



# O CÓDIGO TRANSCENDENTE

Uma introdução prática à programação e arte gerativa

M. P. Berruezo

M. P. BERRUEZO

# O CÓDIGO TRANSCENDENTE

Uma introdução prática à programação e arte gerativa

1º Edição

Belo Horizonte  
Mateus Paresqui Berruezo  
2019

O Código Transcendente, V 1.0

Este arquivo foi gerado utilizando o  $\text{\LaTeX}$  em: 28 de dezembro de 2019, às 11:38h.

Todos os direitos reservados por Mateus Paresqui Beruezo

ISBN-13: 978-65-902096-0-3

O conteúdo deste livro que não esteja diretamente referenciado está licenciado sob a [\*Creative Commons Attribution-NonCommercial 3.0 Unported License\*](#), publicada pela organização sem fins lucrativos, Creative Commons.

Todos os códigos deste livro foram gerados pelo autor e estão licenciados sob a [\*MIT License\*](#) publicada pelo Instituto de Tecnologia do Massachusetts.

Às almas impetuosas,  
sedentas de conhecimento,  
transbordando de paixão,  
incapazes de resistir ao vício da criação;  
  
este livro é para vocês.



# Sumário

<b>Carta-Prefácio</b>	<b>ix</b>
<b>Agradecimentos</b>	<b>xi</b>
<b>Prefácio</b>	<b>xiii</b>
<b>Introdução</b>	<b>xvii</b>
Identidade fragmentada .....	xvii
O gerativo .....	xviii
Nós, artistas .....	xx
<b>I Origem</b>	<b>3</b>
<b>1 Primeiros Passos</b>	<b>5</b>
1.1 Uma breve história da arte gerativa .....	5
1.2 Processing .....	15
1.3 Ambiente Processing .....	19
1.4 Conversando com a máquina .....	24
1.5 Elementos de imagens digitais .....	31
1.6 Estrutura de um programa em Processing .....	35
1.7 Sumário .....	36
<b>2 A Arte da Programação</b>	<b>37</b>
2.1 Variáveis .....	38
2.1.1 Variáveis do sistema .....	41
2.1.2 Vetores de variáveis .....	42
2.1.3 Cores .....	45
2.2 Operadores .....	49

2.2.1	Aritméticos . . . . .	49
2.2.2	Atribuição . . . . .	50
2.2.3	Relação . . . . .	51
2.2.4	Lógicos . . . . .	51
2.2.5	Operadores e cores . . . . .	52
2.2.6	Funções matemáticas . . . . .	53
2.3	Funções . . . . .	55
2.3.1	Interrupções . . . . .	58
2.4	Condicionais . . . . .	60
2.4.1	Múltiplos condicionais . . . . .	62
2.5	Repetições . . . . .	66
2.5.1	Repetições - <i>for</i> . . . . .	69
2.5.2	Múltiplas repetições . . . . .	70
2.6	Tópicos avançados - classes e objetos . . . . .	81
2.6.1	Uma classe na prática . . . . .	84
2.7	Sumário . . . . .	89
<b>II</b>	<b>Evolução</b>	<b>93</b>
<b>3</b>	<b>Caos</b>	<b>95</b>
3.1	Aleatoriedade . . . . .	95
3.2	Distúrbios . . . . .	100
3.3	Ruídos multidimensionais . . . . .	105
3.4	Sob o controle do acaso . . . . .	109
3.5	Sumário . . . . .	113
<b>4</b>	<b>Ordem</b>	<b>115</b>
4.1	Trigonometria . . . . .	116
4.2	Beleza matemática . . . . .	120
4.2.1	Pequenos acidentes favoráveis . . . . .	123
4.3	Simetria radial . . . . .	127
4.4	Espiral . . . . .	134
4.5	Sumário . . . . .	139
<b>5</b>	<b>Ilusões</b>	<b>141</b>
5.1	Linhas etéreas . . . . .	141
5.2	Faça-se a cor . . . . .	146
5.3	Através do espelho . . . . .	152
5.4	Um exemplo para reinar sobre todos . . . . .	157

5.5	Sumário . . . . .	167
<b>III</b>	<b>Transcendência</b>	<b>169</b>
<b>6</b>	<b>Fractais</b>	<b>171</b>
6.1	Autossimilaridade e recursividade . . . . .	172
6.2	Fractal árvore . . . . .	176
6.2.1	Subespécies . . . . .	185
6.3	Sumário . . . . .	192
<b>7</b>	<b>Emergência</b>	<b>193</b>
7.1	Agente autônomo . . . . .	195
7.2	Processo . . . . .	203
7.3	Extrapolação . . . . .	205
7.4	Sumário . . . . .	213
<b>8</b>	<b>O Código Transcendente</b>	<b>217</b>
8.1	Tratamento de imagens . . . . .	218
8.2	Áudio/Visual . . . . .	221
8.3	Visualização de dados . . . . .	224
8.4	Design computacional . . . . .	229
8.5	Sumário . . . . .	237
<b>O Processo Final</b>		<b>239</b>
<b>Índice</b>		<b>241</b>



# Carta-Prefácio

Começo esta carta prefácio com um depoimento pessoal. Lendo o manuscrito de “O Código Transcendente” foi impossível eu não lembrar de minha infância. Meu primeiro contato com computadores aconteceu no começo da década de 1980, quando eu tinha apenas 8 anos de idade e meu pai apareceu com uma caixa misteriosa em casa, cujo conteúdo iria mudar minha percepção e meu modo de interagir com o mundo pelo resto de minha vida. A caixa continha um TK-90X, um clone brasileiro do ZX-SPECTRUM, um clássico microcomputador britânico. Hoje em dia é cada vez mais frequente ver crianças, até menores do que 8 anos, usando computadores, mas naquela época acredito que eu era uma das poucas crianças em Porto Alegre que tinham um microcomputador. E ter um computador significava saber programar. Se você não soubesse programar, a utilidade destes aparelhos era bastante limitada. Assim, desde o dia em que esta caixa misteriosa apareceu em casa, programar é uma atividade a qual eu dedico algum tempo praticamente todos os dias, sempre tentando tornar o mistério contido dentro do aparelho daquela caixa cada vez mais inteligível.

É interessante notar, depois de todos esses anos, que eu sempre percebi o ato de programar como sendo uma atividade extremamente lúdica e criativa, e não somente como um fazer técnico, cuja finalidade seria a otimização máxima dos algoritmos e um uso eficiente dos recursos computacionais. Programar, para mim, sempre foi um fazer experimental, tendo os computadores sendo playgrounds onde eu exercito minha criatividade. Hoje eu leciono a disciplina “Creative Coding”, no mestrado em “Digital Arts” da De Montfort University, em Leicester, Reino Unido, explorando com os meus alunos as diversas possibilidades do uso da linguagem Processing nas artes visuais, sonoras e interativas. Tendo este *background*, é com alegria que vejo este volume nascer, preenchendo uma importante lacuna na literatura brasileira sobre o tema. Mais do que apenas introduzir a linguagem Processing ao público interessado em aprender a programar ou em aperfeiçoar seu conhecimento, Berrueto contextualiza o terreno que possibilitou o surgimento, pelas mãos e mentes de Casey Reas e Benjamin Fry, desta linguagem, trazendo ao leitor um importante panorama sobre a arte generativa, que resgata e contextualiza seus anos primordiais com Georg Nees, Frieder Nake, Michael Noll, Manfred Mohr, Vera Molnár e Ernest Edmonds, entre outros artistas.

Códigos computacionais estão presentes, de maneiras óbvias e também imperceptíveis, em praticamente todos os aspectos da vida contemporânea, sejam eles econômicos, culturais, criativos ou políticos, e programar é uma forma de participar ativamente do processo de construção e compreensão de nosso futuro. Iniciativas como esta obra são essenciais para a alfabetização digital e inclusão de um número cada vez maior de pessoas nas vias computacionais pelas quais trafegam o mundo contemporâneo, e o leitor encontrará em “O Código Transcendente” um ótimo guia para sua viagem!

Fabrizio Poltronieri

Berlim, 2019.



# Agradecimentos

Agradeço a todos que contribuíram de forma direta e indireta com o livro. Dedicar parte de seu tempo para um projeto de outra pessoa é um ato altruísta e nobre. Vocês não sabem o quanto isso realmente significou para mim. Muito obrigado.

Aqueles(as) que contribuíram com a revisão, texto, sugestões e explicações:

Ana Carolina Mendes da Silva, Carlos de Souza Braga, [João Antonio de F. P. e Ferreira](#), Ludimila Bragança, Luís Rocha Dias, [Monica Rizzolli](#) e Renata Ferreira Campos.

Aqueles(as) que me permitiram usar suas imagens neste livro:

[Benjamin Fry](#), [Casey Reas](#), [Fernando de la Torre](#), [Janet Marshall](#), [Jon Mace](#), [LIA](#), [Mark Harris](#), [Martin Krzywinski](#), [Metropolitan Museum of Art](#), [New York](#), [National Gallery of Art](#), [Washington D.C.](#), [NASA](#), [Nervous System](#), [Olga Sytina & Alexey Kljatov](#), [Rafael Alves Girundi](#), [Raven Kwok](#), [Scott Atwood](#), [Sérgio Albiac](#) e [Viorica Hrincu](#).

À [Processing Foundation](#), responsável por manter e distribuir gratuitamente um software tão completo e versátil quanto o Proceesing, além de compilar e fornecer uma grande quantidade de recursos educacionais.

Em especial, agradeço também a duas pessoas com contribuições excepcionais:

À artista e pesquisadora Dr.<sup>a</sup> Analivia Cordeiro pela revisão e complemento do paragrafo do capítulo um, relativo ao seu seu pai, o artista Waldemar Cordeiro, pioneiro da arte computacional no Brasil e no mundo. Adicionalmente, lembrei com carinho de nossa brevíssima conversa e suas brilhantes e sabias palavras.

Ao artista e pesquisador Dr. [Fabrizio Augusto Poltronieri](#) pela recepção, conversa e carta-prefácio. Particularmente o considero um grande exemplo de artista computacional do Brasil e fiquei honrado por ele ter aceitado o convite para escrever a carta-prefácio.



# Prefácio

## Mensagem do autor

Escrever um livro de programação para um público eclético e disseminado nas áreas das artes, humanas, biológicas ou mesmo da vida em si é desafiador, especialmente perante ao contraste com o estranho mundo das exatas. Pessoas da área de exatas, autor incluído, costumam ser criaturas curiosas, movidas por um instinto que nos impelem a desmontar tudo que vemos pela frente para tentar entender como elas funcionam. Nesse processo nós escrevemos equações em cima de equações, redigimos instruções, desenhamos esquemas, fluxogramas e uma infinidade de símbolos que confundem até mesmo outros de nós. Foi mais ou menos isso que ocorreu no nascimento deste livro. Antes mesmo de começar a escrever, eu havia listado todos os tópicos que gostaria de incluir, ordenados pela sua complexidade matemática e filtrados pela computacional. Mal sabia eu que eles eram sementes de um intrincado espiral de letras e números, lentamente florescendo em labirintos de extensas equações e termos técnicos. Em uma autocrítica, percebi que se eu fizesse isso estaria condenando este livro a viver nos confins do diminuto grupo de pessoas que veneram a matemática e a arte que pode ser produzida através dela. Foi nesse exato momento que decidi rever toda a filosofia de escrita.

Após ponderar muito, decidi que iria reformular livro como um guia sobre conceitos necessários para produzir arte através da programação. Em vez de focar na matemática para dissecar tópicos de programação, decidi abraçá-la como ferramenta para auxiliar nos desenhos e animações. Os códigos, que antes eram explicados a nível de complexidade computacional, foram simplificados para cobrir apenas os pontos mais importantes quando se trabalha com programação voltada para visualização e arte gerativa. Em suma, não é mais necessário idolatrar a matemática, apenas entendê-la. O resultado é um manuscrito sucinto e com uma linguagem menos formal, concentrando os pontos mais interessantes desse universo que funde o técnico e o artístico. Todavia, não se engane quanto a abrangência do texto. Apesar de muitos tópicos terem sido abreviados, os restantes tornaram-se mais ricos com conceitos essenciais que combinam o melhor de cada uma dessas áreas.

Consequentemente o livro exige um foco maior de você, leitor, e ao mesmo tempo recompensa essa dedicação com uma aceleração no aprendizado. As várias seções foram meticulosamente desenvolvidas de maneira a expor ideias importantes e ilustradas com exemplos cuidadosamente concebidos para reforçá-las. Você perceberá que este livro, por estar relacionado à arte computacional, apresenta conceitos de programação sob uma ótica diferente daquela presente em textos puramente voltados para a programação. Isso permite que tanto pessoas experientes na área quanto aqueles que nunca tiveram contato com

uma linguagem de programação assimilem os conceitos de forma clara e direta. Este é o principal objetivo do livro, pois se você entender como utilizar as ferramentas técnicas da programação para expressar as suas ideias, o que restará do ponto de vista artístico será unicamente o processo criativo.

## Sobre o livro

Este livro foi projetado visando suprir duas necessidades recorrentes quando lidamos com a arte computacional gerativa. A primeira é apresentar o mundo da programação através de uma perspectiva diferente da habitual, na qual o resultado do seu código não se resuma a uma letra branca realçada pelo fundo preto de um console. Ele será uma imagem, uma animação, uma exposição visual, fator muitas vezes subestimado no estudo da programação e matemática. O que nos leva ao segundo ponto, relativo ao esclarecimento e eventual assimilação de como números, fórmulas e instruções integram a quintessência da beleza estética explícita de um padrão, ou capelada de uma sequência numérica infinita. Introduzir, de maneira prática, ambos os assuntos é a melhor maneira de promover a arte gerativa que ainda é muito tímida, e carente de recursos, no cenário brasileiro.

Em seu conjunto total, o livro está divido em três partes, sendo que a primeira é composta por dois capítulos e as outras duas, cada uma por três. Brevemente, eles cobrem os seguintes tópicos:

### Parte I

- Capítulo 1: Inicia com uma breve contextualização da arte gerativa, seguida da justificativa da escolha do Processing como ferramenta de programação. Posteriormente apresenta pontos relevantes relacionados ao Processing e à arte visual em geral.
- Capítulo 2: Explora os conceitos primais da programação. É um capítulo técnico, balanceado sempre por aplicações na arte digital.

### Parte II

- Capítulo 3: Examina o papel essencial da aleatoriedade e do caos na composição imprevisível da arte gerativa.
- Capítulo 4: Demonstra como abstrações puramente matemáticas são a base para gerar formas e movimentos naturais.
- Capítulo 5: Exemplifica que a escolha do método de visualização de uma peça gerativa influencia profundamente em sua apresentação final.

### Parte III

- Capítulo 6: Introduz os fractais e a programação recursiva. Dedicado a mostrar como regras simples são capazes de originar padrões complexos.
- Capítulo 7: Aplica conceitos avançados de programação para simular comportamentos emergentes, diversas vezes encontrados na natureza.
- Capítulo 8: Extrapola o conteúdo do livro, apresentando aplicações elaboradas da programação criativa. Interpreta a programação pragmaticamente e conta com obras de artistas que a dobram em seu favor.

## Pré-requisitos

Os principais pré-requisitos para você, leitor, é que possua uma mente aberta, vontade de aprender e uma participação ativa, fundamentais para garantir o seu desempenho. A programação direcionada para elementos visuais envolve posicionamentos de figuras que são mais facilmente entendidos se esboçados em papel e, sendo assim, desde agora estimulo que você faça constantemente anotações e desenhos.

Do ponto de vista material, ter um computador capaz de executar o software Processing auxilia no aprendizado uma vez que você terá a liberdade de experimentar com seus códigos, alterando parâmetros e observando as consequências. Mesmo assim, a maioria dos códigos possui uma respectiva figura de sua execução e, com um pouco de análise técnica e imaginação, é possível abstrair os resultados dos mesmos.

