にいつも最初にまずやってみるのは、フラクタルを作ってその限界を試してみることでした。けれど私は、それをアートだとは思っていなくて、ただそれをやっているアーティストたちの活動の広がりにぼんやりと気づいていただけでした。しかし21世紀になると、その動きはますます盛んになって目立ち始めました。そのためのツールもますます強力になり、入手しやすくなってきました。

私は、もしも何かをやりたいと思った時の最良の方法は、断念することなく、ただひたすら続けることだと信じてきました。報酬がなくても、やること自体が楽しければ、妨げるものは何もありません。経験上、もし十分に長い間何かを続け、そのための意欲が持続したなら、遅かれ早かれ誰かがあなたにお金を払ってくれるでしょう。あるいはあなたに本を書くように頼むでしょう。この原則は、自虐や連続殺人には適用されませんが、たいていの芸術的な努力については真実です。

あなたが今（物理的に、あるいは仮想的に）抱えているこの本の起源も、2008年にこの原則に従って開始したプロジェクトだと言えます。私は、もしジェネラティブ・アートを真剣にやっていくならば、ジェネラティブ・アートのブログを始めるべきだと考えました。私はそれを「100の捨てられたアート作品（100 Abandoned Artworks）」と呼んで、ジェネラティブ・アートを毎週作るという仕事を自分に課しました。私の遊び時間に実生活が入り込む前に、それまでやっていたどのような状態のもの（すなわち、捨てられたもの）でも、そこに放り入れてみました。私はクリエイティブ・コモンズのライセンスでソースコードを公開したので、誰もが私が捨てた半完成の作品を手にすることができ、また利用することができます。私自身も、100に達するまでやめないぞと誓いました。この自らに課した厳密なスケジュールは、私自身のものごとの優先順位を意識的に再編しました。かつては、暇をみつけて漫画を読んだり、だらだらとひどい映画を見て何時間も過ごすことがありましたが、しかしジェネラティブ・アートという楽しみを知った後には、そんな暇などありませんでした。

この自制はうまく効きました。スケジュール管理が容易になっただけでなく、Processing（私が選んだツール）への意欲がますます膨らみました。プロジェクトは、印刷やビデオなどさまざまな方向に発展し、私の日々の仕事にフィードバックするようになりました。マニング社のクリスティーナ・ルドルフさんが本の出版の可能性について相談するために連絡をくれたのは、作品が50くらいに達した頃でした。

この本をもって、プロジェクトは完結します。この本は、私がどこにいるのか、これまでのプログラミングのキャリアの中で学んだことや、学ばなかったことすべてのスナップショットです。私はしかし、Processingの本を書くつもりはありませんでした。これまで丸一冊読んだこともないようなプログラミングの本を書くのは、さらにいやでした。もっと感動させるような何か、「どうやって（how）」ということと同様に、「なぜ（why）」というようなことについて書きたかったのです。アートをプログラミングすることは、システムをプログラミングすることとは違った訓練が必要で、強力なツールを自由に使うのに、正しい、あるいは誤った方法などというものがあってはなりません。コーディングというものは、単に構造化され、整然としているだけでなく、それがいかに心を解き放って創造的にしてくれるか、また技術屋だけでなく、もっと多くの人々の手に届きうるものかを知って欲しいのです。この目的が成功したかどうかは、みなさんに決めてもらいましょう。

謝 辞 acknowledgments

貢献していただいたこと、実現可能にしてくれたこと、感じ良くいてくれたこと、プロジェクトをサポートしれくれたこと、くだらないことにつきあってくれたこと、原稿をレビューしてくれたこと、フィードバックをくれたこと、適切な時に適切なことを言ってくれたことに対し、次の方々に多大な感謝を送ります。

Alan Sutcliffe, Rob “SanchoTheFat” O’Rourke, Justin “Soulwire” Windle, Seb Lee Delisle, J4mie Matthews, Ron Wheedon, Robert Hodgin, Aaron Koblin, Jeremy Thorp, Jared Tarbell, Casey Reas, Ben Fry, Manfred Mohr, Reza Ali, Jerome “01010101” Saint-Clair, Dan Shiffman, Shardcore, Norman Klein, Kerry Mitchell, Orhan Alkan, Patrick Steger, Andy Dingley
マニング社：Marjan Bace, Emily Macel, Christina Rudloff, Jeff Bleiel, Mary Piergies, Tiffany Taylor, BarbaraMirecki, Irene Korol Scala,（そして私と接する不運を持った他の方々）
Philip Galanter, Kevin “lomokev” Meredith, Marc Banks, Alice Eldridge, Rich “Text Format” Willis, Rifa Bhunnoo, Chris TT, Ewan Swain, Eric Bates, Iestyn Lloyd, Alec Morrison, atdotBrighton社のみなさん，RuthとJo, さらにFutureDeluxeと（私が執筆していた間、私にデスクスペースを使わせてくれた）TGSiでの私の友人たち

マリウス・ワッツのアートワークと序文、Frederik “w:blut” Vanhoutteのアートワークとテクニカルレビュー、アラン・ワッツの引用に関してクリフ・ピックフォードに、特別な感謝を表したい。

また忘れてしまったかもしれない方々に謝罪も。

そして何よりも、途方もないオタク（massive nerd）である私にもかかわらず、どうにか愛してくれている辛抱強い妻デボラに感謝したい。

本書について about this book

本書から、あなたはジェネラティブ・アートで実験を始める方法を学びます。ツールとアルゴリズムとクリエイティブなアプローチを探り、アイデアを得て、さらにそれをもっと発展させる方法を探求します。けれども、この本から学べないことが1つあります。それは、「アーティスト」になる方法です。

もしもこの本が、みなさんに役に立つレシピを集めただけのものだったり、ある種の美的に面白い結果を生み出すだけのものであるなら、それは私の真意を失してしているだけでなく、傲慢だと言えるでしょう。アートの評価は完全に主観的であり、もし私がアート制作の正しい方法があると宣言したとしたら、平手打ちを喰らうでしょう。

同様に、これは「デザイン」についての本ではありません。デザインは、それを見る誰にでも同じ反応をもたらすビジュアルを生み出すことを意図していて、その意図（「ストリートっぽい」とか「レトロである」とか「洗練されている」だとか）は、たいてい明確でなければなりません。アートでは、何らかの反応を生み出すこと目的にはしていますが、ひとりひとりの反応が異なっていてもたいした問題にはなりません。ある人が冷笑を浴びせたとしても、別のある人が作品を大好きになってくれれば、それは素晴らしいことです。さらに言えば、もしある人が作品を好きになってくれたら、もう一方の人が逆に嫌いになることを望んだりします。もし私たちがこのような極端な反応を生み出すことができたら、成功の1つの基準となるでしょう。おそらく、アートにとっての唯一の大罪は、退屈なことです。

本書中の図版のほとんどは、とりあげているトピックと関連した私のジェネラティブな作品の図ですが、それ以外のものは、息抜きとしてランダムに添えられています。これらのイメージのソースコードの多くは、http://abandonedart.orgと出版社のWebサイトhttp://www.manning.com/GenerativeArtからダウンロードできます。これらもまたクリエイティブ・コモンズ・ライセンスのもとに公開されているので、あなたがこれら作品を利用し、分析し、変更し、ごちゃまぜにしたり、壊したりすることも大歓迎です。また重要なことは、あなた方にはそれらをがらくたであると考える権利もあることです。もしそうであっても、私はそんなに困りません（それが本書の目的ではありませんが）。

たとえあなたが本書のすべての作品を嫌いであったとしても、何かに役に立つインスピレーションか何かしらをこの本から得ることを願っています。極端な反応を引き起こすことができれば、だめなアートも良いものになります。まったく反応をもたらさないのが、アーティストとして失敗することです。

もし私がこの本で理解して欲しいことを1つあげるとしたら、これが何か特定の種類の結果を得るためのルール本ではなく、プログラミングのスタイル本であるということです。私は、どうやったらプログラミングが自分自身に芸術的な感動を引き起してくれるのか、つまりプログラムを書くことでクリエイティブになるために、頭を自由にする方法を探っていきたいのです。

このアプローチのための道筋は以下の通りです。本書の最初のパート「クリエイティブ・コーディング」では、舞台を設定し、事前に不可欠な必要条件を教えます。Chapter 1は、ジェネラティブ・アートのコンセプトをいくつかの異なった角度から見ます。それからChapter 2で、本書の主要なツールとなるシンプルなプログラミング言語Processingを使えるようにします。

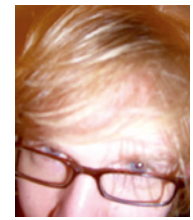
Part 2「ランダム性とノイズ」では、Processingを使ったジェネラティブな視覚表現を作るための私のプロセスを分析し、最も単純な形状のアルゴリズミックな創作を見ながら、新しいアプローチの方法について考えます。Chapter 3では、線の描き方から始めます。これを流体的なアプローチによって、面白い空間へと展開します。それはさらにChapter 4の三角関数の考え方へと拡がっていきます。Chapter 5では、アニメーションと3D表現を探究します。

本書の最後のパート「複雑性」では、機械の限界を超えて、自然界が教えてくれる3つの有機的なプロセス――創発（emergence）、自律性（autonomy）、フラクタル（fractals）――を見ていきます。これらの現象をいかにしてコードの中でシミュレートできるか実践的なアプローチで探り、この実験に必要な最新のオブジェクト指向プログラミングを秘密裏に学んでいきます。

ジェネラティブ・アーティストにとって、正しい方法とか間違った方法とかはありません。規則もレシピもありません。ジェネラティブ・アートは、有機的なもの、創発的なもの、美しいもの、不正確なもの、予測不可能なことに関するものです。それらを論理と正確な仕組みの世界の中で探求することに関するものです。この楽しい逆説によって、コンピューティングは禅に近づいていきます。遊びに満ちていて、有機的で、抑制から自由で、自然の流れを招き入れます。

さらにシンプルに言えば、ジェネラティブ・アートはコーディングを楽しむためのものです。プログラミングは、良い実践、構造、効率のためだけのものではありません。それは自由、創造力、表現についてのものでもあります。プログラミング言語は、ちゃんとした人の手にたどり着けば、深遠なものも陳腐なものも作れるアートのための道具になるのです。

著者について about the author



マット・ピアソンはクリエイティブなプログラマーで、時には著者にもなります。ずっとフリーランスでやってきて、ロンドンの国立肖像画美術館、クリーブランド美術館、パース芸術祭、BBC、ビクトリア＆アルバート美術館、ロンドン、シカゴ美術館、バンクーバー人類学博物館、パラマウント映画、英国政府と仕事をしてきました。彼は英国のブライトンでブロンドの子供たちとMacBookのコレクションと、おそらくウォルヴァーハンプトンが生んだ最も美しい女性と一緒に暮らしています。彼のひっきりなしのおしゃべりをhttp://zenbullets.comで読むことができます。

オンラインフォーラム

マニング社によって運営されているAuthor Online forum（http://www.manning.com/GenerativeArt）で、マットにコンタクトが取れます。マニング社は読者の皆様に、読者同士、そして読者と著者と有意義な会話が行われる場所を設けています。著者の方々は、自主的に（そして無報酬で）このフォーラムに参加してくれています。この本が出版されている限り、出版社のWebサイトからAuthor Online forumと、これまでのディスカッションのアーカイブを見ることができます。

イントロダクション：
有機的なものと機械的なもの

introduction: the organic vs. the mechanical

> タオイストの哲学によれば、自然の形は、作られるのではなく成長するものであり、有機的なものと機械的なものには、根源的な違いがあります。
>
> 家、家具、機械のような作られたものは、部品を寄せ集めて組み立てられたものか、あるいは外側から内側へと、彫刻のようにかたちづくられたものです。
>
> しかし、内から外へと自分で成長していくものは、そもそもが個別の部品の寄せ集めではなく、自らを分割しながら、全体から部分へ、単純なものから複雑なものへと、自らの構造を精緻に作りあげます。
>
> アラン・ワッツ, 1958

アラン・ワッツ（1915–73）。イギリスの哲学者にして禅僧。まさに1960年代的な意味で仏教徒でした。彼は神学のマスターであり、司祭であり、禅についての20冊以上の本の著者でもありました。個人レベルと研究室の両方で、幻覚剤を使った実験もしました。彼はクリエイティビティとテクノロジーに関することについて多くを語ってきましたが、私が知る限り、ジェネラティブ・アートについては特に何も言っていません。

上記の引用では、有機的なものと機械的なものの作り方の違いによる、自然界と人工物の不一致について語っています。形状が「内から外へ」とかたちづくられていく、というこの有機的な成長の概念は、本書のトピックをかなりうまく述べています。けれどもこの明確な2国間主義（バイラテラリズム）の中で、私たちはどのようにしたらコンピュータプログラミングの作品が、機械的なものでなく有機的な世界に属すると言えるのでしょうか？

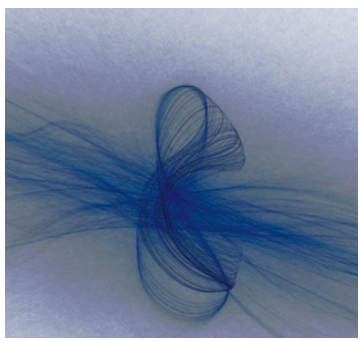
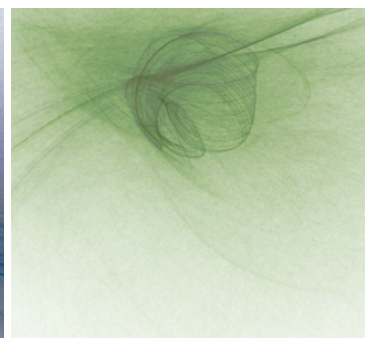
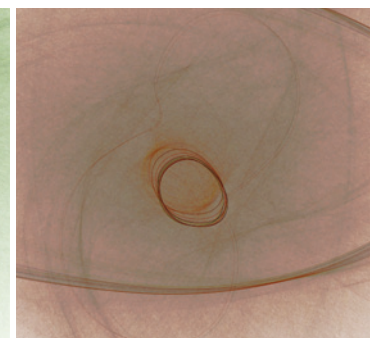
ジェネラティブ・アートは、そもそもはプログラミングでもアートでもありません。それはこれらの両方であり、どちらでもありません。プログラミングは、人と機械の間のインターフェースです。それは明確に定義されていて、クリーンで、論理的な実践です。アートは感情的な対象であり、大いに主観的で、定義に挑むものです。ジェネラティブ・アートは、これら2つが出会うところです。厳密で、冷徹で、論理的なプロセスの実践をひっくりかえして、非論理的で、予測不可能で、表現豊かな成果にする行為です。

ジェネラティブ・アートは、プランや材料や道具を使って組み上げるものではありません。あたかも花や樹木のように、育てられるものです。しかし、その種子は土壌や水分ではなく、ロジックとエレクトロニクスです。論理的な決定と数学によるプロセスの、最もシンプルで創発的な生成物です。ジェネラティブ・アートは、機械を使って有機体を作るようなものです。

ワッツの言う、世界の中で対立するカテゴリーである機械的なものと有機的なものは、容易に見分けられます。建物はまっすぐなエッジと鋭いコーナーを持っています。機能的で、正確です。それは機械的な世界の領域にあります。樹木は不規則で時間的に変わりやすく、葉は風に揺れて秋に落葉します。それは有機的なものの領域にあります。機械的なものは組み立てられ、ワッツが言うように、外から作られます。それらは作られて、描かれて、組み立てられて、彫られて、生産されます。他方、有機的なものは育てられます。それらは自己構造化していて、全体論的です。その形状は意図なしに生まれます。デザインや青写真に従ってはいません。

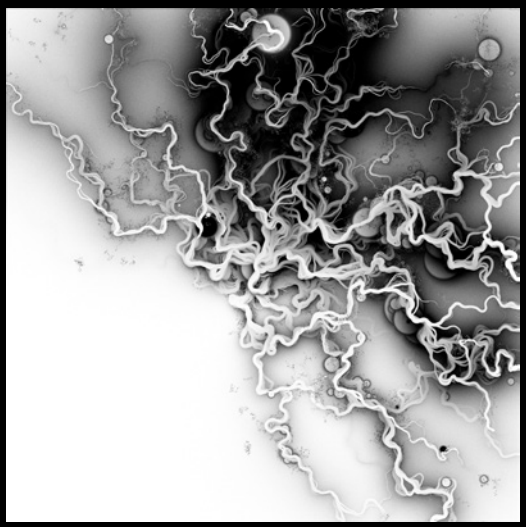
造園師のように、多くのジェネラティブ・アーティストは、自然に発展していく現象を使って、それらを美的に楽しめるものに作りあげます。自然界の美しい気ままさと、私たち類人猿の脳が望む秩序の間のバランスを見い出します。放置されて、繁茂し過ぎた庭は、混沌とし過ぎていて私たちを不快にします。それに対してコンクリートで固められた場所は、整然として秩序づけられてはいますが、すべての美しさを取り去ってしまいます。

芝生がよく手入れされていて、均等にハサミが入れられ、葉は2つとして同じでないけれど、完璧にシンクロしています。花々の色は正確で厳密ではないけれど、均等にバランスがとれています。スイートスポットはこうした2つのものの狭間にあり、そこが「アート」が生きているところです。

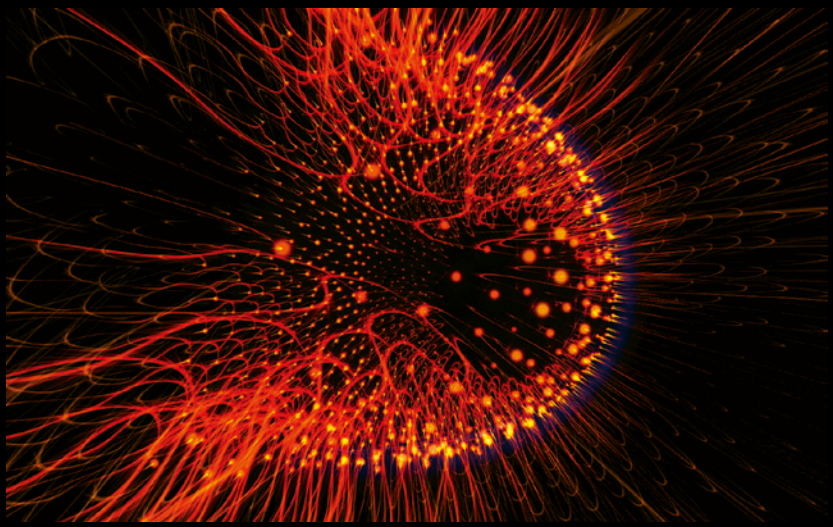




図i.1、i.2、i.3
Opiamas Trangelo (2010)、マット・ピアソン。1つのアルゴリズムによって作られた多様な作品。これ以降、作家名が明記されていない場合、その作品は著者のものだと思ってください。みなさんは、図のキャプションを見て、それがどのように章に関連しているかを考えてしまいがちかもしれません。けれども、テキストが途切れないように、図をただ入れるだけのこともあるので、ご了解ください。

図i.4
Addition/Subtraction, Variant (2010)、ロバート・ホジン。
その作品が強いインスピレーションをもたらし続けるアーティスト。
i.5とi.6も参照。



図i.5
Magnetic Ink（2007）、ロバート・ホジン。
この作品は群れのアルゴリズムを使って制作されたもので、各エージェントがインクの霧を残しています。
詳細はhttp://www.flight404.com/blog/?p=86をご覧ください。
群れについてはChapter 6でさらに学びます。



図i.6
Jelly（Magnetosphere）、ロバート・ホジン。このイメージを含め、
ロバートの初期の作品の大部分が、これから学ぶツールであるProcessingで制作されています。

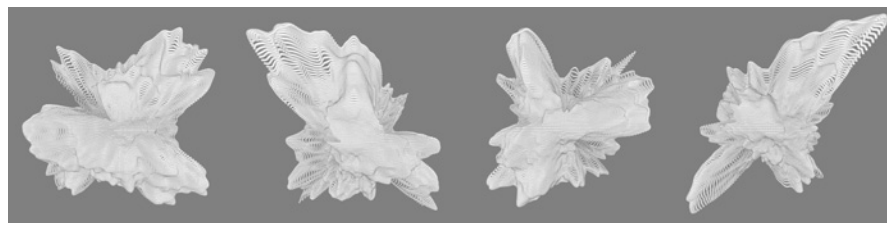
ジェネラティブ・アートは簡単

もしあなたがアーティストになることに決めたなら、あなたは1つだけやることがあります。名前を書いて、その後に「アーティスト」と書き、それを名刺にプリントしてください。おめでとう。あなたはアーティストです。もしあなたがそう選択するなら、この時点から、すべての落書き、仕草、あなたがする発言、あるいは動きが、アートと定義されます。人生の便器に出すどのウンコだって、誰かがどこかで、その意義を見つけるものです。たとえそれがあなただけだったとしても。あなたは特に優れたアーティストではないかもしれませんが、しかしアーティストというのは、あなたそのもののことなのです。

「ジェネラティブ・アーティスト」になるにはもう少し固有のことが必要ですが、しかしそれはそう困難なことではありません。あなたは、何らかの一揃いの技術がこの肩書きには必要とされると思っているかもしれせん。プログラミング言語の習得が、いちばんの障壁だと考えているかもしれません。そんなことはありません。言語は障壁ではなく、むしろ近道です。スキルが作品制作には必要不可欠だと信じている他のアートに比べれば、ジェネラティブ・アートではたいていのスキルを学ぶ必要はなく、それらは既にツールの中に含まれています。

絵画を描いたり、素描したり、彫刻するには、たくさんの年月がかかりますが、ジェネラティブ・アートのプロフェッショナルな結果を得るためのプログラミング能力は、数日で習得できるでしょう。もしあなたが私の言うことを信じられないなら、この本でそれを証明してみたいと思います。図i.7のイメージはたった24行のコードで生成されていて、次のページのリストi.1で見ることができます。実行されるたびに、いつも異なる静止画を生成します。

このコードが行っているのは、2つのループを用いたグリッドによる反復です。14行目の関数呼び出しでそれぞれのグリッド上の点に円を描き、それを数学的に変化させて3D空間内に置き換えています。このコード、あるいは前の文章がわからなかったとしてもご心配なく。この本はまさにそのためにあるのですから。最初のいくつかの章で、これらすべてのコンセプトを



図i.7　リストi.1の24行のコードによって作成された4つのジェネラティブ・アート作品。

紐解いていきましょう。

リストi.1　24行のコードによるジェネラティブ・システム

```
void setup() {
  size(2000, 2000, P3D);
  background(150);
  stroke(0, 50);
  fill(255, 200);
  float xstart = random(10);
  float ynoise = random(10);
  translate(width/2, height/2, 0);
  for (float y = -(height/8); y <= (height/8); y+=3) {
    ynoise += 0.02;
    float xnoise = xstart;
    for (float x = -(width/8);x <= (width/8); x+=3) {
      xnoise += 0.02;
      drawPoint(x, y, noise(xnoise, ynoise));
    }
  }
}

void drawPoint(float x, float y, float noiseFactor) {
  pushMatrix();
  translate(x * noiseFactor * 4, y * noiseFactor * 4, -y);
  float edgeSize = noiseFactor * 26;
  ellipse(0, 0, edgeSize, edgeSize);
  popMatrix();
}
```

たとえあなたがこれまで一度もコードを書いたことがなかったとしても、Chapter 2の終わりまでに強力なプログラミング言語を身につけているでしょう。さらにChapter 3以降で、ジェネラティブ・アートに関するたくさんのテクニックを習熟できます。この技術をマスターできるかはどうかは、あなた次第です。

もし、プログラミングの技能に恵まれているならば、あなたは既にそこまで来ています。実際のところ、経験豊かなプログラマーにとっての大きな問題は、慣れ親しんだ厳密なやり方を忘れることかもしれません。カオスは電子機器にとって、起こってほしくなかったり歓迎できないものですが、これからの百数十ページでは、カオスは私たちの友人であり協力者です。秩序とカオスは互いに相反するものかもしれませんが、それらが一緒に機能しないということはありません。

図i.8
パース国際芸術祭のブランディング（2010）。
プリント／ビデオ／オンラインのジェネラティブ・ブランディング・プロジェクトからのイメージ。
ブライトンのFutureDeluxe（http://futuredeluxe.co.uk）との共同制作。
全容はhttp://vimeo.com/20096257で。

秩序とカオス

図i.9のイメージは、1795年にウィリアム・ブレイクが描いたニュートンです。絵の左側には、明るく着色された動植物、自然界の複雑さがあります。右側には、秩序、幾何学の精度、コンパスがあります。中央部、2つの不調和な要素の間に、男が座っています。

この男はただの男ではなく、『自然哲学の数学的諸原理』（古典力学創始の書）の著者であるアイザック・ニュートンです。ブレイクの絵は、ニュートンの世界観への批判です。彼は自然界の美しさに背を向けています。彼の唯一の興味は、巻物とコンパスにあります。

ウィリアム・ブレイクは、大文字のRで書くRomantic、つまりロマン派の画家です。彼は18世紀後半の啓蒙運動に対する芸術的な、文学的な、そして知的な反応であるロマン主義運動の味方でした。啓蒙主義の時代は近代科学の誕生の前触れとなった時で、純粋理性が哲学の支配的傾向でした。ロマン派は不信心な世界の訪れを恐れ、審美的で、地方的で、絵のような美しさをもつ自然の牧歌的な理念の消え行く残影に執着していました。彼らは自分たちの先にある未来、つまり現在私たちが住んでいる合理的で科学的な世界を見てはいましたが、それに反発していました。

図i.9
Newton、ウィリアム・ブレイク。自然のカオスと、科学と数学による正確さの間の、不似合いを描いた美しい絵です。
William Blake（1757–1827）、Newton、1795年頃、インクと水彩による版画、テイト・ギャラリー、ロンドン

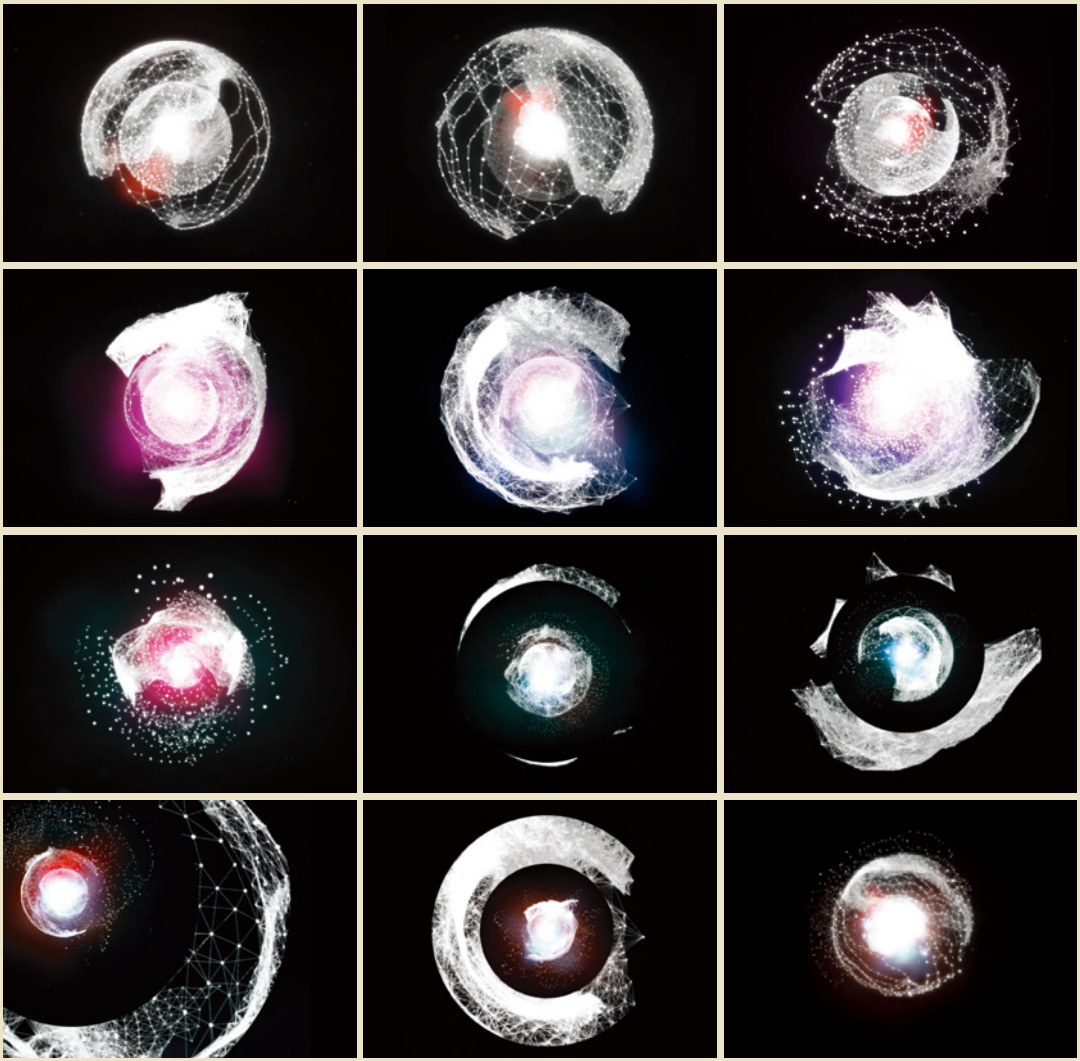
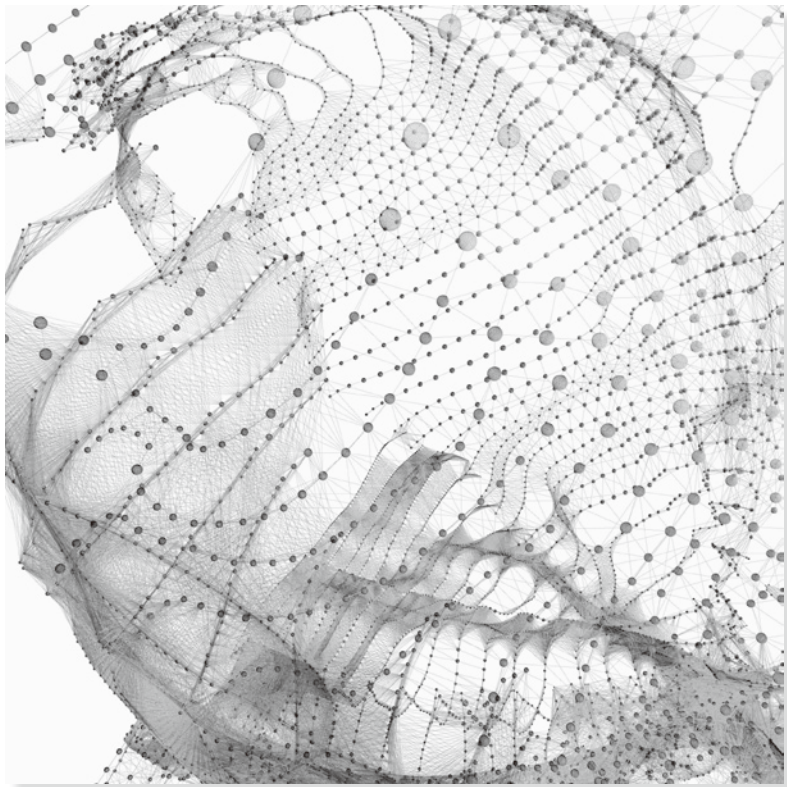


図 i.10

Life in 2050 (2010)。私が映画祭のために作った告知映像。
これも FutureDeluxe (http://futuredeluxe.co.uk) との共同作業。
パーリンノイズと、解体された球を使って作られており、制作プロセスは Chapter 5 で述べています。
完成した映像は http://vimeo.com/10924639 にあります。

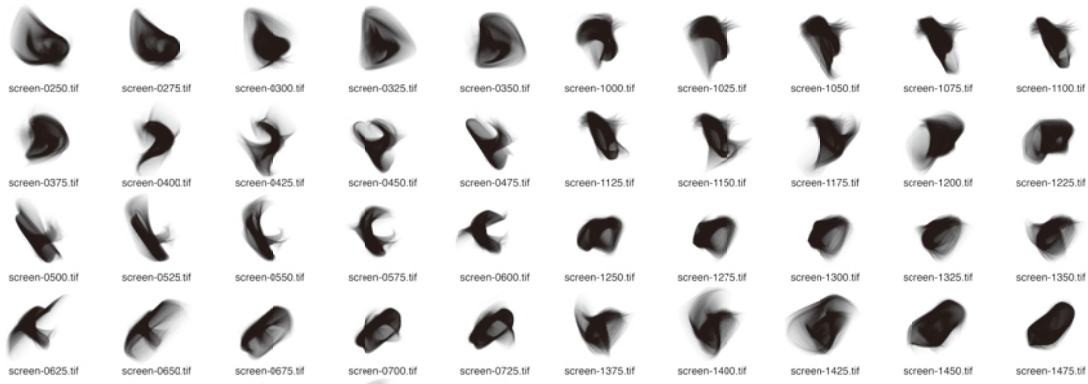


図i.11
Frosti(2010)。図i.10のビデオ作品を作ったのと同じシステムを印刷のために再利用しました。

現代のコンピュータプログラマーたちは、ニュートンと同じポーズをしているかもしれません。彼らはスクリーンの前で魅了され、薄暗いオフィスで鮮やかなモニターを目を細めて見ながら、マウスと手、キーボードと指を微細に動かし、スクリーンの向こう、窓の外、町の外にはほとんど気を向けずに仕事の日々をこなしています。自然界のカオスは、この論理の世界では歓迎されません。予測不可能性はコンピュータに期待するものではないのです。

けれどもジェネラティブ・アーティストは、クリスティーン・キーラー*1のように、その岩にまたがって反対の方向を向いているプログラマーです。ジェネラティブ・アーティストは論理の世界からやってきて、インスピレーションのために自然界を見ています。

＊1　もしあなたが歳をとりすぎて（あるいは英国人しすぎて）いなければ、ジョン・プロヒューモとクリスティーン・キーラーのスキャンダラスな話を覚えているでしょう。Wikipediaで“defiant straddle”（敢然と股にかける）が実際にどのようなものだったかを見てみましょう。http://ja.wikipedia.org/wiki/プロヒューモ事件



図i.12
Twill (2010) のコンタクトシート。本書[原書]の表紙のイメージは、ジェネラティブに実行した数百のイメージの中から選ばれました。

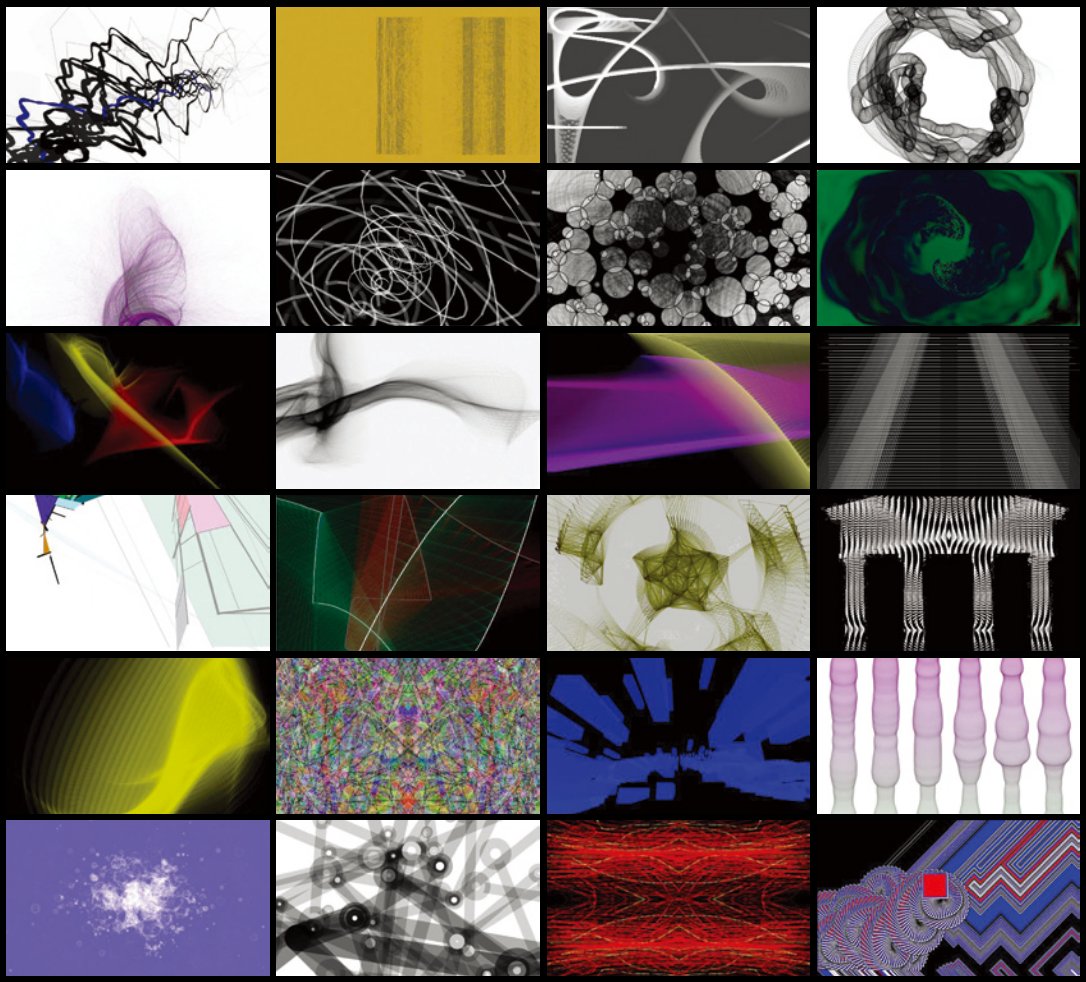
有機体の領域を探るのにコンピュータを使用することには、本質的な矛盾があるのではないでしょうか？　ブンブンいうエレクトロニクスの箱の中の1と0ほど、機械的なものはないんじゃないでしょうか？　ジェネラティブ・アートは、無駄足以外の何ものでもないのでしょうか？

さらに、もし私たちが自分の作品をアートだと宣言するのなら、感情と美学を探求しようとすることが当然必要とされるのではないでしょうか？　私たちは、手順、論理、数学だけを使って、共感を生む作品を作ることができるものでしょうか？　コンピュータに頼ってアート作品を作り、見る人たちと人間レベルで結びつけようとすることは、無数の猿たちが無数のシンセサイザーでパーフェクトなポップソングを書くのを待つのも同然です。

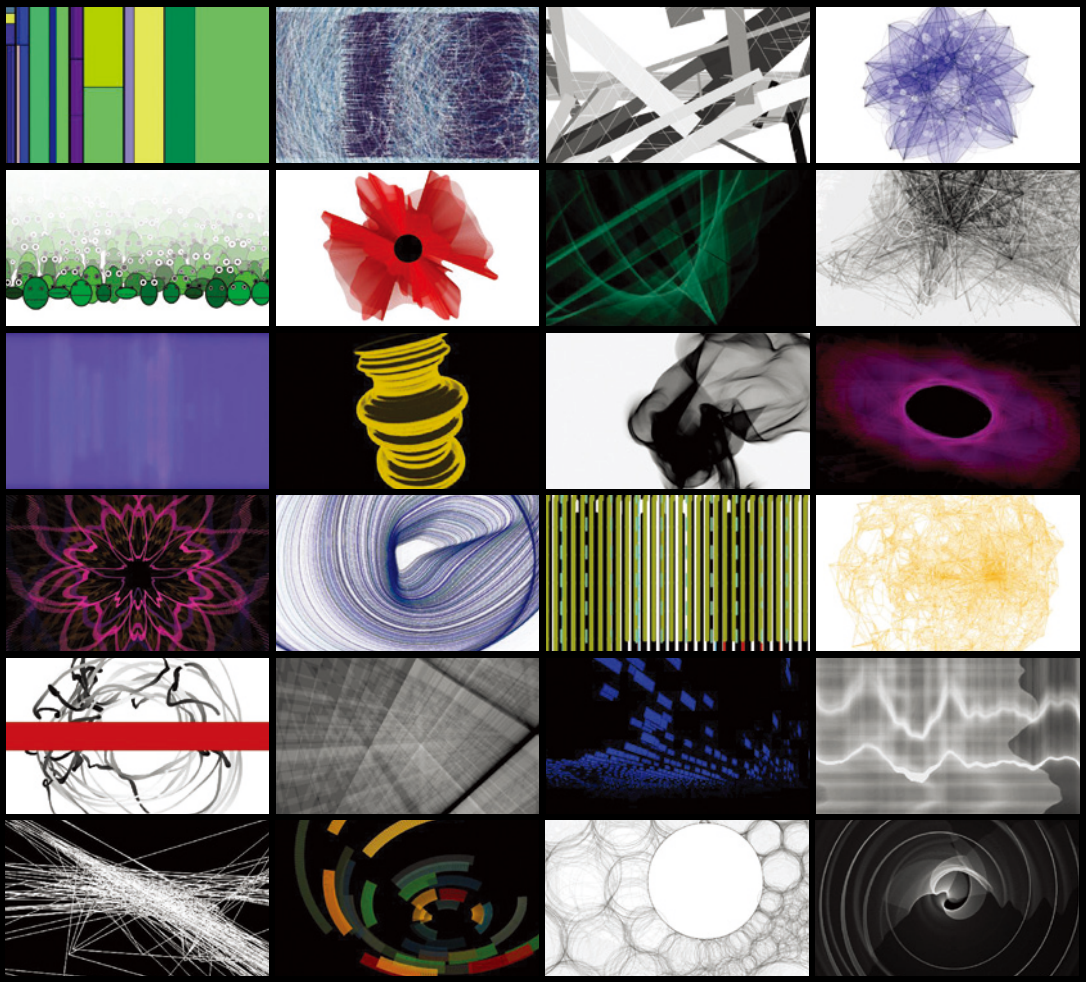
ここに挑戦があります。

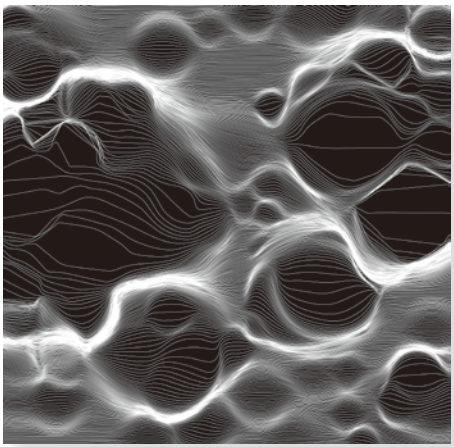
秩序とカオス、単純さと複雑さ、機械的なものと有機的なものは、必ずしもスペクトルの両端にありません。それらは共生的で、からみあっているのです。これら2つの間で私たちが進む進路は、ナイフの刃先のようなものです。私たちの存在そのものは、エントロピーと秩序の間、はむかう乱気流、混沌とした環境、助けなしには生存できない自然界、最も純粋な単純さ、動物としての私たちに必要な喪失感の中をさまよっています。私たちは、カオスに打ち勝ち、純粋な秩序の退屈さと狂気を征服してきたと思っているかもしれません。しかし、あらゆる生きる存在は、これら2つの間に絶え間ない変動があることを示しています。あなたが自分の生活にどれくらいの秩序を課しているかにかかわらず、カオスは自動車衝突事故ほど遠いものではありません。

図i.13
Process 14 / Image 4、C. E. B. Reas (2008)。ケイシー・リースはProcessingプロジェクトの創始者の1人。
このプリントは、ニューヨークのビットフォームス・ギャラリーで
2008年春に展示されたProcessingの作品です。



図i.14
100 Abandoned Artworks (2008–2010)。2年の間、毎週おこなったジェネラティブ・アニメーションの実験。ソースコードはhttp://abandonedart.orgにあります。本書の他の作品の多くのコードも、同じくそこで見ることができます。





図i.15
Grid Distortion 02D 0018、
マリウス・ワッツによるレーザーエッチング（2010）。

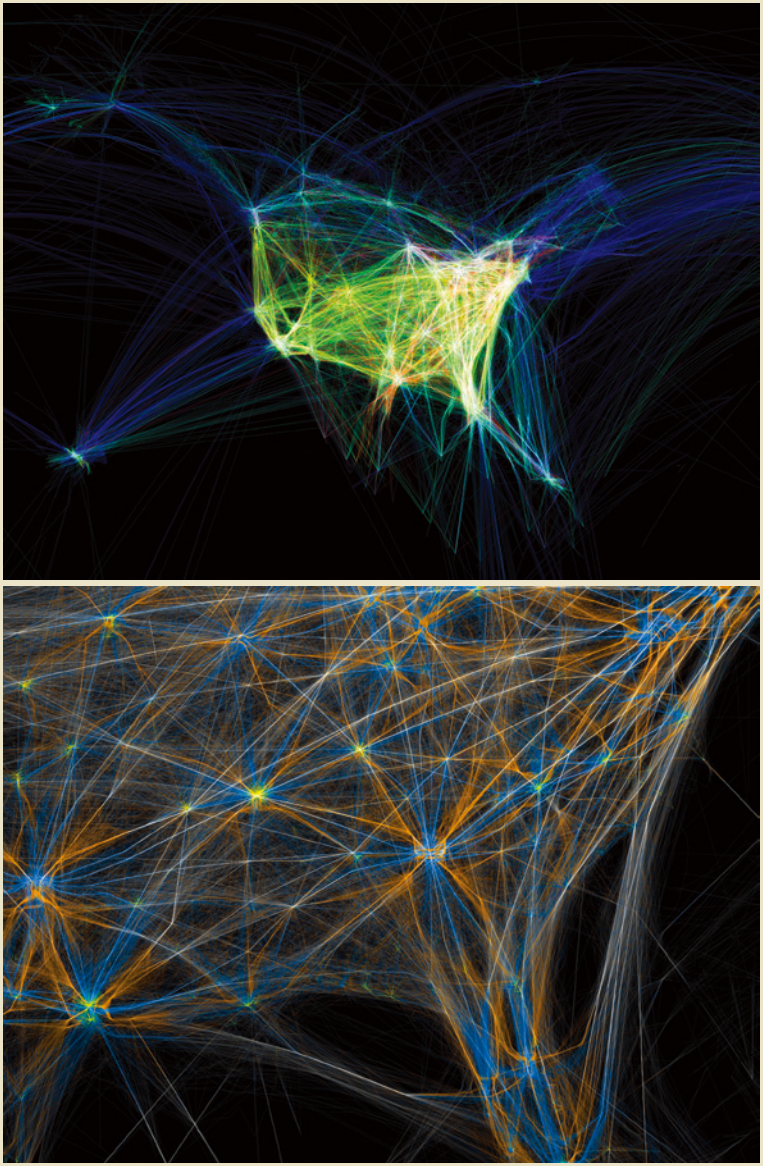
図i.16
KBGD01E 0012、マリウス・ワッツ（2010）。

機械的なものと有機的なものは、秩序とカオスのように共依存（一方は他方がなくして存在できないこと）しています。複雑なものは、単純なものと同じように私たちにアピールします。有機的なものと機械的なものも同様です。流行や雰囲気は、どんな状況でも二者択一的に私たちを揺さぶるかもしれませんが、私たちはその一方あるいは他方を完全に選ぶことはできません。それは生きるのをやめることと同じです。カオスを根絶するためにはロボットにならなければならず、秩序を根絶するには未開人になるしかありません。

幸いなことに、誰も私たちに選択を迫りません。ジェネラティブ・アートの目的は――もしそれが目的を持っているとしたら――、何かを美しくすることです。私たちは、秩序から始めてカオスを目指し、どちらかの方向に迷い込まないよう気をつけながら、機械的なものを使って有機的なものを作ることを試みます。そしてもし私たちが達人になれたら、自分自身をプログラマーとアーティストの両者であるとみなすことができるでしょう。

図i.17

Illuminations B、マリウス・ワッツ（2007）。
本書の序文を書いてくれたマリウスは、
私が多大なインスピレーションを受けたジェネラティブ・アーティストです。
図i.15とi.16にもう2つ作品があります。
彼の他の作品とプロセスはwww.unlekker.netで見ることができます。

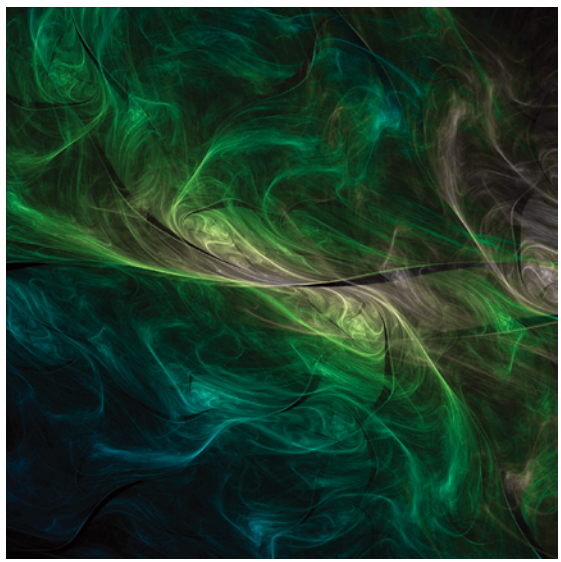


図i.18
アーロン・コブリンのFlight Patterns (2005) からのレンダリングで、
Chapter 7で論じるデータドリブンの作品です。

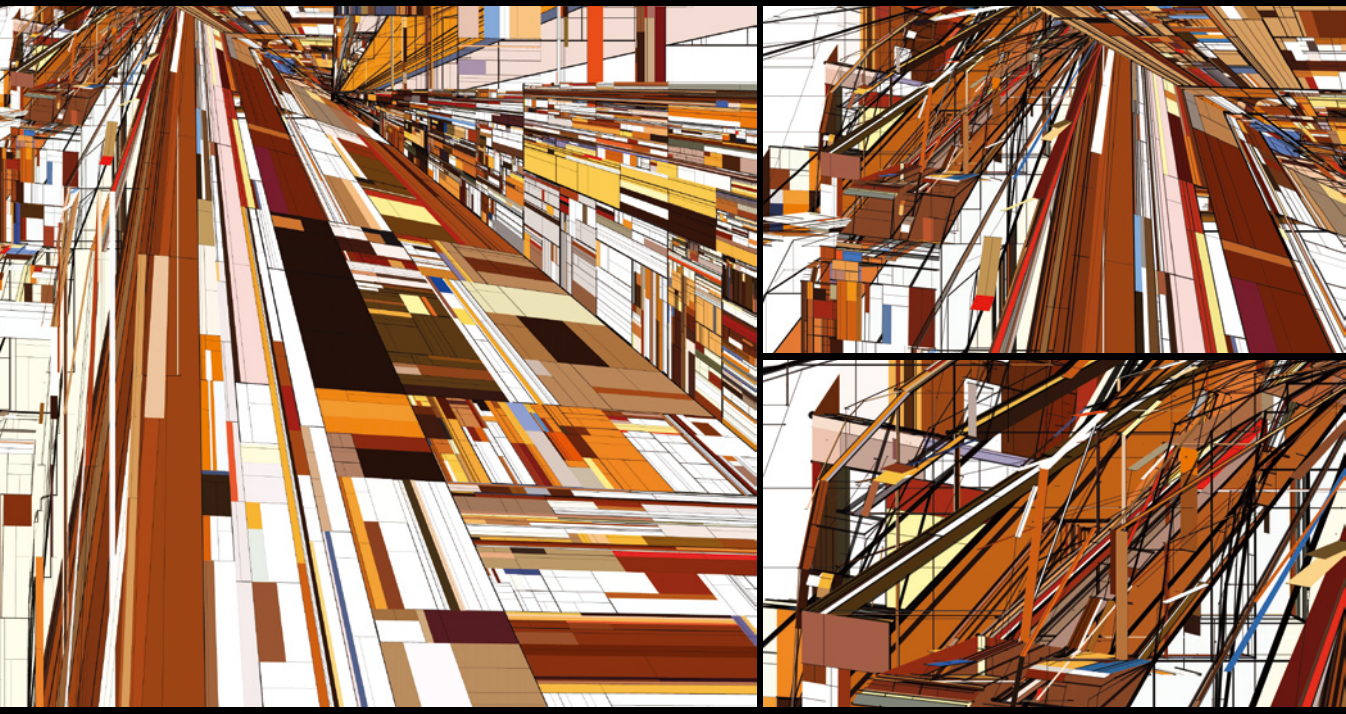
詩としてのプログラミング

プログラミング言語をアーティスティックなツールとして使用するのは、フォークリフトでバレエを踊ったりT定規で曲線を描くようなことで、反直感的だと考える人もいます。伝統的に、プログラミング言語は論理、構造、組織化のためのものだったからです。プログラムは、問題解決、データのモデリング、正確なシミュレーションのために使われています。こうした暗黙の論理的なやり方が、クリエイティブに考える人にとっては必然的にコーディングを魅力のないものにしています。これは馬鹿げたことです。プログラミング言語はただのツールにすぎません。どこかのコミュニティに属しているものではありません。それは、デザイナーにとっても、システム設計者にとっても、等しく有効なものです。しかしそれぞれのツールユーザーたちによって作られた作品は、根本的に異なります。

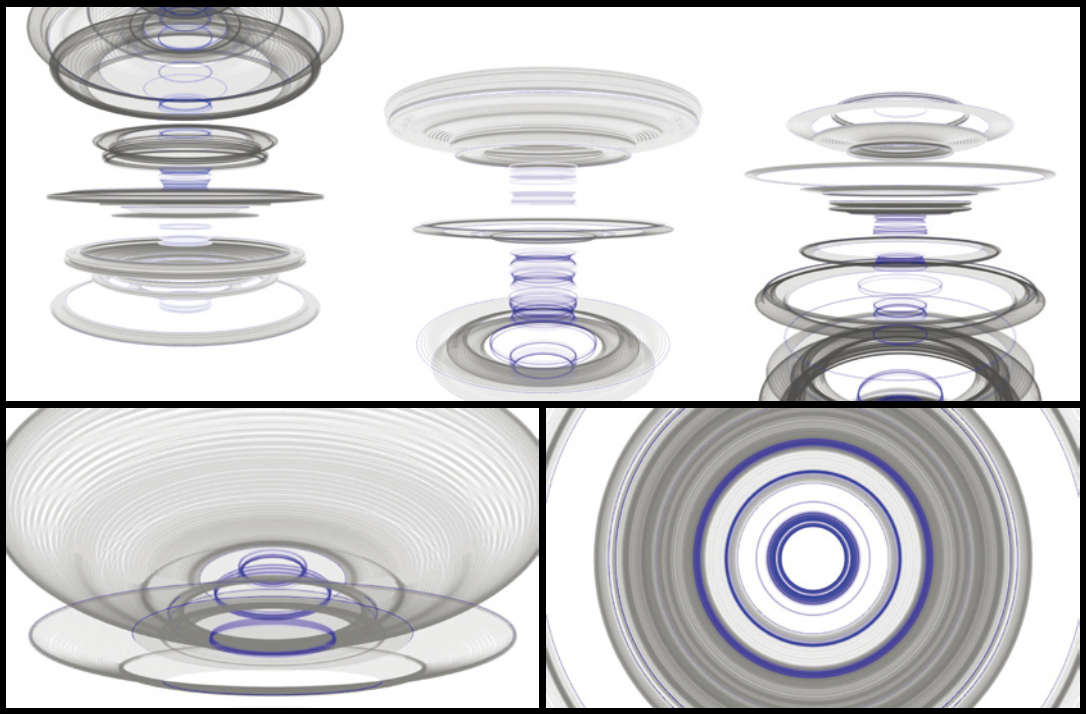
プログラミングの歴史は、常により優れて洗練された組織化へと向かってきました。コンピュータ科学者がデザインパターンについて話をする時、それらは美学を意味してはいません。プログラミング専門用語では、「デザインパターン」とは、プログラミングの課題に対して均質的に取り組むために有用な枠組みのことです。より普遍的なプログラミング言語を用いることで、共通の問題を同じように取り組めるようにすることを目的にしています。試用されテストされたデザインパターンの使用と理解は、典型的な良いプログラマーの証ですが、それらはまた、あまりに厳密な言語の使用方法でもあります。私たちがジェネラティブ・アートでやって



図i.19
Strange Symmetry 2。
フレデリック・バンフート（2008）。



図i.20
Mondrian Architecture (2009)。ピエト・モンドリアンに触発されたフラクタル。フラクタル構造を作る喜びは、ディテールを探ることにあります。それは本当に多くのレベルで作用しています。それぞれの細部が、前のイメージの左上の隅へとズームインしていきます。これらのアイデアについてはChapter 8で説明します。



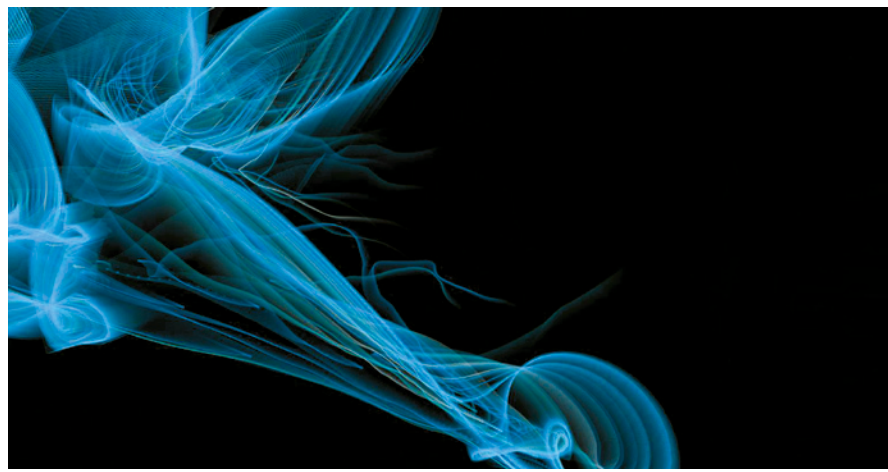
図i.21
Orbitals（2009）。3Dの円にパーリンノイズを適用。
この作品にある必要なすべての原理はChapter 4と5で論じます。

いるのは、この流れに逆らって泳ぐことと、何をするにも正しい方法とか間違った方法というものはないということです。私たちは、こうした素晴らしい経験を利用することができるし、そこから多くを学ぶことができますが、しかし、私たち自身がそれらによって邪魔されてはなりません。もしルールが存在したとすれば、それらは破られるべきものです。

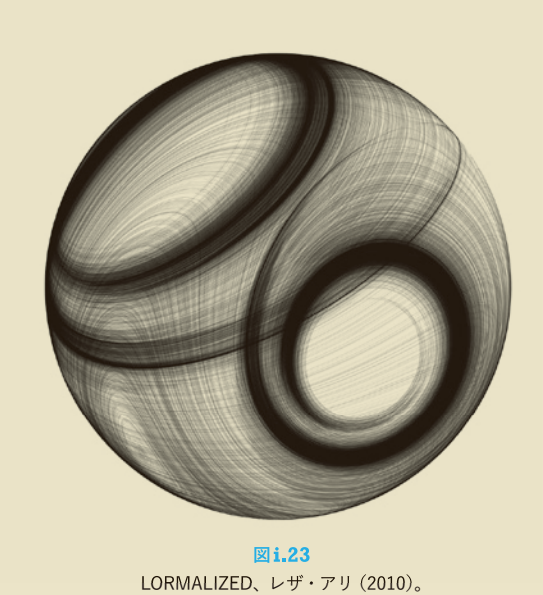
しかしこれは残念ながら、プログラミングの文法についてのことではありません。セミコロンをミスすることについて、因習打破も何もありません。そうしても、それはただコードを壊して終わるだけです。これは不適切なリアリティです。

プログラミング言語は、つまるところ1つの言語にすぎません。そして言語は、いろいろなアクセントや語形変化など、多くの異なった方法で話されます。プログラミング言語は、人と人の間ではなく、人と機械の間のコミュニケーションを促進するためのものである、という点でだけ、英語、ドイツ語、フランス語などのような自然言語と違っています。人と機械のコミュニケーションに必要な明瞭さのために、私たちがJavaやC++のような言語を使って、詩を書くことは想像できません。けれどももし言語に詩を作る能力がないとすれば、それは明らかに、方程式の人間側の重要さを欠いていることになります。

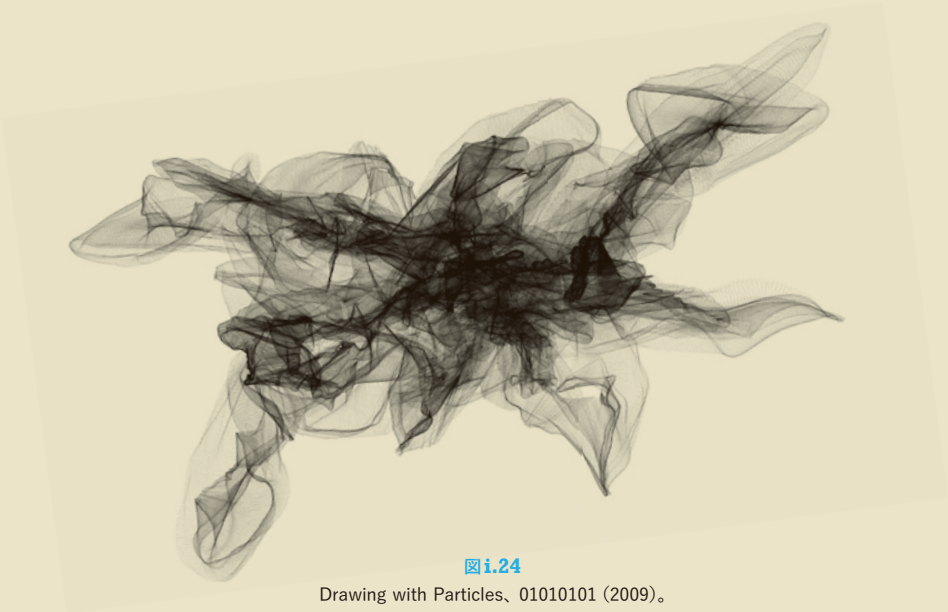
私は、コードは詩でありえると確信しています。そしてコードは楽しいものにもなります。しかし、私たちはそれを可能にするために、専門的でコマーシャルなプログラミングの厳格さ、良い設計、ベストプラクティスを、いくらか犠牲にしなければならないかもしれません。



図i.22
Broken Mirrors 1–4 (2011)。画廊での私のインスタレーションです。
作品を見る観客の動きから作った4つのジェネラティブ・アニメーションの1つ。



図i.23
LORMALIZED、レザ・アリ（2010）。
アリはアプリケーションとしてのこの作品を発表しました。
そして他の人たちも彼のシステムを使って、自分のイメージを作ることができるようになりました。
www.syedrezaali.com からダウンロードできます。



図i.24
Drawing with Particles、01010101（2009）。
01010101として知られるジェローム・サンクレアは、マウスの動きでこのイメージを作りました。
しかしマウスがコントロールしたツールは、01010101自身が作った
ジェネラティブなパーティクル・システムでした。

図 i.25

Happy Place、ジャレッド・ターベル (2006)。
ミクロレベルでのエージェント指向の行動が、創発的な複雑さをマクロレベルで作り出します。
もしこれらの言葉の意味がぱっとわからなくても、Chapter 6 と 7 で説明しています。

カオス芸術家

私たちの作品を「アート」と宣言するのは、大胆で横柄なものです。そうすることで、私たちの作品は単なる有益性を超えていて、私たちの人間性と個性の表現だと言えます。言葉では言い表せない領域にあるかもしれません。残念ながら、それは私たちのうぬぼれの表現かもしれませんが、しかし今のところそのことについては心配しなくても大丈夫です。もしあなたが、人生に特別な目的や測れる価値がなければ意味がないという意見ならば、あなたは間違った本を選択しています。

私たちの作品を「ジェネラティブな」アートだと述べることは、単に私たちの個性を表現するものでなく、カオスと、コントロールなしのプロセスの奔放さを表現することでもあるのです。この予測不可能性なしには、私たちのアートは「ジェネラティブ」とラベルされた箱の外に分類されてしまうでしょう。ジェネラティブ・アーティストはカオス芸術家です。彼らは予想できないものを繁殖させ、歓迎し、制御し、そして好ましい形式にあつらえることができます。

ありふれた、あるいは栽培種の（本書のPart3で簡単にふれる数学的な類のものとは反対の）カオスは、論理学者には呪いのように思えるかもしれません。予測不可能性は、伝統的なプログラミングの慣習では望まれないものです。それが、より経験豊かなプログラマーが本書の文章を読んだ時に、そこでの考え方のいくらかを不愉快に思う理由です。でも、私を信頼してください。それはあなた自身のためでもあります。なじみ深いものに対する非線形のアプローチは、脳を健康にします。そして、ひょっとしたら、これらのクリエイティブな見方のいくらかをプログラマとしての仕事に反映させることができるかもしれませんし、逆にあなた自身を上司や同僚に対してひどく不評にするかもしれません。

カオスは恐れるようなものではありません。結局のところ、私たちはコンピュータで魅力的なイメージを作ること以上に危険なことはしていないのですから。まじめな話、起きうる最悪のものは何でしょう？　ジェネラティブなプロセスに携わることは、予測不可能性を期待することを意味しています。予測不可能性は歓迎されない客ではありません。私たちは、自分の作品を制御しないことに対して満足するようになるでしょう。私たちはこのカオスを受け入れて、それを愛することを学ぶのです。

Part 1 クリエイティブ・コーディング

“機械が音楽を作りました。
でもその機械は私が作ったのです……
こうなると、私はその責任が
どこにあるのかわかりません。”

オウテカのショーン・ブースのインタビュー, 2010

Chapter 1

ジェネラティブ・アート：理論と実践

Generative Art: In Theory and Practice

まず初めに、「ジェネラティブ・アートって何？」という質問はひどく単純に思われるかもしれません。けれどもあらゆるものの性質はジェネラティブなので、その定義さえ創発的複雑さを持っています。

この何年か、フィリップ・ガランター（アーティストでテキサスA＆M大学の教授）の2003年の論文「ジェネラティブ・アートとは何か？　芸術理論の文脈としての複雑系理論」*1が、たびたび引き合いに出されてきました。そこにはこう書かれています。「ジェネラティブ・アートは、アーティストが、自然言語の文法、コンピュータプログラム、機械、その他の手続き的な創意を用いて、芸術作品に対して何らかの自律性を働かせたり、その結果を生み出すようなシステム一式を用いて行う、芸術的な実践を意味している」。

これは正確かつ記述的で、正しい言葉を使った長い文章であるけれど、簡単に言えば、これでは十分ではありません。ジェネラティブ・アート（GenArt）の本質を捉えているとは思えません。それはもっと不明瞭なものです。私の考えでは、ジェネラティブ・アートは、光とピクセルのバレエで表されるような、自然な調和をひねりだそうとするカオスと秩序の力の間の、永遠で巨大な戦いの副産物にすぎません。けれどもそのような美辞麗句は、私たちには役に立たないでしょう。

私たちがこのトピックに踏み入る際には注意しなければなりません。なぜなら、放っておいたほうがいい論議である「アートとは何か？」という定義づけを、どうやってでも避けたいですから。アートの概念は非常に脆くて、曖昧で、私たちがつっつきまわすと消えてしまうようなものです。そこで、含蓄のある表現で問題を分析するのではなく、以下に続く章では、私はより繊細かつ鈍感な方法でいきたいと思います。まずは、ジェネラティブ・アートではないものを吟味することから始めましょう。

＊1　ガランター教授の論文は、www.philipgalanter.com/downloads/ga2003_paper.pdf で読むことができます。

1.1　私たちの芸術様式

例えば彫刻、絵画、映画といった、より伝統的な芸術様式では、アーティストは素材を完成作品にするためにツールを使います。これは確かに難しい作業です。ジェネラティブ・アートでは、自律システムがそうした大変な作業をやってくれます。アーティストは、初期条件を与えて、システムに指示するだけです。

制作プロセスにおけるアーティストの役割は、クリエイターよりもキュレーターに近いかもしれません。あなたはシステムを作り、雛形とし、育て、磨きをかけます。けれども最終的に制作された作品は、子供の仕事であって、その親の誇り以上のものはないかもしれません。

これはもちろん、極めて不公平な見方です。私たちは、共同作業における人間の役割を過小評価すべきではありません。プログラミングに加えて、人はもうひとつ重要なスキルである美的判断を担います。コンピュータが美的センスを磨くこと（進化システムの分野では、この目的で多くの仕事が行われてきました*2）は可能ですが、しかしそれは人間と機械の創造的なパートナーシップにおける、決して良い分担ではないでしょう。もし百万の桁まで円周率を計算する必要があるとしたら、人間の脳をこの仕事に専念させるのは才能の濫用です。同様に、機械に何が美しくて何がそうではないかを決めることを委ねてはいけません。

ジェネラティブ・アートは、その性質上ほとんど常に抽象的ですが、作品のスタイルによって定義することはできません。ジェネラティブな作品の共通要素は、最終結果のスタイルではなく、その作品の制作の方法論です。

例えば図1.1は抽象的なモノクロームの作品です。しかしどのように作られたかについては、私としては、ただジェネラティブな作品であると言えるだけです。この作品のピクセルの配置は、写真、あるいは紙と鉛筆とスキャナーを使って、またAdobe社のIllustratorでもできます（私のへたくそな手でなければ、とあわてて付け加えますが）。そしてどうであれ、これはモノクロの抽象です。こうしたツールが使用された方法によってはジェネラティブであるかもしれませんが、しかしこれは既定のものではありません。

制作に使ったツールは決定的な要因にはなりません。あくまでそれは、共通項をもたらすための方法にすぎません。この本では、プログラミング言語Processingがツールに選ばれています。

＊2　リチャード・ドーキンスの本『盲目の時計職人』（1986）が、遺伝的アルゴリズムとその応用についてのわかりやすい入門書です。

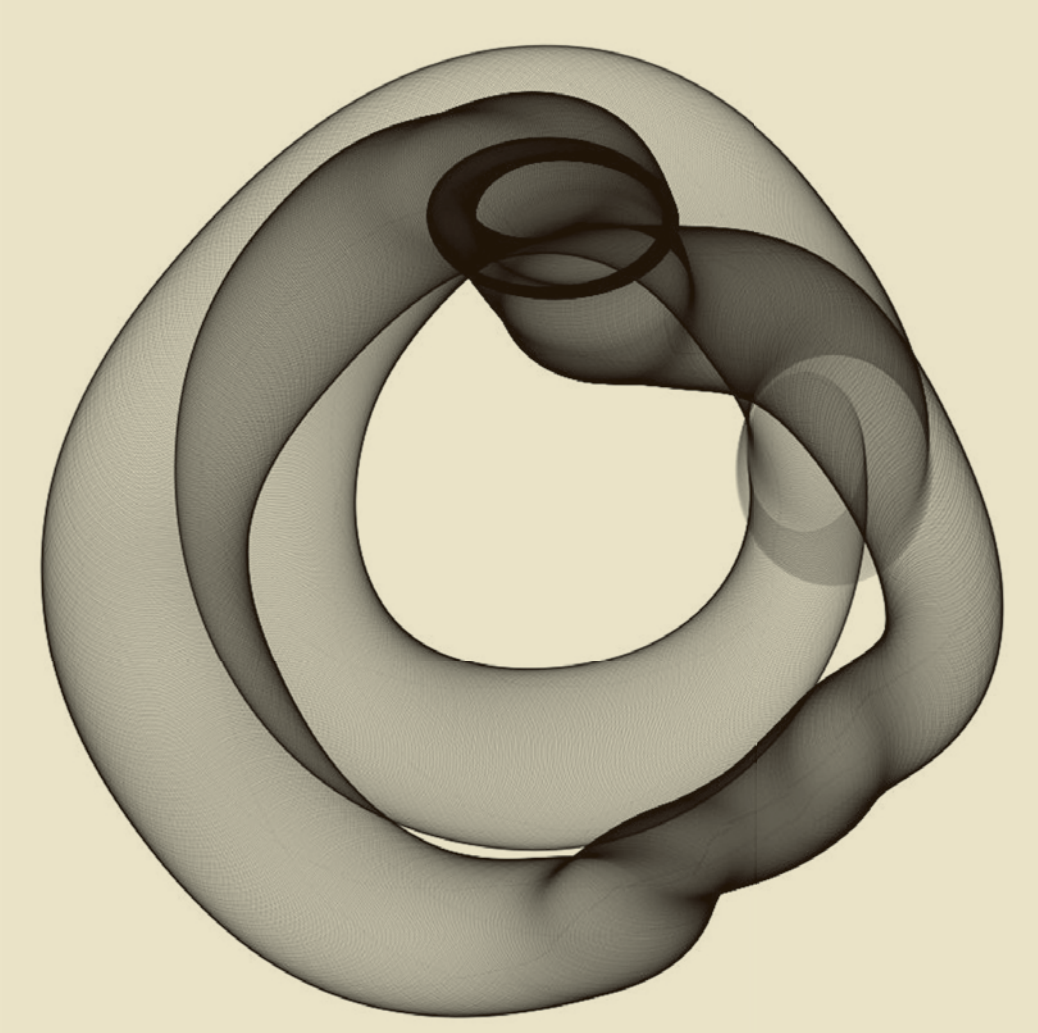


図1.1
Tube Clock (2009)

しかしそれは、そのツールで作ったものがジェネラティブであることを意味しません。プログラミング言語は、ただコンピュータを言われた通りに動かす方法にすぎません。そもそも、命令に従うだけではジェネラティブではありません。

方法論を「ジェネラティブ」と呼ぶには、まず最初の鉄則として自律性が必ず含まれていなければなりません。アーティストは通常、まずランダムあるいはセミランダムな要素を含んでいる基本のルールや定式を作ります。そしてそれから、作品を作るための自律したプロセスを開始します。システムは、完全にアーティストの制御下にあるわけではありません。さもなければ、アーティスト自身が唯一のジェネラティブな要素です。第二の鉄則は、したがって、ある程度の予測不可能性がなければならないということです。アーティスト自身が、他の誰かと同じように、その結果に驚くようでなければなりません。

ジェネラティブなアート作品を作ることは、たとえアーティストが一人で制作したとしても、常にコラボレーションなのです。あらゆるジェネラティブ作品の著作権は、アーティストが使っているメカニズム（それを生成しているシステム）に、必ずその一部が帰属しています。幸いなことに、無節操な芸術家のパートナーがすべての成果を横取りしたとしても、匿名で自律したシステムはそれを気にしません。

意図を明確にしておくため、この本ではただ1つのツール――プログラミング言語――について論じ、結果としてはビジュアルだけを制作します。しかしジェネラティブな方法は、音楽、建築、詩、ダンス、物語、インタラクティブな体験を作り出すためにも使えます。そしてそれらの創作の背後にある自律したシステムは同様に、機械、偶然によるゲーム、自然現象、潜在意識による人間の行動であったりします。こうしたさまざまなアプローチについては、本書の後半で触れたいと思います。

1.2　新しい考え方の歴史

ジェネラティブ・アートの歴史は、他のアートと比較して長くなく、まだ数十年程度です。ですから、ジェネラティブ・アートはまだ、アートの世界の周縁にいます。世界中の美術大学が毎年何万人もの画家、陶芸家、ファッションデザイナー、グラフィックデザイナーを送り出しています。しかし世界の現役ジェネラティブ・アーティストの数は、おそらくカリブ海を航海する豪華客船の一隻にゆったりと乗れるくらいでしょう（もし誰かこれを手配してくれたら素晴らしいのですが）。けれどもこの数は急速に増え始めています。コンピュータの技術が加速し、一般化したため、クリエイティブな人たちがどんどんツールを手に入れて、この斬新な芸術様式

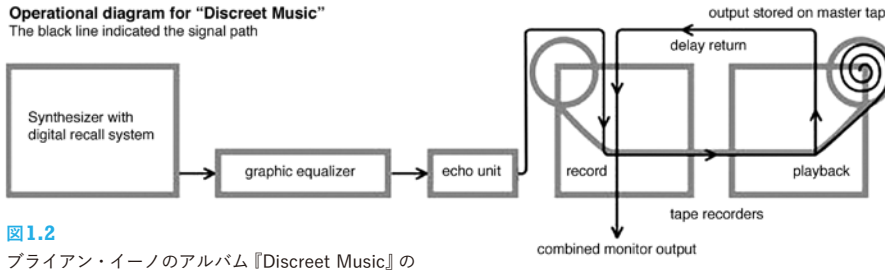


図1.2
ブライアン・イーノのアルバム『Discreet Music』の
スリーブの中の、音楽制作に使われたシンセサイザーと
テープループ・フィードバックシステムを説明するダイアグラム。

を発見し始めました。

「ジェネラティブ・アート」という用語は1960年代から一般的に使われていたのですが、その概念はそれより前からありました。例えば、音楽のジェネラティブな形式はモーツァルトから存在していました。彼の「Musikalisches Würfelspiel（音楽のサイコロ遊び）」はジェネラティブ・アート的なシステムの初期の1つの例です。

そのアイデアとは、既に書いておいた部分をサイコロの目に従ってカット＆ペーストして、メヌエットを作るというものでした。1つの六面サイコロであっても、5回転がせば7,776個、6回転がせば46,656個の組み合わせになって、可能な組み合わせの数は急速に増えていきます。こうしたタイプの芸術的な室内ゲームは、18世紀に人気が高まりました。

前世紀には、ジョン・ケージ、カールハインツ・シュトックハウゼン、ブライアン・イーノたちがジェネラティブ・ミュージックの考えを拡げました*3。ジョン・ケージの「4'33"」（4分33秒）は、ただその長さによって定義されたことで物議をかもした、音符なしの作品です。環境的なアンビエントサウンドを音楽の内容としたもので、作品の演奏が2回と同じになりません。

後に、シュトックハウゼンとイーノ（そして他の人たち）は、音楽が一連のルールや条件によって定義される、という作曲の手順を使った実験を行いました。イーノのLP『Discreet Music』（1975）はその素晴らしい例です。A面は、テープループ・フィードバックシステムによって作られた30分の作品です。シンセサイザーで作られたメロディーはテープレコーダーに録音さ

＊3　Processingはジェネラティブな音楽を作るためにも使うことができますが、この本ではビジュアルの作品のみに焦点を合わせます。ジェネラティブ・ミュージックは大きくて魅力的なトピックで、それだけ書かれた本があっていいでしょう。デイヴィット・トゥープの『Haunted Weather』がその入門に良いかもしれません。

れ、そのアウトプットが2番目のテープレコーダーに送られます。2番目の機械のアウトプットが最初の機械に戻されて、音が上書きされて録音されます。イーノがLPの裏表紙にこのセットアップのダイアグラムを掲載したので、私たちはこのプロセスを知ることができました（図1.2参照）。

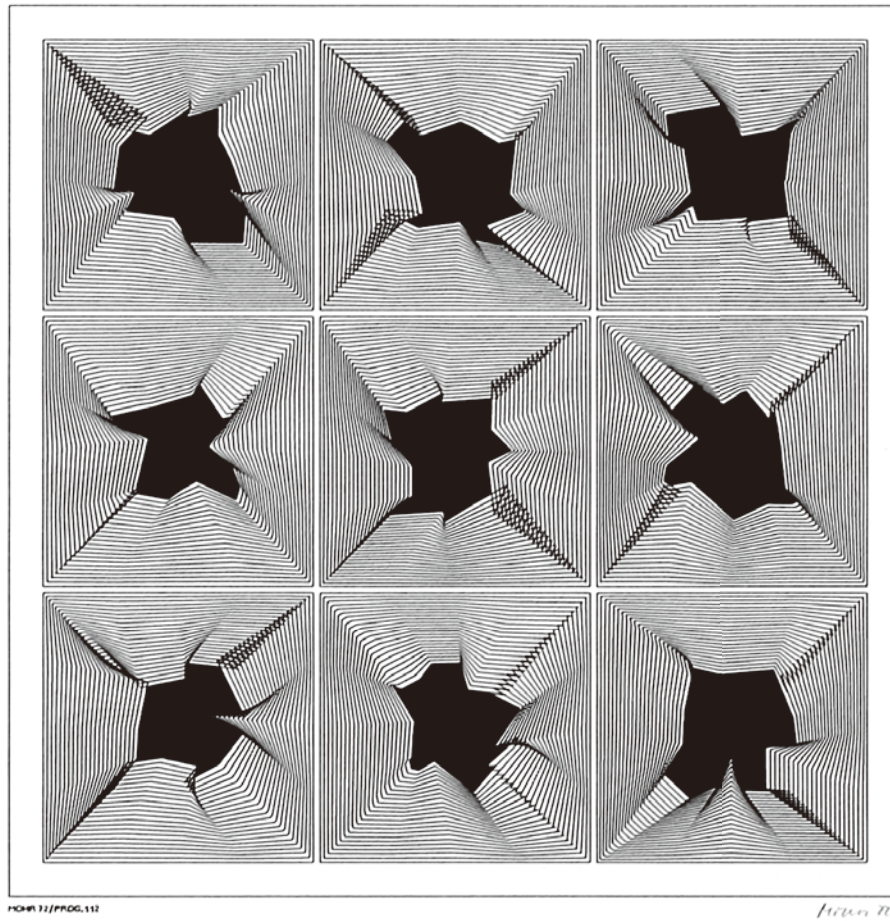


図1.3
Lady Quark、マンフレッド・モール（1972年）。
最も初期のアルゴリスト（コンピュータで制作するジェネラティブ・アーティスト）の一人。
モールは自己のアートを探究するために、自分でソフトウェアを書きました。

視覚的なジェネラティブ・アートは、まずは1960年代にプロッタによる出力、次にディスプレイ画面で、その後は、さらに洗練されてきた印刷とビデオで登場しました。初期のプロッタ時代のパイオニアとしては、フリーダー・ナーケ、ジョージ・ニース、ヴェラ・モルナール、ポール・ブラウン、そしてコンピュータによって生み出した「Artificiata I」という作品集を1969年に出版したマンフレッド・モール（図1.3参照）があげられます。

ジェネラティブ・アートの発展は、コンピュータの進展と密接に結びついていましたが、コンピュータはただ有用で便利な道具にすぎません。ジェネラティブ・アートの本当のツールは、あなたが使っている種々のツールの基礎となっているもの――すなわち「アルゴリズム」です。アルゴリズムは自然界の一部です。それは、メディアを超えた普遍性を持っています。ジェネラティブ・アートを作成するためのシステムは、テクノロジーと共に進化して、ずっと変化してはきましたが、アルゴリズムは同じままです。ジェネラティブ・アートはアートそのものほど古くはありませんが、少なくとも数学ぐらいに古いと言えるかもしれません。

本書は、そのアルゴリズムについての本です。哲学、美学、実験についての本でもありますが、しかし主として探究するのは、アルゴリズムと、あなたが使うツールについてです。理屈では、チョークと大きな平らな岩を使っても、手続き的なジェネラティブ・アートの背後にある数学を行うことができます。しかしそれは次の節で見るような、この21世紀に入ってますます進化する素晴らしいデジタルツールに囲まれて暮らしている私たちにとっては、逆に生活を不必要に難しくしてしまうものでしょう。

1.3　デジタルなツールセット

あなたがデジタルメディアに囲まれた実践家であるなら（今日、そうでないことの方が難しいですが）、おそらく既にクリエイティブな研鑽のための、お気に入りのツールセット（ソフトウェア一式）を持っているでしょう。あなたは、ある特定のワードプロセッサや電子メールクライアントを使っていて、言葉をあやつる必要を感じた時の手段とします。また、ピクセルを使って何か言いたい時のために、グラフィックパッケージの使い方を時間をかけて学んできたかもしれません。もっと基本的なレベルでは、あなたはMP3プレイヤー、電話、ラップトップなどのデバイスとの（おそらく依存）関係を身につけてきました。そしてあなたのニーズを満たすために、それらをうまく使いこなそうとします。ソフトウェアもハードウェアも、それらはあなたのデジタル道具箱の中のツールです。けれども、これらの人間－機械間の密接な結びつきは、どのくらいの期間続くのでしょう？

1.3.1　絶え間ないはかなさ

急速な技術革新の時代（現在ももちろんそうですが、同様に人類史のどの時代もそうです）、私たちは最新のツールにフォーカスを当ててきました。私たちは、それらのクセをマスターすることに多くの時間を投資し、しょっちゅうアップグレードしています。

Appleのラップトップは、CPUの性能や使いやすさのレベルがますます高くなるように、毎年アップデートされています。AdobeのPhotoshopのようなアプリケーションは、平均して18ヵ月ごとに新しいバージョンがリリースされます（しかも毎回かなりの金額がかかります）。彫刻刀や絵筆、三角定規を、2年で古くなると知りながら買うようなことは想像できませんが、こうしたあっという間に陳腐化してしまうことは、ハードやソフトの最先端では疑いもなく受け入れられています。

暗黙の不文律が、ユーザーとテクノロジー製造者の間に存在しています。製造者は、毎年同じ製品を去年購入した同じ客に届けられれば大繁盛しますが、これは市場の要求によってのみ可能となります。ユーザーは熱心に製品のアップグレードを待ち、そしてあらゆるささいな繰り返しに価値を見い出しています。それらは最新の技術を提供しなければならない、という欲望によって促進され、そのために必要な犠牲を払う用意も整えています。こうして、技術を前進させる絶え間ないはかなさの繰り返しが引き起こされます。

デジタルアート作品を作る時、あなたはこのことを心に留めておかなければなりません。もしあなたがピクセルで仕事をするなら、その仕事にはもろさがあります。あなたの作品は、モニタが消される瞬間に消滅してしまいます。そのため、多くのアーティストがプリントや彫刻のような、触れることのできる芸術様式に引き寄せられます（3Dプリンタの価格の低下は、これからの10年、さわれるジェネラティブ・アートのエキサイティングな可能性を約束しています）。さらに、もしあなたが作品を制作するために機械を用いるなら、機械やソフトウェアは箱から出す前に既に陳腐化し始めていることに気づいているでしょう。しかし、それは悪いことではありませ

図1.4
ZXスペクトラム。私がプログラムすることを学んだ機械。1982年にはイギリスのホームコンピューティングの最先端でした。48KB RAM、ハードディスクなし、マウスなし。ビジュアルなオペレーティングシステムではなかったので、ゲームをロードできるようにするためには、プログラミングの基礎を学ばなければなりませんでした。

ん。次の世代の技術は、より優れた数学的な早業能力があって、芸術的な可能性をより広げてくれるからです。

私たちの限られた世俗的な視点からは、デジタルツールの進化はよりいっそうの洗練、スピード、シンプルさ、直感的なわかりやすさを生み出してくれるので、すべて肯定すべきであるように見えるかもしれません。ムーアの法則――コンピュータの処理能力は数年ごとに2倍になり、価格は半分になるという、1960年代の考え――が途絶える兆候は見えず、今のところ、デジタルな世界はより速く！良く！安く！なるだけです。しかし私たちは、この猛スピードのデジタルの潮流によって前へ前へと押しやられることで、私たちの機能的な友人がこの文明が提供する最高のものであり、かつてないレベルの洗練にあると信じるように、自分自身をだましているだけかもしれません。しかしどんな尺度によって、私たちはこのような洗練さを測ればいいのでしょう？

1.3.2　最新の稚拙な技術

私たちは、デジタル技術のまさに初期の段階にいます。コンピュータはまだ生まれたばかりの幼児で、人間とコンピュータのインタラクションの技術は未完成です。Photoshop、Flash、GarageBand、After Effects、Illustrator、Final Cut Pro、Mayaのようなアプリケーション、そしてC++、Java、HTML、CSSのようなシンタックスなど、そのすべては信じられないほどクールで、メディアの見え方や振る舞い方を変えました。けれどもそれらは、絵筆やカッターナイフのシンプルさや直観的なわかりやすさを持ちあわせたツールではありません。それらは明らかに、鉛筆、ペン、ノミ、ギター、ドラム、こて、ハサミから、またこの数十年ではなく、数千年にわたって発展してきた自己を表現する自然で直感的な方法からもかけ離れています。このように比べてみれば、私たちのデジタルツールは、そのユーザビリティが極めて稚拙であり、時にはクリエイティブな作業の妨げになったりもします。ツールの目的は、私たちの能力を伸ばすだけでなく、私たちの創造力の流れを強くするべきものです。最新のソフトウェアは、わずか50年前には想像ができなかったような方法で世界を作る力を与えます。しかし自然の芸術的能力に関しては、それらの使用にはまだまだ改善の余地があります。ほとんどすべての強力なツールには険しい学習曲線が必要で、マスターするのには、専念することと絶え間なく使用することが求められます。子供がすぐにAdobe Illustratorを身につけて線を引くことはできません。ある程度のテクノ通の成人であっても、Mayaのようなソフトウェアに新たに出会って数分以内に自分の表現手法にすることはできません。他方、鉛筆あるいは絵筆の場合、子供でも直ちに（キーボード・ショートカットを学ぶことなくとも）線や色を使ってある種の主張をすることができます。そこで彼らができたことは確かに素人的なものでありますが、しかし大事なポイントは、ツールによって妨げられなかったことと、それよって力を発揮できたということです。

芸術様式は、そのツールによって定義されます。ツールは、芸術様式にその文法を与えます。もしツールの性格が、ある1つのアクションやアイデアを行うのが容易で、他のことを行うのが難しければ、より容易な選択が、よりありふれたものになるでしょう。だから、ほとんどの風景画家が3Dで仕事をしませんし、振付師は果物ボウルに興味を持ちません。私たちのツールは、あなたが思っている以上に制作される作品への大きな影響力を持っています。それは、私たちのクリエイティブな選択を制限すると同時に、拡大することもできます。

いいやり方は、これらのパラメータの良さは認めて、ツールを使う新しい方法を思いつき、疲労やありふれた慣習に陥るのを避けることです。このことは、絵筆と同じようにプログラミング言語にも当てはまります。もしあなたが業界で正式に基礎を積んだ経験豊かなプログラマーであるなら、それはこの本からあなたが獲得して欲しいと私が望むもののひとつでもあります——それはすなわち、コーディングの技芸にアプローチするための新鮮な方法です。そしてもしあなたが一度もコーディングをやったことがないなら、私はあなたがこの新しいおもちゃに、新しい体験を見い出して欲しいと思います。それを使う意志を持った芸術家がいなければ、絵筆には何の意味もありません。

1.4 まとめ

私たちはここまで、何がジェネラティブ・アートではないか、ということを考えてきました。そしてしぶしぶ、私たちの稚拙で永続的でないツールセットの限界を受け入れてきました。確かに、この章から外すべき前向きなものもあるに違いありません。そこで手始めに、ジェネラティブ・アートはおそらく確実にこれだと言える、いくつかの事柄をリストにしてみます。

- 美的なものを作るアルゴリズムの方法
- アーティストと自律システム間のコラボレーション
- 完全に決定論的なプロセスから予想できない結果を引き出す行為
- 秩序とカオスの間のスイートスポットの探求
- コーディングに対する新鮮で面白いアプローチ
- 大きな可能性を秘めた成長するメディア

ジェネラティブ・アートは、デジタルなツールセットとしてはまだ初期段階です。しかしこれらの技術的驚異は、私たちが表現することに新しい力を与えてくれます。同時にそれは、私たちがそこから作り出す作品の種類と大きな関係を持っています。本書でジェネラティブ・アートの実験のために用いるツールは、「Processing」と呼ばれる素晴らしくシンプルな言語です。

Chapter 2

Processing：アーティストのためのプログラミング言語

Processing: A Programming Language for Artist

本章であなたは、プログラマーになります。ジェネラティブ・アートの実験をするために必ずしもプログラミング言語を理解する必要はありませんが、プログラミング言語を習得すれば、ジェネラティブ・アートの実験はずっと容易になるはずです。

この章だけでプログラミング言語をマスターすることは、確かに難しいかもしれません。けれど私はこの約30ページで、コードを使って創作を始めるために必要なこと（ハロー・ワールドから、基本的な構文や関数、自分の作品をプリントやビデオ、あるいはモバイルフォンに出力することなど）をすべて紹介します。

プログラムを学ぼうと思うのであれば、まず最初に制作のための開発ツールを選ばなくてはいけません。ジェネラティブ・アートの開発環境としては、Flashや3ds MAXのような高価なプロフェッショナルツールから、HasCanvasやScratchなど無料のオンラインリソースまで、私たちには既に、豊富な選択肢が用意されています。しかしながら時間は限られていて、限られた時間の中でそれらのすべてを熟知することは不可能です。目的のためには、最良のものがただ1つ必要です。

それを絞り込むために、私はいくつかの基準を考えました。平易で、多くの要求を実現でき、拡張可能で、サポート体制が良いもの。さらに言えば、ジェネラティブなビジュアルを制作するために、スクリーン上だけでなくプリントやビデオ用に書き出すことにも十分に対応できるような環境。そして最も重要なのは、お金を支払わなくても利用できること。こうした無茶な要求を叶えるツールとして、本書では、openFrameworks（www.openframeworks.cc/）、Context Free（www.contextfreeart.org/）、Structure Synth（http://structuresynth.sourceforge.net/）、NodeBox（http://nodebox.net/）、そしてvvvv（http://vvvv.org/）を勧めます。あなたのアイデアを実行するための新しいおもちゃとして、どの開発環境も活躍してくれるでしょう。しかし、これからプログラミングの世界に飛び込む人にとっての最適な開発環境して、私は「Processing」を勧めます。

本書を執筆するきっかけにも、Processingが大きく関わっています。今までのジェネラティブ・アートというのは数学とコンピュータサイエンスの領域の、さらに一部の人たちのものであり、それ以外の一般の人たちには理解されにくいものでした。しかし、Processingというツールがクリエイターやアーティストを中心に一般に広まることによって、ジェネラティブ・アートの可能性が拡がりました。Processingは、コンピュータ科学者以外の人もジェネラティブ・アートを実現できるようにした、最初の言語と言えるでしょう。

2.1 Processingとは？

ProcessingはMITのケイシー・リースとベンジャミン・フライによって始められた、アーティストのためのオープンソース・プロジェクトです。視覚的なフィードバックを即座に得られることが、プログラミングスキルの習得をしやすくしています。もっと難しくてパワフルな言語であるJavaをベースに作られていますが、Processingはさらにシンプルに、プログラミング初心者でも利用できるよう工夫されています。核となる言語のシンプルさのおかげで、Processingの人気が上昇するにつれて、3Dの描画、XMLの読み込み、MIDIやArduinoとのやり取り、他のAPI（FlickrやTwitterなど）やサードパーティーのライブラリ［汎用性の高い複数のプログラムを、再利用可能な形でひとまとまりにしたもの。ソースコードの場合と、オブジェクトコード、あるいは専用の形式を用いる場合とがある］などが拡張され、どんどん活用範囲が拡がっています。

Mozilla社（オープンソースのWebブラウザ「Firefox」を開発・運営する組織）が、“Processing for the Web”という、Processingの言語と環境をオープンに、Webへ移植するプロジェクトを提案したことで、今後JavaScriptとHTML5の「Canvas」タグのような標準技術とも統合することができる予定です。これはつまり、Processingがプラグインなしで、Webブラウザで実行できるようになるということです。このことは既に、JavaScriptで実装されたジョン・レッシグのProcessing.jsプロジェクトで立証されています。Processingの実装については、他にもRuby（http://wiki.github.com/jashkenas/ruby-processing/）、Clojure（http://github.com/rosado/clj-processing）、Scala（http://technically.us/spde/）でも行われています。どちらもたった1人のプログラマが2年ほどで実現した草の根プロジェクトです。これらはProcessingの人気の高まりと、将来の可能性の証です。さまざまなディストリビューションの機会が広がることで、Processing入門のための障壁はますます下がっていくでしょう。

プログラミング言語を習得しなければならないことが、ジェネラティブ・アート入門の大きな障壁になりがちです。すぐにProcessingを自由自在に扱えるようになるのは難しいかもしれませんが、勉強を始めてから習得するまでの学習曲線は、他の言語よりずっと早く伸びていき、

さまざまなことができるようになるでしょう。Processingは、プログラミングを体験するための単なるおもちゃや学習支援ツール以上のパワーを秘めています。とはいえ、どんなに基本的なツールであっても、すべてのツールには学習するためのプロセスが必要です。あなたが初めて鉛筆を持った時、まず初めに取りかかるのは大作のアート作品を描くことではないでしょう。最も順応性のある子供でさえも、たいていの人はまず手描きのお絵かきセットを渡されて、その最も単純なツールでさえも、使いこなすまでに何年もかかります。

プログラミング言語というものは、複雑な現実世界のスペクトラムの一端に位置しています。それは、手で触れることができる物理的なモノをカプセル化することはできません。実行させるためには、多くの複雑なハードウェアを必要とします（そしてハードウェアなしではただの役立たずです）。入力とその応答が抽象的で、直観的にはわかりません。鉛筆をさっと動かして紙に線が引かれたのを見るような、即座のフィードバックとは違って、書く－実行する－修正する（あるいは、書く－実行する－デバッグする）といったプロセスが、アーティストと作品の間に隔たりを生み出します。

しかし前に述べた通り、Processing言語はこれまでのプログラミング言語よりもはるかに使いやすくなっていて、あなたの制作活動の道具や、協力者とすることができます。あなた自身をはるかに超えた数学的で論理的な能力によって、あなたを煩わしてきた困難な作業をプログラムに引き受けてもらうことで、制作に集中することができるようになります。

新人プログラマーであっても、時間と共に左脳の直感やプログラミングに対する興味によって、コーディングの論理に慣れていきます。私は、プログラマーでなかった人がこの本を読み終わった後に、プログラミングに対する恐怖心がなくなっていることを願っています。あなたがツールと仲良くなって信頼関係を結ぶことができれば、きっと新たな協力者との制作が楽しめるようになっているでしょう。

2.1.1　大股歩きとヨチヨチ歩き

あなたのプログラマーとしての出発点はProcessingのWebサイトです（図2.1参照）。Processingのダウンロードも同様に、このWebサイト（www.processing.org/download/）から行います。現在、Mac OS X、Windows、Linuxをサポートしていて、最新バージョンをダウンロードすることができます。本書に掲載しているスクリーンショットはMacバージョンのものです。しかしインターフェースは、サポートされているWindowsやLinuxでもほぼ同一です。

バージョンアップは頻繁に行われているので、Processingのダウンロードページを時々チェックするといいでしょう。2008年11月にバージョン1.0がリリースされて以来、常に改善され続

けています。www.processing.orgには他にも多くのリソースがあります。ダン・シフマンとイラ・グリーンバーグによる素晴らしい入門チュートリアルが「Learning」のページにあり、この言語のさまざまな特色を紹介してくれるたくさんの例が掲載されています。また「Discourse」のページには活発なフォーラムがあります。また、是非あなたのブックマークにzenbullets（www.zenbullets.com）とAbandoned Art（www.abandonedart.org）も加えてください。両方とも私のサイトですが、ここからその他多くのジェネラティブ・アートとそのソースコード、そして本書のために制作したプログラムのダウンロードが行えます。

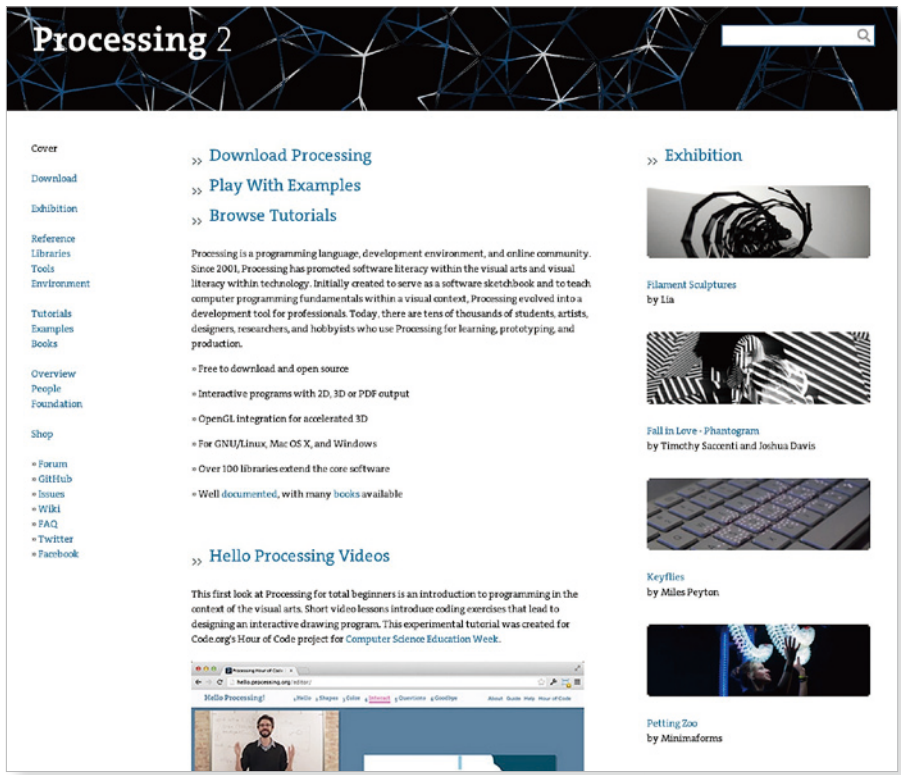


図2.1
Processing.org。チュートリアル、リファレンス、論議のための非常に貴重なリソース。
またProcessingの最新版をダウンロードする所でもあります。

次のセクションでは、あなたにProcessing開発環境（PDE）を紹介しながら、まず最初にどこにコードを書いたらいいのかを説明します。もしあなたがこれまでProcessingを使ったことがあるなら、この章のこれ以降はきっと必要ないでしょう。その後、プログラミング初心者を念頭において、言語の要点を必要に応じてひとつひとつ詳しく説明していきます。Processingのコンセプトとプログラミング言語の文法に触れながら、実験を始める前に最低限必要ものを身につけましょう。あなたが初心者であるなら、このセクションは不可欠です。

あなたがプログラミング経験者でありながら、まだProcessingを使ったことがないなら、ここまでのページを読み飛ばしてもかまいません。けれども、あらゆるプログラミング言語にはそれぞれ固有の特徴があります。例えば、Processingではどのようなコンセプトでフレームをループさせているのかを知ることが必要です。もしあなたがJavaやECMA（ヨーロッパ電子計算機工業会）ベースの言語（ActionScriptやJavaScript）でプログラミングをしたことがあるなら、Processingは自然にあなたになじんでいくでしょう。

Processingのようなオープン・コミュニティ・プロジェクトの良い点の1つは、Web上での無料リソースに事欠かないことです。もし本書でのチュートリアルがあなたのペースに合っていないとしても、既にWeb上に無料のリソースがあり、あなたをサポートしてくれるはずです。ここでもう一度、出発点としてwww.processing.orgを紹介しておきましょう。

さて、あなたはProcessingに「ハロー」と挨拶を済ませました。次は、Processingが応えてくれるか見てみましょう。

2.1.2　ハロー・ワールド

あなたがProcessingをダウンロードして、インストールし、初めて起動すると、まず図2.2のようなウィンドウが現れます。これがProcessing開発環境（PDE）です。この空白部分に何かを書いてみましょう。まずはメイン画面に以下のテキストを入力してください。

```
ellipse(25, 25, 50, 50);
```

それから、Runボタンをクリックしてみます。何が表示されましたか？　出力ウィンドウが現れているはずです（図2.3を見てください）。100×100ピクセルのキャンバスに、（左上から測って）中心点が25, 25、幅と高さが50ピクセルの円が描かれています。これがビジュアルプログラミング環境におけるハロー・ワールドです［Hello world（ハロー・ワールド）とは、画面上に文字列を出力する非常に簡単なプログラムで扱われる文字列であり、ひいてはそのプログラム全体を指す。大抵のプログラミング言語の入門書では、このプログラムを作ることを最初の例題としており、ほとんどの場合、新しくプログラミング言語を習得する際に最初に作るのがこのプログラムである。そのため、

「世界一有名なプログラム」と呼ばれることもある]。

もしあなたがコーディングに不慣れであったとしても、このたった1行であなたは既に2つのプログラムの記述方法をマスターしました。あなたは関数(ellipse)を呼び出し、そこにパラメータ(x、y、幅、高さ)を渡しました[パラメータとは、ソフトウェアを実行したりプログラム内で関数を呼び出したりする時に、その動作を指定するために外部から与える設定値のこと]。関数は、あなたが実行しようしている作業に対する一種のサービスだと考えることができます。時にはパラメータを与え、時にはただ呼び出すだけだったりします。関数の記述よっては数値を返すかもしれませんし、あるいは与えられたパラメータに合わせてスクリーンに描画するというような作業を実行するかもしれません。この章の後半では、あなたに自分自身で関数を書いてもらおうと思います。ですがそこに行く前に、Processingに組み込まれたいくつかのコア関数(ellipseもその1つです)を試すことになるでしょう。Helpメニューをクリックして Referenceを選択すれば、その他の関数を使ったサンプルの全リストを見ることができます。リファレンスはオンライン(http://processing.org/reference/)にもあります。

図2.2
空のPDEウィンドウ。ダウンロードしてProcessingを起動すると、あなたが目にする最初の画面です。

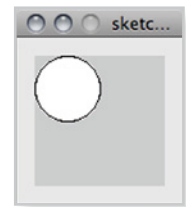


図2.3
ハロー・ワールド。出力ウィンドウは、あなたのコマンド(命令)の結果を示します。1行のellipse(25, 25, 50, 50)は、x:25、y:25に中心点がある直径50ピクセルの円を描きました。

シンタックス問題

たいていのプログラミング言語では、構文（シンタックス）が問題になります。人間の言葉では、もしあなたがある文章の中の単語を誤って大文字にしたとしても、その意味は失われないでしょう。例えばellipseではなくEllipseと書いたとしても、問題なく意味は通ります。けれどもプログラミング言語でこのように記述すると、プログラムは途中で中断してしまいます。

同様に、あなたが何をする時にも、命令の終わりにセミコロンを加えることを忘れないでください。Processingの構文では、セミコロンは終止符を意味します。空白のスペースが間に入ることは問題ありません。あなたが書くすべての文章がセミコロンで終わっている限り、同じ行にあっても実行できます。セミコロンを忘れてしまうことは、Processingではよくあるエラーです。

2.2 プログラムでお絵描き

Processingでプログラミングを始めるために知っておくべきことが、この2.2と続く2.3、2.4内にすべて含まれています。これらの3つのセクションを読めば、あなたはProcessingだけではなく、プログラミング全般の基本ルールを把握したことになります。すべてのプログラミング言語のコアとなる部分は、同じような構造（変数、ループ、関数）に集約されます。ですから、Processingの概念を把握すれば、あなたは他の言語の半分ほどを理解したと言ってもいいでしょう。

最初はあまり直感的には理解できないかもしれません。もしあなたが未知の領域をあてもなく彷徨っているように感じたなら、この章をゆっくりと読み進めることを提案します。読み進めながらコードを入力し、あなた自身がプログラムを操っていると感じられるようになるまで、さまざまなコードをProcessingで遊んでみることに時間を使ってください。それで幸せな気分になれたら、次に進んでください。プログラムを習得するのに時間制限はありません。あなたが辛抱強く繰り返した努力は、きっと報いられるでしょう。経験豊かなプログラマは、この時点で既に本章を斜め読みしているに違いありません。私はまだ彼らにとって価値のある情報を提供していませんから。

次のChapter 3からは、グラフィカルなプログラミングをいじくったりルールを破って、いろいろ楽しむことができます。けれどもルールを破る前に、破ろうとするそのルールをまずは学ばなければなりません。それは退屈なことですが（私も知っています）、もし的確にフランス人を侮辱したければ、まず最初にフランス語を学習すべき、ということを覚えておきましょう。そこで、まずここでは単純な形を描くことから始めます。

2.2.1　関数、パラメータ、色値

前のセクションでProcessingをダウンロードして、初めての1行のスクリプトを入力し、そして出力ウィンドウにそれを表示しました。既にあなたは、初めての関数（**ellipse**）を呼んで、そしてそれにいくつかパラメータ（**x**、**y**、**幅**、**高さ**）を渡したことになります。さて、もう少しやってみましょう。

PDEの新しいウィンドウを開き、以下をタイプしてください（図2.4を参照）。// で始めている行はコメントで、人間が読むためのテキストを書く場所です。// は、Processingがその行に書いてあることを実行してはならないことを示しています。同じく /* 〜 */ は、複数行に渡ってコメントを書く時に使えます。

```
// 設定と背景
size(500, 300);
smooth();
background(230, 230, 230);
// 2本の交差した直線を描く
stroke(130, 0, 0);
strokeWeight(4);
line(width/2 - 70, height/2 - 70, width/2 + 70, height/2 + 70);
line(width/2 + 70, height/2 - 70, width/2 - 70, height/2 + 70);
// 円を描く
fill(255, 150);
ellipse(width/2, height/2, 50, 50);
```

Runボタンをクリックしてください。すると出力ウィンドウ上に図2.5のように表示されます。あなたがPDEに書いたものは関数の一覧です。それぞれの関数が何をしていたのか、一緒に見ていきましょう。