## Códigos

Os códigos seguidos neste livro se baseiam no Processing que foi construído sobre a linguagem Java e beneficiado, adicionalmente, por uma simplificação na sintaxe. Eles serão evidenciados basicamente em dois meios: no texto, onde comandos e termos relacionados à programação estarão escritos na fonte sourcecodepro, e em uma caixa de contenção que destaca códigos completos:

### Código 0.1 Exemplo de código de programação

```
string texto1 = "Olá, eu sou um exemplo de código!";
string texto2 = "Não se preocupe, serei explicado posteriormente.";

// Exibe as variáveis:
println(texto1);
println(texto2);
```

Em geral, códigos desse tipo são funcionais e podem ser copiados diretamente da caixa de contenção e colados no editor do Processing. Caso seja necessária alguma ação adicional ela será mencionada no texto. Todos os códigos estão sob a [MIT License](#) e podem ser utilizados livremente por você. O download do arquivo contendo os exemplos mais relevantes de cada capítulo pode ser feito através do link de [downloads](#).

## Figuras e imagens

Este livro possui aproximadamente 200 imagens, esquemas, fluxogramas ou fotos, das quais 160 foram geradas diretamente utilizando funções do Processing. Dentre as poderosas funcionalidades desta linguagem existe a de exportação de imagens no formato .pdf ou .svg, ambas sem perdas por resolução. Sempre que possível foi dada prioridade a essa classe de imagem, e você verá que é possível magnificá-la em mais de 5000% sem borrar nenhum de seus detalhes. Algumas vezes você poderá perceber que existem figuras que não estão vinculadas explicitamente ao texto, elas servem como um mostruário de possibilidades da arte gerativa e do Processing, e visam provocar o sentimento de busca de conhecimento no leitor.

Os elementos visuais que foram produzidos pelo autor do livro não foram creditados e estão licenciados pela [Creative Commons Attribution-NonCommercial 3.0 Unported License](#). Todas as demais figuras foram obtidas de fontes externas que autorizaram seu uso e a atribuição do crédito é feita imediatamente em suas legendas. Cabe ressaltar que cada uma delas estão sujeitas as suas próprias licenças, consideradas como propriedade do seus autores originais.

## Distribuição livre

Em sua versão atualmente escrita, este livro é absolutamente livre de qualquer tipo de custo, sendo expressamente proibida a restrição de sua distribuição por uma transação de natureza financeira ou compensatória. O autor acredita na filosofia do conhecimento livre e espera que outros possam se beneficiar com as informações contidas neste livro. Ele também incentiva que o público em geral, sempre que possível, abrace a cultura do compartilhamento do conhecimento e da educação comunitária.

Você pode contribuir voluntariamente com este e futuros trabalhos através da página de [perguntas frequentes](#).

O download da versão digital deste livro deve ser feito através do site original do projeto:

<https://codigotranscendente.github.io/livro/>.

# Introdução

## Identidade fragmentada

A arte computacional é uma intrigante forma de expressão artística que combina o pensamento computacional com a criatividade do espírito humano. Naturalmente surge uma pergunta cuja resposta não é tão simples: a qual área ela pertence? Estaria ela ligada à computação, por utilizar de máquinas e um pensamento metódico e calculista? Ou, quem sabe, à arte, pois busca a mais autêntica forma de manifestação própria? A verdade é que é impossível, e improdutivo, categorizá-la em um único extremo, já que ela se nutre do que há de melhor em ambos. Infelizmente ainda é um campo que sofre de preconceitos e paradigmas justamente por buscar uma simbiose entre os opostos desse espectro dual.

É bem possível que grande parte da dicotomia existente seja devida as palavras que usamos, arte computacional, para caracterizar esse tipo de expressão artística. Ambas as palavras são perigosamente ambíguas e a definição de uma delas, “arte”, é tão controversa que alguns autores questionam sua possibilidade e utilidade<sup>1</sup>. Usualmente são adotadas dimensões de discussões, como valor ou impacto histórico, social, ideal, conceitual, sensorial, estético e outros<sup>2</sup>, mas devido a complexidade e fragilidade do termo, uma definição única ainda não foi construída. Deste modo iremos focar no estudo da palavra “computacional” que possui sua origem na palavra computação que, acredite ou não, não tem a ver necessariamente com computadores. Dentre diversos outros significados a computação pode ser definida como o processo de busca objetiva de soluções (saídas), utilizando de dados (entradas), através de um algoritmo que, por sua vez, é composto por uma série de instruções ordenadas. O grande problema é que essa é uma definição extremamente técnica e muitas vezes mal interpretada. A prova disso é que você pode não ter percebido, mas o maior exemplo de computação está lendo essas palavras agora mesmo: *você!* Ao ler qualquer texto você está recebendo informações, processando-as e criando significado. Você inconscientemente emprega o *pensamento computacional* na maioria de suas atividades, como quando cozinha, conversa com outra pessoa, resolve um problema matemático ou até mesmo cria um trabalho artístico. Portanto, pode-

<sup>1</sup>Adajian T. (2018). *“The Definition of art”*. Stanford Encyclopedia of Philosophy, Stanford, California. ISSN: 1095-5054.

<sup>2</sup>Davies S. (2001). *“Definitions of art”*. The Routledge Companion to Aesthetics, New York, New York. ISBN-13: 978-0-41-532798-5. Pages.169-179.

mos adaptar o conceito da computação para um contexto mais holístico, permitindo que ela seja colocada adjacente à palavra arte:

A computação, na arte, tem a função de assistir no processo da própria expressão artística, originada de entradas, que não são nada mais do que a subjetividade, paixão, instinto e imaginação humana, e feito através do uso de um algoritmo, sendo este a quebra do método artístico em passos bem definidos.

Ao fazermos essa reinterpretação da palavra computacional percebemos que existem um ponto chave que une os dois universos, o *processo*, constantemente referenciado durante todo este livro. Talvez seja nesse aspecto que, supostamente, exista a ânsia de se separar o artista tradicional de um computacional, afinal o primeiro ainda é visto sob as lentes do romanticismo: um gênio solitário de aura mística cujas obras surgem sem esforço. Essa mentalidade é reforçada pelo fato de que o artista tradicional não precisa<sup>3</sup>, conscientemente e abertamente, fragmentar sua arte em passos, uma vez que ele é o próprio autor de sua obra final. A maior parte desses preconceitos foram desmistrificados ao longo dos anos, em especial, pelo movimento americano denominado *process art*, no qual os processos, métodos e a racionalização artística possuem um foco maior do que a *obra de arte em si*<sup>4</sup>. Assim como seu par, o artista computacional deve compreender profundamente *o que e como* ele quer se expressar, a diferença<sup>5</sup> é que na maioria dos casos será necessário instruir um agente, normalmente (e em nosso caso) o computador<sup>6</sup>, para que ele possa sintetizar essa manifestação artística. A responsabilidade está dividida entre o agente criativo, a pessoa, e o agente realizador, o computador que possui um papel essencial na arte computacional. No entanto o cerne criativo ainda permanece onde ele sempre esteve, no ser humano. E é desta forma, furtivamente, que a arte computacional encontra seu caminho, não originada puramente da lógica ou do instinto, mas sim de uma estrutura nova, que funde os dois em algo que eles nunca poderiam ser de maneira individual, a beleza da razão e a inteligência da emoção.

## O gerativo

O termo arte gerativa, generativa ou procedural é muitas vezes empregado como sinônimo de arte computacional, mas existem certas diferenças que devem ser evidenciadas. São múltiplas as definições encontra-

<sup>3</sup>Isso não significa que ele não o faça. Cabe lembrar que a arte é um ofício como outro qualquer e pode ser aprendido ou aperfeiçoado.

<sup>4</sup>Grant K. (2017). *"All About Process"*. Penn State University Press, University Park, Pennsylvania. ISBN-13: 978-0-27-107744-4.

<sup>5</sup>Note que esta diferença é incapaz de sustentar um argumento sólido de separação dos dois tipos de classificações. Os processos - método, lógica, execução - por trás de ambos são de mesma natureza.

<sup>6</sup>O computador foi utilizado apenas por conveniência. A computação pode ser empregada em diversas máquinas, agentes ou processos, sejam eles mecânicos ou biológicos.

das na literatura, internet ou até mesmo em explicações orais e aqui são apresentadas as mais relevantes para o entendimento das duas.

O termo “arte computacional”, conforme explicado na seção passada, é usualmente encontrado no sentido incorreto de “arte de computador”, fato relevado em virtude de nossa própria língua nativa. Em outros países, como os Estados Unidos, a distinção é mais evidente visto que o primeiro tipo é escrito como *computational art* e o segundo como *computer art*. De qualquer forma, estas duas expressões, definidas como obras formadas com o auxílio de técnicas computacionais ou de um computador, falham ao serem usados como classificadoras de uma composição pelo infortúnio de serem amplamente genéricas, não remetendo ao estilo, técnicas, qualidades estéticas ou ao porquê<sup>7</sup> (intenção do artista). Empregá-las nessa função seria similar a tentar classificar todos os gêneros e subgêneros da música utilizando uma só palavra. Algumas das categorias cobertas por elas, que inclusive podem ser parte gerativa, são arte digital, eletrônica, evolucionária, procedural, interativa e virtual dentre muitas outras.

No contexto especial da “arte gerativa”, a definição mais referenciada dessa prática na literatura é a do professor e artista Philip Galanter em seu artigo “O que é Arte gerativa?”<sup>8</sup>, posteriormente revisada pelo próprio acadêmico em uma forma mais clara, completa e sucinta no texto “O que é Complexismo?”<sup>9</sup>. Segundo Galanter:

“Arte gerativa se refere a qualquer prática artística na qual o artista cede o controle a um sistema com autonomia funcional que contribui para, ou resulta em, uma obra de arte completa. Nesses sistemas estão incluídas instruções de linguagem natural, processos biológicos ou químicos, programas de computadores, máquinas, materiais auto-organizáveis, operações matemáticas e outras invenções procedurais.”

Note que essa definição é restrita o suficiente para não incluir todo tipo de prática artística e, ao mesmo tempo, contempla atividades originárias, e permite explorações futuras, da arte gerativa. Galanter também é cuidadoso ao evitar blindar o propósito da arte. Ele estabelece que o resultado final é uma obra completa, mas o intento continua sendo de mérito integral do campo das artes e artistas. O principal ponto de Galanter é esclarecer os mecanismos de produção de um trabalho e reforçar aquilo que realmente destaca o gerativo: a total, ou parcial, *delegação do controle* para um sistema que apresenta *autonomia funcional*. Esta última é qualificada pela independência da necessidade de decisões, ou comandos, de momento em momento por parte do artista. Você perceberá que essa mínima ação irá reverberar em consequências colossais. A maior delas é que, em vez de expressar a sua individualidade através de um único trabalho

---

<sup>7</sup>Lambert, N. (2003). “A critical examination of computer art”. University of Oxford, London.

<sup>8</sup>Galanter, P. (2003). “What is Generative Art? Complexity theory as a context for art theory”. International Conference on Generative Art. Generative Design Lab, Milan Polytechnic, Milan (2003).

<sup>9</sup>Galanter, P. (2008). “What is Complexism? Generative Art and the Cultures of Science and the Humanities”. International Conference on Generative Art. Generative Design Lab, Milan Polytechnic, Milan (2008).

artístico, você irá gerar uma família de obras, podendo escolher candidatos que melhor representem a sua intenção.

Abrir mão de parte do processo de execução artística pode parecer contra intuitivo em um primeiro momento, principalmente quando o meio adotado neste livro é o programa de computador que é desenvolvido especificamente para fornecer uma resposta correta. Contudo, adicionar aleatoriedade e autonomia permite abrir o caminho para o desconhecido e surpreender-se com o fato de que não existe a resposta correta, e sim uma irrupção de cenários e horizontes pluralistas repletos de características inesperadas, belas e intrigantes que só podem ser reveladas pelo gerativo. Quanto a sistemas desse tipo que não usam computadores, e possivelmente nem computação, o maior de todos os exemplos é o fenômeno que batizamos como natureza, exímia em criar uma infinidade de formas, cores e detalhes através processos complexos, da emergência e também da aleatoriedade. As manchas nos pelos de uma onça, tigre ou zebra nunca são iguais. Árvores de uma mesma espécie, apesar de serem parecidas entre si, não tem a mesma forma, tamanho ou número de folhas. Até mesmo nós, em parte, somos produtos do acaso, resultante de mutações durante nossa concepção. Essas pequenas variações e incertezas que borbulham a níveis macro e microscópicos serão nossas principais inspirações para permitir que o computador, um máximo do sintético, possa produzir imagens que remetam ao caráter orgânico da natureza.

## Nós, artistas

O tema da estética gerativa, no contexto da programação, possui sua base prática majoritária derivada do plano das exatas conforme você perceberá nos capítulos que se seguem. Seu segmento teórico é uma sobreposição de conceitos das ciências e arte, incluindo profundas observações e emulações do natural, estudos de estética e produção de estilo. Por uma questão de prioridades não entraremos em detalhes na área que alude exclusivamente à arte, como suas formas de manifestações ou intento. Sendo assim, cabe particularmente destacar duas explicações quando eu citar esse termo.

Primeiramente, sempre que me referir a artistas e a arte não estarei apenas indicando pessoas e obras famosas, como o pintor Picasso e sua lendária tela *"Guernica"*. Arte é tratada, neste livro, como qualquer forma de trabalho ou expressão criativa, tais como pintura e desenho, poesia, música, esculturas, grafite, rascunhos em caderno ou até mesmo um vídeo gravado em um celular. Isso não é feito com o intuito de denegrir a arte e dizer que ela aceita qualquer coisa, muito pelo contrário, o objetivo é torná-la mais nobre anunciando que ela aceita qualquer um de nós. Em vista disso, seria muita pretensão minha rotular qualquer figura neste livro como arte. Os exemplos, por mais que projetados para explicar os conceitos, e concretizados através de uma representação estética, não passam de lições. Em uma visão artística, elas não possuem objetivo, nem impacto ou permanência.

Em segundo lugar, a definição do que realmente é arte é uma discussão para doutores no assunto e, em nenhum momento, tentarei defini-la ou restringi-la. O fator "arte", neste livro, limita-se a explicar como usar técnicas de programação para conceber padrões específicos e conferir (ou criar) efeitos visuais característicos ao seu trabalho, provendo uma visão de como o computador pode ser um grande aliado no

seu processo artístico. Na realidade, arte é tão subjetiva e particular quanto a própria vida. Existem pessoas que idolatram a arte renascentista e nunca considerariam a arte moderna como “arte”, e pessoas que veneram o pós-modernismo, mas sentem repulsa pelo grafite. Até o presente momento, não existe um padrão universal quantificável de qualidade para a arte que possa ser mensurável ou mesmo definido. Em última instância, assim como a beleza, a arte está nos olhos de quem vê.









# ORIGEM

"Comece pelo começo," disse o Rei, muito seriamente, "e continue até chegar ao fim: então pare".

- LEWIS CARROLL, ALICE NO PAÍS DAS MARAVILHAS.





[ CAPÍTULO 1 ]:

# Primeiros Passos

Olá Mundo!

O início de toda jornada começa com aquela sensação de que a caminhada será longa, mas que ao final terá valido a pena. É com esta mentalidade que você deve perseguir cada uma das folhas que ler. Ao terminar o livro você terá dominado conceitos que transcendem a arte gerativa, podendo ser usados tanto no desenvolvimento de aplicativos para fins artísticos quanto em programas com objetivos científicos. Grande parte desse conhecimento está inserido nas entrelinhas da lógica e do pensamento computacional, por essa razão você deve sempre se perguntar como pode usar o que é apresentado para impulsionar seu lado criativo e engenhoso.

Neste capítulo começaremos com uma breve introdução sobre a arte gerativa e trabalhos que marcaram sua origem, sua história e suas excitantes aplicações na atualidade. Em seguida será justificado o uso da linguagem de programação central deste livro, o Processing. A partir desse ponto você será direcionado rapidamente para a parte prática da computação e em pouco tempo estará criando seus próprios programas.

Boa leitura e mãos à obra!

## 1.1 | Uma breve história da arte gerativa

---

A prática da arte gerativa, assim como o próprio termo que é usado para descrevê-la, surgiu em meados de 1960 e até hoje está em constante mutação e redefinição. Sua origem formal pode, muito provavelmente, ser traçada a uma exibição em 1965 batizada de *Generative Computergraphik*<sup>1</sup> que mostrou o trabalho de um dos pioneiros da arte digital, o acadêmico alemão Georg Nees. Uma das obras apresentadas por ele, *23-Ecke*, figura 1.1, era composta por uma matriz de polígonos formados por vértices posicionados

---

<sup>1</sup>Boden, M. A. e Edmonds, E. A. (2009). *What is generative art?*. Digital Creativity 20(1): pags.21-46.

aleatoriamente em um pequeno espaço, em cada um confirmando-se as raízes do gerativo. A década de 1960 representou uma época especial para a arte digital como um todo devido ao rápido desenvolvimento dos computadores e da computação. É curioso notar que os precursores desse tipo de prática foram cientistas e acadêmicos, fato justificável uma vez que eram essas pessoas que estavam inseridas em um meio que possuía as ferramentas para o trabalho e exploração da lógica e padrões formados por regras ordenadas ou pelo caos. Além de Nees (fig. 1.2a), outros importantes nomes que podemos citar são o do matemático Frieder Nake (fig. 1.2b), dos artistas Manfred Mohr (fig. 1.3) e Vera Molnár (fig. 1.4a) e do engenheiro Michael Noll (fig. 1.4b). O ponto de marco, o computador, também é um dos motivos pelo qual muitas vezes os termos arte gerativa e arte computacional são utilizados quase como sinônimos, de maneira intercambiáveis. Rigorosamente, a arte gerativa é singular uma vez que não exige o meio computador e sim um processo com algum grau de autonomia. Por exemplo, um robô vibratório com um lápis em sua base constitui a fundação para formar uma peça gerativa, assim como reações químicas em uma placa de Petri, peças musicais em um sintetizador eletrônico ou até mesmo alterações atmosféricas em um ambiente fechado<sup>2</sup>.

No Brasil, é possível que o nome mais conhecido quando citamos a arte computacional seja o de Waldemar Cordeiro, artista plástico e teórico do movimento da arte concreta e do popcreto. Seus trabalhos com sólida base teórica, tantos nos anos 50 com o concretismo quanto nos anos 60 com o popcreto, assim como sua experiência social como paisagista e urbanista mostraram-se ideais para a experimentação da arte com computadores. Em 1968 Cordeiro iniciou uma parceria com o físico Giorgio Moscati, na qual investigaram como técnicas do domínio da matemática, física, psicologia e arte poderiam ser fundidas na expressão artística. Uma das obras da dupla, considerada a primeira obra de *computer art* da América Latina e dotada de uma originalidade que repercutiu internacionalmente, *Derivadas de uma Imagem* (figura 1.5), consistia em derivar numericamente uma imagem, repetindo o processo em até três vezes sobre a digitalização manual realizada por Cordeiro. Segundo Moscati, “Cordeiro insistiu muito em usar uma imagem com forte conteúdo humano e emotivo para ser transformada por uma ‘maquina fria e calculista’.”<sup>3</sup>. Um dos pontos que chama a atenção em Cordeiro é a sua preocupação com questões que ultrapassam a pura expressão da arte; suas obras, como a citada anteriormente, provocaram discussões acerca da natureza e validade social e estética da arte pelo computador, autoria em produções digitais e até sobre o princípio visual da interpretação de estímulos sensoriais<sup>4</sup>.

Inserida em um meio de característica naturalmente permissível, a arte gerativa possibilita que sua história seja enriquecida com teorias daqueles que defendem que a mesma surgiu consideravelmente antes

---

<sup>2</sup>Famosa peça do artista Hans Haacke intitulada *Condensation Cube*.

<sup>3</sup>Velho L. (1993). “Waldemar Cordeiro”. Arteônica - Homenagem a Waldemar Cordeiro. Catálogo da exposição Sibgrapi VI.

<sup>4</sup>Nunes F. V. (2004). “Waldemar Cordeiro: da arte concreta ao ‘popcreto’”. Universidade Estadual de Campinas. Dissertação de Mestrado

de 1960. Na arquitetura de civilizações antigas são evidentes as leis que permeavam escolhas de design em instalações importantes. Templos hindus, como o Kandariya Mahadeva (fig. 1.6a), exibem repetições fractais em sua estrutura<sup>5</sup> que refletem a própria crença de um cosmos autossimilar. Padrões geométricos presentes em construções islâmicas, particularmente o complexo histórico Shah Nematollah Vali ou a praça do Resgistan (fig. 1.6b), podem ser estratificados em regras geométricas bem definidas, e simultaneamente permutáveis, para formar infinitas variações como argumentado pelo artista Roman Verostko. A matemática como geradora de padrões por meio de simetrias, ciclos, repetições, combinações, transformações espaciais e sequências numéricas é encontrada em uma imensa quantidade de artefatos e pinturas das mais diversas culturas como a egípcia (fig. 1.7a), céltica (fig. 1.7b), grega (fig. 1.7c), anglo-saxônica (fig. 1.7d) e outras distribuídas espacialmente e temporalmente ao longo da história humana.

Presente também na música, o gerativo pode ser rastreado ao *Musikalisch Würfelspiel*, figura 1.8, atribuído ao compositor clássico Mozart<sup>6</sup>. Esta peça era sintetizada através do jogar de dados, no qual cada número representava uma pequena seção musical que, posteriormente, eram costuradas sequencialmente formando uma espécie de música do acaso. Uma abordagem similar seria o do artista Marcel Duchamp que, entre 1912 e 1915, criou duas<sup>7</sup> peças musicais baseadas na chance. Em *Erratum Musica* Duchamp retirava, aleatoriamente, cartas de um chapéu que continham vozes musicais, formando uma composição completamente inesperada.

Na classe dos sistemas citados, parte deles estão sujeitos a críticas, em especial aqueles que utilizam completamente de etapas predefinidas (padrões geométricos, fórmulas matemáticas ou regras determinísticas). Margaret Boden e Ernest Edmonds evidenciam que a comunidade artística possui certa objeção quanto a esses tipos de trabalhos por não existem surpresas, e que eles estariam ligados muito mais a visualizações matemáticas, arte algorítmica ou arte computacional *programada* do que à arte gerativa em si. A justificativa seria que um processo definido passo a passo é inteiramente conhecido em qualquer instante de tempo, não havendo nenhum tipo de decisão *autônoma* pelo agente gerador. Em outras palavras, a arte gerativa é formada quando existem regras que limitam ações ao invés de impô-las, e sempre

<sup>5</sup>Rian, I. M. et al (2007). "Fractal geometry as the synthesis of Hindu cosmology in Kandariya Mahadev temple, Khajuraho". Building and Environment 42(12): pags.4093-4107.

<sup>6</sup>Mozart, W. A. (1796). "Anleitung Walzer oder Schleifer mit zwei Würfeln zu componiren, so viele man will, ohne etwas von der Musik oder Composition zu verstehen". N. Simrock

<sup>7</sup>*Erratum Musica* e *La Mariée mise à nu par ses célibataires même. Erratum Musical*

deve existir algum tipo de elemento não controlado durante sua execução<sup>8</sup>. Boden e Edmonds propõem uma conclusão para esse assunto de maneira sucinta e clara: atualmente, tanto os programas que literalmente seguem regras quanto aqueles que apenas as interpretam, costumam ser tão complexos que não possuem distinção no nível mais fundamental, sendo improdutivo insistir em esclarecer tecnicamente a segregação entre arte gerativa e algorítmica. As diferenças entre as duas estariam ligadas mais ao “sentimento” de falta de controle sobre o resultado gerado do que ao controle em si, já que esses dois tipos de prática são capazes de formar obras completamente imprevisíveis pelo autor na etapa de concepção.

No que concerne a nova geração de artistas gerativos, pós 1990, pode-se observar o mesmo comportamento experimental de seus precursores, mas em um espectro de abrangência muito maior. Nos anos que se seguem os computadores e ferramentas digitais começaram a ser atualizados a uma velocidade espetacular que, somados a popularização da Internet, facilitaram o processamento, compartilhamento e acesso a dados, originando a Era da Informação. Essa nova casta de artistas, armada com poderosas ferramentas tecnológicas, está focada em mesclar especialidades da engenharia e computação com a estética e forma da arte. Casey Reas e Benjamin Fry, criadores do Processing, são programadores e artistas com trabalhos (fig. 1.9) relacionados à visualização de dados e exploração da estética através do código, tornando a linha divisória entre o técnico e o artístico cada vez mais tênue. Jared Tarbell, Marius Waltz e Joshua Davis são outros exemplos de artistas que combinam algoritmos, design e estilos únicos para criar belíssimas peças gerativas. Muitos outros casos similares podem ser citados: Patrick Tresset construiu um braço robótico<sup>9</sup> e o programou para rascunhar ilustrações estilizadas de si. Jon McCormack desenvolveu um algoritmo capaz de sintetizar imagens de plantas evoluídas artificialmente por algoritmos genéticos formadas pelos *logos* de companhias de petróleo e derivados<sup>10</sup>. LIA (fig. 1.10) utilizou de uma impressora 3D para explorar o comportamento de esculturas<sup>11</sup> constituídas por filamentos de ABS<sup>12</sup>. Em adição aos tipos passivos de obras, a computação permite também a arte interativa, em que se usam de periféricos poderosos como o *Kinect*, o *Leap Motion* ou o óculos de realidade virtual (VR), para impulsionar uma interação muito mais pessoal e próxima ao espectador. A arte, ou o produto final dependendo do ponto de vista, se torna uma colaboração triádica do artista/programador, do computador e também do usuário. A empresa Nike explorou essa ideia em uma instalação que utilizava de um *Knect* e sensores conectados

<sup>8</sup>Este tipo de discussão deve ser deixada para os acadêmicos no assunto uma vez que existem contra argumentos para ambos os casos. Se um programa que traça um padrão passo a passo é desenvolvido de forma que produza algo tão complexo que seja impossível de ser previsto antes de sua execução, ele pode ser considerado gerativo ou não? Da mesma forma, um sistema composto por agentes autônomos que tomam decisões preestabelecidas (mas não predeterminadas) seria gerativo? Se sim, por que, já que para uma simulação com condições iniciais conhecidas seria possível determinar os estados dos agentes em qualquer tempo? Cabe um questionamento.

<sup>9</sup>Paul the Robot (2011).

<sup>10</sup>Fifty Sisters (2012)

<sup>11</sup>Filament Sculptures (2014).

<sup>12</sup>Acrylonitrile Butadiene Styrene - Copolímero formador de um material termoplástico rígido e leve.

a uma esteira ergométrica para transformar uma pessoa que corria sobre ela em um fluxo de partículas animadas<sup>13</sup>.

Esta pequena introdução teve o objetivo de prover uma visão holística sobre as origens e aplicações da arte gerativa. É desafiador traçar uma linha histórica detalhada de um campo que foi formalizado recentemente, na ordem de décadas, mas cuja essência experimental está presente na própria natureza humana. O que se pode observar é que a cada ano aumenta-se a demanda e o interesse por exibições que combinem o artístico e o científico, essencialmente reforçando a ânsia de uma simbiose orgânica entre forças duals, seja do caos e a ordem ou do homem e a máquina.

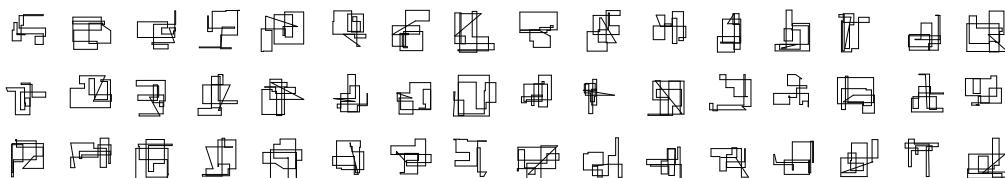
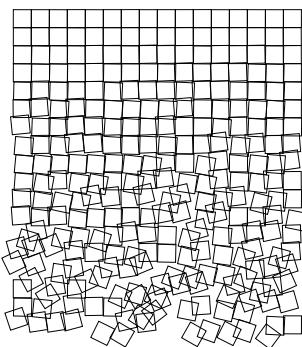
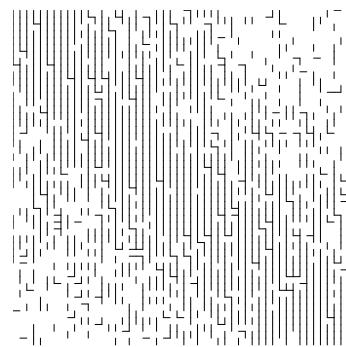


Figura 1.1: Georg Nees, *23 Ecke*, reproduzido utilizando o Processing.



(a) Georg Nees, *Schotter*.



(b) Frieder Nake, *Random Walk Through Raster*.

Figura 1.2: Reinterpretação de obras clássicas do movimento gerativo utilizando o Processing.

---

<sup>13</sup>Nike: *Force of Nature* (2015).

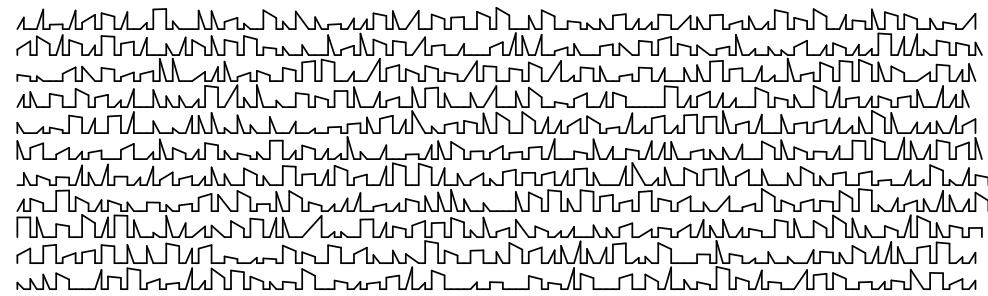
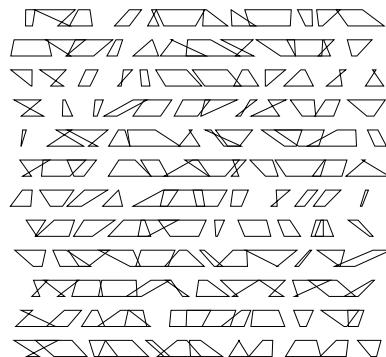
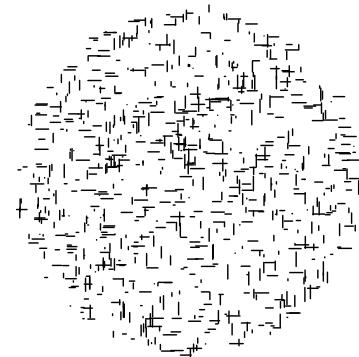


Figura 1.3: Manfred Mohr, *P122+*, reproduzido utilizando o Processing.

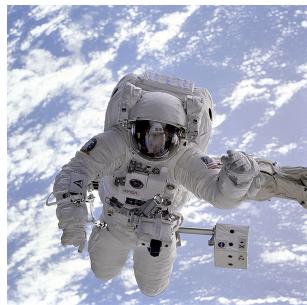


(a) Vera Molnár, *144 Trapèzes*.

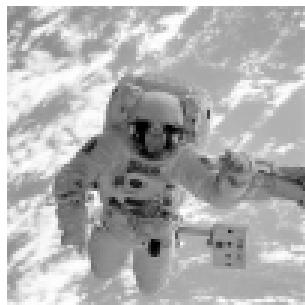


(b) Michael Noll, *Computer Composition With Lines*.

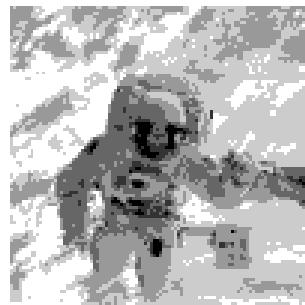
Figura 1.4: Reinterpretação de obras clássicas do movimento gerativo utilizando o Processing.



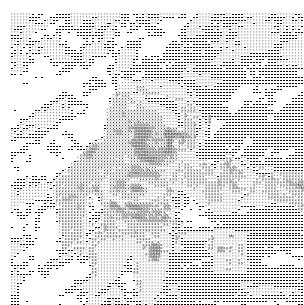
(a) Astronaut Michael Gernhardt during extravehicular activity (EVA)<sup>a</sup>, cortesia da NASA.