まず、キャンバスの大きさを500ピクセル×300ピクセルに設定します。

```
size(500, 300);
```

スムージングをかけて、描く直線にアンチエイリアスを加え、ガタガタに見えないようにします。

```
smooth();
```

そして背景色をグレーに設定します。

```
background(230, 230, 230);
```

background関数への3つのパラメータは、RGB値で赤：Red、緑：Green、青：Blueです。そ

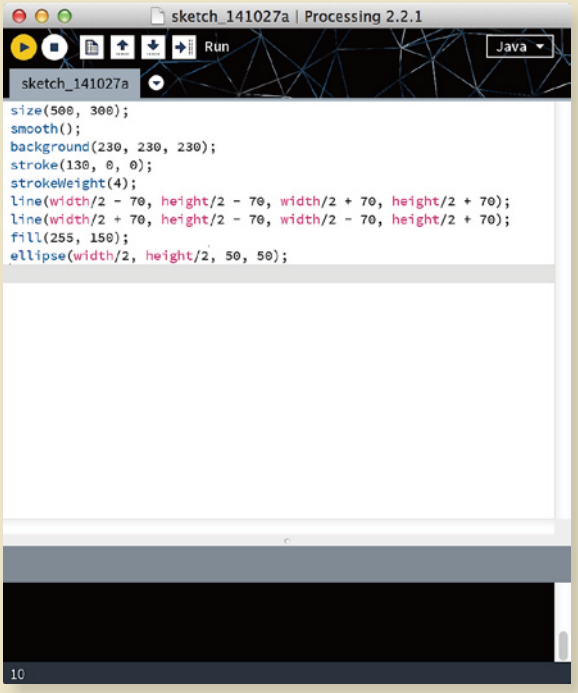


図2.4
新しいスクリプトを記述したPDE。

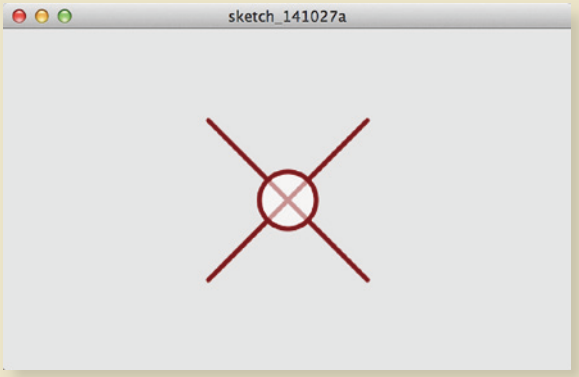


図2.5
出力ウィンドウ。あなたが書いた命令の結果を示しています。

れぞれの値は0から255で指定でき、どれだけその色が混合されるかを定義します。これらの3つの値で1600万の色の組み合わせができます。この例の場合、RGBがすべて同じ値なので灰色になります（RGBを均等に混ぜると、灰色の濃淡になります）。灰色の濃淡で表現する場合は、**background(230);** と、1つのパラメータで短く表現することもできます。こう書いても、3つのパラメータのものと同じ灰色が表示されます。

他のグラフィカルなアプリケーションの多くが、このRGB表記法を使っています。もしあなたがRGB表記に詳しくなかったら、まずは混色の考え方に慣れるため、**background**パラメータに0から255までの数値をいろいろ入力して試してみましょう。同じく、HTMLあるいはPhotoshopなどで、16進法の値が赤を意味する#FF0000のような値を見たかもしれません。この表記もRGB表記と似ています。3つのRGB値は16進数（基数16）の文字列でも表現でき、それぞれ2つの文字でカラー値を表すことができます。例えば、赤：255（ff）、緑：00、青：00[*1]。

Processingでも同じように16進数を使うことができます。**background(#E6E6E6);**は**background(230, 230, 230);**と同じ結果となります（16進数のE6＝10進数の230）。

次に、strokeを見ていきましょう。

2.2.2　線、スタイル、座標

次は以下の2行です。

```
stroke(130, 0, 0);
strokeWeight(4);
```

ここでは、Processing上で線の色（仮想のペンを想像してみましょう）をRGB値で赤に設定し、線の太さを4（ピクセル）にセットしています。

これで、設定したstrokeのスタイルで線が引けます。

```
line(width/2 - 70, height/2 - 70, width/2 + 70, height/2 + 70);
```

この関数（line）のパラメータは、**（始点のx座標，始点のy座標，終点のx座標，終点のy座標）;** の4つです。この段階では、あなたはまだ決まった値を渡していません。4つのそれぞれのパラメータのためのスロット（箱）は計算によって決まり、パラメータとして渡されます。あな

＊1　16進法の表記法についての詳細は、www.web-colors-explained.com/hex.phpを参照。

たが最初にウィンドウの幅（width）と高さ（height）を500×300ピクセルに定義したことを覚えているなら、描かれた直線が点(180, 90)から点(320, 220)に向って描かれるのが計算からわかるでしょう（座標値は常に左上が原点になります）。

ここで、なぜline(180, 80, 320, 220)と書かないのだろう？と疑問に思うかもしれません。これでも同じ結果になるのに、と。その理由は、あなたがキャンバスの中心にバツ印を描こうとしていることにあります。そう、あなたはキャンバスの大きさが500×300であることを知っていますね。数行前に自分でこれらの値を設定したことを覚えているはずです。しかし、キャンバスの幅（width）と高さ（height）の値を、あなたの記憶ではなく、Processingの知識を使っていれば、ちょっと戻ってsize関数のパラメータを変えてみてください。例えばsize(600, 150)など（実際にやってみましょう）。交差した直線は、それでも中心点にありますね。

絶対座標(250, 150)の場所は、決して動きません。けれども相対座標(width/2, height/2)――つまり(幅/2, 高さ/2)――は、キャンバスの大きさに合わせて動的に再配置します。

次の行のコードも同様です。中心点に合わせて、2番目の直線で十字を描きます。

```
line(width/2 + 70, height/2 - 70, width/2 - 70, height/2 + 70);
```

プログラミングのもうひとつの概念「変数」を紹介するのは、このあたりがちょうどいい頃です。

2.2.3　変数

先のコードでは、あなたが何度も同じ計算（width/2とheight/2）を行っていることに気づいたと思います。もしその計算を1度にまとめることができれば、手際が良くなりそうですね。コードを読むのが楽になり、もしプロセッサの計算がより少なく済めば、より速く実行できます。

あなたが今後、もっと複雑なコーディングをするようになった時に迎える限界の1つが「速度」です。どんなコンピュータのCPUでも、毎秒たくさんの計算を行うことができます。ですから、あなたがほんのひと握りの指示を出しているだけの時には、速度はさほど問題ではなりません。けれどもすぐに、毎秒かなりの量の計算を行うようになるので、効率の考え方を念頭に置いて、もし実行する計算の数を簡単に減らす方法がある場合、それを採用すべきです。

2つの交差する直線の描画を、以下のように変えてみましょう。

```
float centX = width/2;
float centY = height/2;
line(centX - 70, centY - 70, centX + 70, centY + 70);
line(centX + 70, centY - 70, centX - 70, centY + 70);
```

ここであなたは、計算の結果を保持する2つの変数を宣言しています。変数は箱で、その中に、ある型の1つのものだけ入れることができると考えてみてください。だけど、ここに入れるものは変化する、だから「変数」というわけです。あなたは変数を使う前に、変数の箱を宣言しなくてはなりません。また、それに名前（何でもいい）をつけなくてはなりません。最初にそこに何かを必ず入れる必要はありません。以下は変数を宣言するためのやり方です。

```
float centX;
centX = width/2;
```

そこに入れるデータの型（この場合はfloat）を定義して、変数を宣言します。「floatって何？」というあなたの声が聞こえてくるようです。表2.1は、あなたが使用できる、組み込み変数のデータ型の一覧です。

表2.1　Processingでよく使われるデータ型

データ型	使用例	説明
char	char varName = 'a';	aや#シングル引用符など、文字やユニコードシンボル。シンボルの周りにシングルクォーテーション（'）が使われているのに注意。
int	int varName = 12;	整数。正の数でも負の数でも可。
float	float varName = 1.2345;	浮動小数点。小数点を持っているかもしれない数。
boolean	boolean varName = true;	正（true）か偽（false）の値。二択になる場合の論理演算に使用されます（不確定でも良いが、その場合は「偽」となる）。
byte	byte varName = -127;	-127から128の間の数値を格納できるだけで、それほど使われないデータ型。シリアルポートでコミュニケートする必要がある時に最も有用になりますが、本書では扱いません。
String	String varName = "hello";	文章のような文字（char）のリスト。Stringの大文字のSはコンポジットタイプ（charの集まり）であることを示しています。

次に、色の塗りつぶしとアルファ値を見ましょう。

2.2.4　塗りつぶし、アルファ値、描画の順序

さあ、次は2本の交差した直線の描画方法について徹底的に分解してみましょう。スクリプトの最後の2行で円を描いています。

```
fill(255, 150);
ellipse(width/2, height/2, 50, 50);
```

円を描く前に行っているのは塗りつぶしの設定です。fillは、円の線の内側の範囲のことです。fillの値と、以前に設定した交差する直線のstrokeを使って、最後のコマンドで円（circle：幅と高さが等しい楕円（ellipse）のこと）を描きます。1つの色値255（グレースケールの表記法で、この場合は白色）を使って、円の内部の色を設定します。これまでと同様に、0から255の数値の範囲でアルファ値（色の透明度）を設定することができる、2つ目のパラメータがあります。

アルファ値が0の場合は完全に透明で、255では不透明になります。あなたが色を定義する場合、どこでもアルファ値を加えることができます。試しにコードに戻って、設定した線（stroke）の色にアルファ値を加えてみてください。3つのパラメータのRGB構文の場合、アルファ値は4番目のパラメータになります。

```
stroke(130, 0, 0, 50);
```

あなたが書いた命令は、最初に2本の実線を描いて、その上に半透明の円を重ねることを意味しています。プログラムを構成する関数のリストは、上から順番に実行されます。この関数の順番を変えれば、円が交差した直線の前に描かれ、交差した部分は半透明の白い円の上に重なることになり、線ははっきりとした実線に見えるでしょう。もし、私を信じることができなかったら、次のコードを試してください。

```
size(500, 300);
smooth();
background(230, 230, 230);

float centX = width/2;
float centY = height/2;
stroke(130, 0, 0);
strokeWeight(1);
fill(0, 40, 0);

ellipse(centX, centY, 30, 30);
line(centX - 70, centY - 70, centX + 70, centY + 70);
line(centX + 70, centY - 70, centX - 70, centY + 70);
```

コードの最初の部分は、描画の準備と背景を設定しています。次に、線（stroke）と塗り（fill）を設定しています。最後に、円を描いた後、2本の直線を引くように指示をしています。

実行の順序は、描画の重なり方だけでなく、あなたが設定したスタイルとも関係しています。

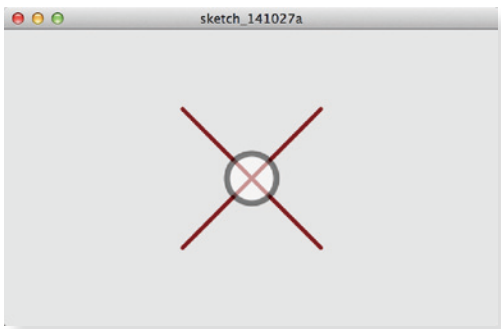
Processingの基本は、線の色や重なりのようなスタイルを設定すると、その設定は次に何か違ったことを設定するまでずっと維持されます。上の例の場合、円と直線を描く前にstrokeとstrokeWeightが呼び出されているので、すべての線は同じスタイルになります。しかしこれらの設定は、いつでも好きなだけ変えることができます。線が描かれるたびに、最後に設定されたスタイルを使うのです。以下はstrokeのスタイルを変えた例です。

```
size(500, 300);
smooth();
background(230, 230, 230);

stroke(130, 0, 0);
strokeWeight(1);
line(width/2 - 70, height/2 - 70, width/2 + 70, height/2 + 70);
line(width/2 + 70, height/2 - 70, width/2 - 70, height/2 + 70);

stroke(0, 125);
strokeWeight(6);
fill(255, 150);
ellipse(width/2, height/2, 50, 50);
```

結果を図2.6に示します。もしあなたが、楕円（あるいは他の形でも）がfillによって色で塗りつぶされるのを望まないなら、関数noFill（パラメータがない関数）によって塗りつぶしをしないようにすることができます。同様に、もしあなたが線（輪郭線）を描きたくないなら、noStrokeを呼び出してください。



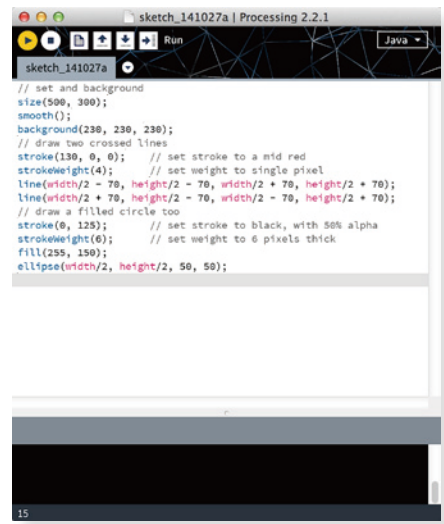


図2.6
形を描く前に、stroke、fill、strokeWeightによって描画スタイルを設定します。

これで、Processingによる描画の基本を理解しました。Processingの座標系に慣れるためにいくつかのコマンドを試したり、Processingのリファレンスから他の形——矩形を描くためのrect、直角ではない四辺形のためのquadなど——を選んで描いたりと、時間を使って存分に試してみてください。triangle関数を使えば何を生み出せるか、などということについてヒントを与えるのは、ここでは止めておきましょう。

Processingの描画方法がわかったら、次にアニメーションの方法を紹介します。どのようにコードを組んで制御していくかを学びましょう。

2.3　構造、論理、アニメーション

これまでのところ、あなたが書いたスクリプトは一方向に命令が実行されていく、ごく短いものでした。これはまだ単純な基本レベルのプログラムなので、ここからさらに先へ進む必要があります。そのためには、複数の関数をグループ化したり、決定を下したりしなければなりません。ステップアップするタイミングです。

2.3.1　フレームループ

このセクションでは、Processing特有の概念（Processing以外のツールにも多くありますが）を紹介します。「フレーム（frame）」という言葉は、コンピュータサイエンスの世界ではいろいろな意味がありますが、ここでは、1枚の静止画であってもそれを充分速く切り替えれば動いているように見える、という映像の原理のことを指しています。

Processingでは、フレームループによって連続的に画面を再描画しています。もしあなたがProcessingに再描画するよう指示すると、それは2次元の画像に時間軸を加えたことになり、成長したり、移動する作品を作ることができます。

画面上でこの描画方法を実現するためには、関数ブロックの中にコードを書く必要があります。「関数ブロック」とは、命令をグループ化してまとめる方法です。Processingには、フレームの基本構造を書くために、2つの特別な関数ブロックsetup()とdraw()があります。プログラムが起動すると、setup()関数ブロック内のコードが一度呼び出されます。そこには初期化のためのコード（キャンバスのサイズ、背景色、変数の初期化など）が含まれます。draw()の中に書くコードは、フレームごとに繰り返し呼び出されます。frameRate関数を使うことによって、draw()が呼び出される速度を設定することができます。もしあなたがフレームレートを設

定したら（12、24、25、30が一般的です）、プログラムはその速度を維持しようと、規則的にdraw()を呼び出します。そうしないと、マシンが可能な限り速くフレームループを実行してしまうのです。

もしあなたがフレームごとに、あまりに複雑な命令を行おうとした時には、指定したフレームレートの速さでコードを処理することができないことがあります。できる限り速くループするように設定した場合、フレームレートが低くなるだけでなく、もしかすると想定外の結果を生み出すかもしれません。

次の例を見てみましょう。新しいスクリプト画面を開いて、そして以下のコードを入力してみてください。関数ブロック内の命令がわからなかったとしても心配しないでください。1つずつ解説していきます。

リスト2.1　setup()とdraw()関数ブロックのそれぞれにコードを書く

毎秒24フレーム

背景はすべてのフレームでクリアされます

次のループのたびに直径を広げます

```
int diam = 10;
float centX, centY;

void setup() {
  size(500, 300);
  frameRate(24);
  smooth();
  background(180);
  centX = width/2;
  centY = height/2;
  stroke(0);
  strokeWeight(5);
  fill(255, 50);
}

void draw() {
  if (diam <= 400) {
    background(180);
    ellipse(centX, centY, diam, diam);
    diam += 10;
  }
}
```

これを実行すると、出力ウィンドウで表示された円がだんだん大きくなり、円の直径が400ピクセルに達すると止まります（図2.7参照）。円の直径と中心は変数で表され、中心座標はsetup()で計算されています。フレームループは、円の直径が400より小さいことをチェック

しながら背景と円を描き、ループが繰り返されるたびに10ピクセルずつ円の直径を大きくしています。直径の値（変数diam）が400に到着するまで、10、20、30…と少しずつ直径が大きくなりながら円を描いていくということです。フレームループは繰り返され、円が規定の直径に達した後も呼び出されている状態なのですが、draw()は既に何も描かなくなっているということに注意してください。

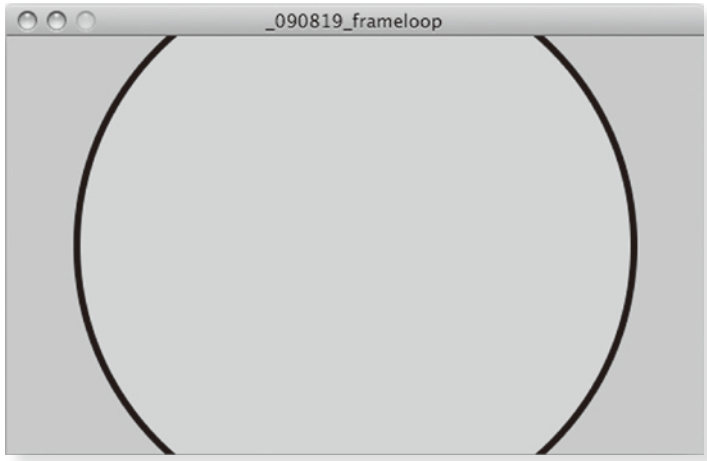


図2.7
フレームループを使って描かれた円。スクリプトを実行すると、円は10ピクセルの直径から400ピクセルまで成長していきます。

関数ブロックを使い始めると、これまでやっていた関数ブロックの外にコマンドを書いていた時の簡単さが失われます。しかしもう過去は振り返らないでください。あれは、わかりやすく学ぶためだけのものです（ありがとう、Processing）。今後も、コマンドが連続する長いリストでプログラムを書き続けることもできます。それは確かに理解しやすい方法かもしれませんが、その方法でできることは限られています。

変数の宣言は、これからも関数ブロックの外で行います。文法上の問題はありません。関数ブロックの外で変数を宣言することは、変数の内容があらゆる関数ブロックで利用可能であることを意味しています。もしあなたが、関数ブロックの中で変数を定義したなら、その変数はそのブロックの中だけで利用できます。もし変数がただ1つの関数の中で必要とされるなら、この方法が良いでしょう（プロセッサがその値を記憶しなくてもよいので、他の目的のためにメモリを開放できるからです）。

このことは「変数のスコープ」と呼ばれています。関数の中にある変数は「ローカル」と呼ばれ、その変数は局所的に存在しています。関数の外の変数は「グローバル」と呼ばれ、いたるところに存在しています。変数のスコープを間違えるのは、初心者にもベテランにもよくありがちなプログラムのバグです。あなたのコードが意図しない振る舞いをしたなら、最初に調べた方

が良いことの1つです。そのため、多くのプログラマーは混乱しないように、グローバル変数の名前の前にアンダーバー（_）をつけています。例えば：

```
int _num = 20;

void draw() {
  int inc = 10;
  _num = _num + inc;
}
```

ここでは、最初の行でグローバル変数（_num）を宣言し、一方、関数ブロックの中のincはローカル変数です。

リスト2.1の一連のスクリプトは、プログラマーではない人にとってたくさんの新しい概念を含んでいます。フレームループで遊び続ける前に、このもっと細部を見てみましょう。

2.3.2 自分の関数を書く

最初に関数ブロックそのものです。関数ブロックの文法は以下のようになります。

```
データ型 関数名( [パラメータ] ) {
  // コードをここに記述
}
```

setup()とdraw()関数のデータ型は「void」です。voidは何も値を返さないことを宣言するProcessingならではの手法です。この場合、パラメータは必要ありません。その他の有効な戻り値［戻り値とは、プログラム中の関数が処理を終了し、呼び出し元に処理の結果として返す値のこと。Processingでは、戻り値を返さない（void）関数を作成することもできる］のあるデータ型としては、String、char、int、floatなど、変数のデータ型と同じです。

これらデータ型についての考え方は、自分自身でプログラミングを始めると、だんだん多くのことがわかるようになってきます。以下の新しいスクリプトを書いて試してください。

```
void setup() {
  int x = AddNumbers(1,2);
  println(x);
}
int AddNumbers(int a, int b) {
  int returnValue = a + b;
  return returnValue;
}
```

最初の関数setup()は、Processingによってスクリプトが実行されると自動的に呼び出されます。しかし、2番目の関数AddNumbersはあなた独自の関数なので、あなたが呼び出すまで何もしません。setup()関数の中で整数の変数xを定義して、そこに自作の関数AddNumbers(1,2);が返す値が代入されるように設定します。同じように自作の関数の戻り値を、intとして定義します（これを行わないとエラーになります）。関数が必要とする2つのパラメータの戻り値のデータ型を定義したことで、このプログラムが何を行うかがわかるはずです。

自作の関数に渡した2つのパラメータを合計し、その値を返すことで、変数に結果を代入します。もしあなたが関数にデータ型を与えたなら、関数の中のどこかにreturnというキーワードがあるに違いありません（さもなければエラーになってしまいます）。関数の仕事が完了し、そして戻り値が変数xに入れられた後、Processingはsetup()の関数ブロックの指令を、続けて実行します。println(x)は、コンソールウィンドウ（PDEのテキストエディタの下の黒い窓）にこの変数の値を出力します。このスクリプトは視覚的なアウトプットを行わないので、出力ウィンドウは空のままです。

2.3.3　演算子

あなたの自作関数の中にもう1つ、新しいプログラミング概念があります。演算子のreturnValue = a + bです。こちらは単なる計算なのでご心配なく。演算子の全リストは、表2.2にまとめています。

演算子をグループ化することで、従来の数学的な表記法でするのとまったく同じように利用できます。括弧を使えば計算する順番を定義することもできます。

```
float a = 45;
float b = 5;
float c = 4;
float sum1 = (a + b) * (b * c);
float sum2 = a / (b + c);
```

この例では、sum1は1000（50 * 20）となり、sum2は5（45 / 9）となります。

合計の結果が整数ではない場合もあるので、小数を含むfloatを使わなければなりません。あなたが使える数学的な関数を以下に紹介します。

- ceil()――最も近い整数まで切り上げ。
- floor()――最も近い整数まで切り下げ。
- round()――最も近い整数まで切り上げ。または切り下げ。

- min()――渡された数の中の最小値を返す。
- max()――渡された数の中の最大値を返す。

表2.2　Processingで使われる演算子

演算子	名前	例	結果
+	加算	num = 45 + 5;	numが50になります。
-	減算	num = 45 - 5;	numが40になります。
*	乗算	num = 45 * 5;	numが255になります。
%	剰余	num = 45 % 5;	numが45 / 5の余り(0)になります。
/	除算	num = 45 / 5;	numが9になります。
++	インクリメント(増分)	num++;	numの値が1だけ増加します。
--	デクリメント(減少分)	num--;	numの値が1だけ減少します。
+=	加える	num += 4;	numの値に4加えて、その結果をnumに戻します。
-=	引く	num -= 4;	numの値から4を引いて、その結果をnumに戻します。
*=	かける	num *= 4;	numの値に4を掛けて、その結果をnumに戻します。
/=	割る	num /= 4;	numの値を4で割って、その結果をnumに戻します。
-	負数	minusnum = -num;	numの値に-1を掛けます。

切り上げ、切り下げに関わる3つの関数は、小数を整数に変換する時に便利です。

```
int centX = floor(width/2);
```

floor()関数なしの場合、もし横幅のピクセル数が奇数であると、結果が小数となるのでエラーになってしまいます[実際には、小数点以下を切り捨てるので、int型に代入してもエラーにはならない]。floor関数を使えばこういったエラーを防ぎ、うまく整数にしてくれます(たとえ中心点が0.5ピクセルずれていても、あなたは違いには気がつきませんから)。

Processing言語には、洗練された数学的な関数がたくさんあります。それらいくつかの関数は、本書の後半の実験で使うことになるでしょう。もしこの段階を飛び越えて自身のプログラミングを楽しみたいなら、Processingのリファレンスを見つつ、利用できる関数で遊んでみてください。

2.3.4　条件文

円を描画するスクリプトにある最後の新しい要素が、「条件文」です。

```
if (diam <= 400) { ... }
```

条件文は、{ }内のコードが実行される前に、まず条件に合っているかをチェックします。この場合、条件文で直径（diam）の値が400より小さいあるいは等しいかどうかを確認し、条件に合っていた場合に{ }内のコードを実行します。もし条件に合ってなければ、{ }内のコードは飛ばされます。

```
// 条件をチェック
if (diam <= 400) {
  // もし条件があうなら
  // {}内のコードを実行
}
```

条件に合わない場合にコードを実行する時は、**else**を使います。

```
if (diam <= 400) {
  // もしdiamが<= 400ならこのコードを実行
} else {
  // もしdiamが> 400ならこのコードを実行
}
```

スクリプトが進む様子を水の流れにたとえれば、条件文を設定することで、水が進めるもう1つ別の水路を作っているようなものです。**if … else**によって水の流れは2つのうちの1つを選んで進むことができます。条件文**if**の内容によって、水路を通り抜けてしまうか、同じ場所を回り続けるかのどちらかになります。

あなたが使うことができる比較演算子は次の通りです。

- ==　等しい
- !=　等しくない
- >　より大きい
- <　より小さい
- >=　より大きいか同等
- <=　より小さいか同等

以下の論理演算子を使って条件をまとめることもできます。

- ||　　論理和、OR演算
- &&　　論理積、AND演算
- !　　論理否定、NOT演算

これがその例です。

```
String likeStr;
if ((likeStr == "pina colada") || (likeStr == "getting lost in ⏎
                                    the rain")) {
  println("come with me and escape");
}
```

条件文は、好きなだけ入れ子構造にすることができます。以下は正しく動くコードです。

```
int man = 5;
int devil = 6;
int god = 7;
if (man == 5) {
  if (devil == 6) {
    if (god == 7) {
      goToHeaven("monkey");
    }
  }
}
```

この場合は次のようにしても同じ結果が得られます。

```
if ((man == 5) && (devil == 6) && (god == 7)) {
  goToHeaven("monkey");
}
```

いよいよあなたもプログラム的な思考で考え始め、論理的にものごとを決めるようになってきましたね。次に、プログラミングに必要不可欠なもので、計算機が人間より得意なもの――「繰り返し」を見てみましょう。

2.4　繰り返し

あなたがプログラマーだと呼ばれるために最低限必要な、最後の基本的なプログラミングの概念、それが「繰り返し（ループ）」です。これされ習得すれば、この勉強も終わりになることを約束しましょう。プログラムで描画する場合、単独でも複数の命令でも、それを何百、何千、

何百万回と繰り返す能力は、他のいかなるクリエイティブな手法を超え得る大きなメリットになります。繰り返しは、あなたのアイデアのスケールを広げるためのいちばんの方法です。

あなたは既に「フレームループ」という繰り返しの機能に出会っていますね。draw()関数の中のコードは、新しいフレームになるたびに呼び出されます。プログラムが動いている限り、コードは常に繰り返し実行されます。あなたはもちろんこのループを使用できますが、あなたが使えるループはこれだけではありません。あなた独自のループをコードで書くこともできます。

ここから数ページ戻って、フレームループを使って円を描画するスクリプトをもう一度リロードして、それを試して試してみましょう。

2.4.1 whileループ

コードループとは、このようなものです。

```
int bass = 99;
while (bass > 0) {
  println(bass);
  bass--;
}
println("how low can you go");
```

これは、コンソールウィンドウにbassの値を99回出力します。whileコマンドで( )内の条件をチェックして、もし条件が合えば{ }内のコードを実行します。ブロックの先頭までループして戻り、条件が合わなくなる（この場合、bassが0になる）まで、実行を続けます。

もし、ループするたびに値から1を引いていくbass--をコード内に記述しなければ、bassの値が99のままで、条件に合うことがなくなって、ループが永遠に続くことになります。これはあなたがRunボタンをクリックした時によく起こるエラーの1つです。コンパイラはこのことを見分けられるほど賢くはありません。あなたのコード内にこうした無限ループが含まれているとシステムが動かなくなってしまうので、あなたはProcessingを強制終了して、そこから逃げ出さなくてはなりません。注意しましょう。

さあ、それではこのコードを動かしてみましょう。円のスクリプトに太字の部分を追加してから、実行してみましょう。

リスト2.2　whileループで描画

```
int diam = 10;
float centX, centY;

void setup() {
  size(500, 300);
  frameRate(24);

  smooth();
  background(180);
  centX = width/2;
  centY = height/2;
  stroke(0);
  strokeWeight(5);
  fill(255, 50);
}

void draw() {
  if (diam <= 400) {
    background(180);

    strokeWeight(5);
    fill(255, 50);
    ellipse(centX, centY, diam, diam);

    strokeWeight(0);
    noFill();
    int tempdiam = diam;
    while (tempdiam > 10) {
      ellipse(centX, centY, tempdiam, tempdiam);
      tempdiam -= 10;
    }

    diam += 10;
  }
}
```

ループの中のループ

次のループへ直径を増やす

このコードには、ループの中にループがあります。直径（diam）の値を他の変数にコピーして、それから第2のループによって、現在の円から10ピクセルずつ直径を減少させながら新しい円を描きます。この効果を図2.8に示します。

strokeとfillをリセットするため、行を追加する必要があります。そうしないと、フレームループが行われるごとに前の設定がそのまま適用されてしまいます。

2.4.2　痕跡を残す

実は、これよりずっと簡単に同心円を描く方法があります。以下のコードを実行してみましょう。

リスト2.3　痕跡を使って同心円を描く

```
int diam = 10;
float centX, centY;

void setup() {
  size(500, 300);
  frameRate(24);
  smooth();
  background(180);
  centX = width/2;
  centY = height/2;
  stroke(0);
  strokeWeight(1);
  noFill();
}

void draw() {
  if (diam <= 400) {
    // background(180);  この行をコメントアウト
    ellipse(centX, centY, diam, diam);
    diam += 10;
  }
}
```

図2.8と同じ同心円が表示されますが、以前よりコードの行数が少なくなっています。重要な変更箇所は、フレームごとにイメージをクリアする（そのため毎フレームすべて再描画しなくてはならない）background(180);がコメントアウトされていることです。イメージをクリアしなければ、前のフレームの上に描画します。成長する円のアニメーションは、フレームごとに直径が成長していきます。もしあなたがフレームをクリアしなければ、その動きの痕跡を見ることができます。

ここから面白いエフェクトを作ることができます。noFill()の行をfill(255, 25);に変更して、円をかなり透明な白色にしてみてください（図2.9参照）。かろうじて見える程度の透明度であっても、効果は累積します。中心部ほど多くの円が重なっているので、端のほうよりもより白くなっています。

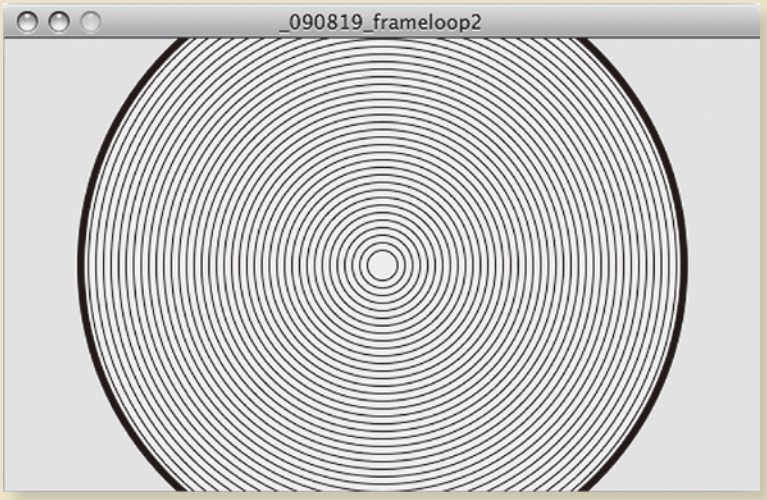


図2.8
サークルの中に描かれたサークル。ループの中のループ。

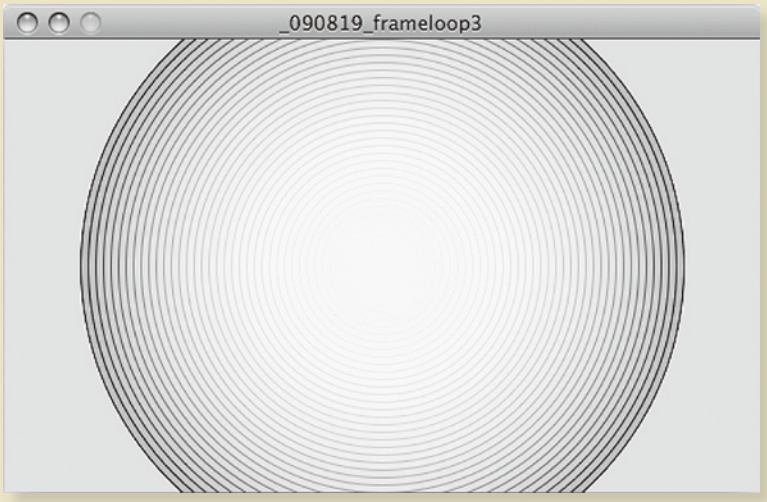


図2.9
再描画してもフレームをクリアしないことによって、
アルファ値を使った微妙なフェードを作ることができます。

2.4.3 forループ

さて、やかんでお湯を沸かしに行く前に、最後にお見せしたいループがあります。forループは、条件が満たされるのをただ待つのではなく、繰り返す回数を設定したい時に使います。構文は次の通りです。

```
for ( [最初の状態] ; [終了条件] ; [回数] ) {
  実行したいコード
}
```

この例は、コードがどのように機能するのかを知る最も良い方法でしょう。新しいスクリプトを開いて、以下のコードを入力してください。

```
void setup() {
  size(500, 300);
  background(180);
  strokeWeight(4);
  strokeCap(SQUARE);
  for (int h = 10; h <= (height - 15); h+=10) {
    stroke(0, 255-h);
    line(10, h, width - 20, h);
    stroke(255, h);
    line(10, h+4, width - 20, h+4);
  }
}
```

線の末端をスクエアに

今回はフレームループは用いず、コードはたった1つのブロックからできています。forループの初期値として、まず繰り返しの変数hを10に設定します。この変数hがh <= (height-15)になるまで、繰り返し実行します(終了条件)。繰り返しが実行されるたび、hの値はh += 10によって10ずつ増加していきます。これはforループの()の中のコードが、28回実行され、そのたびにhが10、20、30…270、280と増加していくことを意味しています。

hの変数がこのパターンに従っていることがわかれば、あなたはこれをもとにさまざまな方法でforループを使えるようになります。あなたが描く直線は、変数hをy軸の値として使っているので、10ピクセルずつ下に描かれています。また、hが変化するにつれて線の透明度もまた変化し、黒い線は次第に明るくなり、逆に白い線は暗くなっていきます(図2.10参照)。

キーワードbreak;を使って、コードの繰り返しを終わらせることができます。例えば黒い線が完全に見えなくなる(そして白い線が完全に見える)と繰り返しを止めたい場合、コード内の繰り返しの部分に以下の行を加えてください。

```
if (h > 255) { break; }
```

図2.10　実行中のforループの表示。

さて新人プログラマーさん、ここまで頭をよく使ってたくさんの新しい概念を覚えましたね。この後の章でももう少し紹介しますが、ここで紹介したものはどれも必要不可欠なものばかりです。これらをマスターし、あなたがコーダーと呼ばれるのに必要な技術を獲得すれば、とっても退屈なディナーパーティーであっても、あなたのまわりに人が集まってくるようになるでしょう。

本書の後半に登場する山のようなプログラムは、あなたにとっては目新しいものかもしれませんが、これまでのように詳しい解説は行いません。もしあなたがまだ命令の実行順序のあたりで頭を抱えているのなら、Processing.org上にある無料のリソース（特に、Learningセクションにあるもの）を使って、このセクションで解説してきた概念を試すことをお勧めします。

次に、あなたがコーディングして作った作品でどんなことができるかを見てみましょう。

2.5　作品をセーブし、パブリッシュし、配信する

Processingの簡単な構文は、アマチュアプログラマーにとっては非常に重要なメリットです。習得するのにかかる時間も減らすことができます。さらにProcessingは、あなたの作品を簡単に公開することができ、世界と共有することができます。

あなたの作品を公開する前に、まずはコードを保存しましょう。保存用ボタンは、ツールバーの左から5番目の下向き矢印のアイコンです(図2.11参照)。最初に保存する時は、保存するフォルダの名前が聞かれます。スクリプト自体は「.pde」という拡張子がついたテキストファイルとして、同名のフォルダ内に保存されます。ファイルだけでなくフォルダが作られる理由は、取り込み(インポート)や書き出し(エクスポート)をする時のわかりやすさのためです。

図2.11 ツールバー

ファイルをロードしたり保存したりすればすぐに気づくと思いますが、Processingではフォルダ内のスクリプトやリソースの束を「sketch(スケッチ)」という用語で呼んでいます。これはアーティストにアピールするためのものだと思いますが、私はこのメタファーは、どちらかと言うとむしろ後々制約になってしまうのではないかと思っています。ですから私がテキストファイルを「スクリプト」、そしてその束を「スクリプトフォルダ」と呼び続けたとしても、大目にみてください。そうすれば、あなたが他のプログラミング言語によくあるコンセプトについて話している時でも、馬鹿みたいに思われないでしょう。

スクリプトフォルダを作ったら、さまざまな方法で目の前のディスプレイの出力ウィンドウから、あなたが作ったピクセル画像を解き放すことができます。まず試みて欲しいのは、Exportボタン(いちばん右の矢印のボタン)をクリックすることです[現在のバージョン2.2.1では、ExportボタンはJavaアプレットではなく、アプリケーションの出力のためのボタンになっている。Processingの開発チームもアプレット出力に対するサポートは打ち切っているため、Webへのアップロードは現在ではProcessing.jsを利用する方が主流となっている]。スクリプトフォルダの中に、Webブラウザでスケッチを動かすために必要なものをすべてを含んだ「アプレット」と呼ばれるフォルダが作成されます(図2.12参照)。もしあなたがブラウザでindex.htmlファイルを開いたら、そこにあなたの制作物が現れます。今すぐに、Webにアップロードすることができるのです(もしあなたがそれを望むなら)。

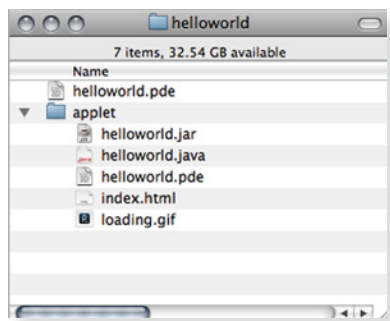


図2.12
スクリプトフォルダ(sketch)の中には、スクリプト(.pde)と、書き出されたアプレットフォルダがあります。

> **注意：JavaとWeb**
>
> アプレットをユーザーのブラウザで動作させるためには、ブラウザに最新のバージョンのJavaがインストールされてなくてはなりません。WebブラウザのJavaは、Flash Playerのようにどこでもダウンロードできるわけではありませんし、インストールされているJavaのバージョンが古くても、自動アップグレードしてくれません。Javaを使ってWeb上に公開することが、よくトラブルの種になります。

デスクトップ上で実行できるアプリケーションを生成するために、File > Exportを選択してください。これによってOS X、Linux、Windows、あるいはこれらすべてのOS上で動く、スタンドアローンのアプリケーションを作成することができます。実行するのに必要なすべてのファイルが入ったフォルダが、あなたのスクリプトフォルダに作られます。

2.5.1 バージョン管理

ツールバーの上のSaveボタンの便利さを語っておきながら、今度は逆に、それを使うのをすぐにやめることをお勧めします。作品を保存、保管するために、もっと便利な機能「Save As...（名前を付けて保存）」が用意されているのです。Save AsはFileメニューからのみアクセス可能です（Macではキーボードのショートカット「⌘+ shift + S」でも可）。Saveボタンの代わりにSave Asを使うと、保存するたびにいつもファイル名の入力を促されることになります。そこで、例えば「HelloWorld_1」「HelloWorld_2」という感じで異なるファイル名をつけていくことで、コーディングを進めていく過程においてバージョンアップの足跡を残せるのです。

このテクニックは「バージョン管理」と呼ばれ、あなたの作品を保管するのにとても役立ちます（これは常に以前のバージョンにたち戻ることができることを意味します）。それはまた、心理的にも安心感を与えます。作品のコードを大胆に改編して新しい可能性を試みたり、馬鹿げたアイデアを試しても、致命的なトラブルを起こすことに頭を悩まさずに済み、あなたの制作をより自由にします。

これは、遊園地のバンパー・カー［お互いに他とぶつかってもいい乗り物］を運転している時と同様に、どんなに激しくハンドルを操っても、誰も傷つけない安全さを感じているのと同じことで、気づかないうちにあなたのコーディングをもっと大胆にし、より信頼のあるものにします。ファミリー用のセダンを運転するよりも、バンパー・カーのほうが、（良い意味で）よりいっそう無謀なドライブができます。

バージョン管理の実践によって、コードの深刻なトラブルを再検証してみるとか、謎の脱線を追跡することができるようになります。バージョンごとにファイルを保存したとしても、スク

リプトはただのテキストファイル（せいぜい数KBの容量）なので、それによってはあなたのディスクがいっぱいになるようなことはありません。

コーディングを始めたばかりのあなたであれば、この驚くべき機能にまずびっくりすると思います。さらに、他の洗練されたプログラミング言語の場合はこんなに簡単にバージョン管理できないと知ったら、あなたはさらに驚くことでしょう（通常のプログラミング言語の場合、依存関係にあるファイルがたくさんありすぎるからです）。言語の違いは、ある部分では好みの違いを生み出します。ある人たちは、規則正しい構造化されたアプローチを奨励します。その一方で、他の人たちはProcessingのような（何はともあれ、ネイティブなPDE形式の）クリエイティブな方法に共感しています。

2.5.2　静止画を作る

ジェネラティブ・アートが最終的に行きつくところは、アプリケーションやアニメーションだけに限らないし、アウトプットの形式はコンピュータの画面だけではありません。あなたは他のさまざまな出力にも興味を持っていることでしょう。

アプリケーションから静止画をキャプチャするのは比較的簡単です。出力ウィンドウ上で実行されているアプリケーションのスクリーンショットを、さまざまなフォーマットで出力することができます。コードに次の関数を加えると、その時点でキャプチャを撮り、スクリプトフォルダに画像を保存します。

```
saveFrame("screen-####.jpg");
```

Processingでは、ファイルを保存する時に#####の部分がフレーム番号に置き換えられるので、画像がフレームごとに上書き保存されないようになっています。また、.png、.tif、.tgaといったフォーマットでの保存もサポートされています。そのためには、ただファイル名パラメータで拡張子を変えるだけで実現できます。同様に、ファイル名にフォルダパスを加えることもできます。

もしあなたがsaveFrame関数をdraw()のフレームループ内に書いて、さらにハッシュ構文（####）をファイル名に使ったら、連続した数がついた画像ファイルがあなたのスクリプトフォルダをいっぱいにしてしまうでしょう。1点の画像を必要としているのなら、どこかで画像を選別しなければなりません。そのためには、あなたは「ある基準に合う時にスナップを撮る」という条件文（if文）を書かなければなりません。キャプチャした画像からコンピュータが自動的に良いものを選び出すには、かなり複雑なロジックが必要なので、完璧な静止画をキャプチャするためにはあなたの目に頼るのが最良の方法です。そのためには、この関数を加えます。

```
void keyPressed() {
  if (keyCode == ENTER) {
    saveFrame("screen-####.jpg");
  }
}
```

keyPressed()関数は、キーが押されてた時にいつでも呼び出すことができる、組み込みのイベント関数です。enterキーが押されたかどうか調べて、押されている時に画面をキャプチャし、スクリプトフォルダに保存します。

2.5.3　代替イメージとして静止画を使う

もしあなたがWebサイトで公開するための作品を制作しているなら、ちょっとしたコツがあります。図2.12のようにアプレットを書き出した場合、loading.gifという画像があることに気づくはずです。これはWebブラウザがアプリケーションをロードしている間、あるいはロードし損ねた場合に表示する画像です。つまり、もしユーザーのJavaの互換性がない場合、loading.gifが表示されることになります。Webにあなたの作品を配信する際に、この画像をロードしている作品の静止画と取り替えると、ちょっといい感じになります。これは生成されたindex.htmlファイルを、少しだけ修正することで実現できます。まずはこの行を探してください。

```
<param name="image" value="loading.gif" />
```

もしあなたがloading.gifをオリジナルの作品の画像に置き換えれば、Javaアプリケーションが実行できないユーザーに対する安全策になります。完成した作品を見ることができなくとも、彼らはあなたの作品の静止画を見ることができます。

次は、映像の出力形式についてです。

2.5.4　ビデオを作る

draw()ループの中のsaveFrame関数によって、アプリケーションを実行した際のフレームごとにスクリーンショットを撮影したために、フォルダが画像でいっぱいになってしまいました。あなたがハッシュ構文を使っていれば、画像のファイル名に連続した番号がついています。これは、QuickTime Proのような、連続するイメージ列から動画ファイルを作る際に役立ちます(QuickTime ProのFileメニューから使用可能)。

ビデオを作れば、あとはコンピュータの処理能力や滑らかなフレームレートが維持されるかどうかを心配する必要はありません。たとえあなたがframeRateパラメータを使ってフレーム

レートを指定したとしても、sketchはそれを、あなたのマシンで実現可能な速さで実行するだけです。もしあなたが、プロセッサに重い負荷をかけるフラクタル生成アルゴリズムを実行したとしたら（本書のPart3でやります。本当ですよ）、アプリケーションはただのろのろと動くだけです。けれども、もしあなたが最終的に映像を制作したいのであれば、各フレームを連続した画像として出力することで、そこから動画ファイルを作成する時のフレームレートだけを考慮していればよくなります。たとえ画像を生成するのに多くの時間を費やしていたとしても、映像はあなたが設定した任意の速さで再生できます。

映像作品を作る際、各フレームのレンダリングは専用のマシンに一晩中まかせて、翌日、生成されたファイルを読み込んで確認するという手順は、映像制作の現場では当たり前のことです。こうすることで、ソフトウェアの作業におけるフィードバックの即時性が失われてしまいます。けれどもこの体験を通して、パンチカードやメインフレーム（大型コンピュータ）でプログラミングを書いていた前世代のコンピュータ科学者たちに対して、あなたは感謝と尊敬の気持ちを持ち始めるでしょう。より複雑な映像を制作するためには、（完成作品を単純化あるいは抽象化して）ワイヤーフレームで表示できるシステムを開発し、最終的にレンダリングする時だけ、精度を上げたり、テクスチャを使用できるようにします。

もしあなたがQuickTime Pro（非常に使えます）や、画像ファイルから映像を作るようなアプリケーションを持っていないなら、ProcessingのVideoライブラリを使ってQuickTimeムービーを作ることができます。またWebカメラを入力装置として使うことも可能です。さまざまな用途に使える有益なProcessingのライブラリについては、数が多過ぎてこの本ではカバーできません。しかし、それらのライブラリを利用するのはそれほど難しくありません。www.processing.org/reference/libraries/video/にあるVideoライブラリの機能を見てください。

2.5.5　フレームレートと画面サイズ

あなたが指定すれば、Processingはどんなサイズの画面でも取り込むので、どんなサイズの映像も制作できます。同様に、連続した画像を読み込む時のフレームレートも自由に設定できます。けれども、Webや放送で使用されている映像にはさまざまな規格があります。

コンピュータモニタやテレビのフレームレートは25fps（1秒あたりのフレーム数）が最もよく使われ、アメリカでは24fpsや30fpsも使用されています（これが、ビデオの24fpsのNTSCフォーマットが標準になっているアメリカと、25fpsのPAL/SECAMが標準である他の地域との違いです）。50～60fpsあたりが、人間の眼がフレームを見ているのがわからなくなるフリッカー融合（flicker-fusion）の限界であると言われているので、HDTVは50あるいは60fps以上のフレームレートを採用しています。黒と白のフレームが交互に現れる映像の場合、フレームレートが50

～60fps以下であればフリッカーに見えるでしょうし、それ以上になればただのグレー画面になります。ローファイなちらつきも、ある程度であればそんなに目に心地悪いものではなく、アニメーションはフレームレート12あたりでも滑らかに見えます。

画面の大きさについてはワイドスクリーンフォーマットが最も一般的で、（1366×768あるいは1280×720のような）通常16：9の比率になっています。最良の方法は、どこであなたの映像を見せるかを考えることです。ギャラリーの壁に作品を投影する時と、携帯電話のためにデザインする時では、間違いなくスクリーンの比率は異なるでしょう。

あなたがWeb、スクリーン、テレビのためにビデオを制作するなら、16：9という比率は既にいたるところで用いられています。例えば、YouTubeは現在1920×1080あるいは1280×720の解像度を推奨しています。もしあなたがYouTubeにアップロードする際に16：9の比率映像を用いないなら、画面に合わせるために、あなたのムービーに黒いボーダーが加えられてしまいます。しかし、Vimeoの場合はかなり違います。現在は640×480が標準規格で、加えて853×480のワイドスクリーンDVと、1280×720HDの規格をサポートしています。

2.5.6 モバイル装置、iPhone / iPadとAndroid

既にあなたもご存知の通り、ProcessingはJavaに基づいています。そして、たいていの携帯電話はJavaを実行することができます。だとすれば、Processingで携帯電話アプリケーションを作ることができるのでしょうか？ まあ、言うなれば、答えはイエスです。

グーグルのAndroidオペレーティングシステムを使って携帯電話用のアプリを開発するのは、それほど簡単ではありません。でもProcessingを使えば、デスクトップ上にアプリケーションを書き出すのと同じぐらい簡単に、Androidエミュレータや携帯電話やタブレットデバイスに、あなたのsketchを書き出せるのです。ProcessingのIDE（統合開発環境）のリリース1.5から、「Standard」と「Android」の間でビューを切り替える新しいボタンがツールバーに加わりました［現在はJAVAモードになっている］。Androidモードを選べば、接続したAndroid端末でプログラムを実行したり、アプリを書き出すことができるようになります。この機能を使うためには、あなたのコンピュータにAndroid SDKをインストールする必要があります。Android SDKのセットアップ方法はhttp://wiki.processing.org/w/Androidにまとめられています。また、creativeapplications.netには、初めてのAndroidアプリケーション制作*2とAndroidマーケットへの公開*3について解説している、2つの素晴らしいチュートリアルがあります。それぞれジャー・ソープとジェローム・サンクレアが解説しています。

もしあなたが、古い携帯電話を含むさまざまな端末をターゲットにした場合も、Processingは

有効です。しかし、Androidのように簡単にはいきません。最近の携帯電話でJavaが動くものは、Java Micro Edition（Java ME）という特別な仕様があって、Processingもそれに対応しています。Mobile Processingはライブラリではなくて、完全に別個のアプリケーションです。それはモバイル開発に特化していて、余分な機能を省いたProcessingのサブセットで、モバイル機器のタッチスクリーンを使ったインタラクションやメモリ管理など、特別な機能を備えています。Mobile ProcessingはProcessingとよく似たIDEを持っています。特殊なミニプラットフォームを開発するために必要な、電話のエミュレータであるwireless toolkit（WTK）をダウンロードする必要があります。

ここではMobile Processingについて、これ以上の詳細は述べません。でもこれだけは言っておきますが、あなたが目標としたいプラットフォームがモバイル機器であるなら、この本の内容のほとんどが、Processingと同じくらい簡単にMobile Processingにも応用できます。モバイル開発を始めるためにはhttp://mobile.processing.org/download/から、Mobile ProcessingのIDEとWTKをダウンロードしましょう。Processingと同様に、多くのチュートリアルとリファレンスが掲載され、最初の一歩からきちんとサポートされていています。

しかし、iPhoneとiPadはまた別の話です。iPhoneがイギリスで発売された時、“iPhoneは「すべてのインターネット」にアクセスできる”というApple社の広告を、広告規制局が禁止しました。その理由として、この装置でJavaやFlashが動かないなら、「すべてのインターネット」をカバーするとは言えないからです。執筆の時点で、JavaとFlashはiOS上でもWebブラウザ内でもiPhone/iPadの上では動きません。アップルのプラットフォーム用にアプリケーションを開発するためには、Objective-Cを学んで、AppStoreでお金を払う必要があります。

この状況はそのうち変わるかもしれません（私の願いでもありますが）。けれども本書の執筆時点では、iPhoneとiPad、またApp Storeの大成功によって、Appleは他の技術を使ってアプリの開発をできるようにする気がありません。FlashがiPhoneアプリケーションを作成するコンパイラを作ったこともあったのですが、Apple社の規定によって使用が禁止されました。同様に、Processingの文法でiPhoneアプリが作成できるiProcessing（http://luckybite.com/iprocessing/）もありましたが、ある言語から別の言語へという非ネイティブ変換は、ネイティブのObjective-Cアプリケーションとはパフォーマンスの点で比較できるものではありませんでした。

＊2　http://www.creativeapplications.net/android/mobile-app-development-processing-android-tutorial/
＊3　http://www.creativeapplications.net/android/from-processing-to-the-android-market-tutorial/