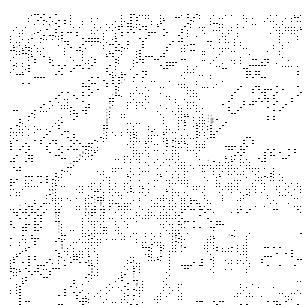
(b) Imagem reduzida.



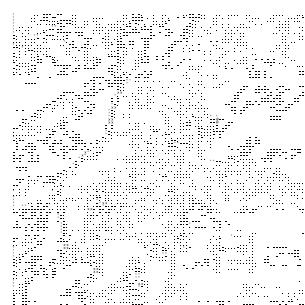
(c) Quantização de cores.



(d) Derivada de ordem zero.



(e) Derivada de ordem um.



(f) Derivada de ordem dois.

Figura 1.5: Reinterpretação da obra *Derivadas de uma imagem*<sup>14</sup>, de Waldemar Cordeiro, utilizando o Processing.

<sup>14</sup>O objetivo do autor é puramente mostrar, de forma visual, o conceito técnico por trás da obra ao ser aplicado em uma imagem qualquer. O processo artístico de Cordeiro em sua obra original, *Derivadas de uma imagem*, é muito mais complexo, e envolveu escolhas cuidadosas que abrangeram, dentre outros pontos, a seleção da imagem, o método de execução das derivadas e a interpretação final das ilustrações geradas.



(a) Templo de Kandariya Mahadeva<sup>b</sup>, Índia. Cortesia do fotógrafo Fernando de la Torre

<sup>b</sup>Todos os direitos reservados pelo fotógrafo original.



(b) Praça do Registan<sup>c</sup>, Uzbequistão. Cortesia da fotógrafa Janet Marshall.

<sup>c</sup>Todos os direitos reservados pela fotógrafa original.

Figura 1.6: Padrões em estruturas arquitetônicas.



(a) Escaravelho, egípcio, 1458–1479 a.C.



(b) Anel, céltico, 500–400 a.C.



(c) Tigela de libação, grega, 400–300 a.C.



(d) Pingente, anglo-saxão, início de 600 d.C.

Figura 1.7: Padrões em objetos culturais, cortesia<sup>d</sup> do Metropolitan Museum of Art, New York

<sup>d</sup>Todos os direitos reservados pelos artistas originais.

ZAHLEN TAFEL.											
TABLE de CHIFFRES.											
A	B	C	D	E	F	G	H	I	J	K	L
2	96	99	141	41	104	129	11	30			
5	32	6	128	63	1+0	46	134	91			
4	69	95	128	13	123	46	120	2+			
5	40	17	112	169	43	80	97	36	100		
6	128	74	169	43	80	97	36	107			
7	104	127	97	167	154	6%	118	91			
8	129	60	171	53	89	133	91	127			
9	219	54	114	50	140	86	169	3+			
10	98	149	42	156	75	129	69	123			
11	3	87	163	61	123	47	147	33			
12	54	120	10	103	88	37	106	5			

TABLE de MUSIQUE.															
1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.	15.	16.
[Musical score with 16 measures, each measure containing multiple notes and rests.]															

Figura 1.8: Mozart, *Musikalischs Würfelspiel*.



Figura 1.9: *Signals*<sup>e</sup>, cortesia dos próprios autores, Casey Reas e Benjamin Fry.

<sup>e</sup>Todos os direitos reservados pelos artistas originais.

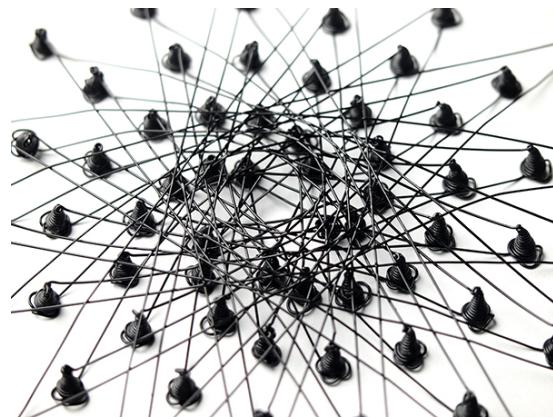


Figura 1.10: *Filament Sculptures*<sup>f</sup>, cortesia da própria artista, LIA.

<sup>f</sup>Todos os direitos reservados pela artista original.

## 1.2 | Processing

---

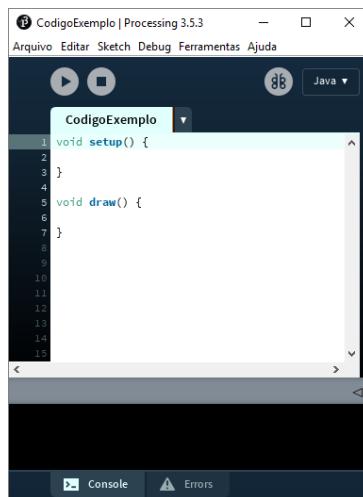
A partir desta seção você mergulhará no vasto oceano da programação voltada para a arte gerativa. O primeiro passo será aprender conceitos universais relacionados à programação e suas aplicações. Sempre que possível foram utilizados exemplos com foco visual para que você possa entender melhor a teoria, afinal, como o livro trata de arte computacional, espera-se pelo menos algum tipo de imagem gerada. O agente responsável por quaisquer desenhos será o computador que, como bem sabemos, não entende diretamente a linguagem humana. Consequentemente você deverá aprender uma linguagem de programação que é um método padronizado para passar instruções para um computador. Posteriormente essas instruções são compiladas, significando que elas serão convertidas para uma linguagem de máquina para serem executadas pelo hardware que compõe o computador físico. Esse processamento gera, a nível de software, uma saída que pode ser um texto, vídeo, música ou, como em nosso caso, uma imagem ou animação.

Em geral, a programação pode ser feita em qualquer linguagem, mas a sua escolha pode ser direcionada para maximizar a velocidade e eficiência do processo técnico e criativo. Existem certas linguagens de programação que se sobressaem devido a recursos nativos para trabalhar com meios que contenham muitos elementos áudio visuais, como é o caso da arte computacional. A maioria desses recursos estão relacionados com instruções que simplificam justamente a produção desses resultados. As opções para quem trabalha nesse campo vem crescendo ao longo dos anos e inclui linguagens, programas e bibliotecas, sendo algumas evidenciadas abaixo:

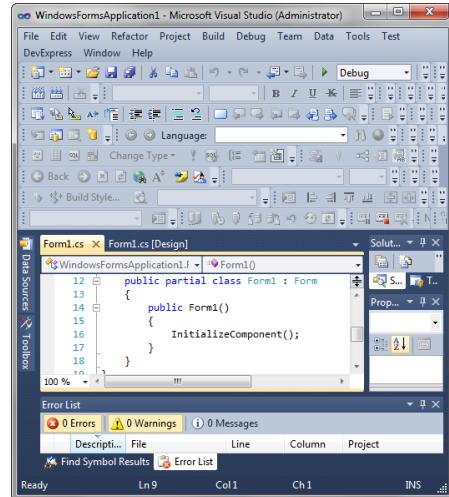
- **Nodebox**: Software composto por uma família de ferramentas desenvolvidas para facilitar o trabalho com design e arte gerativa. Existem duas variações, uma baseada na linguagem Python e outra utilizando uma interface nodal de programação visual<sup>15</sup>.
- **VVV**: Ambiente de programação visual com interface nodal. Especializada na síntese de vídeos, imagens e áudio em tempo real.
- **Cinder**: Biblioteca aberta escrita em C++ voltada para a programação gerativa.
- **OpenFrameworks**: Conjunto de ferramentas abertas desenvolvidas em C++ para auxiliar no processo criativo. É composto por uma série de outras bibliotecas relacionadas a aplicações específicas como geração e processamento de vídeo e imagens, visão computacional e modelagem 3D.
- **StructureSynth**: Aplicação desenvolvida especialmente para gerar estruturas tridimensionais a partir de uma linguagem própria.

---

<sup>15</sup>A programação visual é um tipo de linguagem de programação em que se “escreve” o código através blocos funcionais, ao contrário da forma tradicional que é o texto. A figura 1.12 mostra uma comparação.



(a) Processing.



(b) Visual Studio.

Figura 1.11: Ambiente de desenvolvimento.

- **AdobeFlash:** Plataforma comumente utilizada por web designers para criar imagens, animações e visualizações. Utiliza a linguagem de programação *Action Script*.
- **Processing:** Desenvolvido no MIT<sup>16</sup>, é um software e uma linguagem de programação baseada em Java e idealizada para ensinar os fundamentos relacionados a trabalhos envolvendo artes visuais.

Cada uma dessas opções possuem suas vantagens e desvantagens dependendo do tipo de arte e meio em que se deseja trabalhar. Algumas delas podem se tornar muito específicas ao fazer o uso de linguagens próprias ou focar em um único tipo de aplicação. Por motivos que serão explicados ao longo desta seção, iremos trabalhar exclusivamente com a linguagem Processing.

Processing é uma linguagem de programação cujo desenvolvimento iniciou com Ben Fry e Casey Reas em 2001, no *MIT Media Lab*. Aos poucos foi crescendo a agregando mais e mais entusiastas devido a sua interface simples, sintaxe amigável e grande variedade de funcionalidades. Em 2012 foi criada a Processing Foundation junto com Daniel Shiffman que aumentou a equipe como terceiro líder do projeto. A linguagem

---

<sup>16</sup>Massachusetts Institute of Technology.

em si é baseada em Java que, por sua vez, tem um gigantesco histórico de aplicações em desktop, web e dispositivos móveis como smartphones e tablets.

A premissa do Processing é ser uma linguagem de programação inclusiva, tanto para iniciantes quanto veteranos da programação, cujo o foco é no ensino direcionado à programação criativa e arte gerativa. Seu ambiente de programação é minimalista<sup>17</sup> e limpo para que o programador ou artista possa expressar suas ideias de maneira direta, passando para as etapas de prototipagem, experimentação e prova de conceito o mais rápido possível. Os próprios criadores da linguagem batizaram o ambiente do Processing como um *sketch book*, ou “livro de rascunhos”. Esta maneira de idealiza-lo foi pensada propositadamente para aproximar a arte e a programação de tal forma que o rascunho, o processo de expressar uma ideia, é tão importante quanto o produto final em si.

Em comparação com outras linguagens de programação, como C++ ou Rust, Processing geralmente produz um software com velocidade de execução menor, porém suas vantagens se sobressaem. A linguagem é composta por uma sintaxe simples e amigável com menos regras específicas e estruturais da programação, permitindo um foco maior no processo criativo. Ela também possui uma diversa gama de ferramentas próprias que permitem explorar grande parte das áreas que compõem a computação artística. Em relação a linguagens de programação visual, como as usadas no Nodebox ou VVVV, o Processing transfere mais poder ao usuário através da linguagem escrita, permitindo uma customização cirúrgica das ideias que o programador deseja expressar. Em projetos com maior volume de código a linguagem escrita também é mais fácil de ser gerenciada, organizada e compreendida.

A compatibilidade é outro ponto forte do Processing. O seu ambiente de desenvolvimento compila executáveis para plataformas Windows, Mac OS X e Linux. Somando as suas vantagens, ela também é uma linguagem aberta (*open source*) em que os usuários podem ajudar a corrigir problemas ou *bugs* assim como sugerir novas funcionalidades. Sua popularização encorajou programadores a criar variantes adaptadas para linguagens específicas como o p5.js (JavaScript), Processing.py (Python) e Quil (Clojure). Igualmente relevante, é que Processing é distribuído de graça, sendo mantido por doações.

Em se tratando de ferramentas externas à linguagem, existe uma quantidade respeitável de bibliotecas desenvolvidas por usuários, tanto para Processing quanto para Java, que podem ser instaladas. Bibliotecas são um conjunto de funções e estruturas de dados não nativas à linguagem que, quando adicionadas ao programa, ampliam ou estendem suas funcionalidades. Essa prática tem potencial realmente impressionante nas mais diversas aplicações. Algumas dessas bibliotecas habilitam compatibilidade com periféricos como os detectores de posição Kinect da Microsoft, o LMC da Leap Motion e o microcontrolador Arduino.

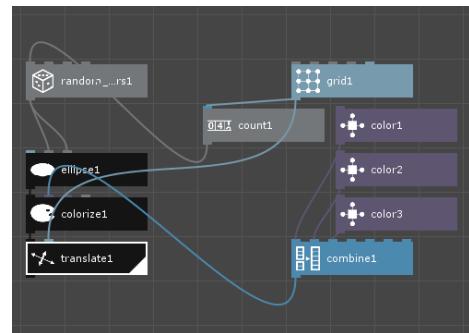
---

<sup>17</sup>Veja uma comparação entre o ambiente do Processing e o Visual Studio, uma ferramenta da Microsoft, na imagem 1.11. Ambos os programas servem a propósitos diferentes, aqui se evidencia apenas as interfaces.

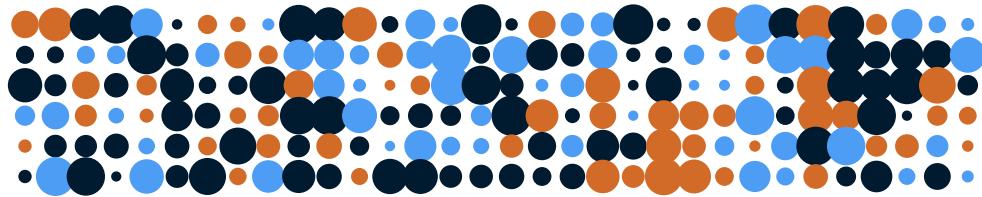
```

1 color[] p = {"#D26B27", "#001A2F", "#4E9DF4"};
2
3 void setup() {
4   size(990,210);
5   background(255);
6   ellipseMode(RADIUS);
7   noStroke();
8   for(int i = 0; i < 32; i++) {
9     for(int j = 0; j < 6; j++) {
10       float r = random(5,20);
11       fill(p[(int)random(p.length)]);
12       ellipse(30+i*30,30+j*30,r,r);
13     }
14   }
15 }
```

(a) Escrita.



(b) Interface nodal.



(c) Resultado dos códigos nodal e escrito.

Figura 1.12: Linguagens de programação.

Outras facilitam o trabalho com especialidades que abrangem a visão computacional, visualização de dados e até mesmo motores de física (*physics engine*) em simulações e jogos.

Todos esses fatores contribuem fortemente na escolha do Processing como a linguagem adotada neste livro para a programação artística. Nada impede que você siga todo o texto utilizando outras linguagens uma vez que, conforme dito, os conceitos ensinados são universais, porém a sintaxe não será a mesma e você terá que realizar as devidas adaptações.

## 1.3 | Ambiente Processing

---

O ambiente de desenvolvimento Processing (*PDE – Processing Development Environment*) é o programa que você irá utilizar para escrever seus códigos, assim como executá-los. No momento em que este livro foi escrito o Processing encontra-se na sua terceira grande versão, mais especificamente a 3.5.3. Você pode obtê-lo no endereço web [processing.org](https://processing.org), na seção de download. O arquivo estará compactado e será necessário extrair o conteúdo para um local do seu computador. Este livro foi baseado na versão do Windows e para abrir o programa basta executar o aplicativo *Processing.exe*. A aplicação será executada em qualquer máquina que suportar o Ambiente de Execução Java<sup>18</sup>, sendo que o Java está incluído no download. O PDE é composto pela interface e itens conforme mostrados na figura 1.13.

Abaixo é feita uma descrição sucinta de cada ponto relevante desse ambiente:

- 1 **Barra de menus:** Contém os menus da aplicação que permitem criar, salvar e carregar projetos, além de consultar exemplos, importar bibliotecas e customizar a interface do PDE.
- 2 **Executar:** Compila o código escrito na seção do editor de texto e mostra seu resultado na janela de exibição e/ou no console.
- 3 **Parar:** Termina a execução do código, fechando a janela de exibição.
- 4 **Abas:** Permite criar novas abas para escrever o código. Utilizado principalmente em programas extensos ou para organizá-los em seções, classes e funcionalidades.
- 5 **Editor de texto:** É o espaço reservado para escrever o código em si.
- 6 **Área de mensagem:** Mostra algumas mensagens do sistema e breves mensagens de erro que possam existir durante ou até mesmo antes da execução do programa.
- 7 **Console:** É onde são exibidos os textos de saída do Processing, como aqueles gerados por algumas funções. Também pode conter eventuais descrições mais detalhadas de certos tipos de erros detectados na execução do código.
- 8 **Erros:** Exibe os erros (em vermelho - figura 1.14) ou mensagens de aviso (em laranja - figura 1.15) do código. Erros são classificados como fatais e impedem ou terminam a execução do código. Alguns deles são a sintaxe incorreta, falta de argumentos nas funções ou tipos incorretos de variáveis. As mensagens de aviso estão ligadas a problemas leves como, por exemplo, o não uso de uma variável declarada, e não impedem a execução do programa.

---

<sup>18</sup>do inglês, *Java Runtime Environment* ou JRE

- 9 **Debug:** Ferramenta adicionada no Processing 3.0 que permite executar o seu programa passo a passo. Essa funcionalidade é muito útil quando se deseja conferir valores de variáveis ou comportamentos de algoritmos.
- 10 **Modo:** Seleciona o modo de programação para o Processing que, por padrão, é a linguagem Java. Modos para JavaScript, Python ou Android podem ser adicionados através desse mesmo botão, na seta lateral, na opção *Adicionar modo....*

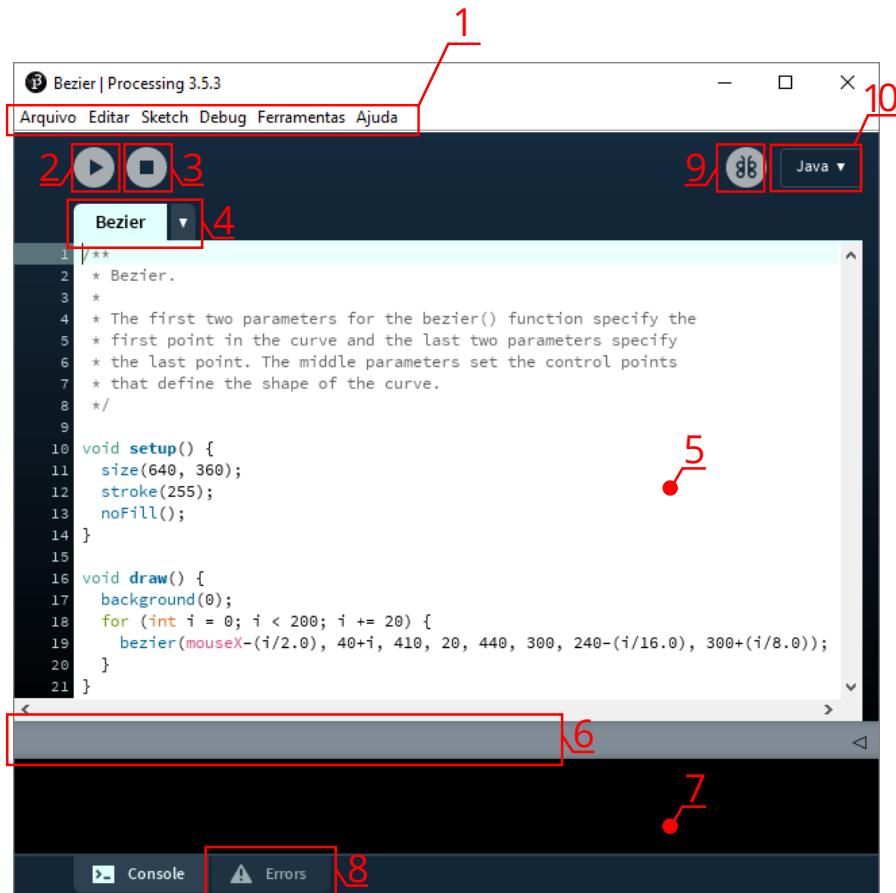


Figura 1.13: PDE Processing para Windows com as principais funcionalidades.

Bibliotecas, como aquelas citadas na seção passada, não fazem parte da versão original do programa e

para usá-las é preciso fazer seu download e instalação. Esse processo é automatizado pelo próprio PDE ao acessarmos a barra de menus, *Sketch* seguido por *Importar Biblioteca...* e *Adicionar Biblioteca....* Na janela que se abrirá, figura 1.16, você pode filtrar os módulos por funcionalidades e adicioná-los através do botão *Install*. Outros recursos a serem carregados ou usados nos seus programas, tais como arquivos de imagens (.gif, .jpg, .tga, .png), texto (.txt, .csv) ou áudio (.wav, .au, .aif, .snd, .mp3) também devem ser explicitamente adicionados através do menu *Sketch* e *Adicionar Ficheiro....* Selecione o arquivo desejado na interface e pressione o botão de *Abrir* para completar o processo.

Você pode salvar os seus códigos para futuras consultas ou reestruturações através do menu *Arquivo* e depois *Guardar como...*, que fará com que o computador crie uma pasta com o nome do seu programa contendo todos os arquivos de código, cuja extensão é *.pde*. Por último, uma vez que sua aplicação estiver completa, você pode exportá-la em forma de um executável, firmando que o código que você escreveu foi transformado em um software **seu<sup>19</sup>**, livre para o compartilhamento com qualquer público. Ao selecionar no menu *Arquivo* o item *Exportar Aplicação...* você verá uma janela, 1.17, com opções para seleção do sistema operacional da aplicação e se você quer ou não incluir o Java para o Windows. Esta seleção aumenta o tamanho final da aplicação, mas também sua probabilidade de funcionar em plataformas que não tenham o Java instalado. O executável criado estará na própria pasta da aplicação. A interface minimalista do PDE mantém o foco direcionado para a programação, evitando preocupações com configurações desnecessárias. Mantenha o Processing aberto, pois na próxima seção você irá escrever seu primeiro programa.

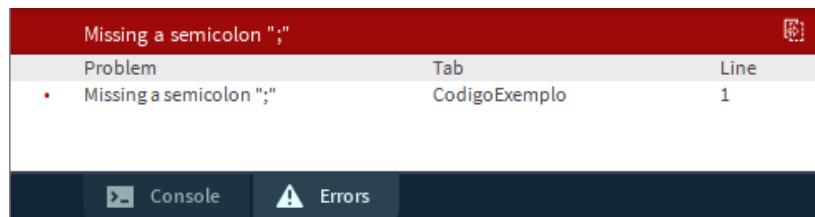


Figura 1.14: Erro - Ausência de ponto e vírgula no final de uma instrução.

<sup>19</sup>O Processing puro é licenciado sob a [GNU Lesser General Public License](#), significando que você pode até comercializar projetos construídos com ele. Mais detalhes sobre ele podem ser encontrados na sua página de [perguntas frequentes](#).



Figura 1.15: Alerta - Variável declarada, mas não utilizada no código.

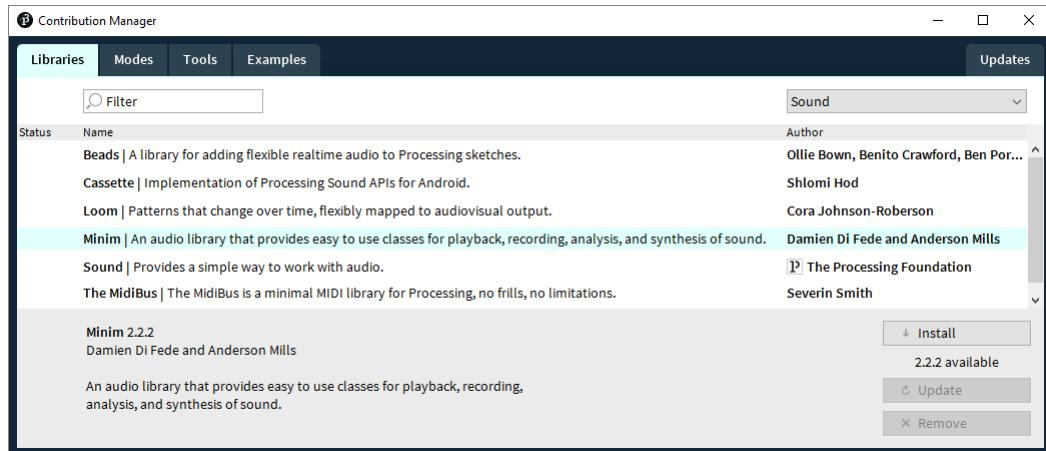


Figura 1.16: Janela de importação de bibliotecas.

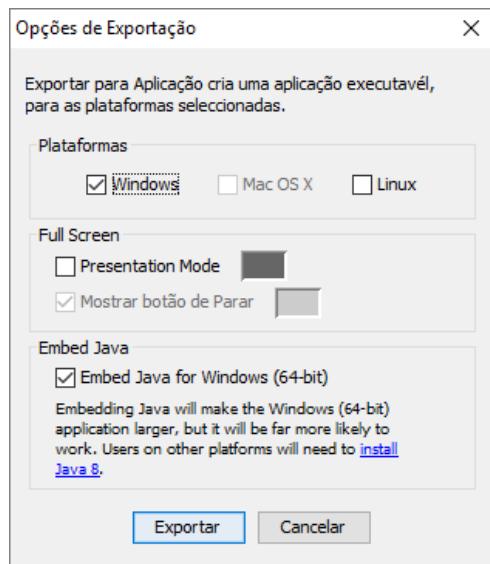


Figura 1.17: Janela de exportação de aplicação.

Nome	Data de modificaç...	Tipo	Tamanho
java	01/12/2019 14:46	Pasta de arquivos	
lib	01/12/2019 14:46	Pasta de arquivos	
source	01/12/2019 14:46	Pasta de arquivos	
Bezier.exe	01/12/2019 14:46	Aplicativo	87 KB

Figura 1.18: Aplicação exportada como um arquivo executável.

## 1.4 | Conversando com a máquina

---

Em virtude da sociedade frenética na qual vivemos, é tentador que qualquer um de nós comece a ler os parágrafos que se seguem utilizando a leitura dinâmica. Eu aconselho que você faça exatamente o contrário e diminua a velocidade do seu raciocínio. Pegue um papel, uma caneta e faça anotações do que julgar importante. Execute os códigos usando o Processing e, ainda mais importante, estude-os com calma. Aprender uma linguagem de programação pode ser intimidador a princípio, mas com o tempo você aprenderá a associar as palavras e comandos. Acima de tudo, seja persistente e você será recompensado com uma moeda muito valiosa, o *conhecimento*. Pronto? Então vamos começar.

Uma das maneiras mais tradicionais de se iniciar na programação é escrevendo um programa que seja capaz de exibir a frase “Olá Mundo!”. No entanto iremos escrever códigos no contexto de arte visual e nada mais pertinente do que exibir um desenho em vez de uma palavra. Sua primeira tarefa será justamente essa. Você pode começar digitando (ou copiando - processo ilustrado na figura 1.22) a linha de código abaixo no editor de texto do PDE:

### Código 1.1 Olá Mundo!

```
// Desenha um quadrado:  
rect(25,25,50,50);
```

Em seguida pressione o botão de Executar (*Run* - ou usando o atalho *Ctrl+R*). Após alguns segundos você verá uma pequena janela como a da figura 1.19. Ela é chamada de janela de saída ou de exibição e mostrará o resultado de tudo que você programar no editor de texto do PDE. Você pode perceber que o computador “desenhou” uma figura (um quadrado), que é o equivalente a exibir um “Olá Mundo!” em uma linguagem de programação tradicional. Saboreie este instante, você está no caminho de se tornar um programador!

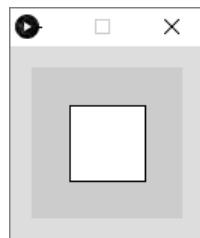


Figura 1.19: “Olá Mundo!” na versão da arte computacional.

Bem, vamos parar por um momento e entender, com mais detalhes, o que as linhas de código 1.1, que você acabou de escrever, realmente significam. A primeira linha começa com duas barras simples (//), definindo um comentário. Comentários são qualquer tipo de anotações que o programador queira fazer, como indicar o objetivo de um código ou o raciocínio por trás do mesmo. Eles são completamente ignorados pelo programa durante sua execução.

A segunda linha contém um comando e depois foi terminada com um ponto e vírgula (;) sinalizando que aquilo que foi escrito antes dele pode ser processado pelo compilador e em seguida interpretado. Essa pontuação é necessária no final de cada uma das instruções que você digitar. Esquecer o ponto e vírgula é uma das principais causas de falhas na compilação do código. A execução dessa segunda linha faz com que o computador trace um quadrado. Isso ocorreu por que você chamou uma função do Processing que desenha um retângulo, `rect()`. Você pode chamar uma função toda vez que desejar fazer uma ou uma série de tarefas específicas. Os números dentro dos parênteses são os argumentos que passamos a essa função. Funções podem ter zero, um, ou mais parâmetros, ou seja, podem pedir (ou não) alguns argumentos ao serem chamadas. Em alguns casos, a função também pode devolver (ou retornar) valores ou outros dados conforme você irá aprender mais à frente.

Conhecer um pouco mais das funções que existem no Processing é especialmente importante. Muitas delas executam ações que você irá usar durante todo este livro e sempre que você trabalhar com a programação. Para funções nativas da linguagem você pode consultar a página de referência que contém uma listagem de todas as funções implementadas até o momento com uma descrição sobre cada parâmetro de entrada e saída, e ainda um exemplo. A página de referência pode ser acessada através da barra de menus em *Ajuda* e depois *Referência*, veja a figura 1.20.

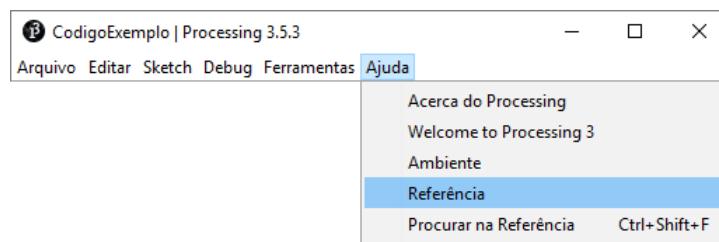


Figura 1.20: Como abrir a página de referências do Processing.

É essencial que você comprehenda como consultar e chamar, de maneira eficiente, as funções do Processing. Em vista disso, investigaremos passo a passo todas as informações presentes na página de referência de uma função qualquer. Para manter a consistência, vamos abrir o link referente à função usada no código 1.1, `rect()`, que está abaixo da seção *2D Primitives* (figuras 1.23 e 1.24), indicando que essa função cria uma forma bidimensional básica. Ao abrir a página dessa função você perceberá diversos tópicos que contém tudo que você precisa saber a respeito da mesma. A tabela 1.1 mostra uma descrição dessas informações, por ordem de importância, para uma função em geral, e para a função `rect()`.

Com base nessas informações, retorno ao código 1.1 e observe que existem números, também chamados de argumentos, entre os parênteses da função `rect()`. Os dois primeiros números, (25, 25) compõem o par (x,y) da origem do desenho do retângulo. Neste caso, 25 pixels de distância no eixo x e 25 pixels de distância no eixo y do ponto (0, 0), também conhecido como origem, da janela de exibição. Os demais valores são referentes ao comprimento e a altura do retângulo em pixels que, por possuírem o mesmo

	<b>Funções em geral</b>	<b>Função rect()</b>
<b>Name</b>	Nome para invocar a função.	Para desenhar um retângulo, rect().
<b>Description</b>	Uma descrição da função e das tarefas que ela realiza.	Explica o que é um retângulo e, rapidamente, os parâmetros da função rect(). Também comenta como tornar as bordas dessa figura mais suaves.
<b>Syntax</b>	Mostra todas as maneiras de chamar a função no editor de texto.	Em nosso caso, escolhemos a versão padrão, que é rect(a,b,c,d), na qual se especificam apenas posição, comprimento e altura do retângulo.
<b>Parameters</b>	Exibe uma explicação sobre cada um dos parâmetros a serem passado para a função, assim como o tipo de cada um deles.	Para a forma utilizada, os parâmetros são: a: Coordenada x do retângulo. b: Coordenada y do retângulo. c: Comprimento do retângulo. d: Altura do retângulo.
<b>Return</b>	Informa qual tipo de variável a função retorna.	Um retorno void indica que a função não devolve nada.
<b>Examples</b>	Mostra a janela de exibição do Processing ao executar os códigos de exemplo.	Você pode ver as diversas maneiras de se desenhar um retângulo.
<b>Related</b>	Lista funções relacionadas à função pesquisada.	rectMode(): Altera a maneira que rect() desenha o retângulo. quad(): Desenha um polígono de quatro lados.