もしあなたがiPhone上でProcessingによる作品を見たい時に、手短かで簡単な、素晴らしい解決方法が1つあります。以前にJavaScriptのライブラリProcessing.jsを紹介しましたね。これがなかなかたいしたものなのです。HTML5のCanvas要素を使って、Webブラウザに直接形を描くことができます。Mobile Processingのように、Processingの関数のサブセットをサポートしていますが、JavaScriptとProcessingの構文はよく似ているので、コードを混ぜることができるという強みがあります［現在のバージョンではPDEにJavaScriptモードが搭載されており、簡単にProcessing.jsが扱えるようになっている。Processing2.2.1の場合は、モード選択ボタンから「Add Mode...」を選択し、JavaScriptモードをインストールすることで利用できる］。

www.processingjs.orgから、Processing.jsのダウンロードと導入ができます。これはあなたのWebページのHTMLに加える数百行のJavaScriptプログラムであり、このライブラリを自分のWebページの中で動くようにするためには、多少経験が必要です。もしあなたにそうしたスキルがなければ、ロバート・オルークによって開発されたオンラインのインターフェースのHasCanvasを経由してProcessing.jsを使うことができます。そうすることで、HTML5のCanvasタグにリアルタイムに直接プログラムを書くことができるようになります。HasCanvasは、www.hascanvas.comから利用できます。Processing.jsを使うことで、Webが使える端末さえあればいつでもどこでも、あなたの作品を見ることができます。

2.6 まとめ

以上でProcessing入門は終了です。今、あなたは、熱く小さな手の中にしっかりとツールを握り締め、クリエイティブなコーディングを始める準備ができました。あなたはプログラムでの描画の基本と、Web、プリント、ビデオ、モバイル端末などの、さまざまなメディアに書き出すための知識があります。そしてProcessingだけではなく、どんなプログラミング言語にとっても役に立つプログラミングの原則を一通り学びました。

続いては、ジェネラティブなスタイルのアート作品の制作方法に焦点を当てていきます。Chapter 3では、ランダム性、ノイズ、そして鉛筆があなたの指から滑り落ちていくような、正しいコンピュータの使用方法を超える方法に取り組みます。

Part 2 ランダム性とノイズ

“人々が、数学が単純であると思えないのは、
私たちの生活がどれぐらい複雑であるかに
気がついていないからなのです。”

ジョン・フォン・ノイマン, 1947年の談話

Chapter 3

線を引く間違った方法

The Wrong Way to Draw a Line

コンピュータの方が優れていることが1つあるとすれば、それはとても正確に命令に従ってくれることです。それが、デザイン、建築、製造、科学、そして無数のジャンルに革命が起きた理由の1つです。

けれどもその正確さは、ジェネラティブ・アートの文脈においてはごく限られた面白さでしかありません。自然界には存在しない完璧な正確さ、そこに喜びはありません。もし私たちのアートがそのような完璧さを求めるのであれば、それはあまりに機械的なものとして捨て去られてしまうでしょう。

反対に、私たちは不完全で予測不可能な形状を望んでいるのかもしれません。しかし、だからといって不明瞭であてもなくさまようような不完全さもいりません。私たちは、有機性（オーガニック）と機械性（メカニカル）の間、カオスと秩序の間の、ちょうど良くバランスがとれたスイートスポットを探しています。これを見つけるために、プログラム的な考え方で効率的に線を描く方法から離れて、決定論とは違ったプロセスでものごとを進めていく必要があります。

この章と次の章では、基本となるプロセスを分解して、そこに多少のカオスを加えていく、私のプログラミング・アートに対する独自のアプローチと、その背景にある基本原則を説明します。マシンによって線を描く「正しい」方法を使えば、常に点Aから点Bへ正確かつ効率的に線を描くことができます。けれどもアートの見地からすれば、「間違った」方法で描くことのほうが面白いことが多いのです。

3.1　デタラメさとそうでもないこと

あらゆるジェネラティブ・アートに、何らかのランダム性が必要です。ランダム性なしに予測不可能なことってあり得ないでしょう？　しかしこの発言は、厳密には正しくはありません。人間の計算能力を超えたアイデアをかたちにするのに、乱数を生成する関数が常に必要というわけではありません。複雑な数式も常に（同じくらいの複雑さを持ったコンピュータなら）予測可能な値を返してくれますが、それは人間が予測できるものを超えているかもしれません。

他のどんなプログラミング言語も、Processingのrandom関数と同じような機能をもっています。しかし、手当たり次第にデタラメな数値を返すことは、機械には不可能です。コンピュータによって生み出されたランダム性は、数式とコンピュータの計算能力を使って、2つとして同じ計算結果にならないと保証している擬似ランダムです。これらの関数の背後にある数式は、それ自身アートと言えるでしょう。そして数式によって戻ってきた乱数値は、私たち馬鹿な人間をだますのには十分です。

私がとりあげた「擬似ランダム」は、主にデータ暗号化のために利用されるので、コンピュータサイエンスにおける重要な問題の1つです。もしあなたが本当のランダム性にこだわりたいというのであれば、ダブリンにあるトリニティ・カレッジのコンピュータサイエンス専攻が運営するrandom.orgプロジェクトを見てください。きっと気に入ることでしょう。ここには、大気中のノイズを基礎にした「本当の」乱数を返すオンラインのAPIがあります。ただし、ここから先は、カオス数学の分野における物議のある量子予測性あるいは自然界の予測不可能さの議論に入り込んでしまいます。

ramdomの使い方は単純です。random関数は一般的に0から1までの範囲で値を返します。さらに、返す値の範囲を指定することもできます。Processingでは、最大値を与える1つのパラメータ、あるいは一定の範囲内で値を返す2つのパラメータを設定します。例えばもし20から480までの範囲の値がほしい場合、このように書きます。

```
float randnum = random(460) + 20;
```

あるいはこのように書くこともできます。

```
float randnum = random(20, 480);
```

指定した範囲が明白ではないのですが、私は最初の表記方法が好きです。ここからは、0と1間の値を返すだけの関数を使います。Processingのように、乱数の範囲を指定できる便利な記述法が必ずしも用意されているとは限りません。そこで次のように、パラメータを用いずに

乱数の値の範囲を表現してみます。

```
float randnum = (random(1) * (max-min)) + min;
```

さて、線を引くのにこれを使ってみましょう。点Aと点Bをつなぐ線を引くのは簡単です(図3.1参照)。

図3.1　線を引く正しい方法。

```
size(500,100);
background(255);
strokeWeight(5);
smooth();

stroke(20, 50, 70);
line(20,50,480,50);
```

パラメータを置き換えれば、ランダムに線を描画することができます。幅が500で高さが100だとわかっていたとしても、ここでwidthとheightの定数を使います。そうすれば、もし後でwidthとheightを変更してもスクリプトがうまく対応してくれます。たとえあなたが後でwidthとheightを変更しなくても、こう書くことを勧めます。

```
stroke(0, 30);
line(20,50,480,50);

stroke(20, 50, 70);
float randx = random(width);
float randy = random(height);
line(20,50,randx, randy);
```

図3.2　どちらかと言うと面白くない、ランダムな線。

図3.2はアウトプットの一例です。この、あなたの最初のジェネラティブな作品についてあまり決めつけたくはないのですが、どちらかと言うと面白くないですね。申し訳ないですが、私は同じことはしたくありません。機械の描画の方法に捕われていて、線がしっかりし過ぎています。これを乗り越えるために、私たちはプログラミング言語の便利さから離れて、線の描き方を少し解体してみる必要があります。

3.2　変化の繰り返し

私たちは、コンピュータが最も効率的なやり方で点Aから点Bに線を引くことを知っていますが、効率が必ずしも有用であり美的であるとは限りません。もっと面白くするために、線を引く「最も良い」方法からいったん離れて、最適とは言えない別の方法を探求してみましょう。まずは線を引く一連のプロセスを、細かな線分に分解してみます。

次のコードは、xの位置を10ピクセルずつ動かしながら繰り返すforループを使って、xとyの値を更新しながら、ミクロな線を連続的に描いていきます。この線は、46の線分の集まりです。出力ウィンドウに表示されるイメージは、図3.1（ポイント20～50、480～20までの線）とまったく同じものに見えるでしょう。けれどもここでは、もう少し多くのコードを使っています。

```
int step = 10;
float lastx = -999;
float lasty = -999;
float y = 50;
int border = 20;
for (int x=border; x<=width-border; x+=step) {
  if (lastx > -999) {
    line(x, y, lastx, lasty);
  }
  lastx = x;
  lasty = y;
}
```

もし、あなたが道路を横切ろうとする年配の婦人の手助けをしようとして、婦人に歩き方を「あなたの左足を数センチメートル動かします。それからあなたの右足を…」と指図したとします。年配の婦人はおそらく、あなたを頭のおかしい人だと思い、自分の財布を点検するでしょう。しかし、コンピュータのプログラムを見下してはいけません。信じてください。私は試してみたのです。無意味で、退屈で、そして見たところ勝手気ままな命令であっても、それが完璧であれば、忠実に実行します。私たちのマシンは愚かな命令であっても、それが「良い」コードであるかのように、滑らかに効率良く実行してくれます。

この短く区切られた描画方法は、あなたが描いた線にランダム性の面白さを加えることができます。1ステップごとにyの値を変える命令を加えてみてください。

```
int step = 10;
float lastx = -999;
float lasty = -999;
float y = 50;
int borderx=20;
int bordery=10;
for (int x=borderx; x<=width-borderx; x+=step) {
  y = bordery + random(height - 2*bordery);
  if (lastx > -999) {
    line(x, y, lastx, lasty);
  }
  lastx = x;
  lasty = y;
}
```

ループの最初の行は、10と90の値の範囲でyにランダムな値を加えます。結果は図3.3に示された通りです。

図3.3　yの位置にランダムさを加える。

いとも簡単に、予測不可能な線へと発展できました。あなたがスクリプトを実行するたびに、いつも線の形が変化します。でもきっと、あなたはカオスの方向性に迷い、とまどっていることでしょう。そこで、ランダム性を少しずつ制御していきましょう。次の例では、ランダム値をyの値に加算または減算することで、もっと自然な変化を実現しています。

```
float xstep = 10;
float ystep = 10;
float lastx = 20;
float lasty = 50;
float y = 50;
for (int x=20; x<=480; x+=xstep) {
  ystep = random(20) - 10; //range -10 to 10
  y += ystep;
  line(x, y, lastx, lasty);
  lastx = x;
  lasty = y;
}
```

結果は図3.4のようになりました。

最初よりも良くなってきました。そう、こんなボロボロでガタガタの線でも、私の文脈においては「もっと良い」と呼ぶことができます。この線はそれほどうまく制御されているようには見えません。しかし、それでもまだ役に立たないほどカオス的ではありません。

あなたはこれまで1本の線だけを描いてきましたが、既にアナログの感触がアウトプットに徐々に表れているのがわかるはずです。これをもとに少しずつ発展させていったり、あるいはこの方法を使って他のさまざまな形を描くこともできます。しかしまず、あなたの線に自然な形状を与えるもっと簡単な方法をお見せしたいと思います。randomが使えるところであればどこでも、次のnoiseを用いることができます。

図3.4　ランダムな変動は、まったくのデタラメさより良さそうに見えます。

3.3　自然な変動

あなたの母親でも知っているような、有名なプログラミングの賞はありません。ですから、ケン・パーリンが開発したアルゴリズムがオスカー（映画芸術科学アカデミー賞）を受賞して、家を買うことができた日、ケンの母親は友達に電話をかけまくって、息子が計算機と親しくすること、プログラムに人生を捧げる決断をしたことを讃えたのは、ごくまれなケースと言えるでしょう。

ケンが（やや遅れて）1997年に受賞したテクニカル・オスカー賞は、SF映画『トロン』（1982年）に用いられたグラフィックス、彼によるプロシージャルな（アルゴリズミックな）テクスチャ開発の業績のたまものでした。『トロン』は、テーマと作り方の双方が先駆的な映画でした。ビデオゲームがばかばかしくて軽薄なものとして見られていた時代に、ゲーマーたちの英雄になりました。ウィリアム・ギブソンの小説『ニューロマンサー』でサイバースペースという概念が広まった2年も前に、バーチャル・リアリティを舞台にした映画が撮影されたのです。けれども最も重要なことは、それがコンピュータグラフィックスによる特殊効果を大々的に用いた最初の映画だったということです。

ケンの仕事は、3Dのオブジェクトを自然に見せるためのテクスチャの開発でした。私たちが前のセクションでやっていたように、彼は制御された中にランダム性を生成する方法を探していました。彼がこれを実現するために思いついた関数が「パーリンノイズ（Perlin noise）」です。これは1つの擬似ランダム関数にしかすぎませんが、非常に優れたアルゴリズムで、特に自然なビジュアルを制作するのに向いています。もしあなたがその背後にあるロジックに興味を持ったなら、彼のWebサイト（http://mrl.nyu.edu/~perlin/）上で、ケンによる応用例と共にそのコードを見ることができます。

1980年代後半まで、ケンのパーリンノイズ関数はどこにでも使われていました。今までグラフィカルなゲーム機でゲームをしたり、あるいはCGアニメ映画を見たり、あるいはここ20年間にCGIを使って制作された映画を見ているとしたら、間違いなくケンによる作品を見たことがあると言えるでしょう。現在Processingでは、ケンのアルゴリズムをシンプルにカプセル化した関数――noiseを用いて、大スクリーンで見たのと同じような自然な変動を自分の創作に取り入れることができます。

3.3.1　Processingでのパーリンノイズ

noise関数は、一般に数列を作るために使われます。それに一連の乱数種（シード値）を渡すと、

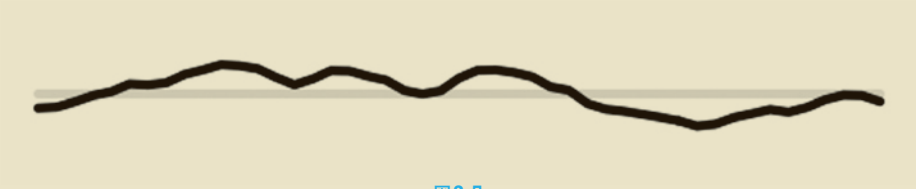


図3.5
パーリンノイズ。リスト3.1のアウトプットの一例。

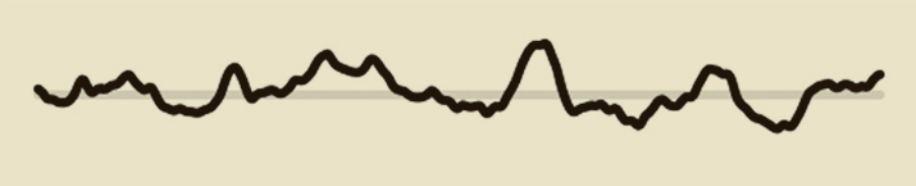


図3.6
xピクセルごとに、**ynoise**の増分を0.03にした時のパーリンノイズ。

そのそれぞれの値によって0と1の間のノイズを返します。数列をどの種から始めても問題ありませんが、しかしスタート地点（乱数種）をランダムにすることをお勧めします。さもなければ、結果として生じる乱数が毎回同じになってしまいます。種の増分（各回ごとの値の変化）を0と0.1の間くらいの小さな値にするのが、最も良い結果を生み出します。

おそらく、noiseの正しい評価を得る最も良い方法は、それを実際に使ってみることです。リスト3.1は、前のセクションのスクリプトのrandomをnoiseに置き換えたものです。

リスト3.1　パーリンノイズ

```
size(500,100);
background(255);
strokeWeight(5);
smooth();

stroke(0, 30);
line(20,50,480,50);

stroke(20, 50, 70);
int step = 10;
float lastx = -999;
float lasty = -999;
float ynoise = random(10);
float y;
for (int x=20; x<=480; x+=step) {
  y = 10 + noise(ynoise) * 80;
  if (lastx > -999) {
    line(x, y, lastx, lasty);
  }
  lastx = x;
  lasty = y;
  ynoise += 0.1;
}
```

10と90の間に

この例では、乱数列を0から10の間のどこか（random(10)）から始めて、繰り返しごとにその値を0.1ずつ増加させています。シーケンス内でさまざまなシード値を与えてnoise()を呼び出すと、0と1の間にある小数を返します。それに80を掛け、10を加えることで、yの値を10と90の有効な範囲内にします。その結果（可能性の1つですが）は、図3.5のようになります。

より滑らかな結果を得るために、あなたはxのステップをたったの1に減らして、線は1ピクセルずつ水平に進み、それに応じてynoiseステップも減らします。図3.6は、ynoiseのステップ

を0.03にして、1ピクセルずつxステップを進めたものです。

これは便利な方法です。けれども、ケン・パーリンだけが面白い変化関数を書くことができるとは思わないでくださいね。

3.3.2　自分のノイズを作る

本章の初めに述べたように、noiseやrandom関数を呼び出す場合、その値はどこか天空から引っ張ってくるのではなく、関数は計算の結果を返すものです。乱数の場合、この計算はとても抽象的で、まったく予想できないように見えます。しかし、それでもそれはただの計算です。私は、この関数が線を面白く引くためのスパイス、しかもその唯一の計算方法だと思ってほしくはありません。あなたには、まだまだ探索すべき数学の世界があります。

それでは、いくつかの簡単な公式から始めてみましょう。noiseのように、三角関数のsin、cosは、あなたが渡す引数によってスムーズに変化する値を返します。これらの2つの数学的な関数に角度（ラジアン）を渡すと、関数は-1と1の間の値を返します。

これを可視化するために、もう一度この滑かな数学的関数を使って直線にアプローチしてみましょう。リスト3.2を実行してみましょう。

リスト3.2　サインカーブを描く

sinが-1から+1を返す

```
float xstep = 1;
float lastx = -999;
float lasty = -999;
float angle = 0;
float y = 50;
for (int x=20; x<=480; x+=xstep) {
  float rad = radians(angle);
  y = 50 + (sin(rad) * 40);
  if (lastx > -999) {
    line(x, y, lastx, lasty);
  }
  lastx = x;
  lasty = y;
  angle++;
}
```

このコードは図3.7に示されるカーブを作ります。

図3.7
サインカーブ

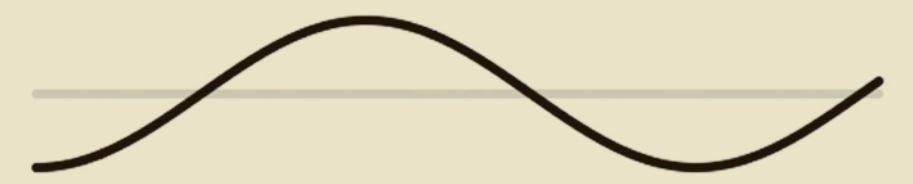


図3.8
コサインカーブ。サインとあまり変わらない。

図3.9
sinが返す値を3乗して作られた線。

図3.10
少量のnoiseを含んだsinの3乗。

図3.11
customRandom関数が実行中。randomほどランダムではありません。

sin関数をcos関数に置き換えても、結果はよく似ています（図3.8参照）。サインとコサインは、円周上の点を計算するのに使われます（次の章でさらに詳しく解説します）。ですので、それらが返す値がとても滑らかであることは驚くに値しません。

カオスを使った実験を始めると、この滑らかな予測可能性はあまり役に立たないように思えるかもしれません。しかしそれは、この計算があまりに単純過ぎるからです。randomやnoise関数の予測不可能性のように、あまり抽象的ではありません。

もしあなたがsinを3乗したら、どのように見えるでしょう（図3.9を参照）？

```
y = 50 + (pow(sin(rad), 3) * 30);
```

あるいはノイズを加えてみたらどうでしょう（図3.10を参照）？　だましているようにみえますか？

```
y = 50 + (pow(sin(rad), 3) * noise(rad*2) * 30);
```

これは基本原則です。いくつかの数学的な変換を混合することによって、より面白くてユニークな線を手に入れることができます。では、この原則を使っていきましょう。

3.3.3　自分だけのランダム関数

randomの問題点は、あなたがおわかりのように、それがただあまりにもランダム過ぎるということです。もしかしたら、あなたがこのデタラメさをもう少し予測可能なものに変えることができたら、それはもっと美的な価値を持つかもしれません。

まず、自分だけの関数を作ってみましょう。

```
float customRandom() {
  float retValue = 1 - pow(random(1), 5);
  return retValue;
}
```

そして、yの位置を計算する時にこの関数を使ってみます［関数を使う際にエラーが出る場合は、これまでに書いたコードをsetup関数の中に入れ込んで実行し直すと良い］。

```
y = 20 + (customRandom() * 60);
```

あなたがこの関数で行ったことは、乱数を5乗にしたことです（図3.11を参照）。乱数を2乗すれば、その戻り値は、もっと0に近づく傾向があります。もしそれを2分の1乗（ルート）にし

たら、値は1に近づきます。5のべき乗に（あるいは、逆に引き下げる方向に、0.1はもう1つの例）にしたら、それほどランダムではない結果を手に入れることができます。

この関数をさらに変換すれば、別の実験をすることもできます。random関数をまったく使わなくとも、代わりにとても乱雑でランダムに見える計算を使ってもいいですし、他の擬似ランダム関数も同じように機能します。

3.4　まとめ

この章でやってきたことは、1つの線を描くさまざまな異なった方法を見てみることでした。Processingに組み込みのline関数の便利さに頼らずに、1ピクセルの線を繰り返し描画することによって、より面白い可能性を見つけることができます。このメカニズムを使って、機械的な完璧さから線を解放して、変化が生み出す魅力的な味わいを付け加えることができます。

次の章では、こうした直線を描くような単純な仕事をあえて台無しにするこの技術を、もう少し実質的なものに応用します。最もシンプルな形状である円をとりあげて、三角関数の楽しさを再発見する準備を始めましょう。

Chapter 4

円を描く間違った方法

The Wrong Way to Draw a Circle

たいていの人にとって、「三角関数」はちょっと恐い言葉です。実際に私も学校では、三角関数にとても手こずらされました。当時私は「こんなガラクタは実社会では決して使わない」と思っていました。実際それは、少なくとも数年の間は、その通りでした。

ところが20代も後半になると、私は生活のためにFlashゲームを作るようになっていました。すると、うろ覚えの三角関数が実は役に立つということがだんだんとわかるようになってきました。

この章では、円を描くことを自由な方法でやってみます。そのためには、少しばかり三角関数が必要になります。前の章でline関数の便利さに頼らずに線を描いたのと同じように、ここではProcessingのellipse関数に頼りません。その代わりに、数学を使って、この形状を描くもっと非機械的で不正確な方法を試してみましょう。私自身の作品の多くが、回転する描画アルゴリズムというアイデアをベースにしています。その1つをケーススタディとして、この章の後半でお見せしましょう。

4.1 回転するドローイング

私の青春期の貧弱な脳に、2年ぶんのAレベルの数学［英国の学力証明の一水準。Aレベルの試験は通常3科目行われ、A～Eまで5段階の合格基準がある。通常3科目ともC基準で合格することが大学入学許可の条件］を課しておきながら、Googleで少し検索してみれば、自分が必要とする三角関数はわずか1枚のポストカードの裏に簡単に収まってしまうくらいのものであることがわかりました。コーディングの思春期（まだそれが過ぎ去ったとは思っていませんが）に、私は図4.1の不可解な走り書きを残しました。私はこのカードをオフィスにある本に10年以上貼り付けていましたが、これ以上の知識が必要だったことはありません（貼り付けた本はよく変わりましたが）。

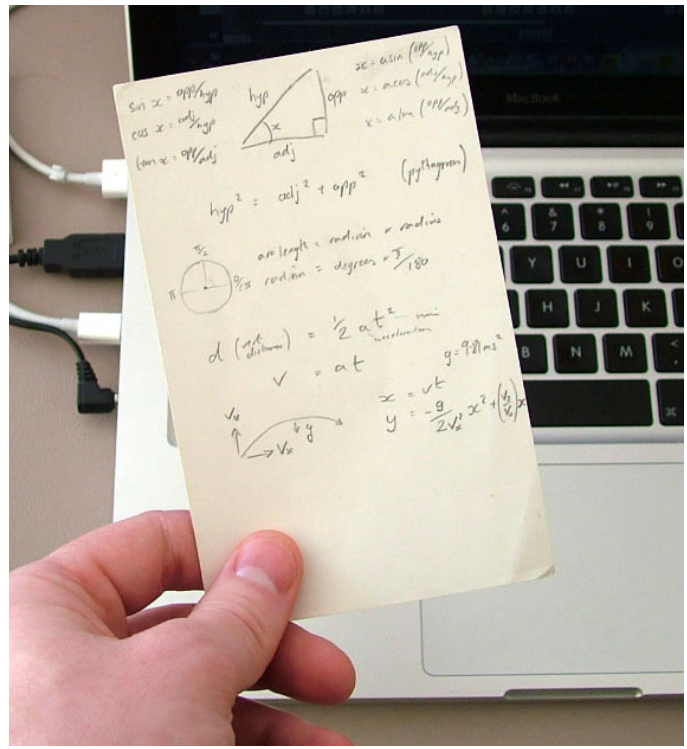


図4.1
これまで必要としたすべての三角関数は、ポストカードの裏にすっぽり。

さて、このカードの三角関数をちょっと使ってみます。まず線を分解したのと同じように、円を壊してみます。

4.1.1　初めて円を描く

円を描く簡単な方法から始めましょう。

```
ellipse(中心のx座標, 中心のy座標, 幅, 高さ);
```

この方法で円を描く時に必要なのは、中心の座標、幅、高さで、円の場合、幅と高さは共に直径に等しくなります。けれどもこの方法をちょっとだけ分解すれば、もっとずっと楽しむことができます。そのために、私のポストカード・カンニングペーパーの中から2つの式を使います。それは sin x = opp/hyp（対辺／斜辺）と cos x = adj/hyp（隣辺／斜辺）です。

円の中心がわかれば、中心と円周上の点を結ぶ斜辺は半径（radius）と等しくなります。対辺と隣辺は、中心からのx軸方向とy軸方向の距離に対応しています。方程式をいじくってみれば、

以下のように中心角（angle）に対応した、円周上のいかなる点の座標も計算できます。

```
x = centerx + (radius * cos(angle));
y = centery + (radius * sin(angle));
```

図4.2は、リスト4.1のコードで描画した円です。実行の仕方がよくわかるように、まずは繰り返しのステップごとに点を描いてマークしてみました。

閉じた円にするために、これらの点をつなげます。この例のように、5度ごとに描けば十分に滑らかですが、後でこの度数の増分をもっと減らすこともできます。

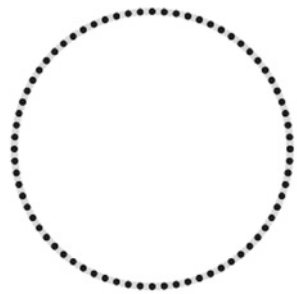


図4.2　三角関数を使って円を描く。リスト4.1のアウトプット。

ラジアンと度

三角関数で使われる角度は、「度」ではなく「ラジアン」です。1ラジアンは180/ π と等しく、円をフル回転（360度）すると2 π ラジアンになります。もしラジアンが苦手なら、普段は角度を使って、三角関数を呼び出す際に変換してもよいです。変換の公式は、私のポストカードに載っています：ラジアン ＝ 角度*（180/ π ）。とはいえProcessingには、既に組み込みの2つの便利な変換関数（**degree**と**radian**）があります。

リスト4.1 円を描く。最初に簡単な方法で、それから三角関数を使って

```
size(500,300);
background(255);
strokeWeight(5);
smooth();

float radius = 100;
int centx = 250;
int centy = 150;

stroke(0, 30);
noFill();
ellipse(centx,centy,radius*2,radius*2);

stroke(20, 50, 70);
float x, y;
float lastx = -999;
float lasty = -999;
for (float ang = 0; ang <= 360; ang += 5) {
  float rad = radians(ang);
  x = centx + (radius * cos(rad));
  y = centy + (radius * sin(rad));
  point(x,y);
}
```

360度のループ

度からラジアンへの変換

これが、あなたが必要なもののすべてです。非常に厳密な方法ですが、こうすれば、円をシステマチックに描くことができます。ここから、もっと自由にこのコードを改良していきましょう。

図4.3 ちょっとした工夫で、円がらせんになります。

4.1.2　円をらせんに変える

このコードを使えば、描画を自由にコントロールできるので、さまざまな工夫をすることができます。円をらせんに変えるためには、角度が回転するにつれて半径を増やしていけばいいだけです。ただ、360度よりたくさん回す必要があります。コードの変更箇所は、以下のリストで太字で強調されています。出力結果は図4.3 の通りです。

リスト4.2　円をらせんに変えるために必要なコードの変更箇所

```
radius = 10;                                          小さく始まる
float x, y;
float lastx = -999;
float lasty = -999;
for (float ang = 0; ang <= 1440; ang += 5) {          ループを4回
  radius += 0.5;
  float rad = radians(ang);
  x = centx + (radius * cos(rad));
  y = centy + (radius * sin(rad));
  if (lastx > -999) {
    line(x, y, lastx, lasty);
  }
  lastx = x;
  lasty = y;
  lasty = y;
}
```

まだ今のところはかなり退屈です。そこでノイズを加えてみます。

4.1.3　ノイズの多いらせん

リスト4.3は、前のように半径を増やしていくだけでなく、かなり大きなノイズで変動させます。結果の一例を図4.4に示します。

リスト4.3　らせんにノイズを加える

```
radius = 10;
float x, y;
float lastx = -999;
float lasty = -999;
float radiusNoise = random(10);
for (float ang = 0; ang <= 1440; ang += 5) {                    (続く)
```

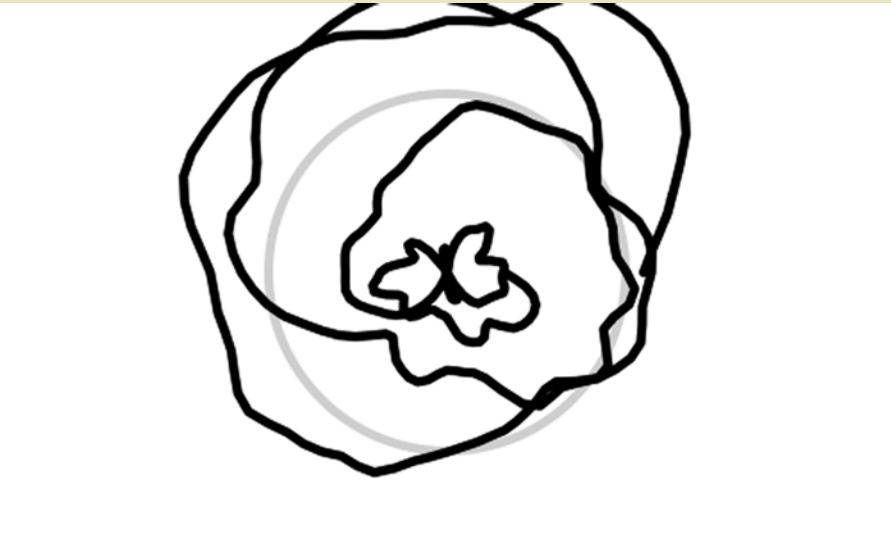


図4.4
ノイズを加えたらせん。真ん中に見えるのは蝶でしょうか？

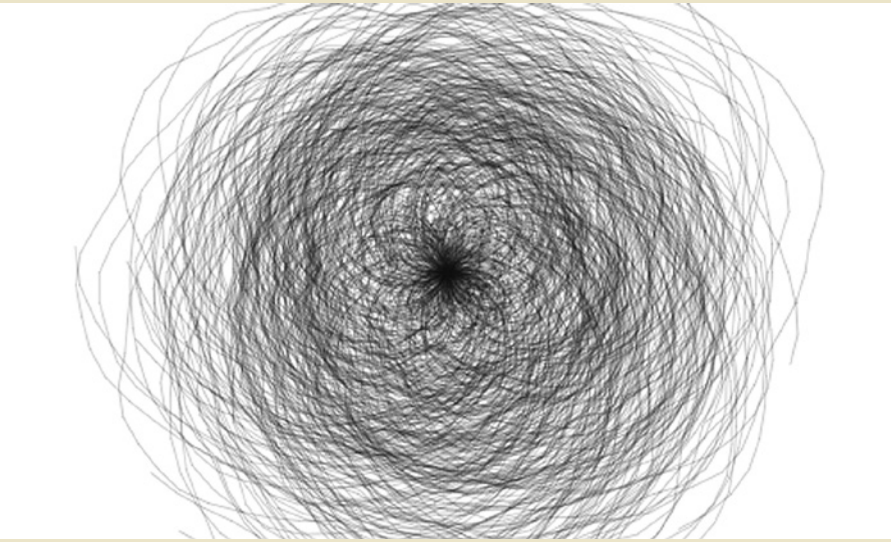


図4.5
らせんを思いっきり重ねる。

（リスト4.3の続き）

```
    radiusNoise += 0.05;
    radius += 0.5;
    float thisRadius = radius + (noise(radiusNoise) * 200) - 100;
    float rad = radians(ang);
    x = centx + (thisRadius * cos(rad));
    y = centy + (thisRadius * sin(rad));
    if (lastx > -999) {
      line(x, y, lastx, lasty);
    }
    lastx = x;
    lasty = y;
  }
```

あなたはとうとう、何か面白い、ジェネラティブと呼べるような何かが現れてくる段階に到達しつつあります。これらの形は、あなたがプログラムした数学的な種を基礎にしていますが、それぞれが独自の形を生み出せるくらい十分なランダムさがあります。

私が信じる、あなたの次のステップは、このすべてに100を掛けることです（図4.5を参照）。

このバージョンでは、同じらせんを100回描くのにforループを使っています。また、線幅を軽くして細くし、各らせんの線の色、アルファ値、開始角度、終了角度、ステップの角度をランダムにしています。コードは以下の通りです。

リスト4.4　ノイズを加えた100のらせん

```
size(500,300);
background(255);
strokeWeight(0.5);                                  より洗練された線
smooth();

int centx = 250;
int centy = 150;

float x, y;
for (int i = 0; i<100; i++) {                       100のらせんを描く
  float lastx = -999;
  float lasty = -999;
  float radiusNoise = random(10);
  float radius = 10;                                （続く）
```

らせんの
要素をランダムにする

（リスト4.4の続き）

```
  stroke(random(20), random(50), random(70), 80);

    int startangle = int(random(360));
    int endangle = 1440 + int(random(1440));
    int anglestep = 5 + int(random(3));
    for (float ang = startangle; ang <= endangle; ang += anglestep) {
      radiusNoise += 0.05;
      radius += 0.5;
      float thisRadius = radius + (noise(radiusNoise) * 200) - 100;
      float rad = radians(ang);
      x = centx + (thisRadius * cos(rad));
      y = centy + (thisRadius * sin(rad));
      if (lastx > -999) {
        line(x, y, lastx, lasty);
      }
      lastx = x;
      lasty = y;
    }
  }
```

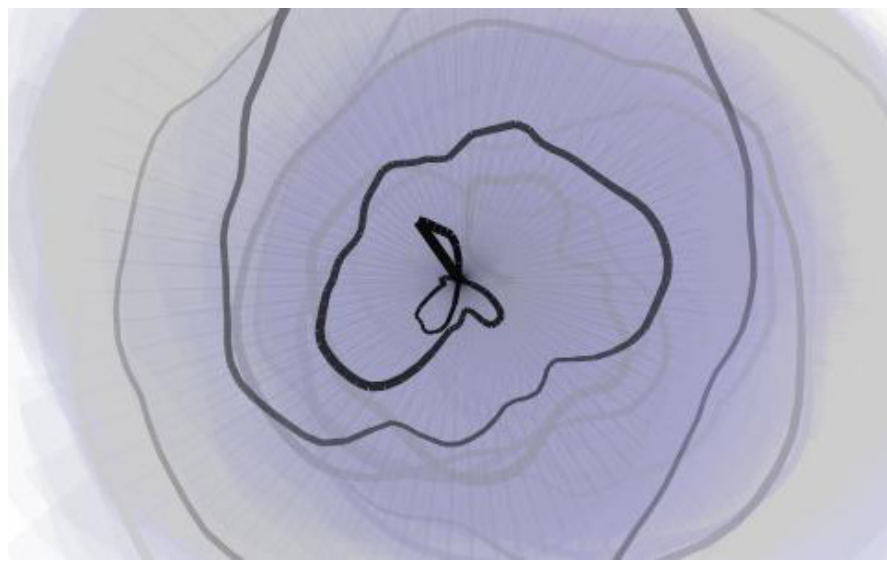


図4.6 Spiral Stairs（2009）

これはめちゃくちゃですが、しかし面白いめちゃくちゃさです。私はまだこれをお祝いのカードにプリントしませんが、これははじめの一歩です。体系的な描画プロセスにノイズやランダムを加えることで、どのように美的に面白い結果が導かれるかがわかってもらえることを願います（例えば図4.6を見てください）。

数学的に形を抽象化するプロセスを通して、秩序からカオスへと形を変化させていくことができます。あなたがこの連続体を無秩序な結果にし過ぎない限り、この抽象は面白い結果を生み出します。そしてもしそれが面白かったら、アートと呼ぶことができるでしょう。

4.1.4　自分のノイズを作る。ふたたび

パーリンノイズは確かに便利ですが、それにあまりにも依存してしまわないように、もう一度、円の描画を試みます。しかし今回は、あなた独自の自然な変化を作り出します。前章で、あなたは線を描画するのに自分独自のランダム関数を作りました。ここではさらに一歩進めて、渡されたシード値によってランダムな戻り値を計算する自分だけのノイズ関数を作ります。そしてこの関数を、円に適用してみましょう。

このために、最初にコードを少し整理します。リスト4.5は、リスト4.1から円の描画コードを取って、そして次のことをしています。

- 関数ブロックの中に円の描画部分を書き直す
- customNoise関数を加える

リスト4.5　独自のノイズ関数で描かれた円

```
void setup () {
  size(500,300);
  background(255);
  strokeWeight(5);
  smooth();

  float radius = 100;
  int centx = 250;
  int centy = 150;

  stroke(0, 30);
  noFill();
  ellipse(centx,centy,radius*2,radius*2);
```

（続く）

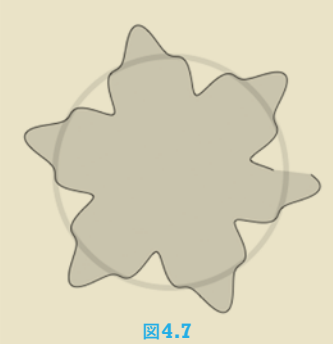


図4.7
リスト4.5のアウトプット。
サイン関数をもとにした、円周の独自な変動。

図4.8
サインの2乗をもとにした、もう1つの独自な変動。

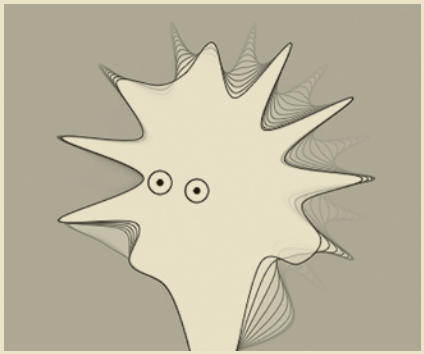


図4.9
このアニメーションのソースコードはhttp://abandonedart.org/?p=549にあります。
このセクションであなたが作ったのと同じように
独自の**noise**関数を使用しています。

（リスト4.5の続き）

```
  stroke(20, 50, 70);
  strokeWeight(1);

  float x, y;
  float noiseval = random(10);
  float radVariance, thisRadius, rad;
  beginShape();
  fill(20, 50, 70, 50);
  for (float ang = 0; ang <= 360; ang += 1) {

    noiseval += 0.1;
    radVariance = 30 * customNoise(noiseval);

    thisRadius = radius + radVariance;
    rad = radians(ang);
    x = centx + (thisRadius * cos(rad));
    y = centy + (thisRadius * sin(rad));

    curveVertex(x,y);
  }
  endShape();
}

float customNoise(float value) {
  float retValue = pow(sin(value), 3);
  return retValue;
}
```

開始点をランダムにします

戻り値を-1から+1へ

あなたはリスト4.5の中にもう1つの重要な変更を加えます。以前やったように、前の位置から線を引く代わりに、今回はbeginShapeとendShapeを、fillを適用する閉じられたエリアを描くのに使います。curveVertex(x,y)で点を結んで形にします。結果は図4.7のようになります。

この最初の例では、customNoiseがサインの3乗の値を返し、規則的に繰り返す変動を作ります。この仕組みを使って、新しい関数の中で数学を使って実験することができます。例えば、サインのべき数をパラメータとして、そのべき乗の値を返す関数を作ってみます。

```
float customNoise(float value) {
  int count = int((value % 12));
  float retValue = pow(sin(value), count);
  return retValue;
}
```

この関数を使うと、図4.8のように描けます。

さて、でもここにはパーリンノイズの美しさはありません。オスカー賞を獲得することはできないでしょう。けれども今あなたには、あなた自身のものと呼べる独自の変動成分があります。それを線、形、動きをはじめ、noiseを使えるさまざまなものに適用してみましょう。

次の段階は、あなたが知っているいろんな数式を関数に投げ入れてみて、そのルーチンがどれくらい壮観なめちゃくちゃさを作れるか、じっくりと見続けることです（例えば図4.9を参照）。あなたの数式がたいしたものでなくても、それは問題ではありません。足し算、引き算、掛け算、割り算だけで、たくさんのことをやり遂げることができます。数学の複雑さは、必ずしもビジュアルの面白さには対応していません。いくつかの数を組み合わせて、形や描画スタイルを変更したら、思いっきりそれを変化させて、それがあなた自身を驚かすことができるかどうかを試してください。

次のセクションでは、パーリンノイズをもとにした私のある作品の制作プロセスを紹介します。自然に変化するシーケンスの中の数の組み合わせを変化させるだけで、たくさんのクリエイティブなことができます（次の章でさらにいくつか見てみましょう）。このたった1つの事例だけでも、あなたの作品にこの章の方法を用いてみようと思わせるのに十分であることを願っています。

4.2 ケーススタディ：Wave Clock

私のWave Clock（図4.10を参照）は、パーリンノイズを使って自然に見える形状を作り出すための、簡単で楽しい例です。私がこのシステムを作った時に、どんな目的でそれに取り組んでいたかを思い出すことはできませんが、しかし私が歩んだ手順を詳しく述べることはできるので、あなたはその過程から何かを得られると思います。

あなたが前のセクションで行ったのと同じように、私は円を描くことから始め、そして円周上のそれぞれの点から反対側の点に線を引きました。現在の角度にπを加えることで反対の点を計算しましたが、それはラジアンで180度回転させるということです。2つの円周上の点の座標は、サインとコサインを使った同じ関数で求めました。

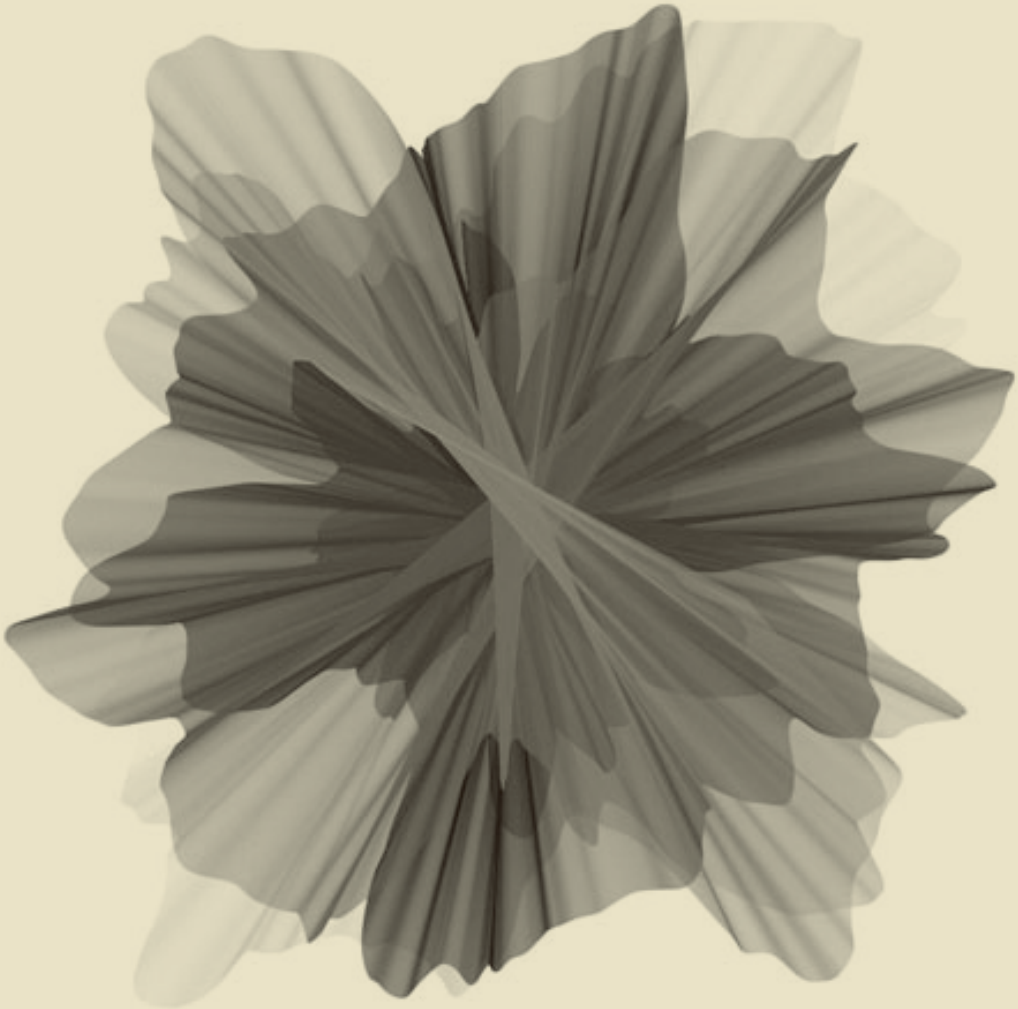


図4.10
Wave Clock (2009)

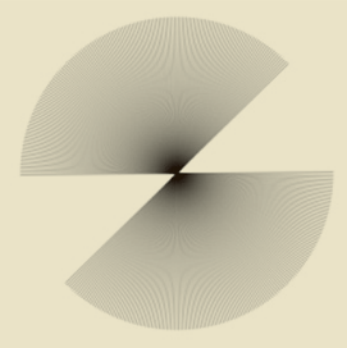


図4.11
円周上の点から反対側の点に
線を引いて描くことで作られた円。

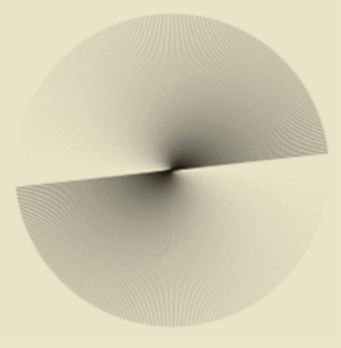


図4.12
線が回転するにつれてストロークの色が
増えることで生まれたフェード効果。

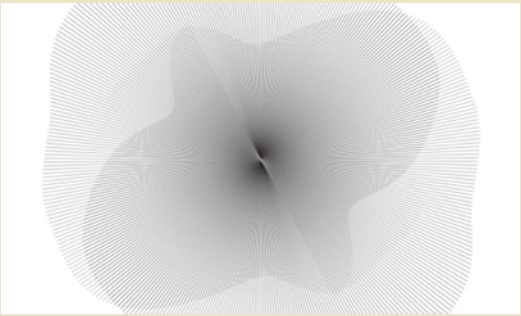


図4.13
半径にパーリンノイズによる変動を加える。

図4.14
回転の速度／方向にパーリンノイズによる変動を加える。

```
float rad = radians(_angle);

float x1 = centerx + (_radius * cos(rad));
  float y1 = centery + (_radius * sin(rad));

float opprad = rad + PI;
float x2 = centerx + (_radius * cos(opprad));
  float y2 = centery + (_radius * sin(opprad));

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

図4.11がアウトプットを示しています。

滑らかで素敵な円が生まれました。フェード効果を与えるために、線の色を変えるいくつかの行を追加しました。まずstrokeCol変数を定義し、255（白）から始めて、毎フレームごとに1ずつ0(黒)に達するまで減少させていき、また逆に255に戻っていくようにしました。その結果、図4.12のようになりました。

次に、2点を結んだ線の長さを、回転するにつれてその半径がノイズ値によって変化するようにしました（図4.13参照）。

```
_radiusnoise += 0.005;
_radius = (noise(_radiusnoise) * 550) +1;
```

それから、さらにひねりを与えるために、回転の角度にノイズ変動を加えて、線の密度を変えました。回転する角度を増減できるようにしたので、反転することもできるようにもなりました。図4.14にこの効果を示します。

```
_angnoise += 0.005;
_angle += (noise(_angnoise) * 6) - 3;
if (_angle > 360) { _angle -= 360; }
if (_angle < 0) { _angle += 360; }
```

最後に、さらに非線形の感じを出すために、ノイズ関数を使ってx軸、y軸方向に最大50ピクセルをプラスかマイナスに動かして、円の中心を揺らしました。

```
_xnoise += 0.01;
_ynoise += 0.01;
float centerx = width/2 + (noise(_xnoise) * 100) - 50;
float centery = height/2 + (noise(_ynoise) * 100) - 50;
```

このシステムの最終的なコードは以下の通りです。

リスト4.6　Wave Clockのための最終コード

```
float _angnoise, _radiusnoise;
float _xnoise, _ynoise;
float _angle = -PI/2;
float _radius;
float _strokeCol = 254;
int _strokeChange = -1;

void setup() {

  size(500, 300);
  smooth();
  frameRate(30);
  background(255);
  noFill();

  _angnoise = random(10);
  _radiusnoise = random(10);
  _xnoise = random(10);
  _ynoise = random(10);
}

void draw() {

  _radiusnoise += 0.005;
  _radius = (noise(_radiusnoise) * 550) +1;

  _angnoise += 0.005;
  _angle += (noise(_angnoise) * 6) - 3;
  if (_angle > 360) { _angle -= 360; }
  if (_angle < 0) { _angle += 360; }

  _xnoise += 0.01;
  _ynoise += 0.01;
  float centerx = width/2 + (noise(_xnoise) * 100) - 50;
  float centery = height/2 + (noise(_ynoise) * 100) - 50;

  float rad = radians(_angle);
  float x1 = centerx + (_radius * cos(rad));
  float y1 = centery + (_radius * sin(rad));
```

（続く）

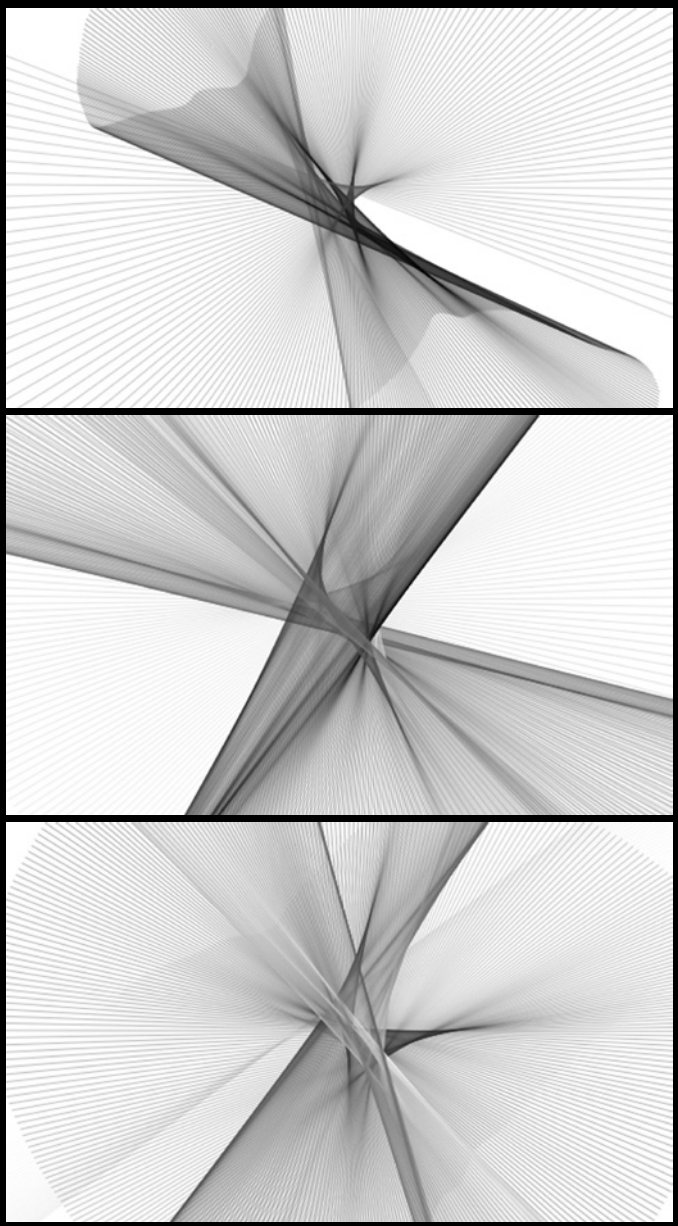


図4.15
完成したシステムの出力結果：Wave Clock（2009）

（リスト4.6の続き）

```
    float opprad = rad + PI;
    float x2 = centerx + (_radius * cos(opprad));
    float y2 = centery + (_radius * sin(opprad));