Tabela 1.1: Informações relevantes da página de referência de uma função do Processing.

número (50), formam um quadrado em vez de um retângulo. Você aprenderá mais sobre pixels e origem da janela de exibição na próxima seção.

O intuito de passar pela explicação dessa função com detalhes foi precisamente para evidenciar que toda informação técnica que você precisar sobre as funcionalidades do Processing se encontra na própria referência do programa. Em diversos exemplos ao longo deste livro são usadas inúmeras outras funções cujas explicações são citadas apenas superficialmente. Neste caso, cabe a você, futuro (ou atual) programador artista, consultar as páginas de referências para uma explicação mais detalhada. Você pode começar a exercitar esse hábito explorando algumas das funções básicas que irá utilizar recorrentemente, como desenhar um ponto (`point()`), uma linha (`line()`), um triângulo (`triangle()`) e um círculo ou elipse (`ellipse()`).

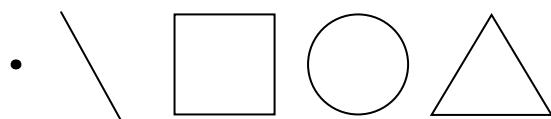


Figura 1.21: Figuras comumente desenhadas no Processing.

Uma das maneiras mais tradicionais de se iniciar é capaz de exibir a frase "Olá Mundo!". No entanto irá ser mais pertinente do que exibir um desenho em vez disso. Você pode começar digitando (ou copiando) PDE:

#### Código 1.1 Olá Mundo!

// Desenha um quadrado:  
rect(25,25,50,50);

Selecione o texto arrastando o mouse enquanto segura o botão da esquerda.

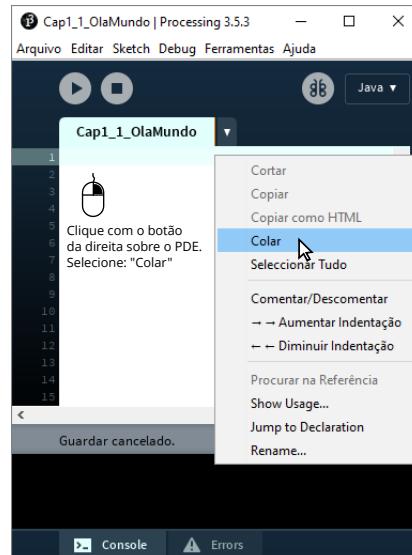


Em seguida pressione o botão de Executar (*Run*) - verá uma pequena janela como a da figura 1.15. Eli... o resultado de tudo que você programar no editor

20

M. P. Berrueto | O Código

(a) Selecionar texto.



(c) Colar texto.

Uma das maneiras mais tradicionais de se iniciar é capaz de exibir a frase "Olá Mundo!". No entanto irá ser mais pertinente do que exibir um desenho em vez disso. Você pode começar digitando (ou copiando) PDE:

#### Código 1.1 Olá Mundo!

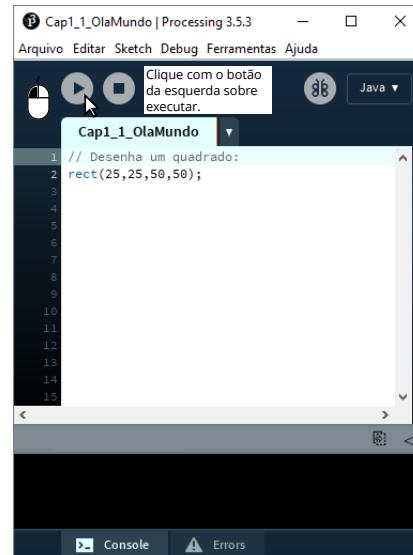
// Desenha um quadrado:  
rect(25,25,50,50);

Clique com o botão da direita sobre o texto.  
Selecionar: "Copiar"

Em seguida clique com o botão da direita sobre o resultado.

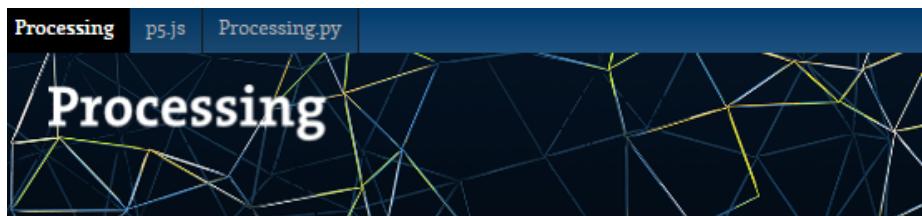
- [Copiar](#)
- [Realçar texto](#)
- [Riscar texto](#)
- [Adicionar nota para substituir texto](#)
- [Adicionar nota a texto](#)

(b) Copiar texto.



(d) Executar programa.

Figura 1.22: Processo para copiar códigos do livro e executar no Processing.



Reference. Processing was designed to be a flexible software sketchbook.

Structure	Shape	Color
() (parentheses)	createShape()	Setting
, (comma)	loadShape()	background()
. (dot)	PShape	clear()
/* */ (multiline comment)		colorMode()
/** */ (doc comment)	2D Primitives	fill()
// (comment)	arc()	noFill()
;(semicolon)	ellipse()	noStroke()
= (assign)	line()	stroke()
[] (array access)	point()	
{ } (curly braces)	quad()	Creating & Reading
catch	rect()	alpha()
class	triangle()	blue()

Figura 1.23: Documentação do Processing.

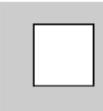
Name	<code>rect()</code>
Examples	 <pre>rect(30, 20, 55, 55);</pre>
Description	<p>Draws a rectangle to the screen. A rectangle is a four-sided shape with every angle at ninety degrees. By default, the first two parameters set the location of the upper-left corner, the third sets the width, and the fourth sets the height. The way these parameters are interpreted, however, may be changed with the <code>rectMode()</code> function.</p> <p>To draw a rounded rectangle, add a fifth parameter, which is used as the radius value for all four corners.</p> <p>To use a different radius value for each corner, include eight parameters. When using eight parameters, the latter four set the radius of the arc at each corner separately, starting with the top-left corner and moving clockwise around the rectangle.</p>
Syntax	<pre>rect(a, b, c, d) rect(a, b, c, d, r) rect(a, b, c, d, tl, tr, br, bl)</pre>
Parameters	<ul style="list-style-type: none"> <li><code>a</code> float: x-coordinate of the rectangle by default</li> <li><code>b</code> float: y-coordinate of the rectangle by default</li> <li><code>c</code> float: width of the rectangle by default</li> <li><code>d</code> float: height of the rectangle by default</li> <li><code>r</code> float: radii for all four corners</li> <li><code>tl</code> float: radius for top-left corner</li> <li><code>tr</code> float: radius for top-right corner</li> <li><code>br</code> float: radius for bottom-right corner</li> <li><code>bl</code> float: radius for bottom-left corner</li> </ul>
Returns	<code>void</code>
Related	<a href="#">rectMode()</a> <a href="#">quad()</a>

Figura 1.24: Referência para a função `rect()`.

## 1.5 | Elementos de imagens digitais

---

A figura que você desenhou na seção passada, produto da função `rect(25, 25, 50, 50)`, ocupava exatamente o centro da janela de exibição. Esse posicionamento certeiro foi fruto da escolha cuidadosa dos argumentos da função, vinculados implicitamente ao conceito de resolução e aos indivisíveis blocos de construção que compõem todas as imagens na computação: os pixels. Quando um computador exibe uma foto ou um vídeo através de seu monitor ele cria a ilusão de uma imagem perfeitamente contínua através da manipulação individual de cores de milhares desses elementos. A figura 1.25 mostra a foto de um monitor magnificado algumas dezenas de vezes, permitindo a visualização do pixel. A resolução de um monitor, e de elementos gráficos digitais, é dada por números que indicam quantos pixels existem, respectivamente, na sua largura e altura. Por exemplo, um monitor com uma resolução de 1280x1024 é formado por 1280 colunas, cada uma com 1024 pixels na vertical, totalizando 1310720 pixels (mais de um milhão de pixels!). Quanto maior a resolução, mais pixels existem para representar os detalhes e melhor é a qualidade de uma imagem digital. A figura 1.26 faz uma comparação de uma mesma imagem com duas resoluções diferentes.

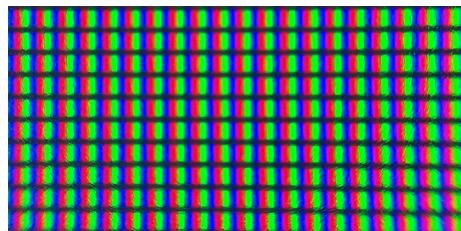


Figura 1.25: Foto ampliada da tela de um monitor de computador.

A janela de exibição do Processing pode ser considerada como um *pseudomonitor* formado por pixels, cuja disposição padrão é de 100 colunas, cada uma com 100 pixels na vertical, gerando uma grade, ou resolução, de 100x100 pixels. Veja que essa janela (figura 1.27) não é nada mais do que um plano cartesiano bidimensional (apenas eixos x e y) e discreto (cada pixel é uma possível posição que pode ser preenchida com uma cor).

Os desenhos de figuras, textos, ou quaisquer elementos visuais na janela de exibição seguem as mesmas leis de desenhos da matemática. Se você conhecer o ponto da origem cartesiana, par (0,0), da janela de saída, assim como o sentido dos eixos coordenados, qualquer desenho pode ser parametrizado por coordenadas, distâncias e equações. Diferentemente das coordenadas cartesianas, a origem em um computador é localizada no canto superior esquerdo da janela, e os eixos crescem (os valores são positivos) da esquerda para a direita (eixo x - horizontal) e de cima para baixo (eixo y - vertical). Isso é tudo que você precisa saber para localizar e posicionar qualquer tipo de imagem ou figura geométrica. Veja na figura 1.28 a diferença entre o plano usado na matemática e no computador.



(a) 770x1015.



(b) 77x102.

Figura 1.26: Imagem com diferentes resoluções.

Fundamentados por estes conceitos, vamos voltar ao desenho da figura geométrica. Suponha que você queira desenhar um quadrado de 50 pixels de lado centralizado na sua janela de saída, e que a mesma possua uma resolução de 100x100 pixels. Sabemos que uma figura que ocupa exatamente o centro de um espaço é aquela que possui distâncias iguais da sua forma aos limites desse espaço. Uma ótima maneira de “visualizar” melhor o que deve ser feito e planejar seus passos antes de programar é criando o hábito de rascunhar suas ideias. Observe a figura 1.29, um esboço rudimentar de nosso objetivo, e verifique como é possível obter todas as coordenadas dos vértices do quadrado.

Uma consulta à referência mostra que o ponto  $(x,y)$  que a função `rect()` recebe para o posicionamento do retângulo é o seu canto superior esquerdo<sup>20</sup>. De acordo com a figura 1.29 é possível determinar que esse ponto é composto pelas coordenadas  $x$  igual a 25 e  $y$  igual a 25, desta forma a função é chamada como:

<sup>20</sup>Para o modo de desenho padrão de um retângulo. Essa informação está disponível na página de referência da função `rect()`.

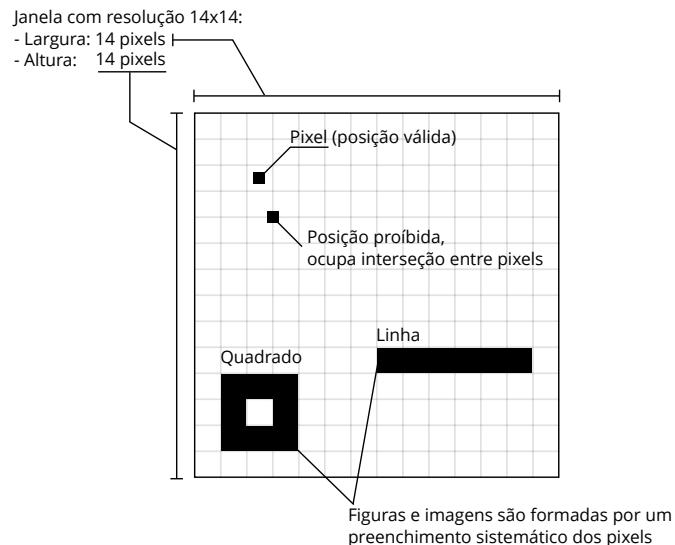


Figura 1.27: Representação da janela de exibição como um plano bidimensional discreto de 14 por 14 pixels.

```
rect(25,25,50,50);
```

Este tipo de procedimento para cálculos e determinação de posições de figuras na janela de exibição é rotineiro na arte computacional e no desenvolvimento de jogos ou interfaces de aplicativos para usuários. A melhor maneira de se habituar a ele é encarando exercícios práticos, então busque, sempre que possível, abordar os programas em um panorama visual. Lembre-se apenas que, se você formular o problema de maneira correta, você sempre poderá utilizar o computador como aliado para solucionar este e outros tipos de cálculos mais complexos.

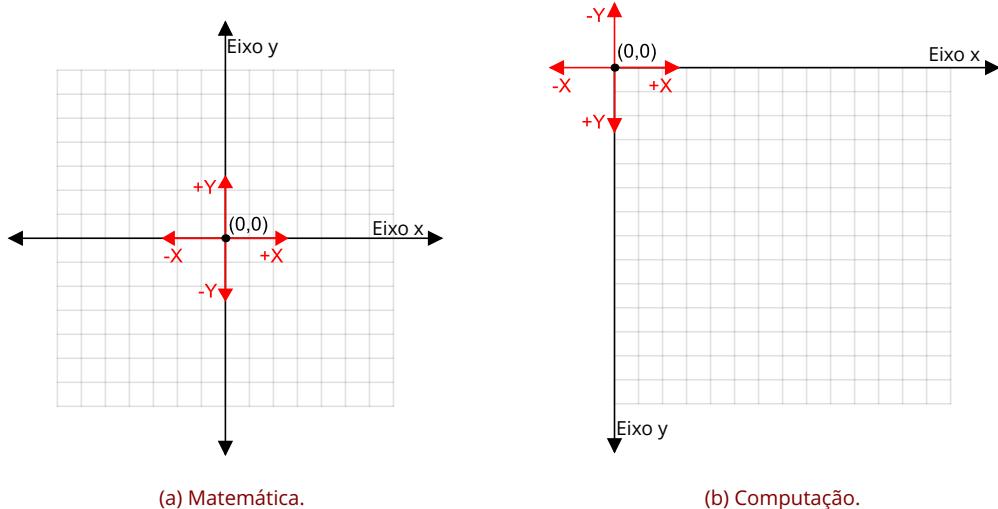


Figura 1.28: Plano cartesiano.

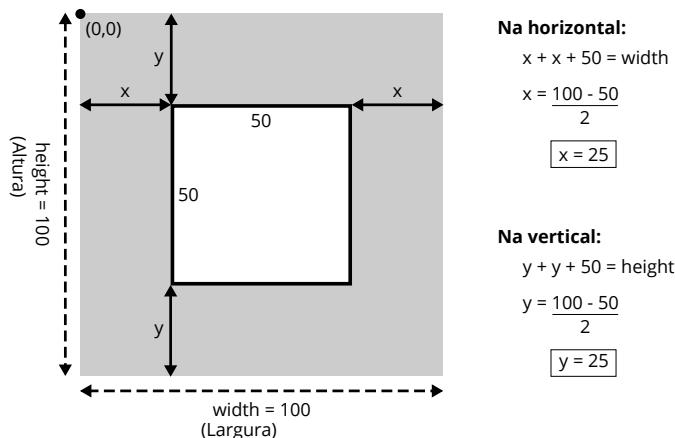


Figura 1.29: Calculando as coordenadas para um quadrado centralizado.