    _strokeCol += _strokeChange;
    if (_strokeCol > 254) { _strokeChange = -1; }
    if (_strokeCol < 0) { _strokeChange = 1; }
    stroke(_strokeCol, 60);
    strokeWeight(1);
    line(x1, y1, x2, y2);
  }
```

図4.15は、その他いくつかのアウトプットを示しています。

4.3 まとめ

この章で私たちは、あてにならない線を引く方法から、どちらかと言うと楽しい結果を生み出してくれる完全にジェネラティブなシステムまで進んできました。前章と本章で確立した基本的な方法は、次の通りです。

- 機械が描く形状を脱構築する
- なんらかの予測不可能性を用いて再構築する

この単純なテクニックを、何かもっと面白いことに応用してみましょう。

円を使いまわすだけでも、さらにたくさんの面白いことができます。飽きることがない形です。でも、コンピュータが描く他のいろいろな形を試してみたいかもしれませんね。ぜひ、あなたが自分自身で実験してみてください。

次の章では、本書のPart 3で紹介するもっとかっこいい有機的な方法論に進む前に、より高次元のノイズを使って、時間と空間の両方を操作すると何が起こるかを見てみます。

Chapter 5

次元を加える

Adding Dimensions

あなたが、大きな球の表面にしがみつくちっぽけな生物だと想像してください*1。あなたは一生をかけて、その表面のあらゆる部分を踏破し、熟知するまで歩き回ります。この段階で、あたかも行くべき新しいところはないし、見るべき新しいものはないと感じられるでしょう。でも、それらはすべてあなたの発見ひとつで、突然まったく違って見えるようになります。

私たちはよく、自分の作品でこのような壁に直面します。考える余地は、もうどこにも残っていないように思いがちです。しかしその時こそが、さらなる次元を加えることを考える機会なのです。もしこれまで1次元で作業していたなら、2次元でやってみてください。もしあなたが2次元で作業しているなら、3次元を試みてください。

コンピュータモニタ、印刷されたページ、展示場の壁は、2次元しかありません。あなたがこれらのメディアのために作るものはすべて、展示のために平面にされてしまいます。けれども、あなたの作品がその次元に限定されているわけではありません。既にChapter 2で、追加の次元（この場合「時間」）を加え、あなたの作品を動かすアイデアを学びました。この章では、Processingがどのように3次元の描画をレンダリングするかを学びながら、再びそのアイデアについて考えます。

これらのアイデアを考える際には、引き続いてパーリンノイズが主役となります。なぜならこの課題に関連して、もっと探索できることがあるからです。これまでのところ、あなたはnoise関数を、1つのパラメータで1次元のノイズを作るのに使っただけでした。では、それにさらなる次元を加えてみましょう。

＊1　想像するのは難しくないでしょう。実際にあなたもそうなのですから。

5.1　2次元のノイズ

2次元でのノイズは、1次元のものと比べてもそんなに難しくはありません。それは単に見方の変更で、概念的には線から平面になります。1次元では、ノイズの値の進行が、水平線上に見える山脈のように揺らぐ線を生み出しました。2次元のノイズは、ヘリコプターで上から山脈を見下ろしているかのようです。まっすぐに見下ろせば、山頂と谷間からなる平面に見えます。

5.1.1　ノイズグリッドを作る

説明の際の最も良い方法は、具体例を見せることです。新しいスクリプトを開き、次のリストの内容を打ち込んでください。

リスト5.1　2次元のノイズグリッド

```
size(300, 300);
smooth();
background(255);
float xstart = random(10);
float xnoise = xstart;
float ynoise = random(10);

for (int y = 0; y <= height; y+=1) {
  ynoise += 0.01;
  xnoise = xstart;
  for (int x = 0; x <= width; x+=1) {
    xnoise += 0.01;
  int alph = int(noise(xnoise, ynoise) * 255);
    stroke(0, alph);
    line(x,y, x+1, y+1);
  }
}
```

各列の開始点でxnoiseをリセット

ここでのノイズ関数は2つの値をとる

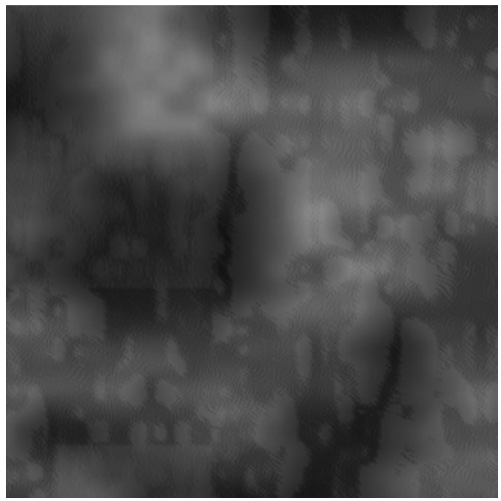


図5.1
アルファ値で視覚化された、
2Dパーリンノイズ。

このスクリプトは、点で格子を作ります（Processingでは、アルファ値は線のほうが良いので、私は点の代わりにごく短い線を使っています）。縦列ごと、横列ごとに、1ステップで1ピクセルずつくまなく渡り歩いて、この300×300の正方形のピクセル上をループします。xとy方向のノイズのシード値種は、古いタイプライターの上のキャリッジ・リターン［1行打ち込んだ後、紙を固定するキャリッジを、次の行の先頭に戻して改行する機構］を打つように、横列の終わりでxシードを元の値にリセットし、ピクセルごとに0.01ずつ増加させていきます。xとyのそれぞれの位置でnoise関数が返す値は、各位置で描画した小さな線のアルファ値の変化として視覚化されています。このスクリプトを実行した結果が図5.1です。

2次元のノイズの楽しさは、それを視覚化する新しい方法を考え出すことにあります。私たちも、その方法を考えてみましょう。

5.1.2　ノイズの視覚化

まず、グリッドを走査する2つのループは変える必要がないので、視覚化する部分だけを関数ブロックに分けることができます。関数にすることで、グリッドに間隔を空けることもしやすくなります。そうすると、そこで何が起こっているかを見ることができます。

リスト5.2　関数ブロックに書き直したノイズグリッド

```
float xstart, xnoise, ynoise;                    (続く)
```

間隔の増加。
ノイズ・ステップの
増加も同様

（リスト5.2の続き）

```
void setup() {
  size(300, 300);
  smooth();
  background(255);
  xstart = random(10);
  xnoise = xstart;
  ynoise = random(10);
  for (int y = 0; y <= height; y+=5) {
    ynoise += 0.1;
    xnoise = xstart;
    for (int x = 0; x <= width; x+=5) {
      xnoise += 0.1;
      drawPoint(x, y, noise(xnoise, ynoise));
    }
  }
}

void drawPoint(float x, float y, float noiseFactor) {
  float len = 10 * noiseFactor;
  rect(x, y, len, len);
}
```

新たに作成したdrawPoint()関数は、x、y、noiseFactorの3つのパラメータをとります。x、yでポイントの位置を決め、2次元のnoiseFactorはそのポイントでの計算に使用します。この場合、noiseFactorを点の大きさを決めるために使い、小さい正方形を描いています。runをクリックしてください。そうすると、図5.2のようなものが現われます。

ひとたび関数を定義すれば、さらなる実験のために、この関数を変更していきます。ノイズを傾きとして描いてみるのはどうでしょう？　そこで、noiseFactorの値によって20ピクセルの線を回転させて描く、新しいdrawPoint関数を作りました。

```
void drawPoint(float x, float y, float noiseFactor) {
  pushMatrix();
  translate(x, y);
  rotate(noiseFactor * radians(360));
  stroke(0, 150);
  line(0, 0, 20, 0);
  popMatrix();
}
```

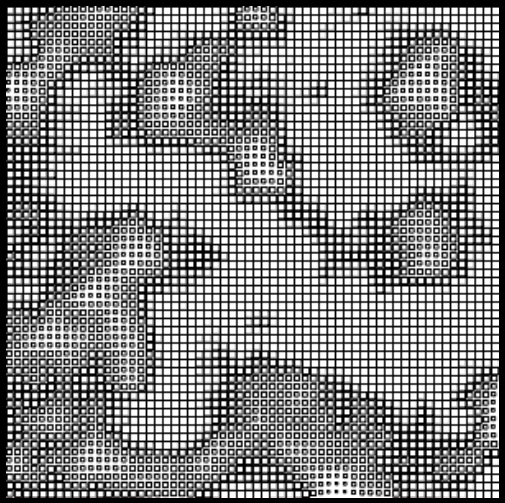


図5.2
さまざまな大きさの正方形で視覚化された、
2Dパーリンノイズ。

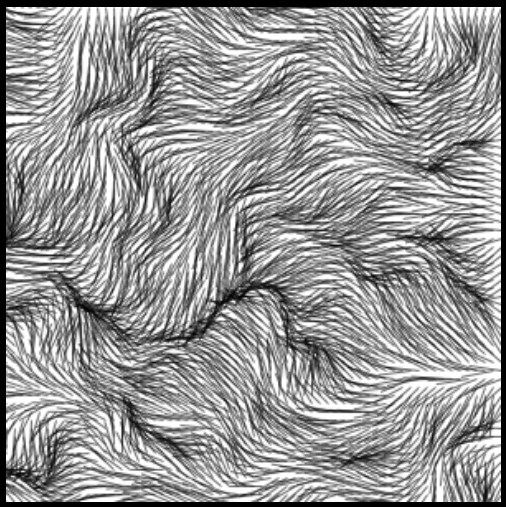


図5.3
回転を使って視覚化された、2Dパーリンノイズ。

回転するために、まずtranslate(x, y)を使って点の描画位置を動かします。そしてpoint(x, y)の代わりに、(0, 0)座標（原点）から描き始めます。そうすることで、左上の(0, 0)のところではなく、今描いているポイントで回転させることができます。関数pushMatrixとpopMatrixは、置き換えや回転を実行する前に、前の描画位置を保存（push）し、その後再び取り出す（pop）ことに使用します。図5.3が実行結果です。

さらにもう1つ。setup関数内のbackground(255)をbackground(0)に変えて、そしてdrawPointを書き直せば、同じノイズファクターを使って、回転、サイズ、色、アルファ値を変えることができます。

```
void drawPoint(float x, float y, float noiseFactor) {
  pushMatrix();
  translate(x,y);
  rotate(noiseFactor * radians(540));
    float edgeSize = noiseFactor * 35;
    float grey = 150 + (noiseFactor * 120);
    float alph = 150 + (noiseFactor * 120);
  noStroke();
    fill(grey, alph);
      ellipse(0,0, edgeSize, edgeSize/2);
    popMatrix();
}
```

その結果生まれたグリッドは、図5.4のようになります。

translateを使う

あなたが片手にペンを持ち、もう一方で1枚の紙を押さえていると想像してください。あなたが、**line(20, 30, 120, 130)**のようなコマンドを使う時、左上の原点(0, 0)ポイントと相対して、ペンがどこからどこまで動くかという座標を特定しているわけです。あなたが、**translate(20, 30)**を使う時、あなたはペンを操っているのではなく、代わりに紙のほうを動かし、ペン先を(20, 30)ポイントにもってきています。(20, 30) ポイントは、描画を始める新しい原点になります。

translate(20, 30)と**line(0, 0, 100, 100)**を実行すると、**line(20, 30, 120, 130)**コマンドと同じ結果になります。そうすることで多くの場合、特にお互いに関連しているものを描く時、座標がわかりやすくなります。

1ポイント回転させるような動きの場合、紙を動かした方がペンを動かすよりかなり簡単にできることに気づくでしょう。

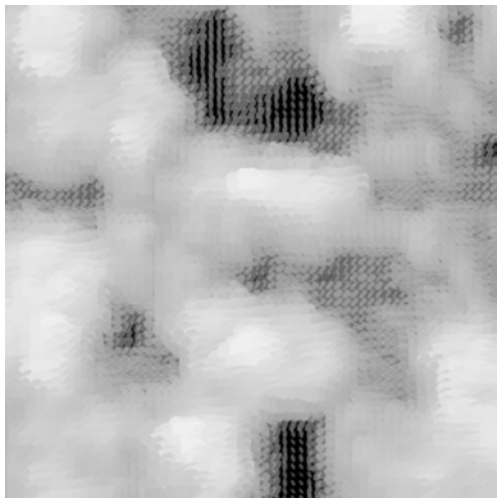


図5.4
ふんわりした小さい雲のような、2Dパーリンノイズ。

コツがわかりましたね。あなた独自のアイデアのいくつかをこの関数に放り込んでみて、何が現れるか見ることをお勧めします。Processingのリファレンスをぱらぱらと見て思いついた、まだ試していない描画方法をノイズファクターに応用することで、どんな変化が現れるのか見てください。

次に、私たちはさらなる次元として、時間に着目します。それにノイズがどのように応用できるか見てみましょう。

5.2 ノイズ・アニメーション

フレームループ——Processingに備わっているdraw関数——で、時間的に変化するイメージを作ることができます。これこそ、私たちが「アニメーション」と呼んでいるものです。聞いたことがありますね。もしもどのようにフレームループが機能するか復習が必要なら、セクション2.3.1に戻ってください。まず、draw関数を使って、2次元のノイズグリッドを何か動くものに変えてみます。

リスト5.3　ノイズグリッドふたたび。フレームループ上で描く

```
float xstart, xnoise, ystart, ynoise;                    (続く)
```

（リスト5.3の続き）

フレームレートを設定

すべてのフレームでバックグラウンドをクリア

x/yのノイズのスタート値を増やす

```
void setup() {
  size(300, 300);

  smooth();
  background(0);
  frameRate(24);

  xstart = random(10);
  ystart = random(10);
}

void draw () {
  background(0);

  xstart += 0.01;
  ystart += 0.01;

  xnoise = xstart;
  ynoise = ystart;

  for (int y = 0; y <= height; y+=5) {
    ynoise += 0.1;
    xnoise = xstart;
    for (int x = 0; x <= width; x+=5) {
      xnoise += 0.1;
      drawPoint(x, y, noise(xnoise, ynoise));
    }
  }
}

void drawPoint(float x, float y, float noiseFactor) {
  pushMatrix();
  translate(x,y);
  rotate(noiseFactor * radians(540));
  noStroke();
  float edgeSize = noiseFactor * 35;
  float grey = 150 + (noiseFactor * 120);
  float alph = 150 + (noiseFactor * 120);
  fill(grey, alph);
  ellipse(0,0, edgeSize, edgeSize/2);
  popMatrix();
}
```

drawPoint関数は、yとxの2重ループの中で、順に走査していくだけなので変化しません。新しく追加したのは、各フレームごとにxとyのノイズのシード値を増やしていく、という点です。このプログラムを実行すると、図5.4のようなイメージが現れます。絶え間ない風が南西から吹くかのように、雲がスクリーンを斜めに漂っていきます。動いている部分はありません。ノイズの平面を横切りながらスクロールすることで、錯覚が生まれているのです。

さすがにこれを印刷で示すのはちょっと難しいので、あなたがこの効果を実際に見るためにはコーディングが必要です。けれどもこの動きは私からすれば、少しばかり予測がつきすぎて退屈です。そこで、ここに滑らかで自然な変化をつけてみます。次のリストを見てください。

リスト5.4　動きにノイズ変動が加わったノイズグリッド

```
float xstart, xnoise, ystart, ynoise;
float xstartNoise, ystartNoise;

void setup() {
  size(300,300);
  smooth();
  background(255);
  frameRate(24);

  xstartNoise = random(20);
  ystartNoise = random(20);
  xstart = random(10);
  ystart = random(10);
}

void draw () {
  background(255);

  xstartNoise += 0.01;
  ystartNoise += 0.01;
  xstart += (noise(xstartNoise) * 0.5) - 0.25;
  ystart += (noise(ystartNoise) * 0.5) - 0.25;

  xnoise = xstart;
  ynoise = ystart;

  for (int y = 0; y <= height; y+=5) {
```

開始位置にノイズを使う

+/- 0.25で、すべてのフレームごとに変わる

（続く）

（リスト5.4の続き）

```
      ynoise += 0.1;
      xnoise = xstart;

      for (int x = 0; x <= width; x+=5) {
        xnoise += 0.1;
        drawPoint(x, y, noise(xnoise, ynoise));
      }
    }
  }

  void drawPoint(float x, float y, float noiseFactor) {
    pushMatrix();
    translate(x,y);
    rotate(noiseFactor * radians(360));
    stroke(0, 150);
    line(0,0,20,0);
    popMatrix();
  }
```

グリッドループの前に追加したコードは、2つの（1次元の）ノイズシーケンスをカメラ位置に与えるのに使っています。上昇気流で漂う葉っぱのような穏やかな不規則さを与え、ノイズの平面が浮いているかのように見せます。あなたはすぐに、私がdrawPoint関数を前のセクションにあった線の回転バージョンに戻したことに気づくでしょう。このように、視覚化に時間の次元を加えることで、毛を逆立てたり、草を通り抜ける風のそよぎのような、素敵な錯覚を作り出すことができます。さっそく実行して、その動きを確認してください。

ひとつのスクリプトの中で使えるノイズのシーケンスの数には限りがありません。リスト5.4には、結びつけることで2次元のノイズ値を得るためのxnoiseとynoise、そして動きの1次元的な方向性を与えるxstartNoiseとystartNoiseの4つを使っています。

次は、Processingがどのように3Dを描くかを学ぶことで、新しい深みを探求してみます。

5.3　3次元

あなたがsize関数でキャンバスを指定する際に、MODEと呼ばれる3番目の追加パラメータがあります。MODEは、今まで特に知る必要はなかったのですが、これはProcessingにどのレ

ンダリングエンジン（レンダラー）を使うのかを教えてくれるものです。

```
size(幅, 高さ, MODE);
```

表5.1　Processingでのレンダラーの選択

MODE VALUE	説明
JAVA2D	デフォルト。最高品質の2Dレンダラー。
P2D（Processing 2D）	より速い2Dレンダラーで、ピクセルデータのために最適化されていますが、通常、JAVA 2Dほど正確ではありません。
P3D（Processing 3D）	高速な3Dレンダラーで、主にWeb上でJavaプログラムを実行するために使用されます。速度とファイルサイズを質よりも優先しています。
OPENGL	より良い高速3Dレンダラーで、OpenGLグラフィックハードウェアがあればそれを利用します。OPENGLはあらゆるグラフィックスを滑らかにしてくれるので、smoothとnoSmoothは不要。このレンダラーを使うためにはOpenGLライブラリをインポートする必要があります（リスト5.5を参照）。
PDF	スクリーンではなく、直接にPDFファイルに書き込むためのレンダラー。特にスクリーンより大きなプリントを作っている時や、とにかく結果を見るためにファイルに保存しなければならない時など、高画質のベクター形状を描く際には理想的。ただしPDFライブラリを必要とします。詳しいインフォメーションはこちら。http://processing.org/reference/libraries/pdf/。

レンダラーのリストが表5.1です。これまで使ってきたデフォルトはJAVA2Dで、最も高品質な2次元のイメージを作成してます。3D画像を描くためには、MODEをP3DまたはOPENGLに変える必要があります。通常OPENGLの方が、見た目がより良くなります。しかし一方で、P3Dの方がWebに載せやすいように、より小さいアプレットを作ってくれます。本章の場合、どちらを選択するかはあまり重要ではありません。

さて、レンダラーを選択したら、何かを描いてみましょう。

5.3.1　3次元空間で描く

これまでの例で、キャンバス上のxとyの位置という2次元のデカルト座標についてはかなり詳しくなったと思います。3D画像を描くためには、2D平面から上下どのくらいの距離にあるのかを指定するのに必要なzの値を、追加の位置情報として加える必要があります。

2D描画と同じように、Processingは一般の3D構造を表現するために、多くの組み込み関数を持っています。3次元空間で描くことの感覚を得るために、まずはこれらを使ってみましょう。最初にsphereを使ってみます。3D空間のある位置に移動するのにtranslateを使い、その位置で描画するという描き方をしてみます。x、y、z座標を扱う時、これが実に理にかなった方

法であることがわかるでしょう。

新しいウィンドウを開いて、以下のコードを入力してください。

```
import processing.opengl.*;

void setup() {
size(500, 300, OPENGL);
sphereDetail(40);
}

void draw() {
background(255);

translate(width/2, height/2, 0);
sphere(100);
}
```

まず最初、OpenGLライブラリのインポートを宣言します。Sketchメニュー（Sketch > Import Library）からライブラリのインポートを選択すれば、この行は自動的に追加されます。**size**コマンドはまずキャンバスの大きさを設定し、OPENGLレンダラーを選びます。

次に、**sphereDetail**コマンドは、球を描く時にどのくらい細部まで描くかをレンダラーに伝えます。3Dレンダラーは滑らかなカーブを描けません。そのため、どんな形でも常に平面の集合で構成されます。このコマンドのパラメータを40とすることで、レンダラーは9(360/40)度ごとに新しいエッジを作ります。

最後の2行は、描画位置をキャンバスの中央に移動し、その周りに半径100の球を描くように指示しています。3つの変数をどのように**translate**コマンドに渡しているかということに注意してください。まだzは使っていません。アウトプットは図5.5に示した通りです。

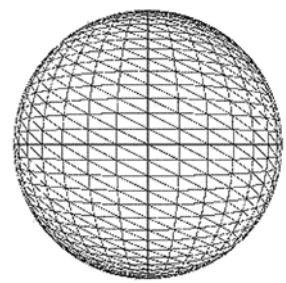


図5.5　OpenGLを使って描かれた球。

これまでのところ、線がなければ2Dレンダラーを使って作られた円とあまり違っているようには見えません。しかし、translateの3番目のパラメータを変えてみれば、その違いがわかるでしょう。translateの行のパラメータをtranslate(250, 150, 50)に変えてください。球は今より大きく見えます。50ピクセル分、あなたの近くに球を動かしたのです（逆に-50にすると遠ざかります）。もし3番目のパラメータを150にすれば、自分が球の中にいるかのように見えるでしょう。

この新しい遠近法で、前のセクションの2Dノイズ平面を、新たな角度から見てみましょう。

5.3.2　3次元ノイズ

新しいウィンドウに、次のコードを入力してください。

リスト5.5　3Dの遠近法による2Dノイズ

```
import processing.opengl.*;

float xstart, xnoise, ystart, ynoise;

void setup() {
  size(500, 300, OPENGL);
  background(0);
  sphereDetail(8);
  noStroke();

  xstart = random(10);
  ystart = random(10);
}

void draw () {
  background(0);

  xstart += 0.01;
  ystart += 0.01;

  xnoise = xstart;
  ynoise = ystart;

  for (int y = 0; y <= height; y+=5) {
    ynoise += 0.1;
    xnoise = xstart;
```

（続く）

（リスト5.5の続き）

```
      for (int x = 0; x <= width; x+=5) {
        xnoise += 0.1;
        drawPoint(x, y, noise(xnoise, ynoise));
      }
    }
  }

  void drawPoint(float x, float y, float noiseFactor) {
    pushMatrix();
    translate(x, 250 - y, -y);
    float sphereSize = noiseFactor * 35;
    float grey = 150 + (noiseFactor * 120);
    float alph = 150 + (noiseFactor * 120);
    fill(grey, alph);
    sphere(sphereSize);
    popMatrix();
  }
```

レンダラーの指定を除けば、このコードはリスト5.4によく似ています。主な相違点は、drawPoint関数です。2D平面上を移動するのではなく、3D空間内を移動します。このコードでは、zの奥行きとしてyの値を再利用します。そうすることで、映画館の座席のような、後ろ向きに段々傾いていく四角を作ります。

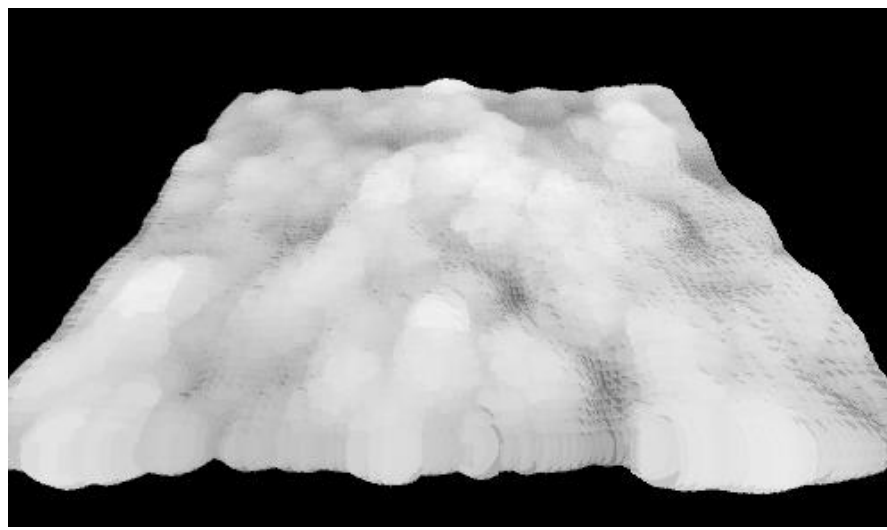


図5.6　3次元の視点から見た2次元のノイズ。

3D空間内のそれぞれの位置で、半径と塗りつぶす色、そしてノイズ値と関連づけられたアルファ値を使って球を描きます。結果は、ほどよい視点から見た、穏やかに動く雲のような景色です（図5.6参照）。

視点は3次元です。しかし、あなたが見ている平面と、それを定義するノイズのシーケンスは本質的に2次元のままです。けれどもnoise関数は最高3つのパラメータをとることができ、それによって3次元のノイズを生み出す能力を持っています。3次元のノイズを表すための視覚的な比喩は、もはや山脈ではありません。雲や、煙が充満した部屋を想像してください。そしてこれらをどうやって視覚化すればいいかを考えてみましょう。まずは、ノイズを制御するブロックを作ることから始めます。

ループの部分をいくらか変更しなければなりません。平面ではなく、立方体におけるすべての点を繰り返すようにします。そして、x、y、zと3次元ノイズ値の4つのパラメータをdrawPoint関数に渡します。変更後のコードは以下の通りです。

リスト5.6　3次元ノイズの立方体を作る

```
float xstart, ystart, zstart;
float xnoise, ynoise, znoise;

int sideLength = 200;
int spacing = 5;

void setup() {
  size(500, 300, P3D);
  background(0);
  noStroke();

  xstart = random(10);
  ystart = random(10);
  zstart = random(10);
}

void draw () {
  background(0);

  xstart += 0.01;
  ystart += 0.01;
  zstart += 0.01;
```

レンダラーをP3Dに変える

（続く）

より良い視点を得るために、位置を変更する

今回は3つのループ

（リスト5.6の続き）

```
    xnoise = xstart;
    ynoise = ystart;
    znoise = zstart;

    translate(150, 20, -150);
    rotateZ(frameCount * 0.1);
    rotateY(frameCount * 0.1);

    for (int z = 0; z <= sideLength; z+=spacing) {
      znoise += 0.1;
      ynoise = ystart;
      for (int y = 0; y <= sideLength; y+=spacing) {
        ynoise += 0.1;
        xnoise = xstart;
        for (int x = 0; x <= sideLength; x+=spacing) {
          xnoise += 0.1;
          drawPoint(x, y, z, noise(xnoise, ynoise, znoise));
        }
      }
    }
  }

  void drawPoint(float x, float y, float z, float noiseFactor) {
    pushMatrix();
    translate(x, y, z);
    float grey = noiseFactor * 255;
    fill(grey, 10);
    box(spacing, spacing, spacing);
    popMatrix();
  }
```

できあがった立方体は、ほとんど透明な箱でできていて、その色は3次元のノイズ値によって与えられています。でもはっきりと見ることができないので、箱をくるくる回すためのコードを加えなければなりません。ともあれ、あなたはこれで箱の中に雲を作ったわけです（図5.7、5.8参照）。

3次元の計算方法は簡単ではありません。3D空間の数学は、2Dのキャンバス上で移動するよりもかなり複雑です。しかし、実際にやってみれば次第に慣れていくでしょう。ちょっといじってみてコツがつかめたら、前の2つの章でやった「分解して、組み立て直す」方法を3Dでもやってみましょう。その一例として球の三角関数をとりあげ、このセクションを終わりにしたいと思います。

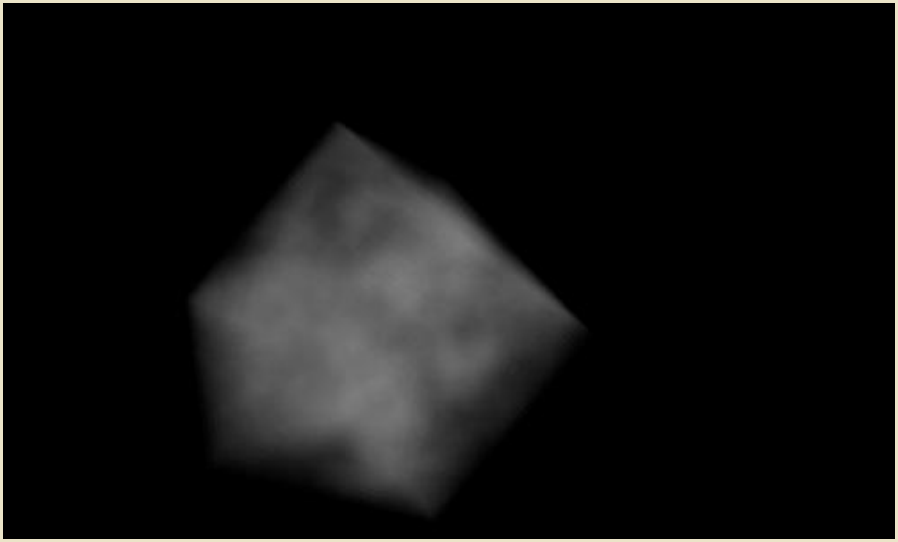


図5.7
3次元ノイズ：箱の中の雲

図5.8
Haunted Fishtank（Abandoned Artwork #49）のソースコードはこちら：
http://abandonedart.org/?p=449。
これは、あなたがこのセクションで開発したシステムによく似ています。

5.3.3　球を描く間違った方法

以下は、球の表面上に点を描くという、あまり魔法っぽくはない公式です。円上の点の公式に似ていますが（Chapter 4参照）、中心座標に加えて、位置を決定するのに2つの角度（sとt）を必要とします。

```
x = centreX + (radius * cos(s) * sin(t));
y = centreY + (radius * sin(s) * sin(t));
z = centreZ + (radius * cos(t));
```

Processingのsphere関数を使わずに球を描くには、セクション4.1でやったのと同様、2つの角度の周りで回転を繰り返す方法を使います。そうすることで、球の表面上の点で線を描くことができます。

この方法の使い方を次のコードで確認してください。角度tは、中央軸の周りを18度ずつ回ります。一方、角度sは、ゆっくり180度回転をします。

リスト5.7　球の周りのらせん

```
import processing.opengl.*;

int radius = 100;

void setup() {
  size(500, 300, OPENGL);
  background(255);
  stroke(0);
}

void draw() {
  background(255);

  translate(width/2, height/2, 0);
  rotateY(frameCount * 0.03);
  rotateX(frameCount * 0.04);

  float s = 0;
  float t = 0;
  float lastx = 0;
  float lasty = 0;
  float lastz = 0;
```

（続く）

（リスト5.7の続き）

```
    while (t < 180) {
      s += 18;
      t += 1;
      float radianS = radians(s);
      float radianT = radians(t);

      float thisx = 0 + (radius * cos(radianS) * sin(radianT));
      float thisy = 0 + (radius * sin(radianS) * sin(radianT));
      float thisz = 0 + (radius * cos(radianT));

      if (lastx != 0) {
        line(thisx, thisy, thisz, lastx, lasty, lastz);
      }
      lastx = thisx;
      lasty = thisy;
      lastz = thisz;
    }
  }
```

三角関数用に角度をラジアンに変換

結果として、図5.9のようにヴァーチャルな球の表面を描くらせんが描かれます。

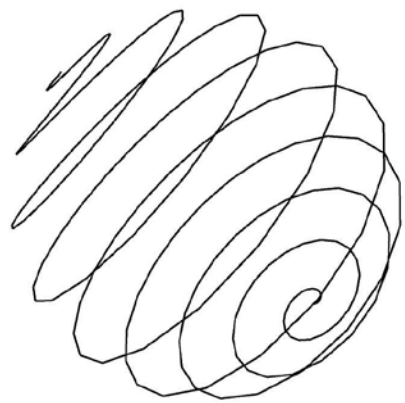


図5.9　らせんで描かれた球。

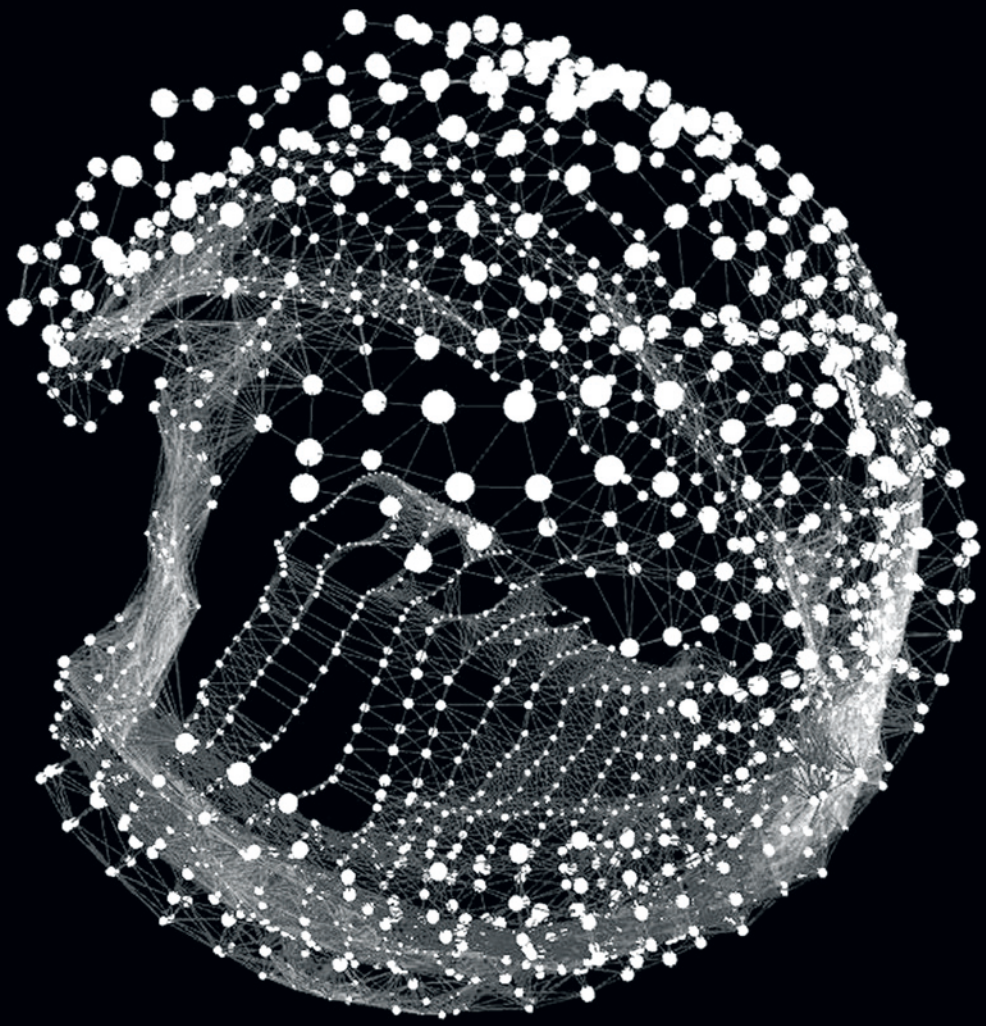


図5.10
Frosti (2010)。分解された球にパーリンノイズを適用して制作したビデオ作品。
http://vimeo.com/9712740

このセクションの冒頭の3行の三角関数が、私の仕事に大きく貢献してくれました。例えば私のビデオ作品Frosti（2010）の主要部分には、この数学の魔術が使われています（図5.10参照）。ぜひあなたもこの魔法を誤用して、自分の作品に活用していってください。

5.4　まとめ

もうこれ以上は実験できない、と感じてしまった時にこそ、空間や時間にさらなる次元を加えてみましょう。この章で、私たちはその両方を試してみました。さらに、Processingでの3次元描画も紹介しました。これは、あなたの弓にもう1本の弦を付け加えたようなものです。

本章と前章で、かなりパーリンノイズを使い込みましたね。ぜひ、このノイズで作り出せる効果の数々を味わってみてください。自然な変化が必要な時には、いつでもどこでも使える便利な道具です。しばらくは、オモチャ箱の奥隅にしまわれないことでしょう。私は、この1つのアルゴリズムを視覚化するための新しい方法を考えるだけで、一生過ごせると思います。もしあなたがそれに飽きてしまったら、今度は別のアルゴリズムで第2の人生を送ることができるでしょう。

本書の中では、まだ当分の間はnoiseとrandom関数を使って、予測不可能性を生み出すためのより有機的な方法を見ていきます。続くPart 3では、もっとすごい数学のオモチャ――「創発（emergence）」を紹介します。自然のプロセスのシミュレーションを通じて、ラップトップの外の世界で生まれている自然の形を呼び覚ましてくれるような、実践的な方法を紹介します。

Part 3 複雑性

“新しいということと、作られたことがない
ということの境界は、決定的ではありますが、
予測し得るものではありません。
私たちはそれらが同じ意味であると思いがちですが、
実はそうではないのです。

何かがある自然の法則に従っていたとしても、
それは予測可能ではありません。
なぜならそこで起きていることは、
何が起きていたかを計算する前に終ってしまうからです。
それくらい複雑なのです。

だから、確かに計算はできますが、
起きていることよりも素早く
世界を計算することはできないのです。”

ルーディ・ラッカー, インタビュー, 2007

図6.1
ブライトン、ウエスト・ピア上空のムクドリの群れ。
ケビン・メレディス（http://lomokev.com）撮影。
単純なルールが高レベルの構造をかたちづくる。

Chapter 6

創 発

Emergence

私はイギリス南海岸のブライトンという街に住んでいます。都市で暮らした後、ここへ引っ越してきました。海と空しかないここの眺めに惚れこんで移り住み、そして今も住んでいます。昔はよく、ブライトンのウエスト・ピアを通って歩いて帰宅したものでした。そこはかつてはヴィクトリア朝の美しい遊技場でしたが、1970年代の頃からずっと放置され、そして2003年には火災で骨組みだけになってしまいました。そして現在は、おびただしい数のムクドリの生息地となっています。毎晩、陽が沈む少し前、ムクドリたちは巣に戻る際に、とても複雑な編成の素晴しい群れの動きを見せてくれます（図6.1を参照）。

私はこの形にすっかり魅了されてしまいました。なぜなら、それが私の脳の両側へ訴えかけてくるからです。片方の脳が、動く生き物の自然感覚的な美しさと、それらが作り上げる形に感銘を受けます。しかしそこには、群れを作り出すアルゴリズムの数学的な美しさもあって、それが私の中の論理的な部分を刺激するのです。群れの中のそれぞれの鳥は、群れを作っているという高いレベルのパターンには気づかずに、単純に、本能に従っているだけです。彼らの行動は、すぐ隣の鳥との関係で方向を決めたり、群れの中心に向ったり、他の鳥の通り道を避けたりという、簡単なルールの組み合わせだけで決められています。これらの単純なルールが群れに適用されると、そこから高いレベルの抽象的な美のパターンが生まれます。

6.1　創発を定義する

低いレベルでの単純なルールが、高いレベルの組織化された複雑さを生み出す現象を「創発」と呼びます。この用語は、19世紀中頃の英国の心理学者、哲学者、そしてジョージ・エリオット［イギリスの女流作家］の恋人でもあったジョージ・ヘンリー・ルイスの論文で生まれました。しかし「全体は部分の総和以上の何ものかである」という考え方の源は、アリストテレスの時代までさかのぼります。ここ数十年では、ジョン・ホランドの本『Emergence』（1989）を皮切りに、その後2001年には、より多くの読者を獲得したスティーブン・ジョンソンによる同じタ

イトルの本が出版され、創発という概念は複雑系理論とも関係づけられながら広範な人気を博しました。

簡潔に言えば、創発とは、複雑な秩序パターンがいかにして非常に多くの小さくて単純な相互作用から生じているのかを観察することです。典型的な例はアリの群れです。2つの異なったスケールで、アリの群れは明確に定義されていて、論理的で一貫した行動が観察できる有機的組織体です。個々のアリのパターンを調べると、それぞれに個体としての行動を定めるニーズ、能力、フェロモン反応があることがわかります。しかし、群れの集合的な行動を調べれば、そこにもまた複雑な行動パターンがあることがわかります。工場、防衛、廃棄物処理などを備えた、まるで都市のような群れです。注目に値するのは、これらのごく小さなパターンが、中心となるデザインや意図によって形成されてはいないということです。それらは、個体の集合体の利己的行動の副産物にすぎないのです。これらの行動は、ミクロレベルでは取るに足りないものですが、集合的に見ると、複雑なマクロ有機体を形成しています。

これは、私たちが以降の章で利用しようとしている原則です。単純な行動をする多数の小さなエージェントを作れば、その集合体がより複雑で創発的な行動をマクロレベルで生み出します。

6.1.1 アリの群れと群集アルゴリズム

今日でも、あまり研究熱心でない人たち（アニメ映画会社を含め）の間で広く信じられている神話は、アリの群れが指揮系統のトップの女王アリのもと、階層的な組織で営まれているということです。人間中心の見方では、これは私たちが目指す組織形態の典型です。しかし、この神話は1970年代の頃の研究で既に疑いをかけられていました。なんと、アリの群れの構造と動きは、中央集権的には決められていなかったのです。グローバルなレベルのパターンと構造はすべて、ローカルなレベルの多数のミクロな決定から生まれます。デザイン、階層、指導者、その他いかなる高レベルの影響も、その中に入っていません。

女王の役割は、大きな全体の中で働いているその他のアリと同じぐらいおざなりなものです。上から下される命令はありません。アリの組織は、ボトムアップになっています。それはまさに草の根の創発です。

同様に、ウエスト・ピアの周りをくるくる回りながら群れをなすムクドリたちも、リーダーに従ってはいません。群れは、個々の行動欲求をもった個体で構成されています。それぞれのムクドリは、飛行の速度と方位を決めるのに、自分に最も近い個体だけを見ます。この効果は、よく知られているクレイグ・レイノルズのBoidsアルゴリズムとして、1987年に計算モデル化されました。レイノルズは、リアルな群れのシミュレーションをコードで書くには、たった3

図6.2
Blonde Boids (Abandoned Artwork #4)。ダン・シフマンによるクレイグ・レイノルズのBoidsコードの応用を、さらに私が改変したもの。

つのルールだけがあればいいことを発見しました。

- 分離――近くの個体を避けて進む。
- 整列――近くの個体の平均の進行方向に向かって進む。
- 結合――近くの個体の平均の位置に向かって進む。

さらにいっそう複雑な規則（例えば障害物の回避など）を加えることもできますが、しかしこの3つが、ウエスト・ピアのムクドリの群れに相当する複雑さを生み出すのに必要な要素のすべてです。ダン・シフマンらがProcessingでBoidsを実装しました。これを改造するのはとても楽しいのですが、しかし私はここでそのコードを転載はしません。なぜならそのコードはかなり長くて、この章で必要とされるものよりずっと複雑だからです。でも、もしそれで遊んでみたいなら、以下のURLからコピーすることができます：http://processing.org/learning/topics/flocking.html。このコードを使った私の実験例が図6.2です。本書のイントロダクションの中には、これをより複雑に応用したものがあります（図i.5）。

オブジェクト指向アプローチによるプログラミングの基本を理解しないと、群集アルゴリズムや他の創発的なコードをさらに深めていくのは難しいでしょう。そこにこの章の主旨があり、セクション6.2でとりあげます。しかしその前に、私は社会学的な観点から、創発が意味することの背景を述べたいと思います。

6.1.2　ローカルに考え、ローカルに行動する

創発の概念は、特に自然界と社会との関係において、政治的な変化を幾度となくされながら、より広く受け入れられるようになりました。この概念は1950年代には異端の分野でした（1954年の死の直前のアラン・チューリングの最後の仕事は、形態発生――創発のルールによって自然界のパターンがかたちづくられること――が主題でした）。1970年代と80年代が革新の時期で、1990年代には熱心に議論が行われました。Webが浸透した21世紀の現在、創発の概念がようやく親しみやすいものになりつつあります。なぜならインターネットが、他のさまざまな組織形態の見方を変える役割を少なからず果たしてきたからです。

単細胞の集合としての粘菌、アリの群れ、都市の住民、あるいはグローバルに分散したWebユーザーといった自己組織集合が、自分自身を秩序的に組織できるという考えは、インターネットのような分散されたノードを介して多くの知的な会話を行っている現代であれば、至極受け入れやすいものです。たとえそれがほとんど意味のない140語のつぶやきであっても、Twitterのようなネットワークがどのくらい重要な社会学的な意味を持っているかを誰もが理解することができます*1。1つのツイートは、単なる自分本位か、あるいは数百人の小さなローカルなグループに影響されたものです。しかし、このつぶやきが何十万人により発せられると、そのメタ・パターンが文化的な動きや世界的な意見を形成します。私たちが必要としているのは、的を得た展望、手段、あるいはそれらを正しく評価するための視覚化なのです。

このようにして出現したネットワークには、信じられないほどの精妙な強さと回復力があります。それらの力は分散性にあります。それぞれのノードは、個々で意味を持つにはあまりにも小さ過ぎます。だから、創発する組織を標的にしたり、影響を与えたり、攻撃するのは難しいのです。インターネットは粘菌と同様に、中央集権化された組織の脆弱性はありません。もし1つのノードが失われたり、あるいは損害を被ったとしても、巨大な全体はその影響をほとんど受けません。分散ファイル共有ネットワークはこの原則を利用しているので、標的となる中心がなく、法の適用が困難です。

明らかに、ここには政治的な意味合いがあります。もしそれが自然界でうまく機能しているなら、人間世界においても、中央組織（政府や諸制度）はかなり必要のないものだということを創発は示唆しています。ちょうど人間が、自分たちのローカルな関心（富、地位、安全等々）を、人間が形成するマクロな集団（集団、都市、国）のニーズより優先するように、アリはローカル

＊1　Twitterのプロトコル自身を創発現象として見ることができるかもしれません。そのシンタックス（ハッシュタグ、@アドレスなど）はTwitterの創設者が設計したものではなく、集団での使用を通してボトムアップに採用されるようになったものです。

なフェロモンに従っていて、群れ全体に対する影響については考えていないのです。私たちは客観的には、自動車を使うことが環境に世界規模の破壊的な影響を与えることを知っています。しかし自動車でスーパーマーケットに行くことの便利さを犠牲にするほどの影響はありません。より大きな全体への理解と、その影響に関する知識があっても、ミクロな行動を変えられないのです。

たくさんの新しい急進的なグループが、創発を組織の原則として認め、受け入れました。例えば反グローバリゼーションや環境問題の抗議グループは、自分たちをファイル・シェアリングの分散的自己組織化システムのようにモデル化しました。

神経科学理論の領域において、意識（言葉を解釈する自己感覚）は頭の中のシナプスの間で発火するミクロなインタラクションによって形成される、マクロなレベルの創発の例であると論じている人がいます。私たちの存在の感覚全体が、創発の結果であるかもしれません。

これは多くの学問領域を横断する、多くの応用と理論にまたがった概念です。でも私たちの関心は、創発が有機的なアート作品を作り出すための重要な方法であるということにあります。まず第一に重要なのは、創発が自然界のいかに多くのかたちを自己組織し、組み立てているか、ということです。もしあなたがこの美学と共鳴したいなら、それがどう機能するかに注意を払うべきです。第二に重要なのは、論理的な種子からアート作品を育てあげることで、十分面白い視覚的な複雑さへと発展させていくことでしょう。プログラミング言語という単純な論理的作業において、創発の原則は、シンプルなコードで複雑な結果を得ることができる理想的なツールです。

今まで、作品に予測不可能性を持ち込むためにランダム化やノイズ関数を用いてきました。しかし、そのようなわかりきったやり方がいつも必要ではないということにも気づくでしょう。動物の脳みそでも理解できるくらい単純なアルゴリズムのプログラムという最小限の抽象からでも、予測できることを超えた複雑さを生み出すことができるのです。

これから私たちは、コードを使った創発を探究していきます。後に続くセクションでは、シンプルなプログラミングの偶発的副産物に出くわすでしょう。Chapter 7では、セル・オートマトンをはじめとする、より組織だったアプローチを利用して制作を行います。Chapter 8では、フラクタルを通じて、自己相似性というさらなるコンセプトを加えたプログラミングの方法を礎にします。本書のこの最後の部分では、章全体を通して、自然界の組織的な原則を反映したたくさんのコードの例を紹介します。

6.2　オブジェクト指向プログラミング

本章のミッションは、私が今説明した創発的複雑性を探求することです。けれどもコードというメディアを通してこのミッションを遂行するために、いくつかの新しい道具が必要です。ここからの数ページで、オブジェクト指向プログラミング（OOP）の基本を学びます。もしあなたが経験豊かなプログラマーであるなら、おそらく既にこのコンセプトに精通しているでしょう。かなりの力と洗練さがあるので、オブジェクト指向の構造を受け入れない言語はほとんどありません。しかし、たとえあなたがOOPに精通していたとしても、どうか私の話につきあってください。なぜなら、OOPは私がこれから説明しようとしていることのたった半分にしか過ぎないのです。

もしあなたが以前に一度もこの用語を聞いたことがなかったとしても、OOPはあなたのコーディングに対する考え方を少し飛躍させればよいだけです。これまでやってきたように、コンピュータに命令を順序だてて与える、というよりもむしろ、メモリの中にオブジェクト（独立したデータを入れるための箱で、通常のリニアなプログラミングスタイルの中でも使えます）を作ることを考えます。

この概念的なシフトは、初心者にとってはちょっととっつきにくいものかもしれません。でも、怖がるようなものではありません。初めにやるのは、いくつか円を描き、それがどうなるかを見ることです。この章で実際にやりたいのは創発的複雑性を求めることなので、いくつかの結果を知るのにそんなに長くはかからないことを保証します。まず最初に、クラスとインスタン

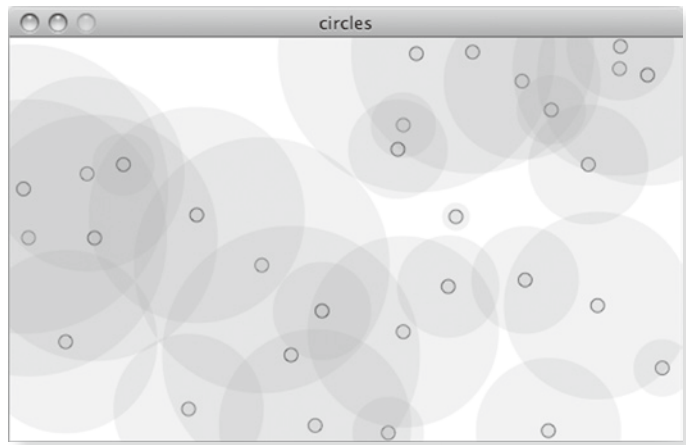


図6.3　リスト6.1のアウトプット：ランダムに生み出された円。

スから始めましょう。

6.2.1　クラスとインスタンス

簡単なところから始めましょう。次のリストは、マウスがクリックされるたびにランダムな位置にいくつかの円を描くスクリプトです。

リスト6.1　マウスクリックのたびにいくつかの円を描くコード

```
int _num = 10;

void setup() {
  size(500,300);
  background(255);
  smooth();
  strokeWeight(1);
  fill(150, 50);
  drawCircles();
}

void draw() {

}

void mouseReleased() {
  drawCircles();
}

void drawCircles() {
  for (int i=0; i<_num; i++) {
    float x = random(width);
    float y = random(height);
    float radius = random(100) + 10;
    noStroke();
    ellipse(x, y, radius*2, radius*2);
    stroke(0, 150);
    ellipse(x, y, 10, 10);
  }
}
```

ここで起こっていることがわかるでしょうか。このスクリプトは、マウスをクリックするたびにスクリーン上のランダムな位置に10個の円を描きます。図6.3に似た感じになるよう、数回試してください。

これまでのところ、この円の描画はまだあまり洗練されていません。スクリーンにばらまかれて、そこに残されるだけで、もうそれ以上何かする手立てを持っていません。より有機的な方法で再アプローチすれば、円をもっと生き生きとできることに気づくでしょう。そこで、Processingに直接円を描けという代わりに、円「オブジェクト」を作成してみましょう。円オブジェクトは、自分自身についての属性すべて（今のところは、中心点のx値、y値、半径）をカプセル化します。

プログラミングでのオブジェクトは、実世界のモノ（オブジェクト）の概念と近いものです。岩を想像してください。これはひとつのモノです。円オブジェクトがx、yや半径を持っているように、岩オブジェクトにも「属性」（色、重量、粗さ）があります。さらに、岩を使ってできることもあります。そう、オブジェクトには属性だけでなく「機能」もあります。投げられたり、ハンマーで打てたり、ペーパーウェイトを使うために机の上に置かれたりしますね。こうしたオブジェクトを、Processingの中でモデル化する方法をやってみましょう。

オブジェクトを定義することは、関数を定義することによく似ています。再利用可能なオブジェクトを定義するのに、classキーワードを使います。あなたのスケッチのいちばん下に、次のリストの内容を加えてください。

リスト6.2　Circleクラス

オブジェクトの属性

オブジェクトのコンストラクタ

オブジェクトの色とアルファ値

```
class Circle {
  float x, y;
  float radius;
  color linecol, fillcol;
  float alph;

  Circle () {
    x = random(width);
    y = random(height);
    radius = random(100) + 10;
    linecol = color(random(255), random(255), random(255));
    fillcol = color(random(255), random(255), random(255));
    alph = random(255);
  }
}
```

ここでは、1つのオブジェクトを定義したのではなく、同じ形式のオブジェクトのクラスを定義しました。「クラス」とは、再利用できるテンプレートであると考えてください。たくさんの異なった円を描く必要はなくなりました。circleクラスの「インスタンス」（つまりあなたが定義したテンプレートによって作られたオブジェクト）として円を作ればよいのです。

クラスとインスタンス

クラスと、そのクラスのインスタンスの違いは重要です。もしこの概念が理解できれば、OOPに入り込んだも同然です。

クラスはオブジェクトの集合に対して定義されるものです。例えば「80年代英国インディーバンド」というクラスを定義すれば、多くのオブジェクトを共有の特徴（ギター、逆立てた髪、感傷的な歌詞など）で記述できます。私たちは、そのクラスからThe Smithsというインスタンスと、Echo and The Bunnymenという2番目のインスタンスを作れます［どちらも有名なイギリスの80年代ロックバンド］。互いに独立してはいますが、**recordAlbum()**、**featureInJohnHughesFilm()**、**splitAcrimoniously()**［レコードアルバム()、ジョン・ヒューズ監督の映画でフィーチャー()、激しい解散()］といった機能を共有することができます。

クラスはテンプレートです。オブジェクトが共有するパラメータ、実行する関数、オブジェクトがどのように組み立てられるかという論理だけを定義します。クラスは、そのインスタンスを作る時だけオブジェクトになります。

怠惰な私たち人間の言語では、かつて80年代の英国インディーズバンドを見たと言うことは、80年代の英国インディーズバンドの1つあるいはもっと多くのインスタンスを見たという意味になります。概念定義では、どんなに自分のことをポスト・モダンだと思っていても、酔っ払ったりダンスしたりはできませんね。

ここまではまだ、純粋に概念的です。あなたはスクリーンに円を描いていませんし、まだ円を作ってさえいません。そのためのテンプレートを定義したところです。このテンプレートから実際のオブジェクト（クラスのインスタンス）を作るには、以下のdrawCircles関数の新しいバージョンのように、newキーワードを使ってコンストラクタを呼びます。

```
void drawCircles() {
  for (int i=0; i<_num; i++) {
    Circle thisCirc = new Circle();
  }
}
```

いいですね。でもあなたはまだ実際には円を描いていません。ただ概念的にメモリの中にそれらを作っただけです。これまでのところ、属性（変数）とコンストラクタ（そのクラスのオブジェクトが作成される時に呼ばれる関数）を定義しただけです。もしクラスで何かしたいなら、次のリストのようにメソッド（関数）を追加しなければなりません。

リスト6.3　初めてのメソッドと共にふたたびCircleクラスを

属性

コンストラクタ

オブジェクト・メソッド

```
class Circle {

  float x, y;
  float radius;
  color linecol, fillcol;
  float alph;

  Circle () {
    x = random(width);
    y = random(height);
    radius = random(100) + 10;
    linecol = color(random(255), random(255), random(255));
    fillcol = color(random(255), random(255), random(255));
    alph = random(255);
  }

  void drawMe() {
    noStroke();
    fill(fillcol, alph);
    ellipse(x, y, radius*2, radius*2);
    stroke(linecol, 150);
    noFill();
    ellipse(x, y, 10, 10);
  }
}
```

オブジェクトを作成した後、このメソッドを呼び出せるようにdrawCircles関数を修正します。

```
void drawCircles() {
  for (int i=0; i<_num; i++){
    Circle thisCirc = new Circle();
    thisCirc.drawMe();
  }
}
```

ほら、図6.3のようにいくつか円が描けました。素晴らしい。数ページ前では、ここまで到着するのにかなりの努力がいるように思われたかもしれません。けれどもこの新しい構造でやり始めると、オブジェクトとして円をまとめてしまうことがいかに便利かがわかるでしょう。これであなたは、円のそれぞれに、動き、振る舞い、生命を与えることができます。さらに、それらにフィーリングや意見なども与えることができますが、これはChapter 7にとっておきましょう。今のところは、それらをあちこちに漂わせたり、相互に作用させることに集中しましょう。

コードを何箇所か更新してみましょう。リスト6.4は以下のことを行います。

- すべての円をしまっておくための配列を作成する
- それぞれの円にx、y方向の動きを与える
- それぞれの円に、drawループでフレームごとに呼ばれるupdateMeメソッドを与える

リスト6.4　動きのある、オブジェクト指向の円描画

```
int _num = 10;
Circle[] _circleArr = {};
void setup() {
  size(500,300);
  background(255);
  smooth();
  strokeWeight(1);
  fill(150, 50);
  drawCircles();
}

void draw() {
  background(255);
  for (int i=0; i<_circleArr.length; i++) {
    Circle thisCirc = _circleArr[i];
    thisCirc.updateMe();
  }
}

void mouseReleased() {
  drawCircles();
}

void drawCircles() {
  for (int i=0; i<_num; i++) {
    Circle thisCirc = new Circle();
    thisCirc.drawMe();
    _circleArr = (Circle[])append(_circleArr, thisCirc);
  }
}

//================================= objects

class Circle {
```

円の配列を定義

オブジェクトを配列に加える

（続く）

（リスト6.4の続き）

すべてのフレームでステップを進める

ランダムなステップ

フレームごとに動かす

ステージの端から出ないようにする

描く

```
    float x, y;
    float radius;
    color linecol, fillcol;
    float alph;
    float xmove, ymove;

    Circle () {
      x = random(width);
      y = random(height);
      radius = random(100) + 10;
      linecol = color(random(255), random(255), random(255));
      fillcol = color(random(255), random(255), random(255));
      alph = random(255);
      xmove = random(10) - 5;
      ymove = random(10) - 5;
    }

    void drawMe() {
      noStroke();
      fill(fillcol, alph);
      ellipse(x, y, radius*2, radius*2);
      stroke(linecol, 150);
      noFill();
      ellipse(x, y, 10, 10);
    }

    void updateMe() {
      x += xmove;
      y += ymove;
      if (x > (width+radius)) { x = 0 - radius; }
      if (x < (0-radius)) { x = width+radius; }
      if (y > (height+radius)) { y = 0 - radius; }
      if (y < (0-radius)) { y = height+radius; }

      drawMe();
    }
  }
```

円は動き回っています。updateMeメソッドは、各オブジェクトのxとyの位置を動かし、スクリーンの外に出そうになると戻します。drawMeメソッドは、画面上にインスタンスを描画します。

配列を使う

「配列」は、ある特定の種類のオブジェクトのリストです。配列は以下のように定義します。

```
int[] numberArray = new int[5];
```

すべてがint型の5つの要素を持つリストを宣言します。
5つの要素は、大括弧（{ }）を使えば以下のように個々の値を定義できます。

```
int[] numberArray = {1, 2, 3, 4, 5};
```

以下のように、それぞれの位置の要素に項目を加えることができます。

```
numberArray[2] = 3;
```

この例では、2番目ではなく3番目の要素を3にしています。1番目の要素はインデックスが0なので、3番目の項目はインデックスが2になります。
項目なしの空の配列は次のように定義できます。

```
int[] numberArray = {};
```

追加のスロットを配列に加え、そのスロットにデータを入れる（リスト6.4でやったように）ためには**append**を使います。新しい要素を配列に加える際には、**int[]**のようにその型を明記する必要があります。

```
numberArray = (int[])append(numberArray, 6);
```

クリックして円を加えても、スクリーン上に既にある円は変化しません。キャンバスをすべてのフレームでクリアしていますが（drawフレームループで、background(255)を呼んでいます）、オブジェクトはメモリにあるままなので、それらはどこにも行きません。ですから、あなたが必要とすればいつでも、円をふたたび何度でも描くことができます。たとえ画面上ですべてのフレームごとに描画しなかったとしても、オブジェクトはずっとメモリにいて、姿を見せずに動いているのです。

どんどんクリックしていって、円の数を何百、何千にしたら、動画のパフォーマンスが落ちていくことに気づくでしょう。もちろんマシンの性能によっては動画が遅くならず、数百の円を画面上で自由に動き回らせることができます。とはいえ、プロセッサの限界については知っておいたほうがいいでしょう。

これであなたはオブジェクト指向の構造を手にしました。たくさんの単純なオブジェクトを生成することができるアプリケーションもあります。次に、どのようにこのシステムをいじくって、そして創発のパターンを活用していくかを考えてみましょう。

6.2.2　ローカルな知識（衝突判定）

Circleクラスから作られたインスタンスには、私たちが「自己認識」と呼んでいるようなものがあります。インスタンスは自分の要素をすべて知ることができますが、それが存在している環境、あるいはスクリプトがどのように使っているかについては何も知りません。群れをなす

ムクドリの1羽であったなら、それは盲目で、群れの他の仲間を見ることができない状態で飛んでいる、ということです。けれどもこの考えを変えて、クラスから作られたオブジェクトに、いくつかの基本的な感覚を与えることもできます。

配列にはすべての円オブジェクトが入っています。このグローバルな配列をループすることで（すべてのオブジェクトに対して）、各オブジェクトは他のすべてのオブジェクト（そのオブジェクトがスクリーン上にあるかどうかにかかわらず）に対して自分自身を認識させることができます。この例は、現在の円が他の円と重なっているかどうかをチェックします。これをゲーム用語では「衝突判定」と呼んでいます。

衝突判定（そして衝突反応――オブジェクトが衝突した時、現実的に反応させること）は、ゲームプログラミングの黒魔術の1つです。現実的な物理システムを実現するためには、オブジェクトが自分の隣人や周囲の環境に対する知識や認知を持って、周囲に反応できる必要があります。環境がいっそう複雑になり、そしてオブジェクトが増えると、各フレーム・ループの実行が（特にオブジェクトが独特の形状をしていると）かなりプロセッサ負荷の高い処理になってしまいます。

このトピックだけをとりあげた本もありますが、私たちはそんなに現実的な物理シミュレーションを必要としているわけではないので、簡単な例をとりあげます。円を用いる場合、2点間の距離を測る時にちょっとだけ三角関数が必要になりますが、Processingでは組み込みのdist関数が距離を計算してくれます。もし2つの円の中心の間の距離をチェックして、2つの円の半径の和よりも大きければ、2つの円が接触していないことがわかります。次のリストのように、この計算をupdateMe関数に数行付け加えてみます。

リスト6.5　Circleクラスのupdateme メソッド（衝突判定付き）

```
    x += xmove;
    y += ymove;
    if (x > (width+radius)) { x = 0 - radius; }
    if (x < (0-radius)) { x = width+radius; }
    if (y > (height+radius)) { y = 0 - radius; }
    if (y < (0-radius)) { y = height+radius; }

    boolean touching = false;
    for (int i=0; i<_circleArr.length; i++) {
      Circle otherCirc = _circleArr[i];
      if (otherCirc != this) {
        float dis = dist(x, y, otherCirc.x, otherCirc.y);
        if ((dis - radius - otherCirc.radius) < 0) {
```

他の円をループする

円の間の距離を計算

現在の円を除外

（続く）

（リスト6.5の続き）

```
            touching = true;
            break;
          }
        }
      }
      if (touching) {
        if (alph > 0) { alph--; }
      } else {
        if (alph < 255) { alph += 2; }
      }

      drawMe()
    }
```

上のリストで加えたループが、配列の中の他のすべての円をチェックして、それらとどのくらい離れているかを調べます。もしある円が重なり合っているなら、touchingフラグがtrueにセットされて、ループを抜けます。最後のif文で、円が交差しているかどうかによって、異なるエフェクトを適用します。もし円が重なっていたら、だんだん消えていきます。そうでなければ不透明度を増やします。これは、動き回る時に他の円と重なることが少ない小さな円はだんだんはっきりとしていくのに対し、大きな円は次第に透明になっていきます。実際にどうなるのか、コードを実行してみましょう。

きっとあなたは、この振る舞いをもっと視覚的に面白くできると思うでしょうが、その前に、もう少しいろいろやってみましょう。まずは円が衝突した時に、その中間点にもう1つ円を描いてみます。

6.2.3　インタラクション・パターン

スクリーンの上（あるいは外でも）のどこかにある2つの円の中間点を計算する時には、少し慎重にならなくてはなりません。まずは基本をおさえておきましょう。もし現在の円のx座標が交差している円のx座標より大きいならば、中間点のxは、

```
x + (otherCirc.x - x)/2
```

となります。逆にもし現在の円のx座標が交差している円のx座標よりも小さければ、中間点のxは、

```
otherCirc.x + (x - otherCirc.x)/2
```

となります。

これは1つの計算方法ですが、たった1行のコードでどちらの可能性もカバーできるもっといい方法があります。2つのx値の平均値はそれら2つの間に中間点をとるので、それらを加えてそして2で割ることによって計算することができるのです。

```
(x + otherCirc.x)/2
```

同じことがyにも当てはまります。xとyの位置がわかったら、重なった部分の長さを半径とする円をその位置に描きます。コードは以下の通りです。

リスト6.6　Circleクラスのupdate Meメソッドで円を描く

重なった部分を計算する
重なりがマイナス＝接触
中間点を計算

```
void updateMe() {
  x += xmove;
  y += ymove;
  if (x > (width+radius)) { x = 0 - radius; }
  if (x < (0-radius)) { x = width+radius; }
  if (y > (height+radius)) { y = 0 - radius; }
  if (y < (0-radius)) { y = height+radius; }

for (int i=0; i<_circleArr.length; i++) {
  Circle otherCirc = _circleArr[i];
  if (otherCirc != this) {
    float dis = dist(x, y, otherCirc.x, otherCirc.y);
    float overlap = dis - radius - otherCirc.radius;
  if (overlap < 0) {
    float midx, midy;
    midx = (x + otherCirc.x)/2;
    midy = (y + otherCirc.y)/2;
    stroke(0, 100);
    noFill();
    overlap *= -1;
    ellipse(midx, midy, overlap, overlap);
    }
  }
}

  drawMe();
}
```

実行してみると、まったく新しいパターンが創発していることがわかります。図6.4がそのアウトプットです。プログラムが与えたシンプルな動きによる円を描いています。しかしさらに、円同士のインタラクションによる創発にも気がつきます。ほんの少しだけ円の動きを抽象化してみることで、複雑さの新しい層に巡り会うことができました。

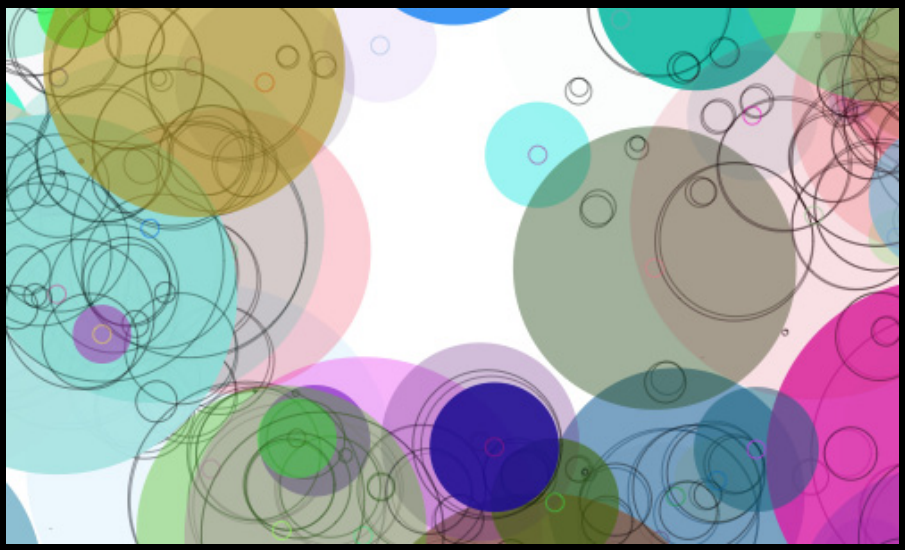


図6.4
円はプログラムした通りの単純な振る舞いをします。
しかし円同士のインタラクションによって、新しい複雑さが創発します。

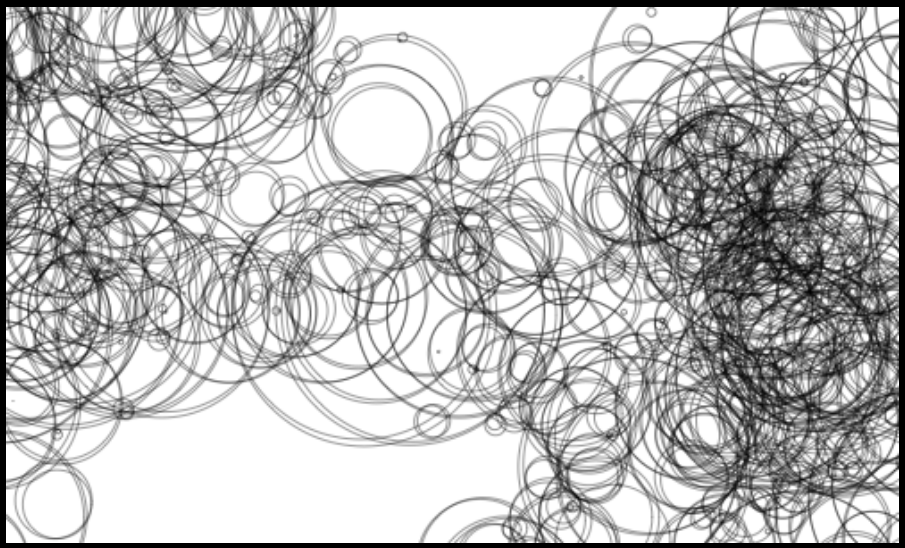


図6.5
もともとの円の動きを取り除いてみると、とても面白い創発のパターンが現れます。

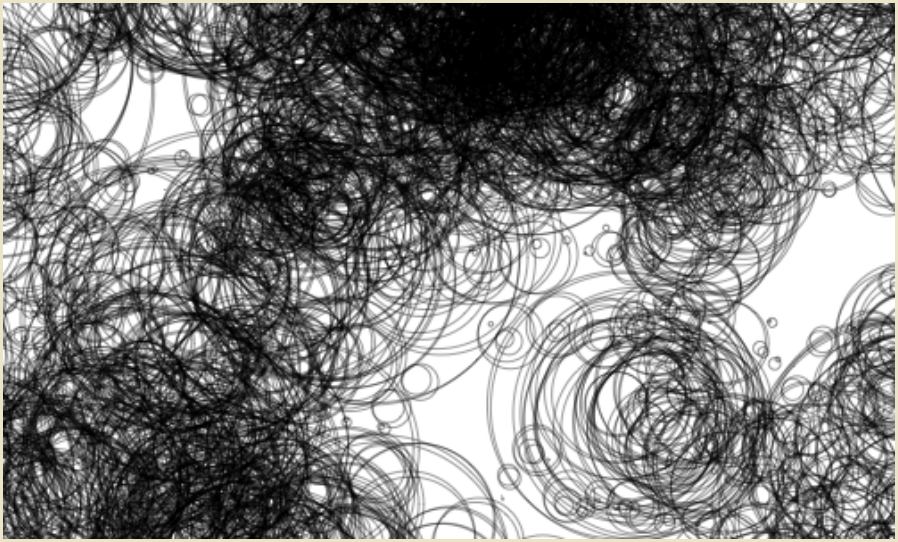


図6.6
170個の円から成る図。
実際にいくつの円が画面上にあるのかは推測しかできません。

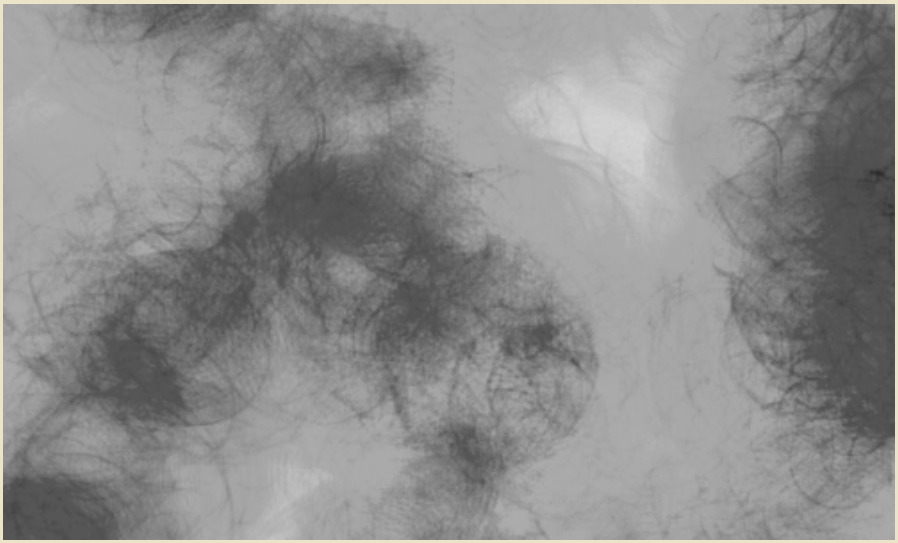


図6.7
創発する円。よりソフトな線と痕跡を使ってさらに抽象化します。

この新しく現れた動きは、コードで作った単純なものより明らかに面白いと思います。それをよりはっきりと見ることができるように、創発した円だけを分けてみましょう。updateMe関数の中のdrawMeの呼び出しをコメントアウトするか削除すれば、オリジナルの円を取り除くことができます。もちろん円自体はまだそこにあります。ただ、もうそれらをスクリーン上には描かないだけです。こうしてみると、図6.5に似たようなものができあがります。

さらに、マウスプレス関数に現在の円の数を表示させる1行を加えることで、今いくつの円が作られたかを知ることができます。

```
void mouseReleased() {
  drawCircles();
  println(_circleArr.length);
}
```

私はなんとなく170が好きだっただけです（図6.6を参照）。

この円で構成する群れのパターンというのは、何かを意図してできた結果ではありません。このパターンは、プログラムされた動きの間に生じたインタラクションの副産物であって、動きそのものではありません。パターンを創発した、エージェントの間のインタラクションの中から生まれたものです。一羽のムクドリは、一連の規則によって飛行パターンを描きます。その規則は何も変わらなくとも、一羽の鳥をその群れの規則の中に置けば、パターンの複雑さが数倍に増えるのです。

この例では、振る舞いはあまり単純にはなりませんが、たくさんのオブジェクトは直線的に動いていますし、描画のスタイルはかなり基本的です。もしさらにいろんなことをやってみたいと思えば、たくさんの道が広がっています。そうし始める前に、いくつかのアイデアを紹介します。

- 痕跡を残す――フレームごとにスクリーンをクリアしない。その代わりに（2.4.2で述べたように）、透明な長方形をその上に描く。
- アルファ値と線の太さを減らす――線をもっと曖昧な感じにして、くっきりした境界を有機的にぼやけさせる（図6.7参照）
- もっと複雑な経路にする――2つの点を直線で結ぶよりも、ずっと面白いルートがあることを前章で見ていきました。例えばカーブに沿って動きを作ってみたらどうでしょう。
- もっと面白い形を描く――単純な円よりも、何かもっとドキドキするようなものを描く。イメージを読み込んで、オブジェクトの視覚表現に使用することも可能です。

6.3 まとめ

本章では、創発の概念を紹介し、いくつかの例をとりあげてきました。冒頭で述べた通り、とても単純に振る舞う単純なオブジェクトから、独自の創発効果が生み出されます。創発というものがどのくらい簡単で、こちらから働きかける必要がなく、ほとんどが偶然の産物でありながらも驚きを生み出してくれるのかをお見せすることができたと思います。たくさんの部分からなるシステムは、それ自身があまり複雑でなくとも、複雑性を創発しやすい傾向があります。それをいかにして引き出すかがポイントです。

この章ではまだ、ランダム性やノイズといったものをコードの生成的な要素を生み出すために利用していません。私たちが行ってきたのは、予想できない複雑さを導くのに十分な抽象化のレベルを適用することです。単純なルールが、複雑な結果を引き起こします。

また、この章でマスターしてもらいたいもうひとつのポイントは、オブジェクト指向プログラミングという非常に有用なプログラミング手法です。これは続く章での、もっと進んだコーディングの可能性、さらにはその先に向かって広がっていくでしょう。

次の章でも創発の複雑さの話題を続けますが、それを作り出すのにより構造的なアプローチをとっていきます。OOPの理解を、さらにエージェント指向のプログラミングの定義に向けて改良していきます。

Chapter

7 自律性

Autonomy

オブジェクト指向プログラミングは、プログラミングのツボです。オブジェクトは、概念的な事柄を概念的な構造に組織化してくれます。パーティクルシステムの1つの粒子から、ゲームのキャラクター、純粋なデータ形式、あるいは変数とメソッドをまとめる簡潔な方法まで、「オブジェクト」は何にでもなれます。

必ずしも、視覚的な形式や同一性を対応させる必要はありません。それはただデータを保持しているだけです。けれども、そうした専門用語でオブジェクトを考えなくてもよいでしょう。あなたがひとたび定義しようと決めたのなら、オブジェクトに感情や願望、弱点や運命を与えることは簡単です。それはとても抽象的なものですが、ヴァーチャルな世界の活発な構成員となり得ます。あなたのオブジェクトは、データを保持する以上のこともできます。それは、自律したエージェントになるということです。

プログラミング用語における「オブジェクト」と「エージェント」の違いは、エージェントが環境を観察し相互作用することにあります。エージェントには、目的、意思、偏見といったような、複雑な行動をとる可能性があります。それらは不正確さ、自己利益、非合理性などのとても面白い行動をも生み出せます。

人間、サル、コンピュータプログラムやロボット、あるいはスーパーモデルであるかどうかにかかわらず、私たちが考える自律性とは、自身で決定する能力のことです。エージェントとオブジェクトの違いは、動物と岩の違いと同じものなのです。より面白くするために、オブジェクトに人間性を与える必要はありません。でも、単なるデータ構成物よりも生物の方に、より複雑で予測不可能な動きを期待してしまいますよね。

あなたにとって、最も馴染み深い自律エージェントは、このページを見ているあなた自身です。ある日の振る舞いが、生物学的あるいは心理学的な要素や、靴の履き心地とかの複雑な要素の総体から導かれる読者のみなさん、あなたこそ、自らが作り出したデータの中で興味深いパターンを生み出す自律したオブジェクトなのです。あなたが最近やっている大抵のことは、データの軌跡（あなたのあらゆる購買、カードについたポイント、クリックするリンク、お出かけなど）を残しています。あなたの財布の中のカードはあなたの経済活動の自叙伝ですし、あなたのツイートやSMSはソーシャルマップの一部にASCII文字の日記を書いていて、ポケットの中の電話はあなたの日常の心理地理学のGPS画像を描いています。

この章の後半で、私たちはエージェントの概念をさらに詳しく見ていきます。コンピュータ内外を問わず、特徴的でクリエイティブな例について議論し、人々がアートで利用するために生み出した大量のデータを扱う方法も含めて考えます。しかし最初に、自律性の概念と、この種のオブジェクトの創発的複雑性についてよく知ってもらうために、コンピュータサイエンスの初期の室内ゲームの1つである「セル・オートマトン」で遊んでみましょう。

7.1 セル・オートマトン

前章で、特定の意図なしに創発的複雑性を偶然見つけることは、そう難しくないことを紹介しました。ここでは、セル・オートマトン（CA）グリッドの助けを借りて、この現象を観察する構造的な方法を学びます。1970年代、コンピュータサイエンスの分野はCAに取りつかれていました。スーパーコンピュータが何週間もの期間にわたって、ジョン・コンウェイのゲーム・オブ・ライフ（セクション7.1.2を参照）を繰り返し実行するために使われていました。今日では、その程度の計算能力なら私たちの携帯電話にもあるので、Processingの軽量なスケッチ1つで簡単にセル・オートマトンをシミュレートすることができます。

2DのCA[*1]は、セルのグリッド（図7.1を参照）です。それぞれはたった2つの状態要素（オンとオフ、黒と白、生か死）しかありません。それぞれのセルは、隣接する8個のセルの状態を知ることだけできる、ローカルな知識しか持ちません。各ステップごとに、それぞれのセルは、それを囲む現在の状態をもとに次の状態を決めます。

＊1　CAには1次元のものと多次元のものがありますが、私たちは2次元のもののみを考えます。

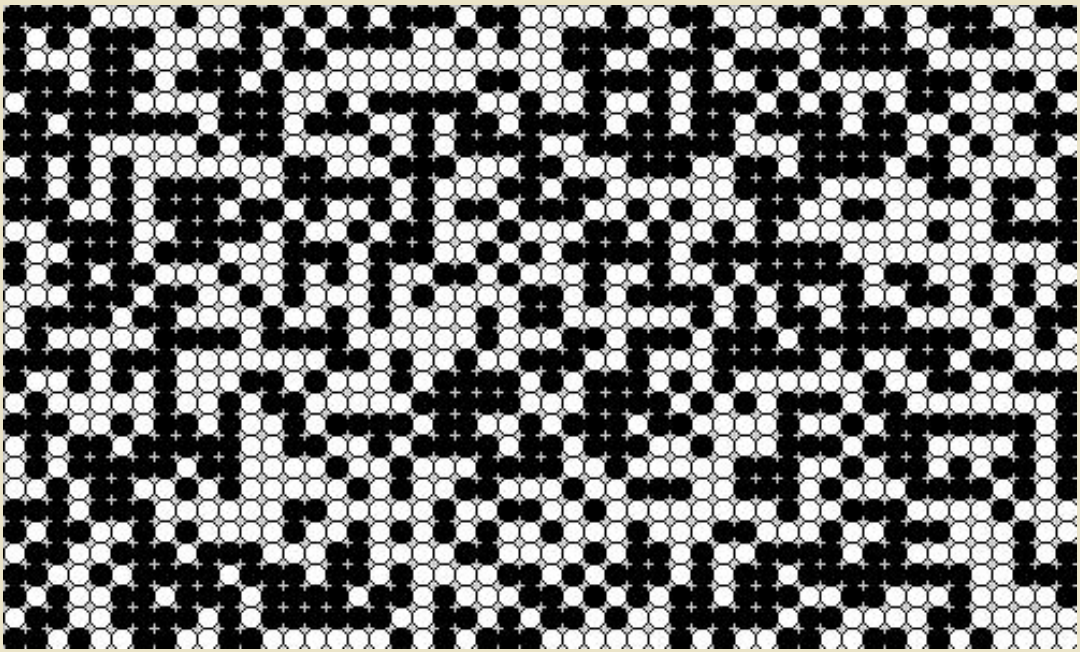


図7.1
セル・オートマトングリッド

7.1.1 枠組みをセットアップ

最もわかりやすい説明は例を見せることでしょう。以下のリストは、オン／オフ状態のあるセルのグリッドと生成し、隣接するセルの状態を参照する方法を定義しています。

リスト7.1 CAフレームワーク

```
Cell[][] _cellArray;
int _cellSize = 10;
int _numX, _numY;

void setup() {
  size(500, 300);
  _numX = floor(width/_cellSize);
  _numY = floor(height/_cellSize);
  restart();
}

void restart() {
  _cellArray = new Cell[_numX][_numY];
  for (int x = 0; x<_numX; x++) {
    for (int y = 0; y<_numY; y++) {
    Cell newCell = new Cell(x, y);
    _cellArray[x][y] = newCell;
    }
  }

  for (int x = 0; x < _numX; x++) {
    for (int y = 0; y < _numY; y++) {

    int above = y-1;
    int below = y+1;
    int left = x-1;
    int right = x+1;

    if (above < 0) { above = _numY-1; }
    if (below == _numY) { below = 0; }
    if (left < 0) { left = _numX-1; }
    if (right == _numX) { right = 0; }
```

セルのグリッドを作る

オブジェクトごとにその隣接セルを教えるループ

上下左右の位置を設定

画面端の位置をつなげる

（続く）

（リスト7.1の続き）

```
      _cellArray[x][y].addNeighbour(_cellArray[left][above]);
      _cellArray[x][y].addNeighbour(_cellArray[left][y]);
      _cellArray[x][y].addNeighbour(_cellArray[left][below]);
      _cellArray[x][y].addNeighbour(_cellArray[x][below]);
      _cellArray[x][y].addNeighbour(_cellArray[right][below]);
      _cellArray[x][y].addNeighbour(_cellArray[right][y]);
      _cellArray[x][y].addNeighbour(_cellArray[right][above]);
      _cellArray[x][y].addNeighbour(_cellArray[x][above]);
      }
    }
  }

  void draw() {
    background(200);

    for (int x = 0; x < _numX; x++) {
      for (int y = 0; y < _numY; y++) {
      _cellArray[x][y].calcNextState();
    }
  }

    translate(_cellSize/2, _cellSize/2);

    for (int x = 0; x < _numX; x++) {
      for (int y = 0; y < _numY; y++) {
      _cellArray[x][y].drawMe();
      }
    }
  }

  void mousePressed() {
    restart();
  }

  //================================== object

  class Cell {
    float x, y;
    boolean state;
```

参照したものを周囲に渡す

まず次の状態を計算

すべてのセルを描く

オンかオフ

（続く）

初期状態をランダムに

（リスト7.1の続き）

```
    boolean nextState;
    Cell[] neighbours;

    Cell(float ex, float why) {
      x = ex * _cellSize;
      y = why * _cellSize;
      if (random(2) > 1) {
        nextState = true;
      } else {
        nextState = false;
      }
      state = nextState;
      neighbours = new Cell[0];
    }

    void addNeighbour(Cell cell) {
      neighbours = (Cell[])append(neighbours, cell);
    }

    void calcNextState() {
      // to come
    }

    void drawMe() {
      state = nextState;
      stroke(0);
      if (state == true) {
        fill(0);
      } else {
        fill(255);
      }
      ellipse(x, y, _cellSize, _cellSize);
    }

  }
```

長々しい出発点ですが、ここにあるのはどれも、これまであなたが出会ってきたものばかりです。スクリプトの最後で、x、yの位置とオン／オフいずれかの状態を持ったオブジェクトCellを定義します。それはまた、次の状態であるnextState（次にそのdrawMeメソッドが呼ばれる時に入る状態）のための入れ物も持っています。

setupで、幅と高さにもとづいて横列と縦列の数とセルの大きさを計算します。そして、2D配列をこのグリッドで埋めるrestart関数を呼び出します。グリッドが作られた後、配列を通して各セルに、隣の状態（それらの上、下、左、右）という2番目の情報が渡されます。

drawループは、配列に対して2回の処理を行います。最初の処理では、各セルの次の状態を計算します。2度目の処理は、データをアップデートして、自分の状態を描画します。これは2段階に分けて行う必要があります。各セルは、次の状態を計算するのに必要な隣接セルの状態を、その計算が終るまで保持していなければなりません。

さて、これでセルのグリッド（図7.1に示されるように）が手に入りました。ここからセルに簡単な動きを与えることができます。最もよく知られたCAはジョン・コンウェイのゲーム・オブ・ライフ［日本ではライフゲームという］なので、私たちもこれを参考にしていきます。

多次元配列

1次元の配列（Chapter 6を参照）は、要素のリストを格納するためのデータ構造です。これらは次のように定義します。

```
int[] numberArray = {1, 2, 3};
```

しかし、リストよりも多くの要素を格納したいなら、あるいは記録したい要素の行列があるならば、配列に次元を追加することができます。次のように2D配列の初期値を設定できます。

```
int[][] twoDimArray = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
```

基本的には、配列の配列を定義します。リスト7.1に、2D配列を作ったり繰り返したりする例があります。

さらに多くの次元を持たせることもできます。あなたの頭で考えられるだけの次元を、自由に付け加えることができます。

```
int[][][] threeDimArray;
int[][][][] fourDimArray;
...
```

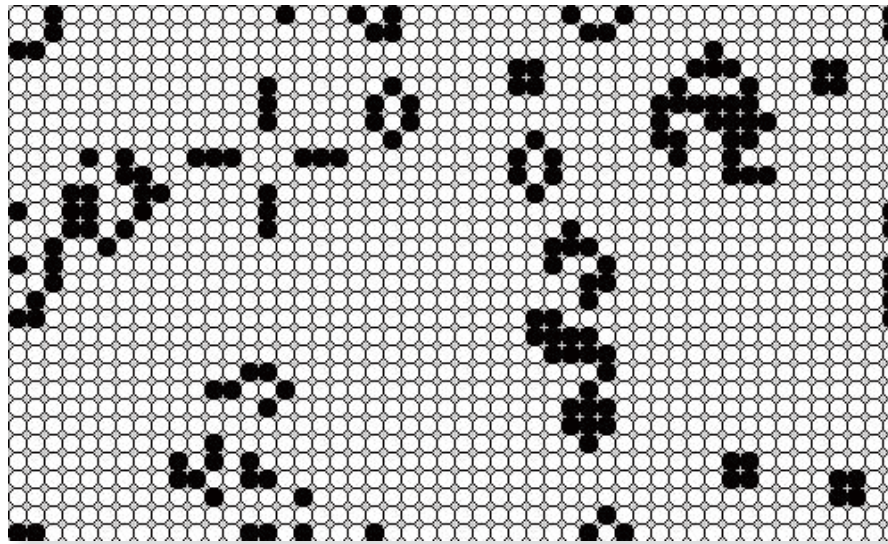


図7.2　100回繰り返した後のコンウェイのゲーム・オブ・ライフ。

7.1.2　ゲーム・オブ・ライフ

コンウェイのゲーム・オブ・ライフ（GOL）の人気は、その意味が数学以外の分野で発見されたことと大きく関係しています。とりわけ生物学者、経済学者、社会学者、神経科学者が、GOLの規則にしたがったCAの動きと、それぞれの専門分野での研究結果の類似点を論じました。それは本質的には単純な計算による人工生命であり、自律エージェントに満ちた数学的な培養皿です。

GOLの規則は次の通りです。

- ルール1：生きている（黒い）セルは、2つか3つの隣接セルが生きていればそのまま生き続ける。さもなければ、過疎か過密のどちらかで死んでしまう。
- ルール2：死んだセルの周囲に、ちょうど3個の生きた隣接セルがあれば奇跡が起こり、それは生き返る。

この規則をコードでは次のように書きます。以下のリストのようにcalcNextState関数を完成してください。

図7.3
セルの年齢に対応して三角形の高さが変化し、回転する。三角形の幅は生きている隣接セルの総数。
三角形の色は、その2つパラメータの組み合わせで決まる。

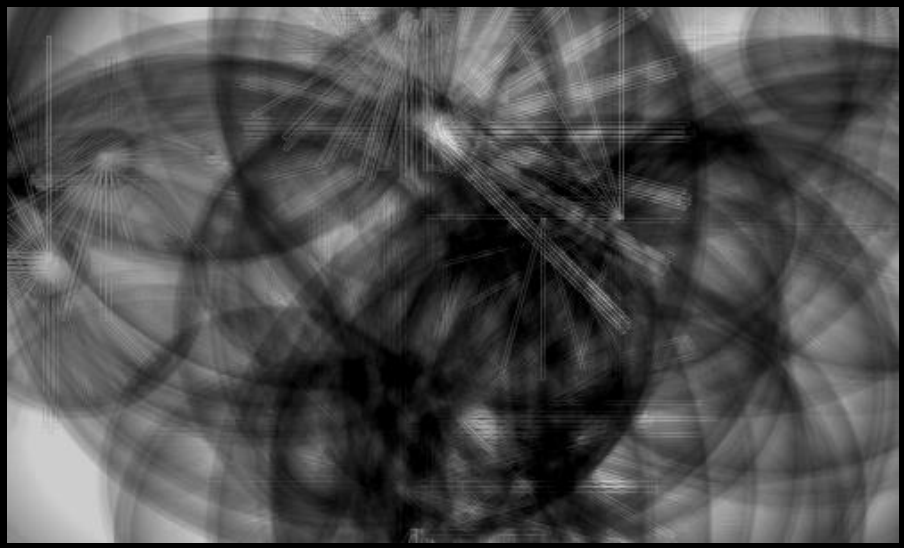


図7.4
円の大きさは生きている隣接セルの総数で決まり、線とそれらの回転はセルの年齢から決まる。

リスト7.2　GOLルールを使って次のステップの状態を計算する

```
void calcNextState() {
  int liveCount = 0;
  for (int i=0; i < neighbours.length; i++) {
    if (neighbours[i].state == true) {
      liveCount++;
    }
  }

  if (state == true) {
    if ((liveCount == 2) || (liveCount == 3)) {
      nextState = true;
    } else {
      nextState = false;
    }
  } else {
    if (liveCount == 3) {
      nextState = true;
    } else {
      nextState = false;
    }
  }
}
```

このコードは、まず生きている隣接セルの数を数えてから、次にその数にGOLルールを適用しています。このコードを数秒間実行してみると、まるで抽象的な生物を顕微鏡で覗いているかのように見え始めるでしょう（図7.2を参照）。いくつかの部分が泡立つように動き、そして次第に消えていきます。他の部分は、一定の動きを繰り返す安定状態に達します。このゲームに関するたくさんの研究があり、数学的な「生命」のかたち――グライダー、ヒキガエル、船、ブリンカー、ビーコンなど――を同定しました。WikipediaのGOLのページには、この単純な規則から生じた、複雑な形やパターンの例の動画がたくさんあります：http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life#Examples_of_patterns。

今あなたが書いたコードは、本質的には人工知能プログラムなのですが、横道に逸れるのは止めておきましょう。人工知能はジェネラティブ・アートの創造にとっても、たくさんの可能性を持っている分野ですが、この本の範囲からは明らかに逸脱しています*2。私たちは引き続き、視覚的に面白い方法で描くため、各セルのローカル（行動が制限された）性に焦点を当てていきます。そのため人工知能についてはやや浅く、興味程度に扱っていきましょう。

＊2　このテーマに興味を持った人には、エンターテイメントとしてダグラス・ホフスタッターのピューリッツァー賞を受賞した『ゲーデル、エッシャー、バッハ』（1979年）をお勧めします。

単純な白黒の点は、出発点でしかありません。Chapter 5の多次元のパーリンノイズでやったように（回転、形、サイズ、アルファ値などでノイズ値を利用）、セルの属性を使ってさまざまな表現の実験をすることができます。図7.3と7.4の2つの例では、セルが持っている生きている隣接セルの数と、セルの年齢（それまで生きてきたステップ数）を、セルの形状やサイズ、回転、色を決めるのに使いました。ぜひ、こうした実験に挑戦してください。

GOLは多くの可能性がある規則の中の1つにしかすぎません。視覚化の実験だけでなく、独自の規則を考えてみてください。次に、さらに2種類のよく知られたパターンと、1種類の動きを紹介しましょう。それぞれが、自然界に似た形状や動きを生み出してくれます。

7.1.3　ヴィシュニアク・ヴォート

最初のパターン、「ヴィシュニアク・ヴォート」（最初にこれを実験したジェラード・ヴィシュニアクにちなんでいます）は、服従に対する教訓です。それぞれのセルは集団からの圧力を受けやすく、隣接セルの傾向に流されます。セルの色が多数派であればそれは変化しません。少数派なら変化します。変化を決めるセルの数を奇数にするため、8個の隣接セルの状態に加え、自分の状態も含んで計算します。

ヴォートの動きを試してみるために、次のコードのcalcNextState関数を書き直してください。人工的な不安定さを生み出すために、生きた隣接セルの数が4つか5つの場合、自分の状態を反転します。そうしないと、急速に静的なパターンに入り込んでしまうのです。

リスト7.3　ヴィシュニアク・ヴォート・ルール

```
void calcNextState() {
  int liveCount = 0;
  if (state) { liveCount++; }
  for (int i=0; i < neighbours.length; i++) {
    if (neighbours[i].state == true) {
      liveCount++;
    }
  }

  if (liveCount <= 4) {
    nextState = false;
  } else if (liveCount > 4) {
    nextState = true;
  }
```

自分を含めて隣接セルを数える

自分は多数派か？

（続く）

（リスト7.3の続き）

生きたセルが4と5の場合、状態を反転する

```
    if ((liveCount == 4) || (liveCount == 5)) {
      nextState = !nextState;
    }
  }
```

結果は図7.5の通りです。密集効果によって、牛の皮膚のような模様や、上空からの風景に似たパターンが、どのように生まれるかを見ることができます。

ヴィシュニアク・ヴォートのルールは、各エージェントが隣接セルグループからの圧力に屈するという意味でとても社会学的です。しかしその結果は、生物学や地質学以上に親しみやすく美しいものです。これは、共通の計算原理がこれらの規律を支えていることを意味しているのでしょうか？　本章、そして次章で、あなたが思わずこの仮説を支持したくなるアルゴリズムをたくさん紹介します。まず初めは「ブライアンの脳」です。

7.1.4　ブライアンの脳

ブライアンの脳は、3状態のセル・オートマトンです。セルにオン／オフだけなく、もう1つ第三の条件があります。ブライアンの脳のCAの状態は、「発火」「休息」「オフ」の3種類です。セルは、脳のニューロンの動きのように発火し、再び発火する前に休息します。ルールは次の通りです。

- もし状態が発火なら、次の状態は休息
- もし状態が休息なら、次の状態はオフ
- もし状態がオフで、2個の隣接セルが今まさに発火していれば、そのセルは発火する

そのために、次のコードのようにcellクラスを修正します。状態を表す変数の型をintに変えなければなりません。そうすることで2つ以上の値を持つことができ、整数の0、1、2が、それぞれオフ、発火、休息を表します。

リスト7.4　ブライアンの脳のために変更されたセルオブジェクト

状態1，2あるいは0

```
class Cell {
  float x, y;
  int state;
  int nextState;
  Cell[] neighbours;
```

（続く）

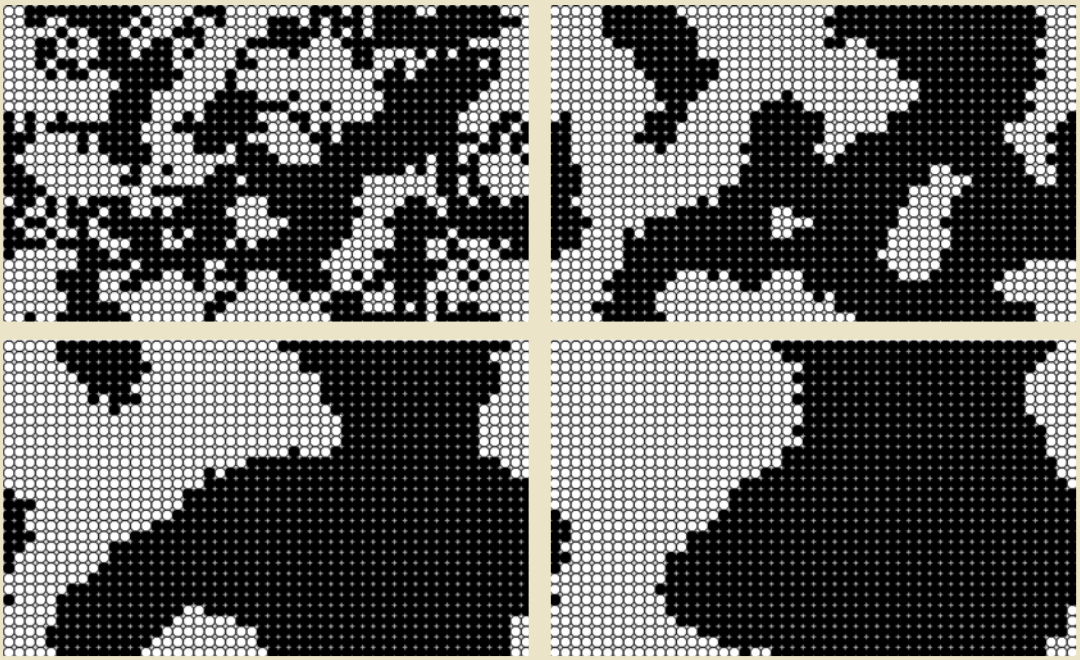


図7.5
3回、11回繰り返し、41回、166回繰り返したの後のヴィシュニアク・ヴォート。

（リスト7.4の続き）

```
    Cell(float ex, float why) {
      x = ex * _cellSize;

    y = why * _cellSize;
    nextState = int(random(2));
    state = nextState;
    neighbours = new Cell[0];
  }

  void addNeighbour(Cell cell) {
    neighbours = (Cell[])append(neighbours, cell);
  }

  void calcNextState() {
    if (state == 0) {
      int firingCount = 0;
      for (int i=0; i < neighbours.length; i++) {
        if (neighbours[i].state == 1) {
          firingCount++;
        }
      }
      if (firingCount == 2) {
        nextState = 1;
      } else {
        nextState = state;
      }
    } else if (state == 1) {
      nextState = 2;
    } else if (state == 2) {
      nextState = 0;
    }
  }

  void drawMe() {
    state = nextState;
    stroke(0);
    if (state == 1) {
      fill(0);
    } else if (state == 2) {
      fill(150);
    } else {
      fill(255);
    }
```

発火している隣接セルを数える

もし2個の隣接セルが発火していたら発火

それ以外の場合は変化しない

発火したなら休息

休息したならオフ

発火＝黒

休息＝グレー

オフ＝白

（続く）

（リスト7.4の続き）

```
    ellipse(x, y, _cellSize, _cellSize);

  }
}
```

結果は図7.6のようになります。宇宙船（starship；CA用語。私の用語ではありません）が、水平、垂直、ときおり対角線方向に平面を横切ります。

この動きは、発火するシナプスの電気的な刺激でかたちづくられる人間の考えを反映しているでしょうか？　あなたが見ているものは、意識の数学的モデルでしょうか？　おそらく違うでしょう。しかし私たちの目的、ジェネラティブな視覚表現の基本として使うことができる、もう1つの創発的な動きです。

先に進む前に、さらに1つの例を紹介します。それは有名な規則ではなく、カスタムな動きです。しかしそれが生み出すパターンには、なぜか親しみがあります。

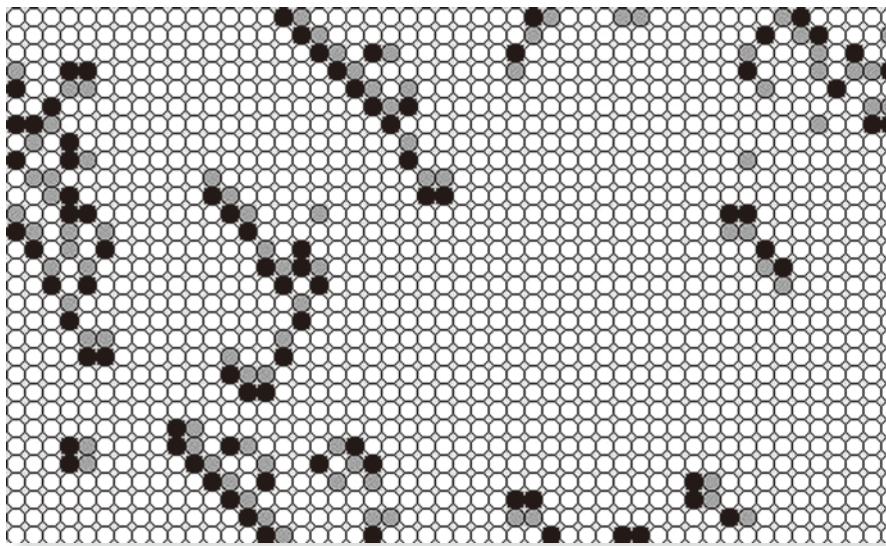


図7.6　ブライアンの脳の動き。19回繰り返し後。

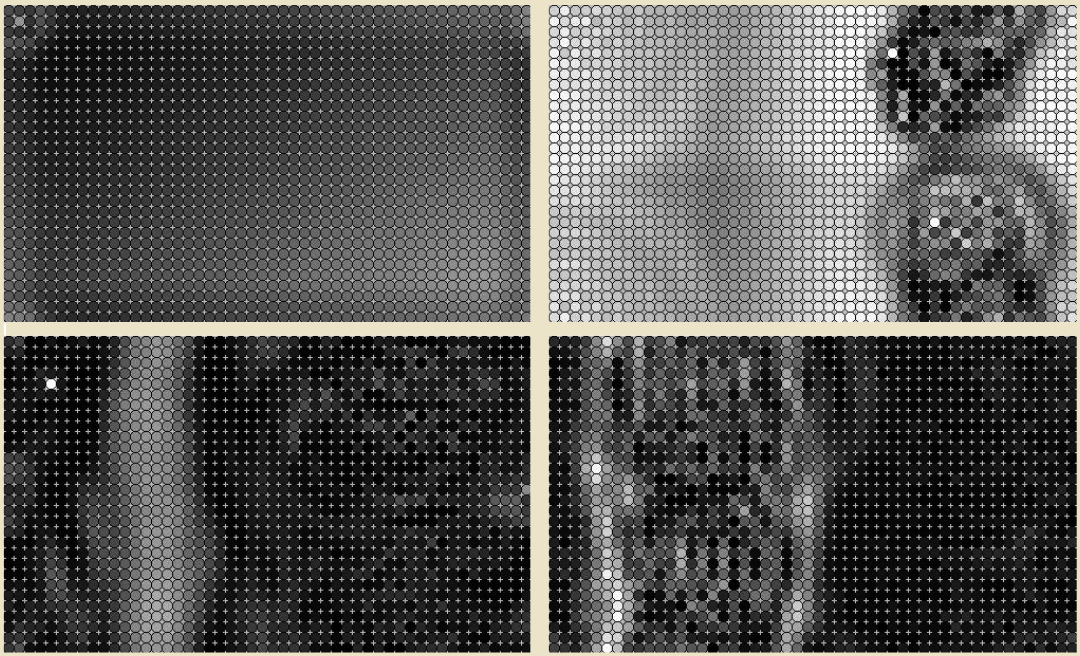


図7.7
カスタマイズされた波ルール。6, 26, 49, 69回の繰り返し後。

7.1.5　波（平均化）

最後のパターンは、連続した動きをもつCAの例です。セルが2あるいは3種類のとびとびの値でなければならない理由はありません。その状態は、値の範囲をまたがって様々あってもよいのです。この例では、255の値、つまり白から黒までのグレースケールを使います。これは独自の動きなので、固有の名前はありませんが、周囲の影響で混沌が静まっていく標準的な物理モデルの「平均化」を基本にしています。

ルールは次の通りです。

- もし隣接するセルの状態の平均が255であれば状態は0に
- もし隣接するセルの状態の平均が0であるなら状態は255に
- そうでなければ、新しい状態＝現在の状態＋隣接セルの状態の平均ー前の状態の値
- もし新しい状態が255を超えたら255にし、もし新しい状態が0以下ならそれを0にする

物理モデルであれば、上記のルールの3番目だけがあれば十分です。ここでは、安定を阻止する不安定性を作り出すために、他の3つのルールを加えました。コードは以下の通りです。その結果を図7.7に示します。

リスト7.5　波のような独自の動き

```
class Cell {
  float x, y;
  float state;
  float nextState;
  float lastState = 0;
  Cell[] neighbours;

  Cell(float ex, float why) {
  x = ex * _cellSize;
  y = why * _cellSize;
  nextState = ((x/500) + (y/300)) * 14;
  state = nextState;
  neighbours = new Cell[0];
  }

  void addNeighbour(Cell cell) {
    neighbours = (Cell[])append(neighbours, cell);
  }
```

初期状態を作る

（続く）

（リスト7.5の続き）

隣接セルの平均を計算

前の状態を格納

状態の値に応じた色をつける

```
    void calcNextState() {
      float total = 0;
      for (int i=0; i < neighbours.length; i++) {
        total += neighbours[i].state;
      }
      float average = int(total/8);

      if (average == 255) {
        nextState = 0;
      } else if (average == 0) {
        nextState = 255;
      } else {
        nextState = state + average;
        if (lastState > 0) { nextState -= lastState; }
        if (nextState > 255) { nextState = 255; }
        else if (nextState < 0) { nextState = 0; }
      }

      lastState = state;
    }

    void drawMe() {
      state = nextState;
      stroke(0);
      fill(state);
      ellipse(x, y, _cellSize, _cellSize);
    }