## 1.6 | Estrutura de um programa em Processing

---

O último ponto antes de transformos para o cerne da programação é atentar para os elementos da estrutura básica de um programa do Processing. O código 1.1 que você escreveu anteriormente é válido, mas raramente ele será tão simplório quanto apenas uma linha de texto. A maneira mais comum de começar a escrever um programa no Processing é através das seguintes linhas de código:

### Código 1.3 Estrutura padrão de um programa no Processing

```
void setup() {  
    // Seu código vai aqui.  
}  
  
void draw() {  
    // Seu código vai aqui.  
}
```

Essa estrutura padrão faz o uso de duas funções primordiais do Processing, `setup()` e `draw()`, que adicionam uma série de características e funcionalidades na execução de seus programas. O ponto principal é que você entenda qual tipo de código você escreve dentro das chaves de cada uma dessas funções, então vamos por partes.

A função `setup()` representa o modo estático da aplicação e, se incluída no programa, é geralmente chamada no começo de cada uma de suas execuções (sempre que você pressionar o botão de executar). Dentro do `setup()` são colocadas as funções de configuração do Processing e normalmente são feitas as inicializações de variáveis declaradas anteriormente. Uma função frequentemente encontrada dentro de `setup()` é a `size()`. Ela permite alterar o tamanho da janela de exibição para uma largura e altura arbitrários. Por exemplo, a função `size(640, 480)` gera uma janela de saída com resolução de 640x480 pixels. Outras funções de configuração incluem maneiras de desenhar figuras geométricas, filtros de suavização e mudanças nos espaços de cores.

A função `draw()` representa o modo dinâmico da aplicação e é aquela que define a animação da janela de exibição. Todo código que você escrever dentro das chaves dessa função será repetido múltiplas vezes por segundo. O Processing, como qualquer outro sistema de vídeo, cria animações a partir de imagens estáticas exibindo-as a uma alta frequência. Por padrão, temos o valor de 60 quadros por segundo. A progressão rápida através desses frames cria a ilusão de que as imagens estão se movimentando, gerando um vídeo ou animação. Portanto, se você quiser que um objeto se move em tempo real na janela de exibição será necessário escrever o código dentro da função `draw()`. É possível alterar a frequência de repetição ou o número de frames por segundo chamando a função `frameRate()` dentro de `setup()`.

Em alguns casos você perceberá essas funções sendo utilizadas individualmente e isso é perfeitamente válido se atender aos objetivos visados. Se você desejar gerar apenas uma imagem estática, um quadro, você pode muito bem fazer isso escrevendo todo código dentro da função `setup()` e omitir a função `draw()` já que não existirá uma animação. Por outro lado, você pode simplesmente omitir a função `setup()` e

manter a `draw()` se quiser uma animação com as configurações padrão do Processing, porém isto não é muito comum. Existe ainda o caso de você usar uma função nativa do Processing sem o `setup()` e o `draw()`, como exemplificado na primeira linha de código apresentada neste livro, em que são usadas as configurações padrão e não existem animações.

## 1.7 | Sumário

---

Neste capítulo você conheceu um pouco mais da origem da arte computacional e seu desenvolvimento ao longo dos anos. Em seguida foram apresentadas linguagens de programação, ferramentas e softwares que podem ser usados para trabalhar com a arte computacional e gerativa. Dentre eles o Processing foi escolhido devido as suas inúmeras vantagens, que se estendem desde a interface e sintaxe minimalistas até sua filosofia *opensource*.

O próximo passo foi se familiarizar com o ambiente Processing, o PDE, que será usado para escrever todos os seus códigos. Abordando esse tema, você escreveu e executou seu primeiro programa para exibir um "Olá Mundo!" no estilo da arte computacional. Esse foi um exercício simples, mas que aludiu a assuntos como sintaxe e funções, bem como consultas as referências do Processing.

Os últimos temas explicados trataram resolução, grade de pixels e estrutura básica de um programa. Esses três pontos são importantes na medida que permitem, respectivamente, entender a janela de exibição, posicionarmeticulosamentefigurassnessajanelaeconfigurar/animarsusprogramas. No próximo capítulo serão introduzidos os alicerces que compõe a programação e como utilizá-los no contexto da computação criativa.

[ CAPÍTULO 2 ]:

# A Arte da Programação

No princípio era o Código

Quando um pintor se sente impelido a expressar suas ideias em uma tela ele utiliza de seu pincel, paleta e instinto artístico. Ao longo dos anos esse profissional aprendeu a usar, naturalmente, dois conceitos muito importantes em sua carreira artística, o de *ferramenta* e o de *processo*. De maneira simplificada, as ferramentas do pintor são o pincel e a paleta. Elas permitem que o artista deposite a tinta com traços longos ou finos, com uma textura leve ou carregada e com cores vibrantes ou brandas que adicionam múltiplas dimensões de riqueza visual. Essas ferramentas são externas ao pintor, mas com o tempo, o treino e a experiência, tornam-se uma extensão dele. O segundo conceito que ele usa é o de processo, constantemente confundido com instinto simplesmente por ser difícil de explicar de uma maneira metódica ou racional. Porém, o processo artístico pode ser decomposto em regras<sup>1</sup> e padrões internos que o pintor segue (mesmo que criadas inconscientemente por ele) para expor suas ideias. Por exemplo, ele sabe que objetos contra a luz projetam uma sombra que está na mesma direção dessa luz. Sabe também quais cores utilizar para expressar um espectro infinito de emoções e motivações (vermelho para paixão e energia ou azul para razão e serenidade)<sup>2</sup>.

Acredite, o programador também utiliza desses mesmos conceitos. Todavia, suas ferramentas são as estruturas de dados, como variáveis e classes, e as estruturas de controle da programação, como funções, condicionais e repetições. O seu processo é o algoritmo e os passos lógicos que ele rationaliza para atingir os seus objetivos. As únicas diferenças entre o programador e o artista são a natureza das ferramentas e o detalhamento do processo. O detalhamento é especialmente importante uma vez que o computador não é capaz de expressar conceitos artísticos abstratos sem que um humano o designe para isso. Nesse ponto, o programador não pode ser puramente intuitivo como o artista, pois ele deve explicar para a máquina, em

---

<sup>1</sup>Nada garante que essas regras estejam no reino da lógica e da razão. As vezes elas se fundem no que origina essencialmente o *estilo* do artista, como as composições de Piet Mondrian ou de Pollock.

<sup>2</sup>Esses são apenas exemplos simplórios. Existem campos da arte e da psicologia inteiramente dedicados ao estudo das cores que, dependendo do contexto, podem apresentar diversos significados.

minúcias, como ele deseja que as ferramentas sejam utilizadas. O primeiro passo, portanto, é conhecer bem as ferramentas que estão disponíveis para ele. Algumas delas você já viu, são elas as funções (`rect()`, `setup()` e `draw()`). Neste capítulo você irá aprender os conceitos básicos da programação, sempre regados a exemplos que visam demonstrar suas aplicações na prática artística.

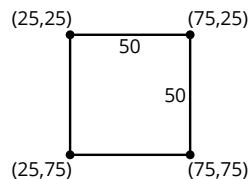
## 2.1 | Variáveis

Vamos começar pelo conceito de variáveis que basicamente são estruturas para armazenamento e consulta de dados, tais como números, letras e símbolos. Elas permitem guardar essas informações e usá-las quando for proveitoso, sendo regularmente aplicadas na parametrização de elementos de código. Por exemplo, suponha que você deseje desenhar um quadrado através de quatro retas em vez de usar diretamente a função `rect()` do Processing. Para facilitar essa ação vamos definir que seu vértice superior esquerdo esteja no ponto (25,25) e que ele possua 50 pixels de lado. Com essas informações você poderia escrever o seguinte código para desenhar um quadrado como o da figura 2.1.

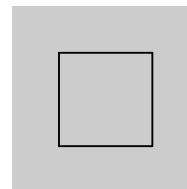
### Código 2.1 Desenhando um quadrado

```
// Desenha um quadrado formado por quatro arestas (linhas):
line(25,25,75,25);
line(75,25,75,75);
line(75,75,25,75);
line(25,75,25,25);
```

Agora imagine que a posição inicial, ou até mesmo do tamanho do lado desse quadrado não ficou como você esperava, e que você gostaria de experimentar outros valores. Para isso você terá de recalcular todos os pontos que compõem essas linhas, além de reescrever os dezesseis números nas chamadas das funções `line()`. Se você quiser testar apenas um ou outro caso você pode até optar por esse caminho, mas e se quiser testar dez, ou vinte, ou cem outras combinações? Isso se torna rapidamente inviável. Felizmente



(a) Coordenadas das linhas.



(b) Janela de saída.

Figura 2.1: Quadrado formado por quatro linhas.

as variáveis nos permitem armazenar e parametrizar valores que estão continuamente mudando e sendo usados no código, como a posição do quadrado ou o tamanho de seu lado.

Uma variável necessita de ser criada ou, no jargão da computação, declarada, que é feito escrevendo o tipo da variável seguido pelo seu nome. A inicialização da variável, que significa definir uma valor para ela, é feita escrevendo o nome da variável seguido por um sinal de igual (=) e o valor que se deseja que ela adquira. Veja os exemplos:

```
// Declaração de uma variável do tipo "float", batizada de "numero":  
float numero;  
// Inicialização arbitrária da variável "numero":  
numero = 10;
```

Também é possível fazer uma declaração de variável que incorpore sua inicialização através da combinação dos dois métodos:

```
// Declara a variável "numero", e a inicializa com o valor 10:  
float numero = 10;
```

Múltiplas variáveis de um mesmo tipo podem ser declaradas desde que separadas por vírgula e, após a última, finalizadas com um ponto e vírgula:

```
// Declaradas três variáveis do tipo float e inicializada apenas uma:  
float variavel_1, variavel_2 = 10, variavel_3;
```

O tipo da variável indica se ela é um número, letra, ou um dado de outra natureza. Você pode consultar os principais tipos de variáveis primitivas do Processing na tabela [2.1](#). No exemplo acima, a palavra `float` firma que a variável é do tipo ponto flutuante, podendo armazenar qualquer valor numérico, inclusive aqueles que possuam casas decimais.

O nome da variável é aquele que será usado sempre que você pretender consultar o valor dela, podendo ser qualquer frase, palavra ou letra que você quiser, desde que siga as seguintes regras:

- Pode começar com subtração (\_), cifrão (\$) ou qualquer letra (A - Z ou a - z), porém não pode começar com números.
- O restante do nome pode conter qualquer uma das condições anteriores e ainda números (0 - 9).
- Não pode conter caracteres acentuados, espaços, operadores ou qualquer outro símbolo ou palavra reservada da linguagem.
- Os nomes são sensíveis a letras maiúsculas e minúsculas, implicando que `var1` é diferente de `Var1` ou de `vAr1`.

Em geral, o nome da variável é uma abreviação ou referência ao propósito dela. Por exemplo, se você quiser guardar a posição `x` de um objeto em uma variável você poderia usar alguns nomes como `posicaoX`, `posicao_X` ou  `posX`. Este último é especialmente interessante, pois nomes curtos e explicativos reduzem o código e tornam sua leitura mais rápida e eficiente.

Nome	Tipo	Exemplo	Descrição
Inteiro	int	int num = 2;	Define um número inteiro, sem casas decimais.
Ponto flutuante	float	float num = 2.1;	Define um número de ponto flutuante, que pode ter casas decimais.
Booleano	boolean	boolean cond = true;	Define uma variável booleana, que pode ser verdadeira ( <i>true</i> ) ou falsa ( <i>false</i> ). É utilizada em operações lógicas, condicionais e repetições.
Caractere	char	char carac = 'b';	Define um caractere Unicode como letras, números e símbolos. Note que um caractere com um número Unicode ainda é um caractere e não pode participar de operações matemáticas no sentido convencional. Um caractere deve ser único (só um) e declarado entre aspas simples.
Palavra	String	String frase = "Olá Mundo!";	Define uma palavra que é, efetivamente, uma sequência linear de caracteres. Diferentemente do caractere, pode possuir uma quantidade arbitrária de letras, números e símbolos. O declarador String deve ser escrito com um "S" maiúsculo e a palavra deve estar entre aspas duplas.

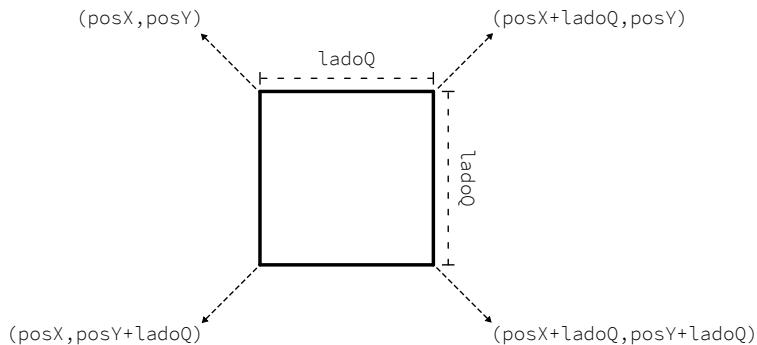
Tabela 2.1: Tipos de variáveis no Processing.

Utilizando o conceito de variáveis você pode reescrever o código 2.1 de maneira parametrizada. Agora as coordenadas do quadrado não são mais números fixos, veja a figura 2.2.

#### Código 2.5 Desenhando um quadrado parametrizado

```
// Declaração e inicialização das variáveis:  
float posX = 25;  
float posY = 25;  
float ladoQ = 50;  
  
line(posX, posY, posX + ladoQ, posY);  
line(posX + ladoQ, posY, posX + ladoQ, posY + ladoQ);  
line(posX + ladoQ, posY + ladoQ, posX, posY + ladoQ);  
line(posX, posY + ladoQ, posX, posY);
```

Desta forma, se você decidir que o lado de 50 pixels do quadrado não é mais apropriado, basta alterar o valor da variável ladoQ que todas as posições das retas serão recalculadas automaticamente, sem a necessidade de reescrever uma a uma.



**Figura 2.2:** Desenhando um quadrado usando variáveis.

### 2.1.1 | Variáveis do sistema

A linguagem Processing conta com variáveis nativas que possuem informações úteis sobre a janela de saída e o programa em execução. Essas variáveis podem ser diretamente acessadas a qualquer momento através de seu nome, sem a necessidade de uma declaração prévia. Algumas delas podem ser consultadas na tabela 2.2. Elas se sobressaem na medida em que você escrever códigos que dependam do seu ambiente de programação. Por exemplo, se você desejar desenhar um ponto centralizado na tela, poderá encontrar suas coordenadas x e y através das operações `width/2` e `height/2` independente do tamanho da janela de exibição.

Nome	Tipo	Descrição
<code>width</code>	<code>int</code>	Armazena o comprimento atual, em pixels, da janela de exibição.
<code>height</code>	<code>int</code>	Armazena a altura atual, em pixels, da janela de exibição.
<code>mouseX</code>	<code>float</code>	Fornece a posição X (horizontal) do mouse na janela de exibição.
<code>mouseY</code>	<code>float</code>	Fornece a posição Y (vertical) do mouse na janela de exibição.
<code>pmouseX</code>	<code>float</code>	Fornece a posição X do mouse na janela de exibição no frame anterior.
<code>pmouseY</code>	<code>float</code>	Fornece a posição Y do mouse na janela de exibição no frame anterior.
<code>mousePressed</code>	<code>boolean</code>	Retorna verdadeiro se o mouse estiver pressionado, ou falso caso contrário.
<code>keyPressed</code>	<code>boolean</code>	Retorna verdadeiro se se alguma tecla estiver pressionada, ou falso caso contrário.
<code>frameCount</code>	<code>int</code>	Armazena o número de frames que foram exibidos desde o início da execução do código. Esta variável é automaticamente incrementada a cada quadro.