  }
```

この動きに最も似ているものは液体です。浅い水たまりに降る雨のように、揺れ動きながら変化します。

この例を出した理由は、私たちは規則だけでなく、規則の枠組みそのものから設定できる、ということを強調したかったからです。グリッドも円も必要としません。グレースケール値の限られたセットも必要としません。あなたのシステムは、細心の注意を払って作ることで、繊細で洗練されたものになり得るのです（図7.8参照）。次のセクションでは、私たちは整然としたグリッドのはるか彼方に出て、シミュレーションとビジュアライゼーションについて考えていきます。今までのように、創発的効果を生み出すローカルなエージェントを利用してはいますが、同時に、とてもよく知っているけれど忘れがちなオブジェクトである「あなた」を利用する可能性も、合わせて考えていきます。

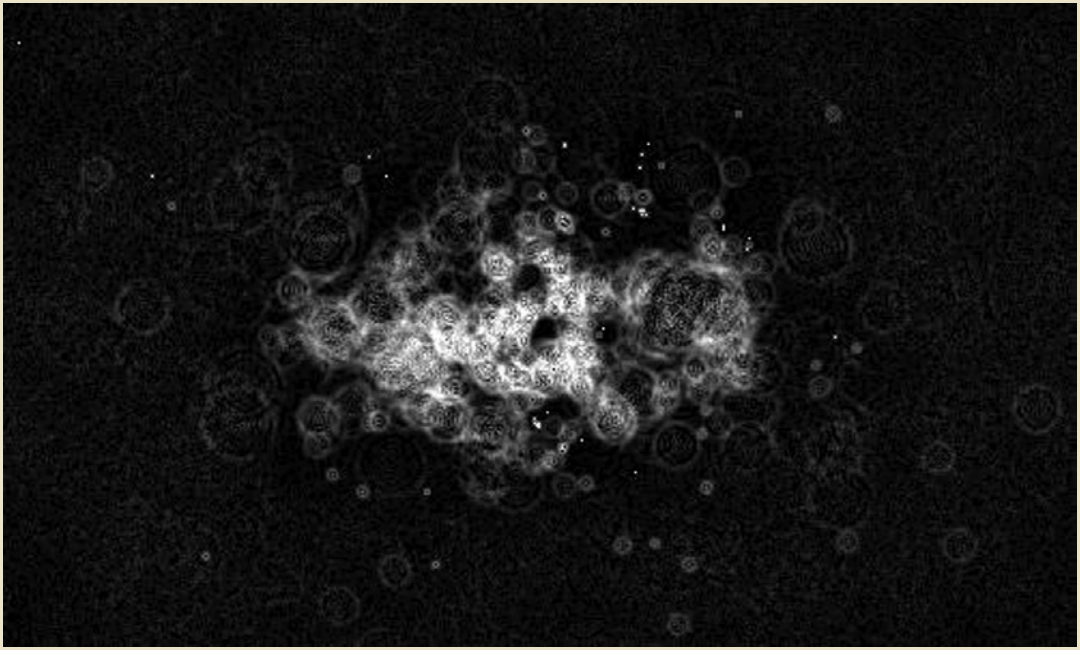


図7.8
1ピクセルサイズのセルが、とても繊細な（しかしプロセッサ負荷の高い）パターンを作り出します。

7.2　シミュレーションとビジュアライゼーション

私たちにとってCAは、単なる視覚的な面白さなのかもしれません。しかしこの実験の本当の目的は、エージェント指向アプローチに取り組むこと、そしてオブジェクトが知らず知らずのうちに作り出すパターンを学ぶことでした。でも、オタクっぽく楽しめるCAは、単にこうやって視覚化できるシステムのうち、最も単純なものでしかないのかもしれません。

この章のコーディング部分は終わりです。しかし、このアプローチをさまざまな方向に発展させていける余地がたくさんあります。この章の残りのページを利用して、そのうちのいくつかの考えを紹介しましょう。

7.2.1　ソフトウェア・エージェント

これまで何度もやったように、コードでクラスを定義する時には、属性とメソッド――x、y、height、width、alpha；update、draw、calcWidth等々――を与えます。しかし、オブジェクトの属性とメソッドは以下のように単純なものでした。

```
class Agent {
  String political_leaning = "right";
  int num_years_in_education = 12;
  int salary = 30000;
  Agent[] neighbors;
  Agent[] coworkers;
  Agent[] friends;

  Agent() {
    num_years_in_education = 11 + int(random(8));
    if ((num_years_in_education < 13) {
      if (random(1) > 0.8)) {
        political_leaning = "right";
      } else {
        political_leaning = "left";
      }
    } else {
      if (random(1) > 0.3) {
        political_leaning = "right";
      } else {

        political_leaning = "left";
```

（続く）

（続き）

```
            }
        }
    }
}
```

もしエージェントに実世界の属性を与えることができれば、シミュレーションは途端に有意義なものになるでしょう。単純化された精神との相互作用から、本物の精神との、単純ではない類似性について、何か言えるかもしれません。例えば、あなたはエージェントの考えや感情を聞き出すことができます。それはおそらくこんな感じになるはずです。

```
boolean isHappy() {
if ((political_leaning == "right") {
  if (salary > 60000)) {
    return true;
  } else {
    return false;
  }
} else {
    if (friends.length > 25) {
      return true;
    } else {
      return false;
    }
  }
}
```

もし無数のプログラマーに無数のMacBookを与えたら、という論理的な極限を突き詰めるために、全人類をコード化し、それをCAグリッドの中に詰め込んで、何が起こるかを観察してもいいでしょう。さらに面白くするために、人工的な都市環境をコード化してそこに人々を住ませて、どんなパターンが生まれるかを試してみてもいいかもしれません。しかしあなたがこれまで見てきたように、面白いエージェントにとって人間（あるい人間もどき）レベルの知性は不要です。複雑さを生み出すかどうかは、オブジェクトの精巧さとは関係ないのです。重要なのは、それらの量や相互作用する方法です。

同様のエージェント指向のシステムを用いた、私が好きな例の1つにジェレミー・ソープのColour Economy（2008）があります。彼は、色のついたたくさんの立方体がそれぞれ利益の追求をすることで、自分自身の色の価値を売買する能力を持ったシステムを作りました（図7.10参照）。それぞれのキューブが従う規則は同じですが、それぞれの立方体が、利己的な行動に影響を与える異なる傾向と特殊性をもっていました。この集合体は、不安定な経済システムに相当しています。

ソープは、立方体とそれらの取り引きを、2次元と3次元の世界で視覚化しました。Processingで作られたこのシステムの素晴らしいビデオがジェレミーのブログにあります：http://blog.blprnt.com/blog/blprnt/the-colour-economy-the-gap-between-the-rich-and-the-poor。

ソープの立方体間の相互作用は、本当の意味での人工知能ではないですが、明らかに文化に対応するモデルと言えるだけの十分な複雑さがあります。彼の作品は、ただ美的に素敵なものであることを超えていました。それは消費者運動、自由貿易、金持ちと貧乏人の間の格差を反映した、社会政治学的な論点をもつ作品だったのです。

あなたがエージェントを人間らしくしようとすれば、きっと最良の近道が見つかると思います。

7.2.2 人間エージェント

アポロ17号のミッション（NASAの最後の月面着陸）は、20世紀で最も重要な写真の1枚をもたらしました。図7.9の地球の写真（地球の軌道から外に出た時に撮影）で、毛のないサルたちが初めて集団として自分たち自身の惑星を見たのです。この写真の以前にも、とても詳細な世界地図によって地球がその外側からおおかたどんなふうに見えるかを知ってはいました。しかし何といってもこの写真が、遥か遠くから自分たちを眺望した最初の経験となりました。それは、私たち人類の主観性にとっての大躍進でした。

図7.9 地球の「ブルーマーブル」写真。1972年12月、アポロ17号によって撮影。

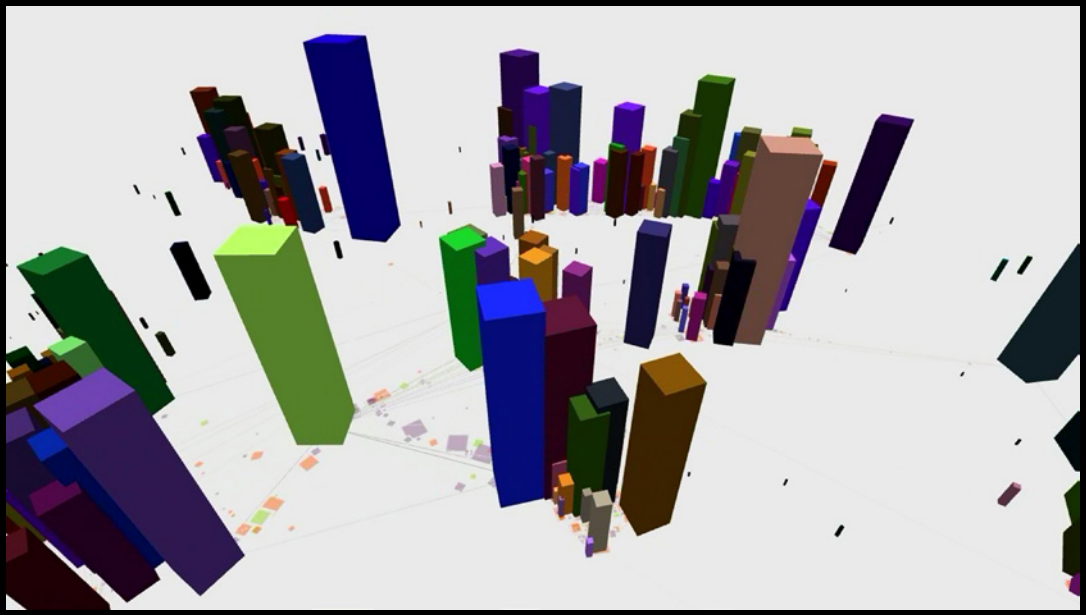


図7.10
Colour Economy、ジェレミー・ソープ（2008）

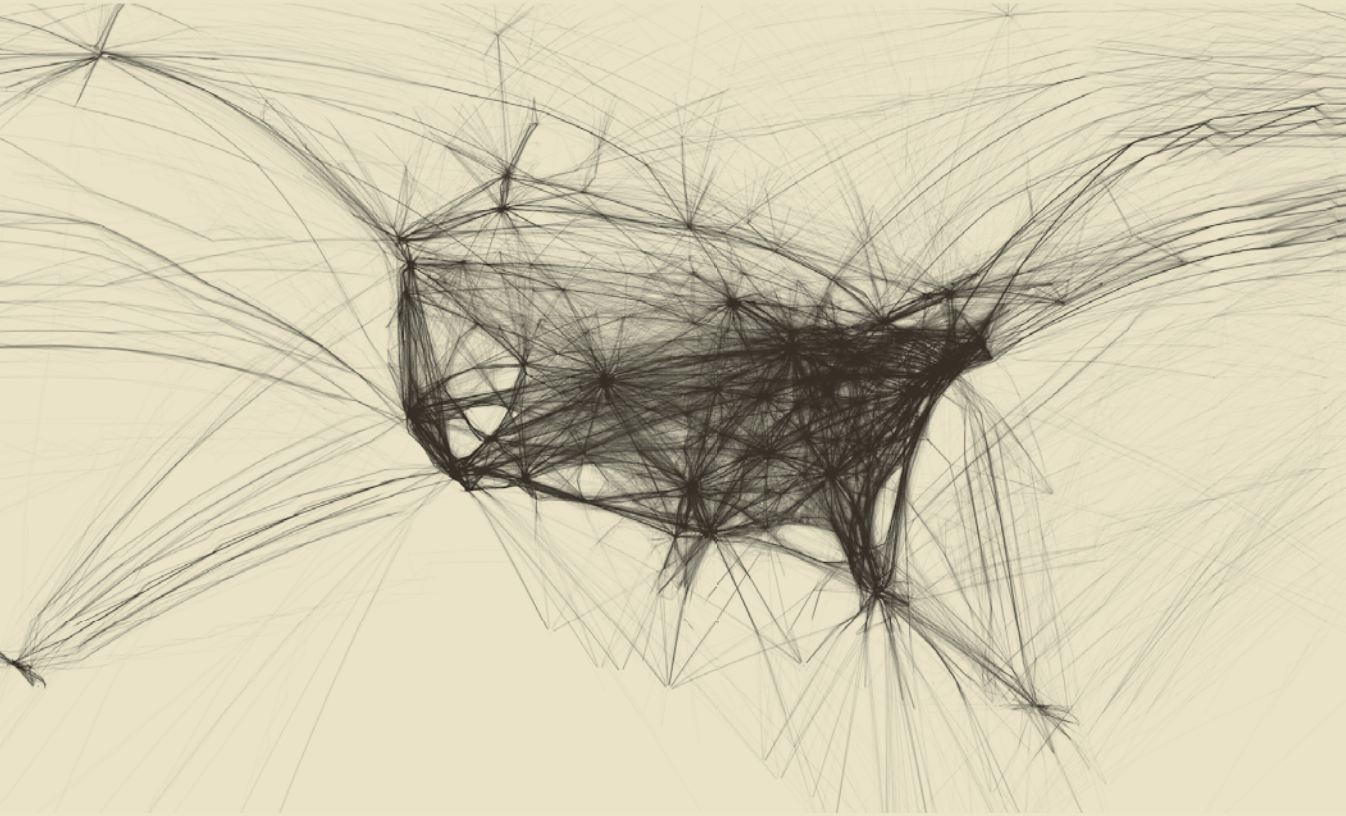


図7.11
アーロン・コブリンのFlight Patterns (2005)。
北アメリカを飛んでいる航空機の飛行経路の視覚化。

Googleマップ世代にとって、人類のこのメタ・ビューはもはやありふれたものになってしまいました。衛星画像の威力があらゆるラップトップにやってくるまで、わずか40年しかかかりませんでした。情報技術が驚くべきスピードで加速することで起こった、ちょっとした奇跡です。これを、全地球測位システム（GPS）という関連技術と結びつけてください。するとあなたは、知らず知らずのうちに新しい形の芸術表現の可能性を手にしています。

あなたの毎日の動きが、惑星の表面に足跡を残します。そしてこの足跡はパターンを描きます。もしあなたが通勤していれば、明けても暮れても同じパターンを繰り返すように、あなたのGPSの足跡が深く道筋を残します。けれども週末には、あなたのランダムな心理地理学が、ずっと複雑なバレエを見せてくれるかもしれません。アーロン・コブリンの、北アメリカを渡っている航空機の飛行経路の視覚化のように、全体として見れば、あなたのデータは雄大な表情を作り出すでしょう（図7.11、図i.18参照）。

本書の冒頭で、私がジェネラティブ・アートとは何か（何でないか）を定義した章を思い出してください。重要な基準の1つは、システムが自律していて、外部からの操作がなく、導き手から自由であることでした。人間をジェネラティブなシステムの一部に含めると、人間の生まれついての本能が、それに影響を与えようとするでしょう。もし人間のシステムへの関与が意識的であれば、それは自律的ではなくなります。もし人間エージェントが、自分たちの行動がシステムに与える影響に気づかないか、あるいは無関心であれば、他の自律したオブジェクトと同様に、有効なデータソースになり得ます。

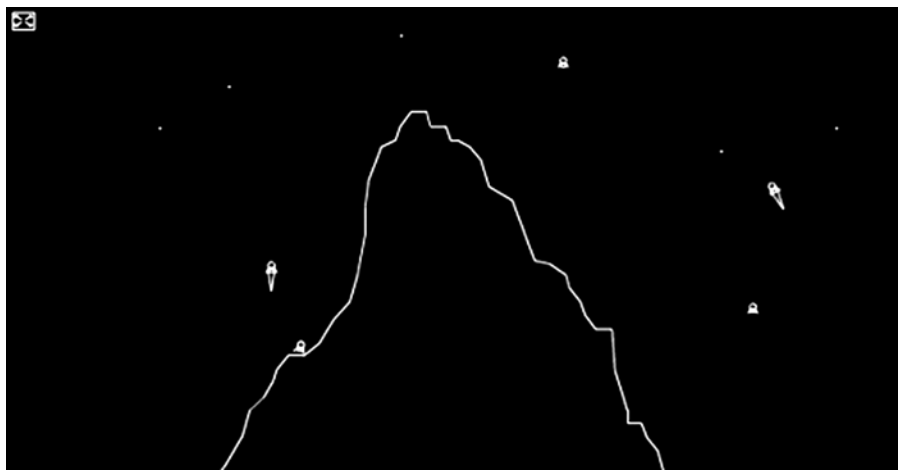


図7.12　Lunar Lander。ActionScriptで再現された典型的なアーケード・ゲーム。

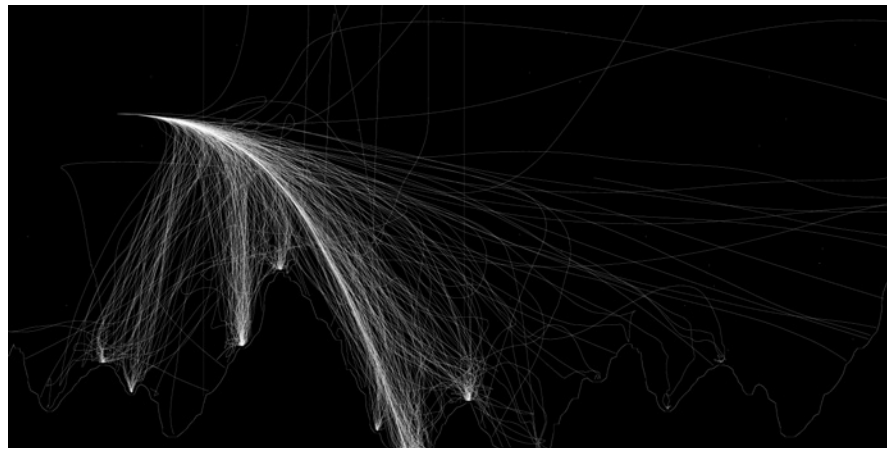


図7.13　Lunar Lander Trails (2010)、セブ・リー・デリスル。多くの自律した人間エージェントの行動。

この素晴らしい例は、私の友人であり同僚のセブ・リー・デリスル (http://sebleedelisle.com) が作ったものです。コーディング・ミニマリズムの実習で、彼はわずか5KB以下で、古いLunar Landerアーケード・ゲームをAS3 (ActionScript3.0) で作りました、その後彼は、Web上で多くのユーザーがゲームをするためのシステムである、マルチユーザーゲームの実験の基盤にこれを使いました。セブは、彼のサイトを訪れる一般の人々に遊んでもらうために、このオモチャをWebの世界に投げ出しました。そして彼は、ユーザーが安全に月の表面に船を着陸させた時にとった道筋を、ユーザーたちに知られずに、システムが記録するようにしました。

彼がこのデータを合成して描き出したイメージは、意図やデザインなどがなしに、驚くほど美しいものでした (図7.13参照)。

ジェネラティブ・アートを制作する際に、人間のエージェントを使うことにおいて2つの難しさがあるように思います。人間の自律性を保証することと、データを集めることです。かつては後者の方が困難な作業でした。しかし今日、そちらはとても簡単です。現代においてデータは、(人間の消費や習性については特に) 最も価値があるかのように日々大量に保存されています——異論はあるかもしれませんが。

たいていのWebや携帯機器のユーザーは、データの軌跡を定期的に残します。Twitterストリーム、Webブラウジング、GPS位置データには、プログラマーが利用可能なさまざまなAPI (通常は無料です) を通じてアクセスできます。例えばジェレミー・ソープによる最近の作品は、ニューヨーク・タイムズのAPI (http://blog.blprnt.com/blog/blprnt/7-days-of-source-day-6-nytimes-graphmaker) をもとに、新聞のデジタルアーカイブ中の主要語句の移り変わり

をProcessingで視覚化したものです。このシステムのエージェントは、仕事をし、記事を書き、視覚化するシステムに気づかずに言語用法の共通メタパターンを作っているジャーナリストたちです。

データ・ビジュアライゼーション自身が大きな可能性を持った芸術様式と言えるでしょう。もしあなたがさらにこのエリアを探索したいなら、私はその出発点として次のいずれかを試してみることを提案します。

- ベン・フライの本『ビジュアライジング・データ——Processingによる情報視覚化手法』(オライリー、2007年) は、本書同様にProcessingを使用して、このトピックを深く掘り下げています。
- コーヒーブレイク版としては、ジェレミー・ソープが書いたブログがあります。とりわけhttp://blog.blprnt.com/blog/blprnt/your-random-numbers-getting-started-with-processing-and-data-visualizationは、TwitterのデータをGoogleDocsのスプレッドシートに取り込んで、そのスプレッドシートをProcessingで読み、結果を視覚化するという内容です。
- もし一直線に美しい絵を作りたいなら、www.informationisbeautiful.netとwww.visualcomplexity.comで、あなたの目を楽しませてくれるものが見つかるでしょう。

7.3 まとめ

オブジェクト指向アプローチによるプログラミングは、出発点にしかすぎません。このアプローチを用いることで、いっそう洗練されたシステムを実現することができるでしょう。あなたのオブジェクトは、人工環境の中で相互作用し、あなたが視覚化しようとする創発的複雑さ(セル・オートマトンの実験を通して探究したアイデア) を生み出す、自律したエージェントになり得ます。そして、エージェントは厳密な構築物である必要はなく、またそれらがソフトウェアをベースにしている必要さえないことも知りました。データ・ビジュアライゼーションの可能性は、芸術の新しい分野を切り拓いていきます。

自分自身を生データの中の小さなピクセル (神の大きなゆで卵タイマーの中の砂粒) として見る経験という小旅行の後では、自分自身を貴重な小さな花のようには感じられなくなったかもしれません。その経験をふまえて、次に私たちは、マクロからミクロに再び焦点を合わせ、もう1つの自然構造の複雑さ、「フラクタル」へと向かいます。最終章では、フラクタルが何であり、それをどのように作るべきか、そしてあなたがそれで何ができるかを検討します。

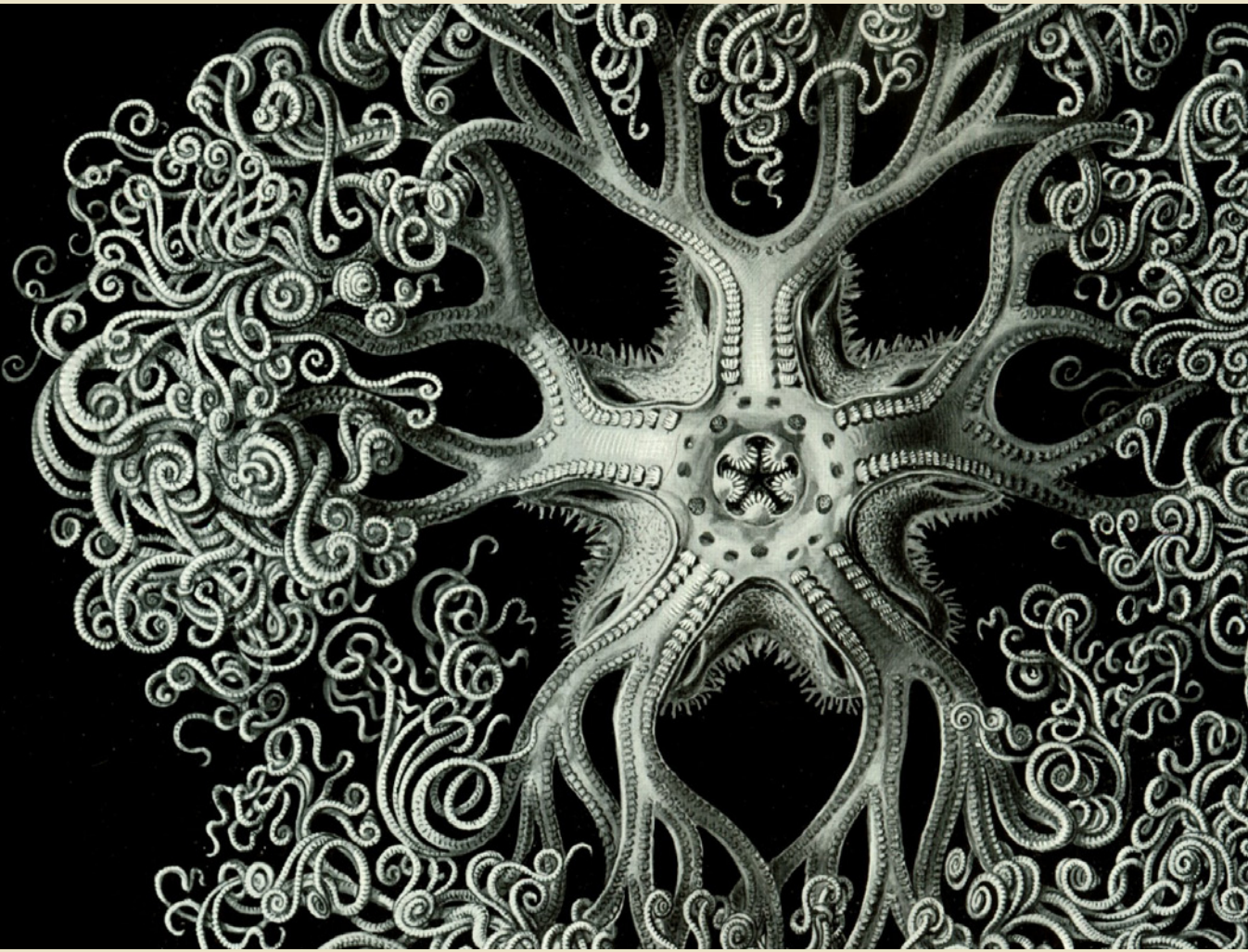


図8.1
フラクタル構造の実験を行えば、自己相似再帰がどこにでもあることに気づき始めるでしょう。このイラストは、エルンスト・ヘッケルの1904年の本『生物の驚異的な形』に収められているものです。

Chapter 8

フラクタル

Fractals

私は、フラクタル芸術にはとても「70年代」的なものがあると思っています。ジェームズ・グリックの非常に人気のある本『カオス――新しい科学をつくる』(1987) ――この本はカオス理論の数学を世に広めました――以前にも、フラクタルイメージ、特にマンデルブロ集合は、数学専攻卒のヒッピーの壁によくある類いのポスターとしてよく知られていました。

初期のフラクタル芸術の表現者が、なぜこのようなけばけばしいサイケデリックな色を好んだのかはわかりませんが、私はそれがLSDの過剰摂取と大いに関係があったと推測しています。ともかく、この章で自己相似性を探究して、私はフラクタル芸術に対するこの先入観が、みなさんの心から洗い流されることを願います。フラクタル芸術は、緻密で美しいものでもあります (図8.1と8.2参照)。

本書の最後の章では、Chapter 6で手に入れたオブジェクト指向プログラミングを使って、あなたのプロセッサの能力が許す限り、このことにもう少し関わっていきましょう。しかし、これはジェネラティブ・アートがそうであるのと同じぐらい難しいことです。もしあなたが自己相似再帰的なフラクタル構造にとりつかれて、それをコードで再現できるようになったら (もう数ページで、自信をもってできるようになります)、物理ライブラリ、機械システム、流体シミュレーション、パーティクル、AI、サウンドに反応するアート、あるいはこの本を置いた後にあなたが追求したいかなる探検であっても、その利用の仕方を発見できるようになるでしょう。そうすればあなたには、もう何も障壁はありません。

「フラクタル (fractal)」は、ラテン語のfractus (「砕けた」の意味) からきていて、たくさんのレベルで反復する形やパターンのことです。パターンは、必ずしも異なるスケールで同一である必要はありません。それらはある特定の種類の、自己相似構造を共有しています。創発のパターンと同じように、フラクタル構造は自然のいたるところにあります (雪片、木の枝、川、海岸線、血管)。シダの葉 (図8.3参照) はとりわけ良い例で、それぞれの葉の葉状体は、全体の葉の形を反映しています。より近くで詳細に葉状体を調べてみてください。そうすれば、その下位部分に再び同じ形状が現れていることがわかります。この自己相似性は、人間の目の限界を

図8.2

Mandelbulb、トム・ベダード（2009）、www.subblue.com。
3Dバージョンのマンデルブロ・フラクタル。
数学的処理についてはベダードのサイトあるいはwww.skytopia.com/project/fractal/mandelbulb.htmlを参照。

図8.3
Ghost Ferns（Flickrユーザーのロウ・ブリューの写真）。自然の中のフラクタル構造。

越えて、小さいスケールへと繰り返し続いていきます。

本章では、私たちはフラクタルの反復を、芸術を制作する方法として見ていきます。あなたは既にforループ、whileループ、drawループを通じて再帰に精通しています。しかし、それは異なった種類の繰り返しです。抑制なく、無限へと繋がっていく傾向がある、再帰的な繰り返しです。

8.1　無限の再帰

あなたは鏡の前に立っていて、2枚目の鏡があなたの後ろにあると想像してください。あなたは、反射の反射の反射の、そのまた反射の、という認識の限界に向かって突き進んでいく反射を見ます。あなたがこのような無限再帰の中にいると、奇妙なことが起こります。小さな変化、ミクロな動きが、この閉じられたループの中でものすごく拡大されます。こうした鏡の反復の中に炎を持ち込んでみてください（さあ、本を下に置いて、やってみましょう。とてもクールですね）。炎のゆらぎのミクロなランダムさが無限に繰り返されると、奇妙な抽象を作ります。同様に、多くのエージェントの間の小さな相互作用が、前章にあった創発的複雑さを呼び起こしたように、無限の中の小さな変化や自己相似構造が、よく似た複雑さを、しかしより言葉にできない

スケールで作り出します。

コンピュータは無限の概念を扱うことができません。コンピュータに無限の回数で繰り返すようには頼めません。なぜなら、命令に盲目的に従うコンピュータは、決して停止しないからです。コンピュータは、あなたが中断するまで永遠にループし続けます。これはアラン・チューリングの停止性問題です。彼は、無限ループが決して結果を返すことができないことに加えて、与えられた命令の集合（プログラム）が無限ループになるのか、あるいは最終的に停止するのかどうかを決定することが不可能であることを証明しました。これはAI分野における重要な問題であり、これまで慎重に議論してきた複雑性の究極の形式です。問題が有限に解決可能であるかどうかは人間が直感的にできることですが、機械はこの能力を簡単にはシミュレートできません。

例えば、あなたはひと目で、次のコードは決して停止しないことがすぐにわかります。

```
int x = 1;
while (x>0) {
  x++;
}
```

これは無限ループの一例です。文法としては完全に正しいのですが、何かがこれを中断するまでプロセッサを終わることのない反復の世界に閉じ込めてしまいます。無限ループをコードするのは簡単ですが、それこそ無限につまらない行為です。

次のセクションで再帰的な繰り返しのコーディングを行う際、このことは覚えておかなければなりません。自然界の中のフラクタルは無限に再帰できる可能性を持っているけれども、あなたはただ電子機器の中で無限の再帰をシミュレートできるだけ、ということです。停止性問題については、特に心配しなくてもいいです。なぜなら、チューリングは停止するかどうかの完璧なテストが不可能であることを証明したけれど、現代の構文エンジンは通常、プロセッサをおかしくするようなほとんどの状態に対して警告を出します。そして、たとえエンジンが警告しなかったとしても、あなたはただProcessingを終了すればいいだけです。

それでは、面倒なことはここまでにして、さっそく何かコーティングし、あなたのマシンが実行したものを見てみましょう。

8.2　自己相似性のコーディング

人生の中で避けられないものは、ご存知のように、死、税金、そして親になったりすることです。それこそ、子オブジェクトが親オブジェクトにそっくりなのは、そんなに不思議なことではありません。どのようにしたら自己相似性を作り出せるでしょうか。そのためには、子と親の両方として振る舞い、自分のコピーをたくさん作るオブジェクトを定義します。そしてもちろん、オブジェクトの決定的な特徴の1つが増加することであれば、オブジェクトは必然的に、ウサギのように繁殖していきます。

その構造と、よく知られた比喩から始めましょう。

8.2.1　幹と枝

コードの最初の行で、いくつかの制限を設定します。制限なしではあっという間に無限の再帰の罠の中に落ち込んでしまいます。新しいスケッチを開いて、以下のコードを入力してください。

```
int _numChildren = 3;
int _maxLevels = 3;
```

それぞれのオブジェクトが持つことのできる子の数と、システムがどのくらいの再帰の深さのレベルまでいけるかを制限します。初めは数を少なくしておけば、あなたはそこで起こっていることをよく理解することができます。何か面白いことが起こり始めたら、数を次第に多くしていけばいいのです。けれども、子やレベルの数を1つ増やすことで、プロセスの数は1つ増えるどころか、その掛け算にもなりません。プロセスの数はべき乗で増えていくのです。レベルを3、子を3つ増やすことは、3の3乗、すなわちオブジェクトが27増えることを意味します。もう1つ子を付け加えると3の4乗になり、それは81のオブジェクトに相当します。5つのレベルで5つの子では、3,125のエレメントを操作しなければなりません。これが慎重にやらなければならない理由です。

これから作るクラスをBranch［枝］と呼ぶことにします。木は役に立つモデルです。最初のBranchは木の幹です。それをコード化して、この比喩がどのくらい拡張できるかを見てみましょう。次のリストがその出発点です。

リスト8.1　フラクタル構造をコーディングする。ステップ1

```
int _numChildren = 3;
int _maxLevels = 3;

Branch _trunk;

void setup() {
  size(750,500);
  size(750,500);
  background(255);
  noFill();
  smooth();
  newTree();
}

void newTree() {
  _trunk = new Branch(1, 0, width/2, 50);
  _trunk.drawMe();
}

class Branch {
  float level, index;
  float x, y;
  float endx, endy;

  Branch(float lev, float ind, float ex, float why) {
    level = lev;
    index = ind;
    updateMe(ex, why);
  }

  void updateMe(float ex, float why) {
    x = ex;
    y = why;
    endx = x + 150;
    endy = y + 15;
  }

  void drawMe() {
    line(x, y, endx, endy);
    ellipse(x, y, 5, 5);
  }
}
```

このコードがこれまでのところでやっているのは、

- Branchオブジェクト（幹）のインスタンスを定義して作成する。
- そのオブジェクトのdrawMeを呼び出す。

Branchオブジェクトには、4つのパラメータ（level、index、x、y）があります。最初の枝は、つまり幹は1つしかないので、そのlevelは1で、indexは0です。xとyパラメータは、木を植えたいと思う任意の場所です。オブジェクトは、作られる時にupdateMeを呼んで、その終点をあなたが渡した開始点をもとに計算します。drawMe関数はそれら2つの点を線で結んで、その出発点に小さな円を描きます。これまでのところは簡単ですね。

次に、自己参照しましょう。Branchの属性にchildren配列を加えて、Branchの作成関数を、自分自身の新しいバージョンを生成するようにアップデートしてください。それぞれの子には、親の枝の終点をその開始点として渡して、現在のレベルを＋1します。条件文（level <= _maxLevels）を含んでいれば、無限ループは避けられるでしょう。

```
Branch [] children = new Branch[0];

Branch(float lev, float ind, float ex, float why) {
  level = lev;
  index = ind;
  updateMe(ex, why);

  if (level < _maxLevels) {
    children = new Branch[_numChildren];
    for (int x=0; x<_numChildren; x++) {
      children[x] = new Branch(level+1, x, endx, endy);
    }
  }
}
```

この変更の結果、1つのlevel-1の枝が3つのlevel-2の子を作り出します、それぞれのlevel-2の枝がさらに3つのlevel-3の子を作り出します。枝の合計数はしたがって、1＋3＋9＝13です。

すべての枝を描くために、drawMe関数を修正します。

```
void drawMe() {
  line(x, y, endx, endy);
  ellipse(x, y, 5, 5);
  for (int i=0; i<children.length; i++) {
    children[i].drawMe();
  }
}
```

アプリケーションは、木の幹のdrawMeだけを呼び出します。命令は徐々に子に流れていきます。

それぞれの枝を区別できるようにするために、外観を少し変えてみましょう。Branchが自分のlevelとindexを知ることができると便利です。そのために以下のようにupdateMeを、レベルに応じた重みをかけた乱数で終点を計算するように変えてください。

```
void updateMe(float ex, float why) {
  x = ex;
  y = why;
  endx = x + (level * (random(100) - 50));
  endy = y + 50 + (level * random(50));
}
```

さらに、drawMe関数にstrokeWeightの変化を加えてください。

```
strokeWeight(_maxLevels - level + 1);
```

このプログラムを実行すれば、フラクタルの木が始まります（図8.4参照）。

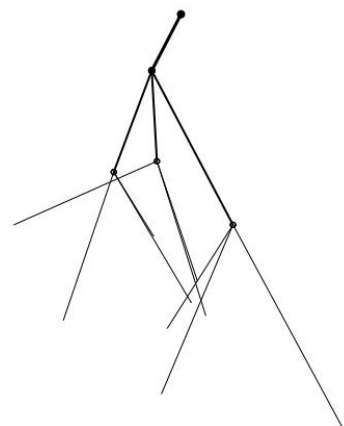


図8.4　フラクタルの木の始まり。

次に、これを動かしてみましょう。

8.2.2　あなたの木を動かす

オーケー。あなたには既に木の構造があって、まだ機械も壊れていません。さあ、今からdrawループでこいつを動かしてみましょう。

```
void draw() {
  background(255);
  _trunk.updateMe(width/2, height/2);
  _trunk.drawMe();
}
```

枝を繋ぐ場所をステージの中央に移動しました（いい感じにするために、newTree関数も変えたいかもしれませんね）。次に、描画スタイル、線の長さ、回転を操作するために、いくつかの新しい属性をBranchクラスに加えます（以下の行をfloat, endx, endy;の下に加えます）。

```
float strokeW, alph;
float len, lenChange;
float rot, rotChange;
```

まず、これらの属性を初期化しなければなりません。枝のレベルに応じたstrokeWeightとalphaを与え、さらにランダム値による長さと回転の属性を与えます。Branchコンストラクタ関数の中の行updateMe(ex, why);の前に、このコードを加えてください。

```
  strokeW = (1/level) * 100;
  alph = 255 / level;
  len = (1/level) * random(200);
  rot = random(360);
  lenChange = random(10) - 5;
  rotChange = random(10) - 5;
```

この新しい属性を用いて、updateMe関数が毎フレームごとに呼ばれるたびに、実際に何かをさせることができます。また、コマンドを子に伝えるコードを書けば、drawMe関数でやったように、子もまた何かしらをすることができます。この新しいバージョンでは、終点は角度と長さによって計算され、半径lenの円の円周上の点として描かれます。その点への角度と線の長さは、毎フレームごとに変わります。新しいupdateMe関数は以下の通りです。

リスト8.2　回転角度によって終点を計算するupdateMe

回転を増やす

長さを増やす

```
void updateMe(float ex, float why) {
  x = ex;
  y = why;

  rot += rotChange;
  if (rot > 360) { rot = 0; }
  else if (rot < 0) { rot = 360; }

  len -= lenChange;
  if (len < 0) { lenChange *= -1; }
  else if (len > 200) { lenChange *= -1; }

  float radian = radians(rot);
  endx = x + (len * cos(radian));
  endy = y + (len * sin(radian));

  for (int i=0; i<children.length; i++) {
    children[i].updateMe(endx, endy);
  }
}
```

あなたが定義、初期化したstrokeWとalph属性を使うように、オブジェクトのdrawMe関数をアップデートします。同様に、新しい変数のlengthの値に応じて、楕円の半径が変わるようにします。

```
void drawMe() {
  strokeWeight(strokeW);
  stroke(0, alph);
  fill(255, alph);
  line(x, y, endx, endy);
  ellipse(endx, endy, len/12, len/12);
  for (int i=0; i<children.length; i++) {
    children[i].drawMe();
  }
}
```

スキップして先に進みたい人には、最後のコードは次のセクションの終わりのリスト8.4です。この段階でのアウトプットは図8.5のようになります。ただしあなたのバージョンでは、それは「宇宙のゼンマイ時計」のように、狂ったように回転するでしょう。今、私たちは意図的に木の比喩から離れています。私がこれまでに読んだことがあるフラクタルプログラミングの教科書はすべて、木を描いて、それを風の中で揺らす方法を教えていました。でも、コンピュータが自然よりうまくできるものもいくつかあります。その1つが、この「宇宙のゼンマイ時計」

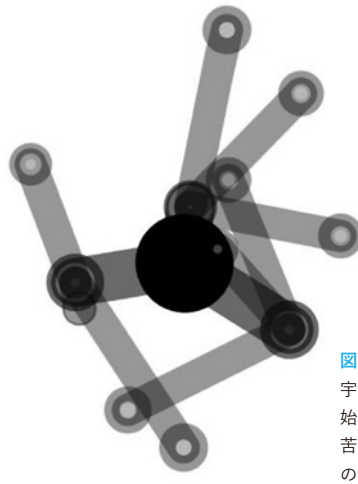


図8.5
宇宙のゼンマイ時計。動くフラクタルの始まり。ダルウィーシュ［イスラム教の苦行僧で、身体を回転させて祈祷する］のように回転します。

です。それをさらに突き進めてみましょう。

8.3 指数的成長

通常、章の中でのこの時点でのメッセージは、あなたのプログラムで何ができて、そのためにはどのファイルを参照すればよいのかを伝えることでしょう。けれども今回は、もう少し注意深く進みます。もし _numChildren と _maxLevels をゼロにすれば、システムはすぐに停止します。その代わりに、少しずつ数を増やしていってみてください。次のページに、_num Children = 4; _maxLevels = 7;（図8.6）と _num Children = 3; maxLevels = 10;（図8.7）の時のイメージを示します。そろそろ私の機械が不平を言い始めました。この不定形のかたまりはまだ回転していますが、もう活気がありません。

strokeW = (1/level) * 10; の1行を入れて線を細くすると、図8.8のようになります。さらに条件文（if (level > 1)）を加えて、drawMe 関数が幹（図8.7では、はっきりと見えます）を描くのを止めました。

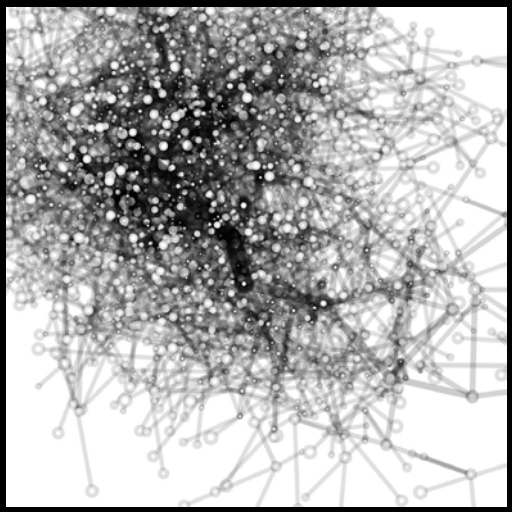


図8.6
4つの子、7つのレベルへと少し大きくしたもの。
この構造物がまだスムーズに動いていることを願っています。

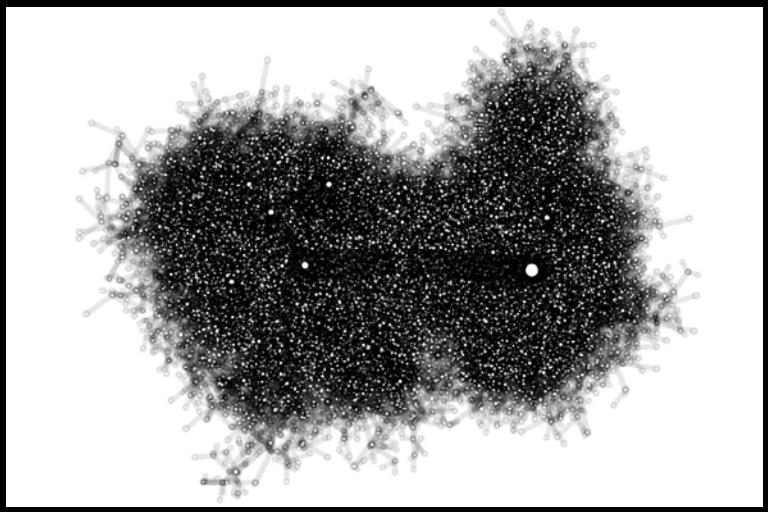


図8.7
3つの子、10のレベル。

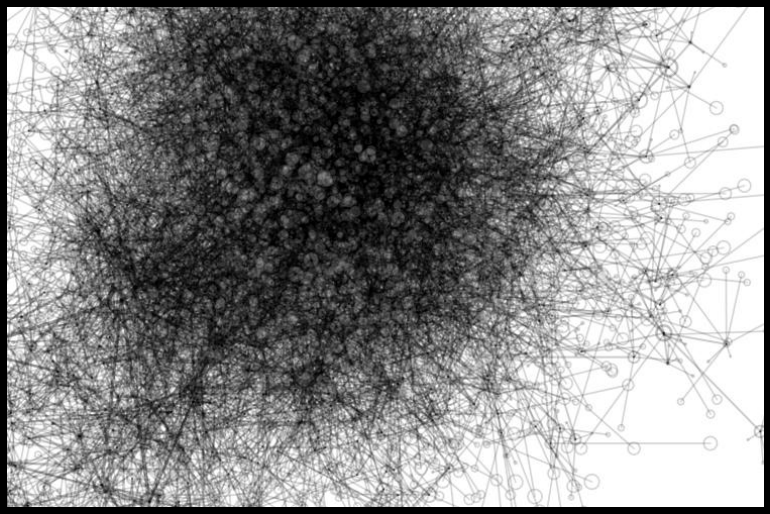


図8.8
より細い線にしたもの。7つの子と6つのレベル。

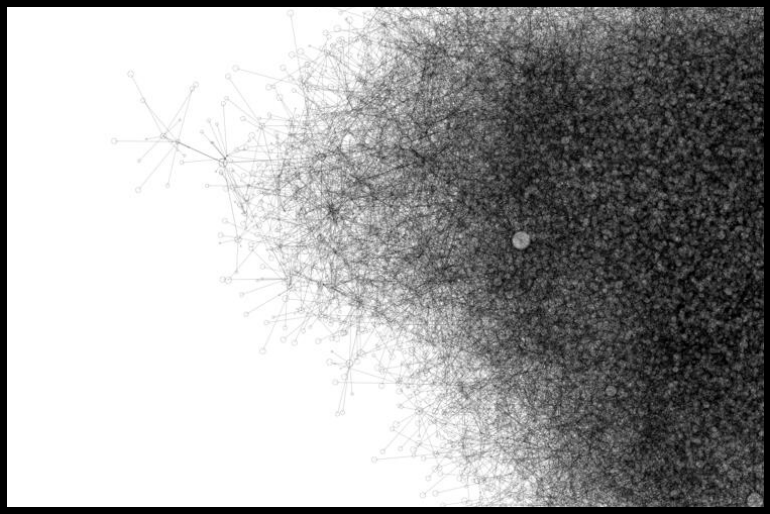


図8.9
7つの子、7つのレベル。動くことが期待できるほぼ限界。

図8.8は、7つの子と6つのレベルにしたものです。さらにそれを押し進めて7つのレベルにすると、図8.9のようになります。それが私の機械の上で動く、ほぼ限界でした。

この最終バージョンの完全なコードを以下に示します。

リスト8.3　歯車フラクタルのためのコードの最終版

```
int _numChildren = 7;
int _maxLevels = 7;

Branch _trunk;

void setup() {
  size(750,500);
  background(255);
  noFill();
  smooth();
  newTree();
}

void newTree() {
  _trunk = new Branch(1, 0, width/2, height/2);
  _trunk.drawMe();
}

void draw() {
  background(255);
  _trunk.updateMe(width/2, height/2);
  _trunk.drawMe();
}

class Branch {
  float level, index;
  float x, y;
  float endx, endy;

  float strokeW, alph;
  float len, lenChange;
  float rot, rotChange;

  Branch[] children = new Branch[0];

  Branch(float lev, float ind, float ex, float why) {
```

（続く）

（リスト8.3の続き）

```
      level = lev;
      index = ind;

      strokeW = (1/level) * 10;
      alph = 255 / level;
      len = (1/level) * random(500);
      rot = random(360);
      lenChange = random(10) - 5;
      rotChange = random(10) - 5;

      updateMe(ex, why);

      if (level < _maxLevels) {
        children = new Branch[_numChildren];
        for (int x=0; x<_numChildren; x++) {
          children[x] = new Branch(level+1, x, endx, endy);
        }
      }
    }

  void updateMe(float ex, float why) {
    x = ex;
    y = why;

    rot += rotChange;
    if (rot > 360) { rot = 0; }
    else if (rot < 0) { rot = 360; }

    len -= lenChange;
    if (len < 0) { lenChange *= -1; }
    else if (len > 500) { lenChange *= -1; }

    float radian = radians(rot);
    endx = x + (len * cos(radian));
    endy = y + (len * sin(radian));

      for (int i=0; i<children.length; i++) {
        children[i].updateMe(endx, endy);
      }
    }

    void drawMe() {
```

（続く）

幹を描かない

（リスト8.3の続き）

```
      if (level > 1) {
        strokeWeight(strokeW);
        stroke(0, alph);
        fill(255, alph);
        line(x, y, endx, endy);
        ellipse(endx, endy, len/12, len/12);
      }
      for (int i=0; i<children.length; i++) {
        children[i].drawMe();
      }
    }

  }
```

これまでは、1つずつ枝分かれをしていくフラクタル構造を見てきました。たぶんこれは、私たちが考える中では最も単純なフラクタルです。でも、あなたが探究できる他の形の、長ったらしいコードを載せるのは止めておきましょう。いじくり回すのはあなたに任せます。その代わりに、私はある1つの例で、この章とこの本を終えたいと思います。それは、ある暗く曇った夜に、賢明な老数学者が私に見せてくれた形です。それ以来、私はこの形でずっと楽しんできました。

8.4　ケーススタディ：サトクリフ五角形

2008年、私はロンドンでコンピュータ・アーツ・ソサエティ（CAS）の会議に参加しました。コンピューティングに非常に貢献してきたこの学会の、40周年記念のお祝いイベントでした。私は、学会の創始者で最初の会長、アーティスト、数学者であるアラン・サトクリフ*1の短い講演を聴きました。この講演で、彼は私に素晴らしい形を紹介してくれました。それ以来、私は心の中でそれを「サトクリフ五角形」と名づけています。

私はこのことについてはっきりとさせたくて、アランと話をしたことがありますが、私がアランの名前を使うということについて、特に天にも昇るような心持ちには見えませんでした。彼は、その形は自分が思いついたけれども自分の発明ではない、と強く主張しています。自分よ

＊1　アランはこの本で、CASの初期の話について書いています：『White Heat Cold Logic: British Computer Art 1960-1980』, Paul Brown, Charlie Gere, Nicholas Lambert, and Catherine Mason, editors（MIT Press, 2008）。

りも前に、ケネス・スティーブンソンのcircle packing*2があり、そこではスティーブンソン自身が、フロイド、キャノン、パリーによる先行作業をクレジットしています*3。

そこで私は、この話を再調査してみました。サトクリフの先人たちは、分割された五角形については述べていますが、誰もが彼の考えの半分にも達していませんでした。それに、もし彼らがそうだったとしても、あの夜、サトクリフ五角形が私の脳裏に焼きついたことは変わりません。

そこで、アランの謙虚さと、この作図を厳密には「サトクリフ五角形」と呼ぶべきではないという彼の意見を尊重しながらも、まずはサトクリフ五角形がどのようなものかを紹介するところから始めましょう（図8.10参照）。次に、サトクリフ五角形の組み立て方を説明し、最後にサトクリフ五角形の構造を使って作られた、いくつかの作品をお見せしましょう。

サトクリフは、まず五角形を描いて、その5つの辺のそれぞれの中点から垂線を引き、この垂線上の点を結べば別の五角形を作れると言っています。こうすれば、形状の残りの部分も同じく、さらなる五角形に分けられるので、結局それぞれの五角形がさらに6個のサブ五角形に分けられます。サブ五角形の内にはさらにサブサブ五角形ができるので、この分割を無限小に向かって繰り返すことができます。

あなたがまだこの数学にあまりスリルを感じていなくとも、もうしばらく我慢してください。次のセクションで私は、この単純なルールを使って何ができるかを紹介します。さあ、コーディングを始めましょう。私がとった、いくつかの方法を説明します。

8.4.1　組み立て

まず、通常の五角形を描くことから始めます。けれども私がやったのは、5つの点の間に5本の線を引くのではなく、中心の周りを360度回転しながら、ある特定の角度で点を打っていくという、もう少し一般的な描画方法です。例えば、もしあなたが72度ごとに点をプロットすれば、五角形を描くことができます。

＊2　Kenneth Stephenson,『Introduction to Circle Packing: The Theory of Discrete Analytic Functions』(Cambridge University Press, 2005)

＊3　J. W. Cannon, W. J. Floyd, and W. R. Parry,「Finite subdivision rules」, Conformal Geometry and Dynamics, vol. 5 (2001)

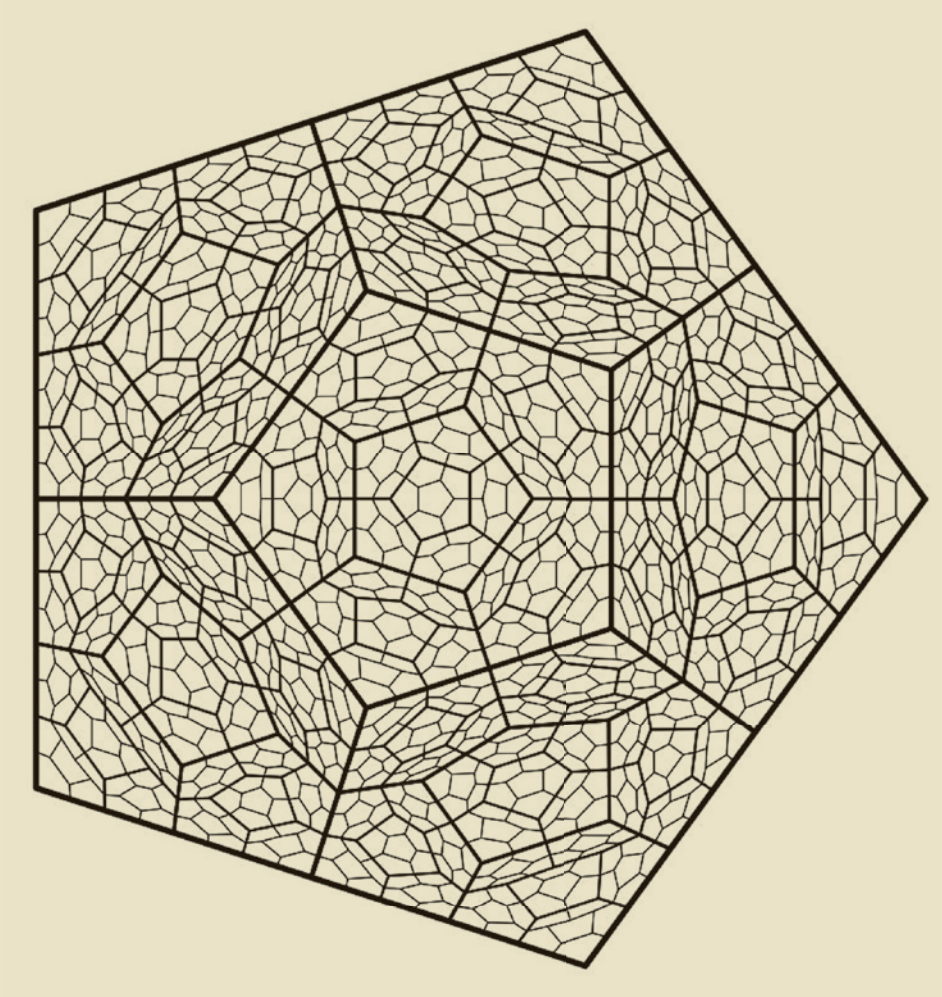


図8.10
サトクリフ五角形

リスト8.4　回転を使って五角形を描く

```
FractalRoot pentagon;
int _maxlevels = 5;

void setup() {
  size(1000, 1000);
  smooth();
  pentagon = new FractalRoot();
  pentagon.drawShape();
}

class PointObj {
  float x, y;
  PointObj(float ex, float why) {
    x = ex; y = why;
  }
}

class FractalRoot {
  PointObj[] pointArr = new PointObj[5];
  Branch rootBranch;

  FractalRoot() {
    float centX = width/2;
    float centY = height/2;
    int count = 0;
    for (int i = 0; i<360; i+=72) {
      float x = centX + (400 * cos(radians(i)));
      float y = centY + (400 * sin(radians(i)));
      pointArr[count] = new PointObj(x, y);
      count++;
    }
    rootBranch = new Branch(0, 0, pointArr);
  }

  void drawShape() {
    rootBranch.drawMe();
  }
}

class Branch {
  int level, num;
```

ルートの五角形を作る

そこにdrawShapeメソッドを呼ぶ

x、yの位置を格納するオブジェクト・クラス

Branchオブジェクトを定義する

（続く）

（リスト8.4の続き）

Branchオブジェクトを定義する

Branchはそれ自身を描く

```
    PointObj[] outerPoints = {};

    Branch(int lev, int n, PointObj[] points) {
      level = lev;
      num = n;
      outerPoints = points;
    }

    void drawMe() {
      strokeWeight(5 - level);
      // draw outer shape
      for (int i = 0; i < outerPoints.length; i++) {
        int nexti = i+1;
        if (nexti == outerPoints.length) { nexti = 0; }
        line(outerPoints[i].x, outerPoints[i].y,
               outerPoints[nexti].x, outerPoints[nexti].y);
      }
    }
  }
```

FractalRootオブジェクトには5つの点の配列があります。その配列には、中心点の周りを72度ごとに回転して作られた点が入っています。その配列を使って最初のBranchオブジェクトを作ります。Branchオブジェクトは、自己複製フラクタルオブジェクトです。それぞれの枝がouterPoints配列の中の点を結ぶことによって、自分自身を描きます。

すると図8.11のように、五角形が描かれます。

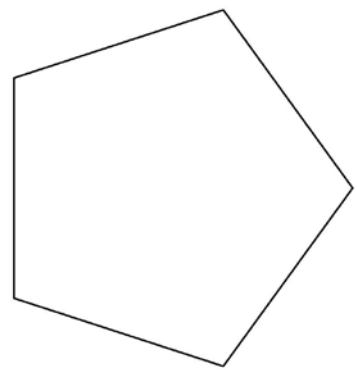


図8.11　普通の五角形がどのようなものか知らないといけないので。

次に、それぞれの辺の中点を図上に記入します。Branchオブジェクトに以下の関数を加えてください。

リスト8.5　頂点（辺）の中点を計算する関数

```
PointObj[] calcMidPoints() {
  PointObj[] mpArray = new PointObj[outerPoints.length];
  for (int i = 0; i < outerPoints.length; i++) {
    int nexti = i+1;
    if (nexti == outerPoints.length) { nexti = 0; }
    PointObj thisMP = calcMidPoint(outerPoints[i],⏎
                          outerPoints[nexti]);
  mpArray[i] = thisMP;
  }
return mpArray;
}

PointObj calcMidPoint(PointObj end1, PointObj end2) {
  float mx, my;
  if (end1.x > end2.x) {
    mx = end2.x + ((end1.x - end2.x)/2);
  } else {
    mx = end1.x + ((end2.x - end1.x)/2);
  }
  if (end1.y > end2.y) {
    my = end2.y + ((end1.y - end2.y)/2);
  } else {
    my = end1.y + ((end2.y - end1.y)/2);
  }
  return new PointObj(mx, my);
}
```

最初のcalcMidPoint関数は、外周の点の配列の中点を求めて、2番目のcalcMidPoint関数にその値を渡します。calcMidPoint関数は、あらゆる点の位置の組み合わせに対してその中点を求めます。そして最後にcalcMidPointsが、それぞれの結果を配列に集めて返します。

Branchコンストラクタの中から新しい関数を呼び出して、そしてmidPointsという名前の配列にその結果を入れます。

```
PointObj[] midPoints = {};

  Branch(int lev, int n, PointObj[] points) {
  ...
  midPoints = calcMidPoints();
}
```

次に、drawMe関数の中のそれらの点を図上にプロットします。

```
strokeWeight(0.5);
fill(255, 150);
for (int j = 0; j < midPoints.length; j++) {
  ellipse(midPoints[j].x, midPoints[j].y, 15, 15);
}
```

すると、図8.12のようになります。

次に必要になる点の集合は、中点から延ばす支柱の終端です。辺の頂点の角度を扱う必要はありません。その代わりに、辺と向かい合った頂点を使うことができます。

最初に、支柱を押し出す長さを全体の長さの比から決めるために、グローバルな_strutFactor変数を定義します。

```
float _strutFactor = 0.2;
```

Branchオブジェクトに、点を追加して描く関数を加えます。

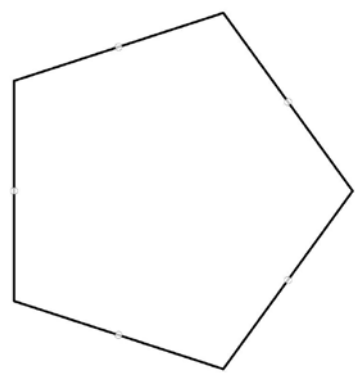


図8.12　辺の中間点を計算する。

リスト8.6　辺の中点を向かい合った頂点に向かって伸ばす関数

```
PointObj[] calcStrutPoints() {
  PointObj[] strutArray = new PointObj[midPoints.length];
  for (int i = 0; i < midPoints.length; i++) {
    int nexti = i+3;
    if (nexti >= midPoints.length) { nexti -= midPoints.length; }
    PointObj thisSP = calcProjPoint(midPoints[i],⏎
                          outerPoints[nexti]);
    strutArray[i] = thisSP;
  }
  return strutArray;
}

PointObj calcProjPoint(PointObj mp, PointObj op) {
  float px, py;
  float adj, opp;
  if (op.x > mp.x) {
    opp = op.x - mp.x;
  } else {
    opp = mp.x - op.x;
  }
  if (op.y > mp.y) {
    adj = op.y - mp.y;
  } else {
    adj = mp.y - op.y;
  }
  if (op.x > mp.x) {
    px = mp.x + (opp * _strutFactor);
  } else {
    px = mp.x - (opp * _strutFactor);
  }
  if (op.y > mp.y) {
    py = mp.y + (adj * _strutFactor);
  } else {
    py = mp.y - (adj * _strutFactor);
  }
  return new PointObj(px, py);
}
```

中点を向かい合った頂点に向けて伸ばす

三角法の計算

斜辺に沿って伸ばす

さらに、Branchコンストラクタにこのコードを加えます。

```
PointObj[] projPoints = {};

Branch(int lev, int n, PointObj[] points) {
  ...
  projPoints = calcStrutPoints();
}
```

drawMe関数で、これらの点を図の中に描いて、正しいかどうか見てみましょう。

```
strokeWeight(0.5);
fill(255, 150);
for (int j = 0; j < midPoints.length; j++) {
  ellipse(midPoints[j].x, midPoints[j].y, 15, 15);
  line(midPoints[j].x, midPoints[j].y,⏎
         projPoints[j].x, projPoints[j].y);
  ellipse(projPoints[j].x, projPoints[j].y, 15, 15);
}
```

図8.13のように見えるはずです。

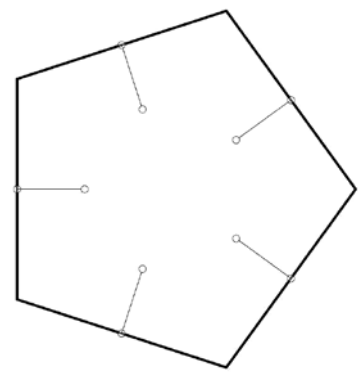


図8.13　支柱を反対側の点に向けて伸ばす。

これで、再帰構造のために必要なすべての点を描くことができるようになりました。5本の支柱の端点を内側の五角形の頂点とすることで、これを試してみることができます。Branchオブジェクトのコンストラクタに以下のコードを加えてください。

```
Branch[] myBranches = {};

Branch(int lev, int n, Point[] points) {
  ...
  if ((level+1) < _maxlevels) {
    Branch childBranch = new Branch(level+1, 0, projPoints);
    myBranches = (Branch[])append(myBranches, childBranch);
  }
}
```

さらに、子の五角形が画面に表示されるように、drawMe関数に同様のコードを加えてください。

```
void drawMe() {
...
  for (int k = 0; k < myBranches.length; k++) {
    myBranches[k].drawMe();
  }
}
```

結果は図8.14のようになります。

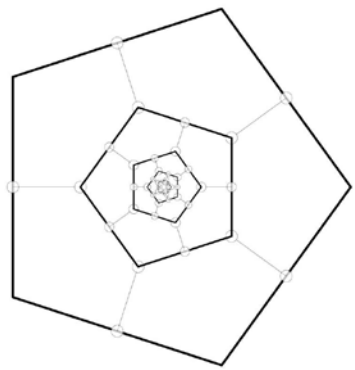


図8.14　再帰的な五角形

完全なサトクリフ五角形フラクタルを作成するためには、他の5つの五角形の内側にも、同様に五角形を描かなければなりません。そのためには、さらに5つの新しいBranchオブジェクトに点を渡せばいいだけです。コードは以下の通りです。

```
if ((level+1) < _maxlevels) {
  ...
  for (int k = 0; k < outerPoints.length; k++) {
    int nextk = k-1;
```

（続く）

（続き）

```
      if (nextk < 0) { nextk += outerPoints.length; }
      PointObj[] newPoints = { projPoints[k], midPoints[k],⏎
                   outerPoints[k], midPoints[nextk],⏎
                   projPoints[nextk] };
      childBranch = new Branch(level+1, k+1, newPoints);
      myBranches = (Branch[])append(myBranches, childBranch);
    }
  }
```

以上で、このセクションの初めに図8.10で見せたフラクタルができあがります。さあ、これを使って実験を始めましょう。

8.4.2　探求

まずは、支柱の比率を表す便利な変数があるので、これを変えることから始めましょう。新しい支柱の長さで、毎フレームごとにフラクタルを再描画する新しいdrawループを加えて、支柱の長さを乱数（ノイズ）で変えていきます。この実験を行うために、オブジェクトを修正する必要はありません。ただ、以下のようにコードの初期化部分を改良します。

リスト8.7　サトクリフ五角形の支柱の長さを変える

実験しやすくするために4に下げる

0から1ではなく、0から2の範囲

```
FractalRoot pentagon;
float _strutFactor = 0.2;
float _strutNoise;
int _maxlevels = 4;

void setup() {
  size(1000, 1000);
  smooth();
  _strutNoise = random(10);
}

void draw() {

  background(255);

  _strutNoise += 0.01;
  _strutFactor = noise(_strutNoise) * 2;
```

（続く）

（リスト8.7の続き）

```
    pentagon = new FractalRoot(frameCount);
    pentagon.drawShape();
  }
```

フラクタルを
毎フレームごとに再描画する

ここで1つだけ、FractalRootオブジェクトにframeCountを渡す、というちょっとした変更をしてみます。このようにFractalRootを変更して角度を少しずつ増加していくことで、形全体を『2001年宇宙の旅』風に、くるくる回らせることができます。

```
FractalRoot(float startAngle) {
...
    float x = centX + (400 * cos(radians(startAngle + i)));
    float y = centY + (400 * sin(radians(startAngle + i)));
...
}
```

同じく、支柱の比率を、0と1の間だけでなく、2まで行けるようにしました。そうすることで、既存の形の制限を超えることができます。うまくいきますか？　やってみてください。支柱の比率が負の値になったらどうでしょう？　何が起きるでしょうか？　以下のコードで試してみましょう。

```
_strutFactor = (noise(_strutNoise) * 3) - 1;
```

この場合、比率は－1と＋2の間で変化します。すると、図8.15のような形のアニメーションができあがります。

面白くなってきましたね！

五角形には素敵な対称性がありますが、最初の形が5つの辺を持たなければならないという理由はありません。私が「五角形だけでなく任意の多角形を描く方法」を採択した理由の1つは、多角形に簡単に辺を追加できるようにするためでした。まず、この辺の数を表すグローバル変数を作成してください。

```
int _numSides = 8;
```

そして、以下のようにFractalRootクラスを修正してください。

リスト8.8　5つの辺に限らないサトクリフ五角形

```
class FractalRoot {
  PointObj[] pointArr = {};
```

（続く）

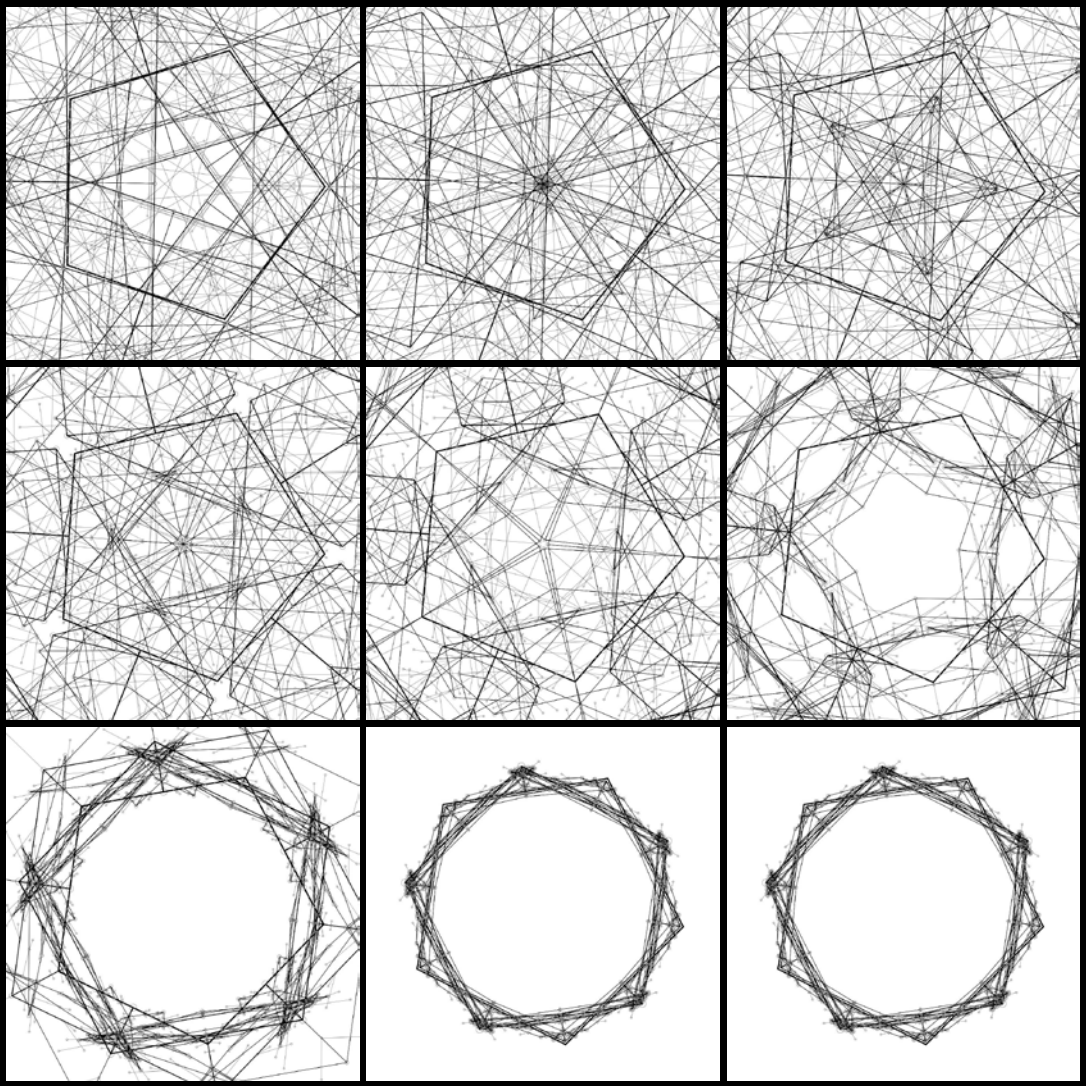
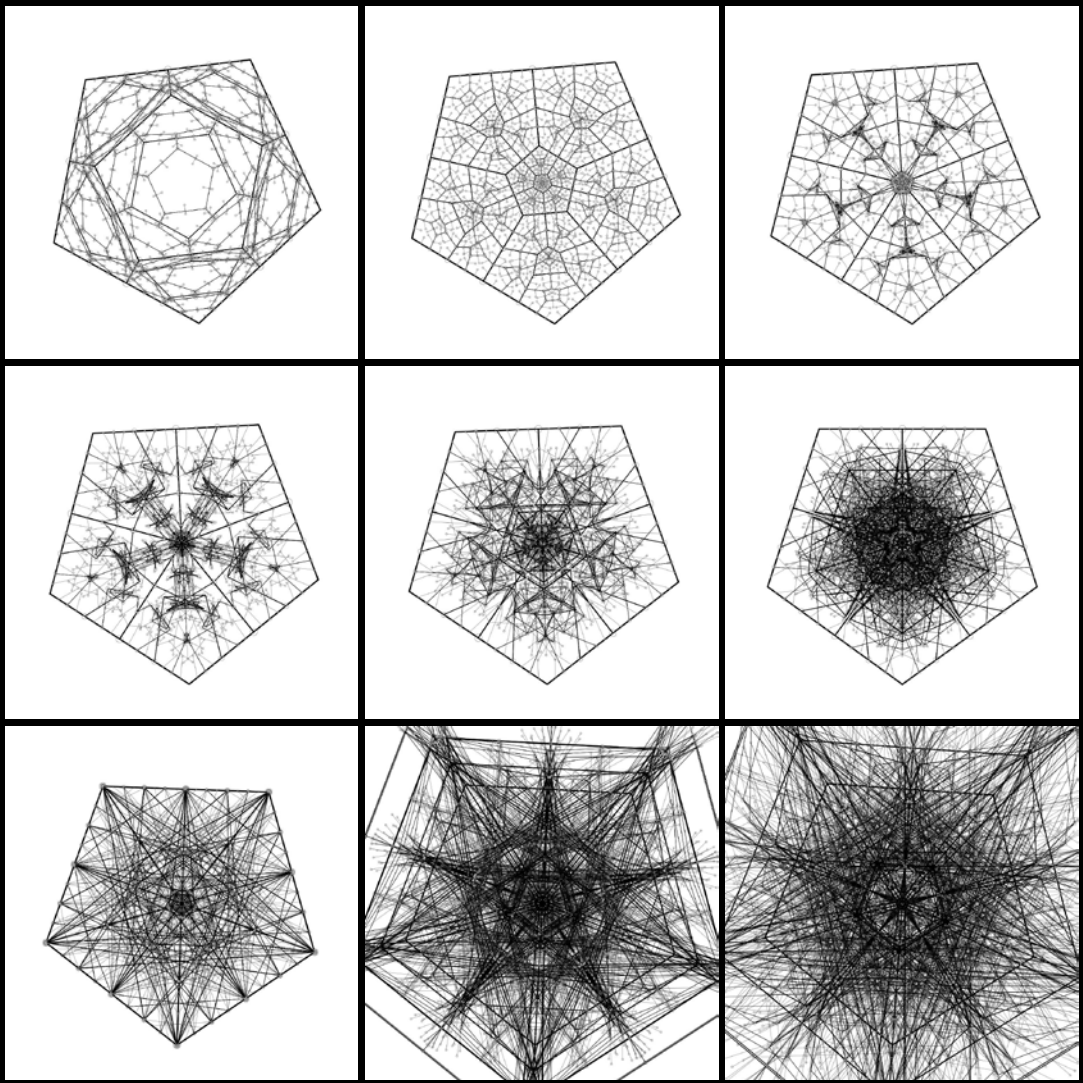


図8.15
支柱要素が－1から＋2まで変化させた例。



（リスト8.8の続き）

辺の数によるステップ数

```
    Branch rootBranch;

    FractalRoot(float startAngle) {
      float centX = width/2;
      float centY = height/2;
      float angleStep = 360.0f/_numSides;
      for (float i = 0; i<360; i+=angleStep) {
        float x = centX + (400 * cos(radians(startAngle + i)));
        float y = centY + (400 * sin(radians(startAngle + i)));
        pointArr = (PointObj[])append(pointArr, new PointObj(x, y));
      }
      rootBranch = new Branch(0, 0, pointArr);
    }

    void drawShape() {
      rootBranch.drawMe();
    }
  }
```

これで、限りなく多くの辺を扱うことのできる構造になりました。図8.16のように、8はどうですか？

あるいは32は？（図8.17を見てください）

さらに別のアプローチをとってみます。なぜ一周を360度に限定しなければいけないのでしょうか？　角度のステップを133度にセットして、8回ステップを繰り返すと（ほぼ3回転します）、図8.18のようになります。

支柱と同じように、角度と辺の数をフレームごとにノイズ要素によって変えてみることを試してみました。あなたも試してみようと思えば、コードの修正は難しくはありません。この実験を通して、私は終わることのない革命のようなものを発見しました（図8.19参照）。

レベルの数をあまり大きくしなければ、私が（サトクリフの仕様で）組み立てたフラクタル構造物は、極めて数学的に頑丈であることがわかりました。フェレット［ウサギ狩りやネズミの駆除に使われるシロイタチ］で狩りをするように、さまざまな結果を引きずり出すことで、予想外の結果が生まれれば生まれるほど面白くなっていきます。

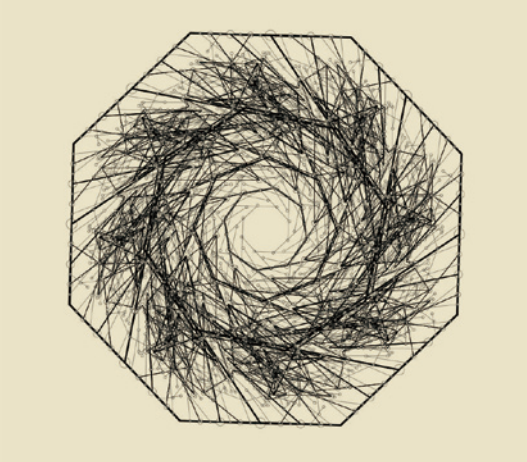


図8.16
8つの辺：サトクリフ八角形？

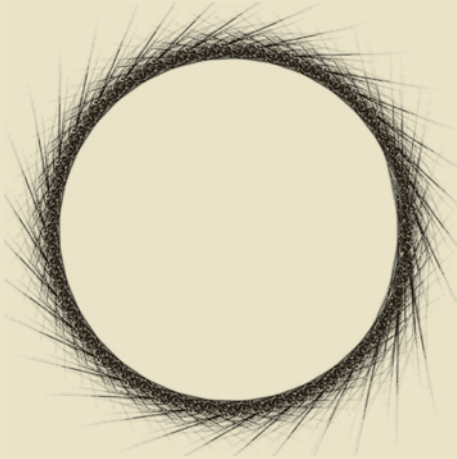


図8.17
32角形による構造。

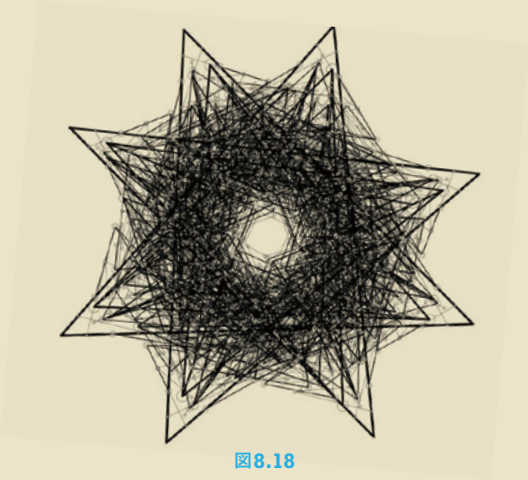


図8.18
133度ごとのステップで1024度まで増加させます。

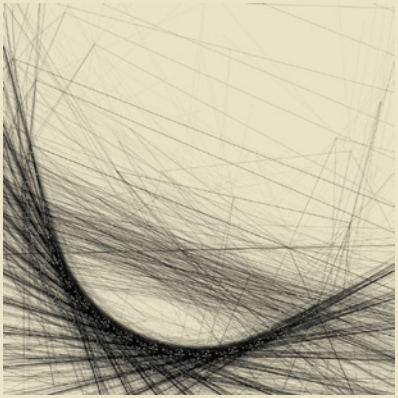


図8.19
このシステムには、不規則だったり
不完全な形を作り出せる柔軟性があります。

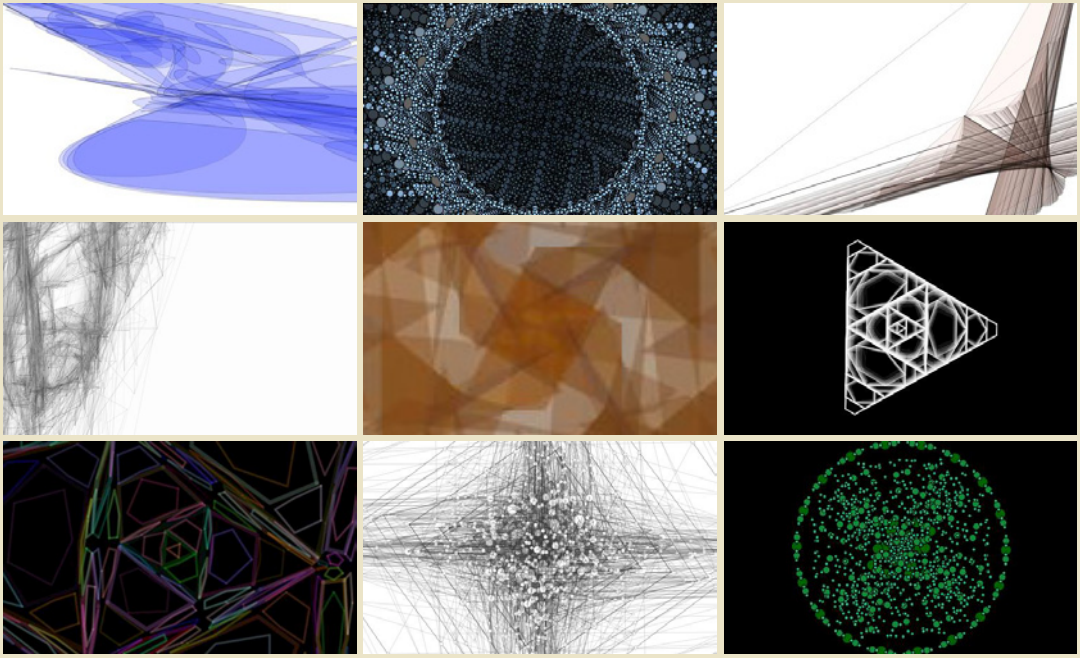


図8.20
サトクリフ五角形を応用した多くのabandonedart.orgの作品。

私が遊び始めたこのたった1つのフラクタルが、なぜそれほど私を喜ばせてくれたかがあなたにも伝わるといいなと思います。図8.20に、私が試みたもののいくつかを紹介します。http://abandonedart.orgに、このセクションのコードの完成版と共に、ここにあるすべてのコードを置いておきます。

ここで、私たちはフラクタル、そして本書を終りにします。もしあなたがこの章のコードを理解することができたのなら、Processingで征服できないものは、本当にもう何もありません。

8.5　まとめ

フラクタル――私たちが自然界から借用した1つの組織構造――は、私たちが以前の章で実験した創発的構造とよく似た原則を共有しています。単純なコードでも、結果が再帰的再帰によって拡大されると、そこから複雑な結果が引き起こされます。

本書の主なテーマは、予測不可能性、創発的複雑さ、そしてプログラミングに対する新たなアプローチでした。Part 1で、ジェネラティブ・アート、Processing言語とIDE、そしてプログラミングという堅苦しい分野に、芸術的な華やかさを持ち込むためのアイデアについて紹介しました。Part 2でとりあげたのは、コンピュータが思いがけずに洗練された美しさを生み出すことで私たちを驚かしてくれるよう仕向けるために、頭で解決できることを超えた結果を産み出してくれるランダム関数、ノイズ関数、数学的構造でした。

この最後の部分で私たちが育んだ予測不可能性は、創発――私たちのつたないプログラミングの技の能力を超えて、さまざまな複雑さのレベルを生み出してくれる、たくさんの単純な要素の相互作用――でした。ジェネラティブ・アートを作る際に用いる私たちの論理と手順の操作がいかに抽象的だとしても、それは自然界のより洗練された計算と原理とつながっています。クレイグ・レイノルズのBoids（Chapter 6）は、群れをなすムクドリをかなりうまく模倣しています。2状態のセル・オートマトンは、社会学者、生物学者、地質学者にとっても、同じようによく知られたものです（図8.21とChapter 7参照）。あなたがこの章で見たように、再帰的なフラクタルの構成は、自然の組織化から手に入れた考えを根っこにしています。

画面を通して見る時、それらがどう見えようとも、私たちはデジタルな世界に住んではいません。私たちの現実は、かたくなにアナログです。明らかに、コード化が可能なものではありません。それは複雑で、ねじくれていて、不明確です。自分たちの狂気や、混沌とした存在を、どうにかして反映させなければならないアートを、デジタルな手段で作りたいと思うなら、不

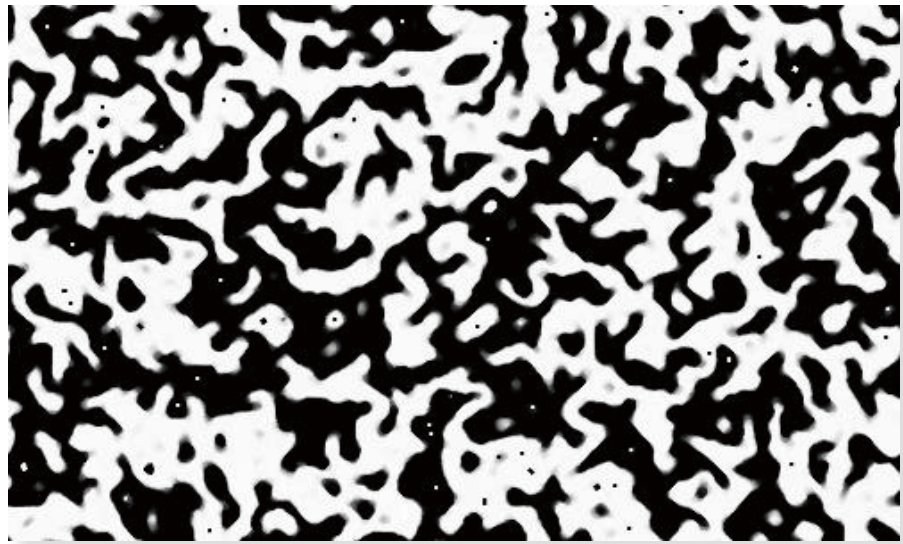


図8.21　Chapter 7のヴィシュニアク・ヴォート・ルールを使って生成されたパターン。

完全さと予測不可能さを方程式に取り入れる必要があります。機械的なものの中に有機的なもののための余地があって、プログラミングが単に効率や秩序のためのものではないことを、うまく伝えられたことを願っています。私たちの計算機――それは生物の貧弱な模倣しか作れないかもしれませんが――は、自然界と「計算による普遍性（computational universality）」でつながっています。

ジェネラティブ・アートをプログラムすることは、その要点をポケットサイズのかたまりにして述べるなら、カオスを受け入れることに関わっています。生活の中のすべてのものが、構造化されたやり方の恩恵を受けているわけではありません。少しくらい混沌とした方が、いっそう良くなるものがたくさんあります。もしあなたが、こうした自由を許容するやり方で仕事をしていて、アイデアからアイデアへと渡り歩いていくためのインスピレーションを大切にしていれば、きっとどこか面白いところに行けるでしょう。

読んでくれて、ありがとう。

Note：もしあなたがこの本に価値を見い出して、友だちとシェアしたいと思うなら（もしそうしてくれたら、ありがたいです）、いくつかの章がクリエイティブ・コモンズのライセンスで配布自由になっています。以下のサイトからダウンロードできます［英語版の原書に関する記述です］。http://zenbullets.com

訳者あとがき

コンピュータ関連の技術書は山のようにあっても、技術解説だけが際立って、その背後にある文化の多様性を語る書物は多くない。マット・ピアソン氏のこの本を読んでいるうちに、まさにその稀有な存在であることに気づき、彼にコンタクトをとって、翻訳したい旨を伝えた。おそらく彼も、いきなり日本からそのような申し出があるとは予想していなかっただろうが、なにかと気があって、いいよ！ということに。

カルチャーとして面白くても、すさまじい技術の陳腐化の速度に追い越されてしまい、読者の興味が失われないようにと思って、迅速にオリジナルの出版社にコンタクトをとると同時に、長年の友人である久保田晃弘さんに相談して国内で出版することになった。

もともとこの本はマットのブログから発展したもので、芸術、ポップス、社会思想史、形態発生、生物学などが、プログラミング言語の解説と共に散りばめられた不思議なものだ。たとえ読者が初心者で、プログラミングがすぐにわからなかったとしても、そこにあるカルチャーだけでも楽しんでもらえるものではないだろうか。

ぼくはマットの本を読んでいて、まずガツンときたのは、ウィリアム・ブレイクの絵画によるアイザック・ニュートン批判のエピソードで、マットはコンピュータ・プログラマーはブレイクによって描かれたニュートンのように自然に背を向けているのではないかと指摘していることだった。自然のカオスの把握は、コンピュータに期待できるものではなく、だからジェネラティブ・アーティストは、むしろそのカオスの方から自然に向きあっていこうということだ。

ぼくは、情報テクノロジーをベースにしたアートやデザインにたずさわると同時に、サステナブルでソーシャルなデザインにも関わっている。いわば、ウィリアム・ブレイクが憂慮した未来の真っ只中で、ある時はニュートンのように、またある時はロマン派のアーティストのような文化と心の葛藤の中にある。そこには18世紀から21世紀まで続いている自然と人との関係の連なりが見えてくる。

ここで結論づけられるものではないが、ぼくは現在、ブレイクたちロマン派の憂慮に応えられるのは、自然に背を向けずに正面から取り組むコンピュータ・プログラミングができるアーティストなのではないかと思ったりする。そして、そういうプログラマーも増えているだろう。

こういうことに思考回路を向けることのできるこの本を、読者に紹介できたのは素晴らしいことだ。

2012年11月

アーティスト／デザイナー／ミュージシャン

沖 啓介

アートワークリスト

Matt Pearson（zenbullets）
100 Abandoned Artworks（2008–10）[図i.14] …… 36
Broken Mirrors 1–4（2011）[図i.22] …… 44
Frosti（2010）[図i.11, 5.10] …… 33, 150
Mondrian Architecture（2009）[図i.20] …… 42
Opiamas Trangelo（2010）[図i.1, i.2, i.3] …… 25
Orbitals（2009）[図i.21] …… 43
Spiral Stairs（2009）[図4.6] …… 120
Tube Clock（2009）[図1.1] …… 53
Twill（2010）[図i.12] …… 34
Wave Clock（2009）[図4.10, 4.15] …… 125, 129

zenbullets / FutureDeluxe
Life in 2050（2010）[図i.10] …… 32
Perth Arts Festival Branding（2011）[図i.8] …… 30

Reza Ali
LORMALIZED（2010）[図i.23] …… 45

Tom Beddard（subblue）
Mandelbulb（2009）[図8.2] …… 204

Robert Hodgin（flight 404）
Addition/Subtraction, Variant（2010）[図i.4] …… 26
Jelly（Magnetosphere）（2007）[図i.6] …… 27
Magnetic Ink（2007）[図i.5] …… 27

Aaron Koblin
Flight Patterns（2005）[図i.18, 7.11] …… 40, 198

Manfred Mohr
Lady Quark（1972）[図1.3] …… 56

Casey Reas
Process 14 / Image 4（2008）[図i.13] …… 35

Jerome Saint-Clair（01010101）
Drawing With Particles（2009）[図i.24] …… 45

Jared Tarbell
Happy Place（2006）[図i.25] …… 46

Jeremy Thorp（blprnt）
Colour Economy（2008）[図7.10] …… 197

Frederik Vanhoutte（w:blut）
Strange Symmetry 2（2008）[図i.19] …… 41

Marius Watz
Grid Distortion 02D 0018（2010）[図i.15] …… 38
Illuminations B（2007）[図i.17] …… 39
KBGD01E 0012（2010）[図i.16] …… 38

索 引

■ 記号

+演算子 …… 80
-演算子 …… 80
*演算子 …… 80
%演算子 …… 80
/演算子 …… 80
++演算子 …… 80
--演算子 …… 80
+=演算子 …… 80
-=演算子 …… 80
*=演算子 …… 80
/=演算子 …… 80
=演算子 …… 80

■ 数字

01010101, Drawing With Particles …… 45
100 Abandoned Artworks（マット・ピアソン）…… 36
16進法 …… 70
3Dプリンタ，さわれるジェネラティブ・アート …… 11

■ A

Add1ition/Subtraction, Variant（ロバート・ホジン）…… 26
After Effects …… 59
Android …… 13, 14, 94
AndroidのためのProcessing …… 94
append関数 …… 167
Apple …… 58, 95

■ B

beginShape関数 …… 123
Blonde Boids（Abandoned Artwork #4）…… 157
Boidsアルゴリズム，リアルな群れのシミュレーション …… 156
boolean …… 72
break …… 87
Broken Mirrors 1-4（マット・ピアソン）…… 44
byte …… 72

■ C

char …… 72
classキーワード …… 162
Clojure …… 62
Colour Economy（ジェレミー・ソープ）…… 195, 197
Context Free …… 61
cos関数 …… 108
curveVertex(x,y) …… 123

■ D

dist関数 …… 168
draw()関数ブロック …… 76
Drawing With Particles（01010101）…… 45

■ E

ellipse関数 …… 66
『Emergence』（ジョン・ホランド）…… 155
『Emergence』（スティーブン・ジョンソン）…… 155
endShape関数 …… 123

■ F

fill関数 …… 74, 84, 123
Flight Patterns（アーロン・コブリン）…… 40, 198
float …… 72
forループ …… 87, 102, 119, 205
frameRate関数 …… 75
Frosti（マット・ピアソン）…… 33, 150

■ G

Grid Distortion 02D 0018（マリウス・ワッツ）…… 38

■ H

Happy Place（ジャレッド・ターベル）…… 47
HasCanvas …… 61, 96
Haunted Fishtank …… 147

■ I
if ... else …… 81
if …… 81
Illuminations B（マリウス・ワッツ）…… 39
Illustrator …… 59
Information Is Beautiful …… 201
int …… 72
Introduction to Circle Packing（Kenneth Stephenson）…… 219
iPad，開発 …… 94
iPhone，開発 …… 94
iProcessing …… 95

■ J
Java …… 62, 90, 92, 94, 141
Java Micro Edition（Java ME）…… 95
JAVA2D …… 141
Jelly（Magnetosphere）（ロバート・ホジン）…… 27

■ K
KBGD01E 0012（マリウス・ワッツ）…… 38
keyPressed() 関数 …… 92

■ L
Lady Quark（マンフレッド・モール）…… 56
Life in 2050（マット・ピアソン）…… 32
line 関数 …… 70
loading.gif …… 44
LORMALIZED（レザ・アリ）…… 45
Lunar Lander …… 199
Lunar Lander Trails（セブ・リー・デリスル）…… 200

■ M
Magnetic Ink（ロバート・ホジン）…… 27
Mandelbulb（トム・ベダード）…… 156
Mobile Processing …… 95
MODE …… 140
Mondrian Architecture（マット・ピアソン）…… 42
Mozilla, Processing for the Web …… 62

■ N
Newton（ウィリアム・ブレイク）…… 31
NodeBox …… 61
noFill 関数 …… 74
noise 関数 …… 105, 108
――アルファ値の変化 …… 133

■ O
openFrameworks …… 61
OpenGL ライブラリ …… 141, 142, 143
OPENGL レンダラー …… 141
Opiamas Trangelo（マット・ピアソン）…… 25
Orbitals（マット・ピアソン）…… 43

■ P
PDF，レンダラー …… 141
Photoshop …… 58, 59, 70
popMatrix 関数 …… 136
Process 14 / Image 4（ケイシー・リース）…… 35
Processing 2D（P2D）レンダラー …… 141
Processing 3D（P3D）レンダラー …… 141
Processing for the Web …… 62
Processing
――ジェネラティブ・アートを作る手段 …… 52
――チュートリアル …… 64
――モバイル …… 95
Processing.js …… 62
――iPhone …… 96
pushMatrix 関数 …… 136

■ R
random.org プロジェクト …… 100
random 関数 …… 100
――自分だけの …… 111
RGB …… 68
Ruby …… 62

■ S
Save As... …… 90
saveFrame 関数 …… 91
Scala …… 62
setup() 関数ブロック …… 76
sin 関数 …… 111
size 関数 …… 71, 140
smooth 関数 …… 68

sphereDetail関数 …… 142
sphere関数 …… 141, 148
Spiral Stairs（マット・ピアソン） …… 120
Strange Symmetry 2（フレデリック・バンフート） …… 41
String …… 72
strokeWeight関数 …… 70
stroke関数 …… 70
Structure Synth …… 61

■ T
translate関数 …… 136
——3Dでの描画 …… 141
Tube Clock（マット・ピアソン） …… 53
Twill（マット・ピアソン） …… 34
Twitter，創発 …… 158

■ V
Vimeo …… 94
visual complexity …… 201
vvvv …… 61

■ W
Wave Clock …… 125, 128, 129
——ケーススタディ …… 124
whileループ …… 83

■ Y
YouTube …… 94

■ あ
アーロン・コブリン，Flight Patterns …… 40, 198
アイザック・ニュートン …… 31
アニメーション
——ノイズ …… 137
——基本 …… 75
アポロ17 …… 196
アラン・サトクリフ …… 218
——White Heat Cold Logic …… 218
——サトクリフ五角形 …… 218
アラン・チューリング …… 16, 158, 206
——形態発生 …… 158
——停止性問題 …… 206
アラン・ワッツ …… 24
アリの群れ
——創発の例 …… 156
——中央集権性の欠如 …… 156
アルゴリスト …… 29
アルゴリズム，ジェネラティブ・アートのツール …… 57
アルファ値 …… 72
アンダーバー，グローバル変数 …… 78
イラ・グリーンバーグ …… 64
インクリメント演算子 …… 80
インスタンス …… 161
インターネット，創発 …… 158
ヴィシュニアク・ヴォート …… 185
ウィリアム・ギブソン …… 105
ウィリアム・ブレイク，Newton …… 31
ヴェラ・モルナール …… 57
エージェント …… 194, 196
——ellipse関数で描く …… 116
——オブジェクトとの違い …… 175
——ソフトウェア …… 194
——実世界の属性 …… 195
——人間 …… 196
——ジェネラティブ・アート …… 199
——データを集める …… 200
——使う …… 200
エクスポート …… 89
円
——らせんに変える …… 117
——円周上の点を計算 …… 115
——描画
——描画　ellipse関数で …… 116
——描画　OOPで …… 161
——描画　三角法で …… 116
演算子
——グループ化 …… 79
——リスト …… 80
オブジェクト
——エージェントとの違い …… 175
——プログラミングでの …… 162
オブジェクト指向プログラミング …… 160
——インスタンス …… 161
——インタラクション …… 169

——クラス …… 161
——円を描画 …… 161
——配列 …… 167
音楽，ジェネラティブな …… 55

■ か
カールハインツ・シュトックハウゼン …… 55
カオスと秩序 …… 31
加算演算子 …… 32
画面サイズ …… 93
関数 …… 66
——自分の関数 …… 78
関数ブロック …… 75
擬似ランダム …… 100
クラス …… 161
クレイグ・レイノルズ，Boidsアルゴリズム …… 156, 235
グローバル変数 …… 77
ケーススタディ …… 124, 218
ゲーム・オブ・ライフ …… 176, 182
『ゲーデル、エッシャー、バッハ』（ダグラス・ホフスタッター）
…… 184
ケイシー・リース …… 35, 62
——Process 14 / Image 4 …… 35
携帯電話，開発 …… 94
形態発生 …… 158
啓蒙主義 …… 31
ケン・パーリン …… 105
コサイン，定義 …… 114
コサインカーブ …… 109
コメント …… 68
コラボレーション …… 54
コンピュータ・アーツ・ソサエティ（CAS）…… 218

■ さ
再帰，無限の …… 205
サイン，定義 …… 114
サインカーブ …… 109
——描画 …… 108
作品をセーブする …… 88
サトクリフ五角形 …… 218
ジェネラティブ・アート
——カオス芸術 …… 47
——ゲーム・オブ・ライフ …… 184
——コラボレーション …… 54
——さわれる …… 58
——人間 …… 199
——人工知能 …… 184
——秩序とカオス …… 31
——方法論 …… 64
——目的 …… 38
——歴史 …… 54
ジェネラティブ・ミュージック …… 55
ジェラード・ヴィシュニアク …… 185
ジェレミー・ソープ …… 195, 197, 200
ジェローム・サンクレア …… 45, 94
自己相似性 …… 207
実行の順序 …… 73
自分の関数を書く …… 78
シミュレーションとビジュアライゼーション …… 194
斜辺 …… 114
ジャレッド・ターベル，Happy Place …… 46
出力ウィンドウ …… 65, 66
ジョージ・ニース …… 57
ジョージ・ヘンリー・ルイス，創発 …… 155
条件文 …… 81
乗算演算子 …… 80
衝突判定 …… 167
除算演算子 …… 80
ジョン・ケージ …… 55
ジョン・コンウェイ …… 176, 181
ジョン・ホランド，『Emergence』…… 155
ジョン・レッシグ …… 62
自律性
——ジェネラティブな方法論に必要 …… 54
——定義 …… 175
スクリプト …… 89
スケッチ …… 89
スティーブン・ジョンソン，『Emergence』…… 155
静止画 …… 91, 92
——ビデオ作成で使用 …… 92
絶対座標 …… 71
セブ・リー・デリスル，Lunar Landing Trails …… 200
セミコロン …… 67
セル・オートマトン …… 176

——ヴィシュニアク・ヴォート …… 185
——ゲーム・オブ・ライフ …… 176, 182
——ブライアンの脳 …… 186
——波（平均化）…… 191
相対座標 …… 71
創発 …… 155
——アリの群れ …… 156
——インタラクションのパターン …… 173
——と意識 …… 159
——と複雑系理論 …… 156
——社会学的な意味 …… 158
——政治的な意味 …… 158
——組織的な原則 …… 159
——定義 …… 155
創発的ネットワーク …… 158
創発的複雑性 …… 51, 160, 176, 201, 205, 235

■ た

ダグラス・ホフスタッター，『ゲーデル、エッシャー、バッハ』 …… 184
ダブリンのトリニティー・カレッジ …… 100
ダン・シフマン
——Boids …… 157
——Processing チュートリアル …… 64
秩序とカオス …… 31
デイヴィット・トゥープ，Haunted Weather …… 55
データ・ビジュアライゼーション …… 201
データ型
——boolean …… 72
——byte …… 72
——char …… 72
——float …… 72
——int …… 72
——String …… 72
——コンポジット …… 72
デクリメント（減少分）演算子 …… 80
デジタルなツールセット …… 57
——学習曲線 …… 59
——急速な陳腐化 …… 58
トム・ベダード …… 204
——Mandelbulb …… 156
トロン …… 105

■ な

ニューヨーク・タイムズ，API …… 200
『ニューロマンサー』…… 105
人間，ジェネラティブなシステム …… 199
人工知能，ジェネラティブ・アート …… 184
塗りつぶし …… 72
ノイズ
——2次元 …… 132
——3次元 …… 143
——アニメーション …… 137
——作る …… 108
——視覚化 …… 133
ノイズグリッド …… 132, 137
ノイズ立方体 …… 145

■ は

バージョン管理 …… 90
パース国際芸術祭のブランディング（マット・ピアソン）…… 30
パーリンノイズ …… 105, 121, 131, 181
配列
——プログラミング …… 167
——多次元 …… 181
パラメータ …… 66
ハロー・ワールド …… 65
比較演算子 …… 81
『ビジュアライジング・データ』（ベン・フライ）…… 201
ビデオ …… 93
描画
——3次元空間 …… 141
——回転する …… 113
——基礎 …… 67
フィリップ・ガランター，ジェネラティブ・アートの定義 …… 51
複雑系理論と創発 …… 156
ブライアン・イーノ …… 55
——Discreet Music …… 55
ブライアンの脳 …… 186
フラクタル
——サトクリフ五角形 …… 218
——宇宙のゼンマイ時計 …… 212
——指数的成長 …… 213
——自己相似性 …… 203
——無限の再帰 …… 205

——木 211
フリーダー・ナーケ 57
フリッカー 94
フレームループ 75, 83, 137
フレームレート 76, 93
フレデリック・バンフート 41
プログラミング，詩としての 41
変化の繰り返し 102
——ランダム性 103
ベンジャミン・フライ 62, 201
——『ビジュアライジング・データ』 201
変数
——グローバル変数 77
変数　ローカル変数 77
変数のスコープ 77
ポール・ブラウン 57
方法論，ジェネラティブ 64

■ ま

マット・ピアソン
——100 Abandoned Artworks 36
——Broken Mirrors 1–4 44
——Frosti 33, 150
——Life in 2050 32
——Mondrian Architecture 42
——Opiamas Trangelo 25
——Orbitals 42
——Spiral Stairs 72
——Tube Clock 53
——Twill 34
——Wave Clock 125, 128, 129
——パース国際芸術祭のブランディング 43
マリウス・ワッツ
——Grid Distortion 02D 0018 38
——Illuminations B 39
——KBGD01E 0012 38
マンデルブロ 203, 204
マンフレッド・モール
——Artificiata I 57
——Lady Quark 56
ムーアの法則 59
ムクドリ，群れ 157
無限の再帰 205
無限ループ 83, 206, 209
群れのシミュレーション 156
モーツァルト 55

■ や

有機体，機械を使って作る 25
予測不可能性 47

■ ら

ラジアンと度 115
らせん 117
乱数，生成 100
リチャード・ドーキンス，盲目の時計職人 52
ルーディ・ラッカー 153
ループ
——for 87, 102, 119, 205
——while 83
——終了 39
レザ・アリ 45
レンダリングエンジン
——JAVA2D 141
——OPENGL 141
——P2D 141
——P3D 141
——PDF 141
——リスト 141
ローカル変数 29
ロバート・オルーク，HasCanvas 96
ロバート・ホジン
——Addition/Subtraction, Variant 26
——Jelly (Magnetosphere) 27
——Magnetic Ink 27
ロマン主義 31
論理演算子 33

[監訳者]
久保田 晃弘（くぼた あきひろ）

1960年生まれ。1988年東京大学大学院工学系研究科博士課程修了・工学博士。多摩美術大学情報デザイン学科メディア芸術コース教授。衛星芸術（artsat.jp）、バイオアート（bioart.jp）、デジタル・ファブリケーション（fablabjapan.org）、自作楽器によるサウンド・パフォーマンス（hemokosa.com）など、さまざまな領域を横断・結合するハイブリッドな創作の世界を開拓中。主な著書に『消えゆくコンピュータ』（岩波書店，1999）、『ポスト・テクノ（ロジー）ミュージック』（大村書店，監修，2001）、『デザイン言語』（慶應義塾出版会，共著，2002）、『創造性の宇宙——創世記から情報空間へ』（工作舎、共著、2008）、『Beyond Interaction —メディアアートのためのopenFrameworksプログラミング入門』（BNN新社、共著、2010）『FORM + CODE—デザイン／アート／建築における、かたちとコード』（BNN新社、監訳、2011）、『ビジュアル・コンプレキシティ—情報パターンのマッピング』（BNN新社、監修、2012）などがある。

[訳者]
沖 啓介（おき けいすけ）

アーティストとして多摩美術大学の学生のころから現代美術の作品を国内外で発表しはじめ、またビデオアート、パフォーマンス、電子音楽などのメディアアートに深く関わる。学生時代に「鉄腕アトム」の効果音の音響制作で知られる綜合社で当時は貴重で珍しかったシンセサイザーに触れたり、身体とテクノロジーをテーマにして名高いアーティストのステラークのロボティクス作品「第三の手」のプロジェクトを手伝ったり、また初期のパソコンソフトのインターフェース・デザインなどもてがけたりしているうちに、次第にコンピュータが自分の表現の重要なツールとしての位置を占めるようになる。カーネギーメロン大学STUDIO for Creative Inquiry研究員を経て、東京造形大学特任教授、名古屋造形大学客員教授のほか、早稲田大学理工学術院基幹理工学部表現工学科、武蔵野美術大学デザイン情報学科などで教えている。

本書は、2012年12月刊行書籍『ジェネラティブ・アート―Processingによる実践ガイド』を、Processingのバージョン2.2.1に対応させた［普及版］です。構成内容に変更はありません。

［普及版］ジェネラティブ・アート
Processingによる実践ガイド

2014年11月23日　初版第1刷発行

著者：マット・ピアソン
監訳者：久保田 晃弘
訳者：沖 啓介

発行人：籔内 康一
発行所：株式会社ビー・エヌ・エヌ新社
〒150-0022
東京都渋谷区恵比寿南一丁目20番6号
E-mail：info@bnn.co.jp　Fax：03-5725-1511
http://www.bnn.co.jp/

印刷・製本：シナノ印刷株式会社

日本語版コーディネート：加藤 久美
日本語版デザイン：松川 祐子
日本語版編集協力：石郷 祐介、新見 大暉、松田 千紘、森岡 麻紗子、吉村 マサテル
日本語版編集：村田 純一

●本書の内容に関するお問い合わせは、E-mailまたはFaxにてご連絡ください。
●本書の一部または全部について、個人で使用するほかは、（株）ビー・エヌ・エヌ新社および著作権者の承諾を得ずに無断で複写・複製することは禁じられております。
●乱丁本・落丁本はお取り替えいたします。

＊定価はカバーに記載してあります。

ISBN: 978-4-86100-963-1
Printed in Japan