**Tabela 2.2:** Variáveis de sistema do Processing.

No tópico de exibição de variáveis, além da maneira gráfica de visualizá-las, como tamanho e posição de uma figura, você pode consultar o conteúdo delas através do console do PDE. Para isto se usa a função `println()` como mostrado no código abaixo:

```
void setup() {
    size(150,150);
}

void draw() {
    // Exibe, no console, o comprimento e altura da janela de saída:
    println(width);
    println(height);
    // Exibe, no console, a posição (x,y) do mouse:
    println("Posição x do mouse:", mouseX);
    println("Posição y do mouse:", mouseY);
}
```

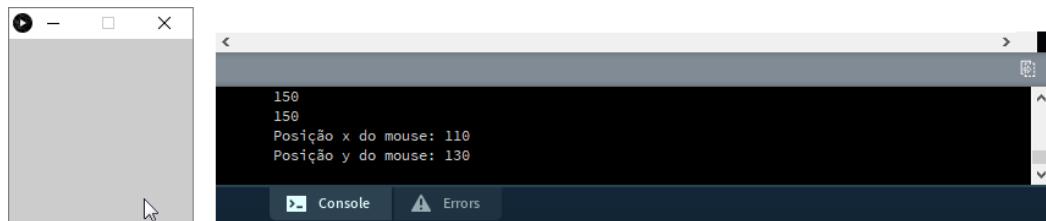


Figura 2.3: Valores da posição do mouse sendo mostrados no console.

## 2.1.2 | Vetores de variáveis

---

A declaração de variáveis uma a uma, como feito nas seções passadas, é uma prática costumeira que satisfaz boa parte dos programas. Contudo, à medida que você adicionar complexidade em seus códigos, você perceberá que existem casos que requerem um número elevado de elementos. Eventualmente manter a estratégia de declará-las individualmente não será mais satisfatório, pois você terá de memorizar muitos nomes de diferentes variáveis, além de digitar consideravelmente mais. A solução para esses problemas está na possibilidade de agrupar variáveis de um mesmo tipo e, em vez de armazenar apenas um valor por estrutura, você pode armazenar diversos se esta for um vetor (ou do inglês, um *array*). A figura 2.4 ilustra esse conceito.

Um vetor de variáveis pode ser de qualquer tipo, como os relatados anteriormente na tabela 2.2, e é declarado da seguinte maneira:

```
tipoDoVetor[] nomeDoVetor = new tipoDoVetor[quantidadeDeValoresArmazenados];
```

Por exemplo:

```
int num1 = 1  
int num2 = 10  
int num3 = 5  
int num4 = 22
```

(a) Variáveis individuais.

```
int[] nums = { 1 | 10 | 5 | 22 }
```

(b) Vetor de variáveis.

Figura 2.4: Armazenamento de dados.

```
// Cria um vetor de variáveis que pode armazenar até 10 valores do tipo "float":  
float[] numeros = new float[10];
```

Você pode perceber que essa estrutura é bem semelhante à declaração de uma variável normal. No lado esquerdo da declaração você só precisa adicionar os colchetes após o tipo, e no lado direito usar a palavra new (indicando que gostaria de criar um novo vetor) e o tipo da variável seguido de colchetes com a quantidade de elementos que se deseja armazenar. A atribuição de valores em um vetor deve ser feito individualmente por posição, sendo esta referenciada dentro dos colchetes:

```
nomeDoVetor[indice] = valorDaVariavel;
```

E para acessá-los:

```
nomeDoVetor[indice];
```

Tais como:

```
// Cria um vetor de variáveis que pode armazenar até 10 valores do tipo "float":  
float[] numeros = new float[10];  
  
// Atribuição do valor de 4.998 para o índice 2 da variável "numeros":  
numeros[2] = 4.998;  
  
// Acesso - Imprime 4.998 no console do PDE:  
println(numeros[2]);
```

A seguir é realizada uma comparação de declarações e atribuições entre o método convencional e utilizando vetores. Também é mostrado que é possível declarar um vetor preenchido com valores, sem a necessidade de usar o qualificador new. Observe que todos os códigos abaixo são equivalentes:

```
// Método convencional:  
float num1 = 1;  
float num2 = 10;
```

```
float num3 = 5;
float num4 = 22;

// Vetores - Declaração separada da atribuição:
float[] num = new float[4];
num[0] = 1;
num[1] = 10;
num[2] = 5;
num[3] = 22;

// Vetores - Declaração já com a atribuição:
float[] nums = {1,10,5,22};
```

Um ponto que deve ser evidenciado é que o índice do valor a ser acessado sempre começa do zero. Logo, se você declarar um vetor de 4 posições, os índices válidos serão 0, 1, 2 e 3, sempre indo de zero até o tamanho total do vetor subtraído de um. Acessar um índice que não seja válido (menor que zero, fracionário ou acima do tamanho total) irá causar uma falha durante a execução do seu programa:

```
// Cria um vetor de quatro posições e inicializa apenas a primeira delas:
float[] nums = new float[4];
nums[0] = 1;

// Erro - Tentar acessar um índice maior que o tamanho do vetor:
println(nums[4]);
```

Por que você deve fazer o uso de vetores? O principal motivo é o fato de eles serem uma ótima forma de organizar e simplificar o seu código. Vamos a um caso concreto, imagine que você possua 50 figuras, todas com posições x e y na tela, e que você queira armazenar essas posições. Se você for declarar todas essas variáveis, somente para a posição x, terá:

```
float posX01, posX02, posX03, posX04, posX05, posX06, posX07, posX08, posX09, posX10;
float posX11, posX12, posX13, posX14, posX15, posX16, posX17, posX18, posX19, posX20;
float posX21, posX22, posX23, posX24, posX25, posX26, posX27, posX28, posX29, posX30;
float posX31, posX32, posX33, posX34, posX35, posX36, posX37, posX38, posX39, posX40;
float posX41, posX42, posX43, posX44, posX45, posX46, posX47, posX48, posX49, posX50;
```

Agora, para as mesmas 50 figuras, mas incluindo ambas as posições, você pode declarar as variáveis usando vetores:

```
float[] posX = new float[50];
float[] posY = new float[50];
```

É bem clara a vantagem de se utilizar vetores. Conforme você verá em diversos exemplos deste livro, eles são empregados estrategicamente para reduzir e estruturar o código. Antes de finalizar este tópico você deve saber que é possível consultar o número total de variáveis que um vetor previamente declarado pode

armazenar. Isso é feito escrevendo o nome do vetor, seguido de um ponto e a palavra `length`. Veja essa propriedade no exemplo abaixo:

```
float[] posX = new float[50];
float[] posY = new float[50];
// Exibe o tamanho do vetor (50) no console do PDE:
println("O tamanho de posX é:", posX.length);
```

### 2.1.3 | Cores

Cores são elementos primordiais quando se almeja a elaborar mensagens visuais. Suas combinações podem causar sentimentos e emoções, e sua ausência pode ser capaz de transmitir mensagens através de contrastes acentuados. No Processing as cores são igualmente significativas e possuem um tipo próprio de variável chamada de `color`.

Cabe lembrar que computadores lidam apenas com números e não sabem o que é a cor vermelha ou azul, então como eles interpretam informações sobre elas? Eles trabalham com o que é especificado de canais de cores que são estruturas dedicadas ao armazenamento de uma imagem, definindo se ela é ou não colorida e se ela possui ou não transparência. O Processing permite dividir imagens em até quatro canais:

- **1 Canal:** Imagens formadas pelo preto, branco e tons de cinza.
- **2 Canais:** Imagens formadas pelo preto, branco, tons de cinza e com suporte para transparência.
- **3 Canais:** Imagens coloridas.
- **4 Canais:** Imagens coloridas e com suporte para transparência.

Um computador constrói uma imagem digital colorida básica como três grandes matrizes de cores, em que cada uma é uma cópia da imagem original, mas somente com valores referentes a uma das cores primárias: Vermelho, Verde ou Azul (Espaço RGB - *Red, Green, Blue*)<sup>3</sup>. Quando os canais são unidos ocorre a sobreposição das cores, revelando uma imagem colorida, veja a figura 2.5. Uma imagem em tons de cinza possui os mesmos valores RGB nos três canais e, portanto, um canal é o bastante para representá-la. A variável `color` do Processing armazena as cores com uma filosofia muito similar ao que acabamos de discutir. Ela pode ser declarada da seguinte maneira:

---

<sup>3</sup>Existem outros espaços de cores muito populares, como o HSV (*Hue, Saturation, Value*) ou o CMYK (*Cyan, Magenta, Yellow, Black*) sendo que este é representado por quatro canais e usa um sistema de cores subtrativo.

```
// Declaração de uma variável "color", RGBA, genérica:  
r = 100; // r = Canal R: Vermelho;  
g = 220; // g = Canal G: Verde;  
b = 220; // b = Canal B: Azul;  
a = 255; // a = Canal A: Transparência.  
color cor = color(r,g,b,a);
```

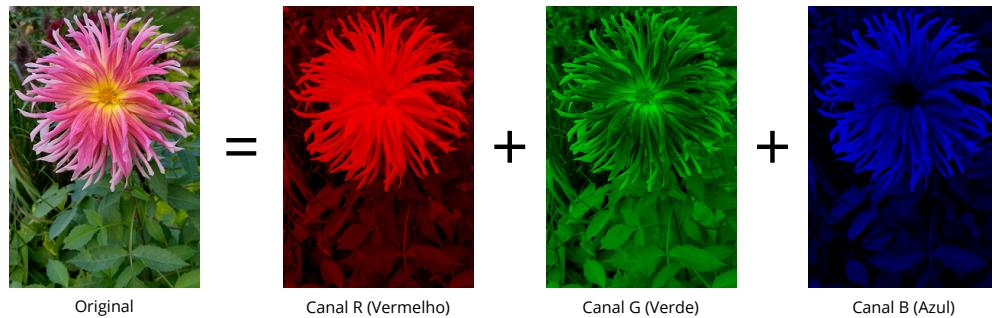


Figura 2.5: Foto decomposta nos canais RGB.

Cada um dos argumentos da variável `color` emula um canal de cor (vermelho, verde e azul) cujo valor é um número inteiro que varia entre 0 e 255<sup>4</sup>. O número 0 indica a completa ausência da cor enquanto o número 255 indica uma total presença dela, podendo ser vistos na figura 2.6. Todas as outras cores são formadas através da combinação dessas três cores primárias que, no Processing, é um sistema aditivo de cores, ilustrado na figura 2.7a. Uma cor RGB com valores `color(255,0,0)` indica o vermelho puro, e `color(255,0,255)` é a combinação do vermelho com azul, formando o violeta. As cores monocromáticas seguem essas mesmas regras, mas você pode especificar a cor em apenas um canal. Por exemplo, o preto é dado por `color(0)`, o branco por `color(255)` e o cinza por qualquer valor entre eles, como `color(127)`. Você pode usar a ferramenta seletora de cores do próprio Processing para identificar qual combinação de RGB leva a sua cor desejada. Ela pode ser acessada através do menu *Ferramentas* e depois *Seletor de Cór...*, e sua janela é mostrada na figura 2.8.

<sup>4</sup>O modo padrão do Processing considera uma profundidade de cores de 24 bits chamada de *Truecolor*. Isso significa que existem  $2^{24} = 16777216$  cores distintas.



Figura 2.6: Representação das cores de acordo com seus valores.

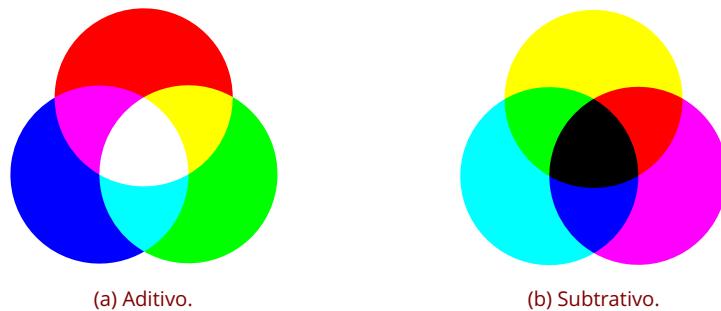


Figura 2.7: Sistema de cores.

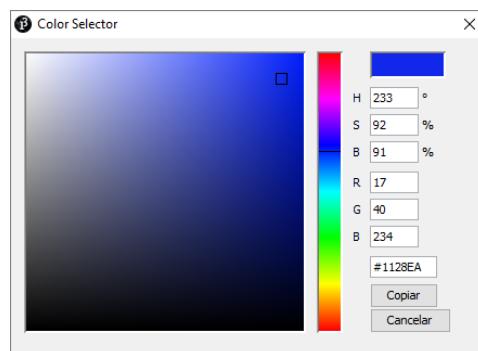


Figura 2.8: Ferramenta *Color Picker* do Processing.

O canal relativo à transparência, muitas vezes chamado de alfa, funciona de maneira invertida. O número 0 indica uma cor completamente transparente (invisível) e 255 indica uma cor totalmente opaca (ou sem transparência). Você pode acompanhar alguns exemplos de como usar a variável do tipo color no código abaixo:

```
void setup() {  
    size(325,175);  
  
    // Preenche o fundo da janela de branco:  
    background(255);  
  
    // Declara a cor vermelha como 255 apenas no canal R:  
    color vermelho = color(255,0,0);  
    // Preenche a figura com a cor vermelha:  
    fill(vermelho);  
    rect(25,50,75,75);  
  
    // Declara a cor azul como 255 apenas no canal B:  
    color azul = color(0,0,255);  
    fill(azul);  
    rect(125,50,75,75);  
  
    // Usa diretamente a cor violeta como 255 no canal R e no canal B:  
    fill(255,0,255);  
    rect(225,50,75,75);  
}
```



Figura 2.9: Figuras coloridas usando a variável color.

Ao incluir um valor de alfa, as figuras desenhadas serão em parte transparentes, permitindo que ocorra a combinação das cores durante a sobreposição de figuras, veja 2.10:

```
void setup() {  
    size(175,175);  
    background(255);  
    // Remove a borda das figuras:  
    noStroke();  
  
    // Vermelho 50% transparente:
```

```

fill(255,0,0,127);
rect(25,25,75,75);

// Verde 50% transparente:
fill(0,255,0,127);
rect(50,50,75,75);

// Violeta 50% transparente:
fill(255,0,255,127);
rect(75,75,75,75);
}

```

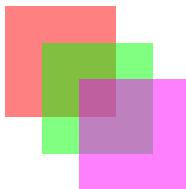


Figura 2.10: Figuras transparentes usando a variável `color`.

## 2.2 | Operadores

---

O próximo tópico a ser discutido são as operações e operadores, ambos utilizados frequentemente na programação. Conforme você viu anteriormente, às vezes é preciso escrever uma variável em função de outras. Em 2.5 você definiu parte do vértice da figura como uma soma, (`posX + ladoQ`), usando o operador da adição, um dos muitos que existem. A maioria dos deles você já conhece do ensino médio e outros você usa no seu dia a dia sem perceber que são formalmente operadores. Eles podem ser divididos em quatro categorias intuitivas.

### 2.2.1 | Aritméticos

---

Operadores aritméticos são os operadores básicos da matemática que você aprende na escola, como soma, subtração, multiplicação e divisão. Você pode ver uma lista completa na tabela 2.3.

Ao escrever operações com números ou variáveis é possível usar parênteses para controlar a precedência de todos os tipos de avaliações. Veja alguns exemplos:

Nome	Operador	Exemplo	Resultado
Adição	+	num = 1+2;	num será 3.
Subtração	-	num = 10-3;	num será 7.
Multiplicação	*	num = 4*5;	num será 20.
Divisão	/	num = 16/8;	num será 2.
Módulo	%	num = 8%3;	num será o resto da divisão inteira de 8 por 3, ou seja, 2.
Incremento	++	num++;	num terá seu valor aumentado em 1.
Decremento	--	num--;	num terá seu valor diminuído em 1.
Negação	-	num = -num;	num terá seu valor multiplicado por -1.

Tabela 2.3: Operadores aritméticos no Processing.

```

int num1 = 10, num2 = 5, num3 = 3;

int result1 = (num1 + num2) * num3;
println(result1); // Será igual a 15*3, ou 45.

int result2 = (num1 * num3) / (num2 * num3);
println(result2); // Será igual a 30/15, ou 2.

num1++;
println(num1); // Exibirá 11, que é o resultado de 10 + 1.

num2--;
println(num2); // Exibirá 4, que é o resultado de 5 - 1.

println(-num3); // Exibirá -3, que é o resultado de -1*3.

```

## 2.2.2 | Atribuição

O segundo tipo, operadores de atribuição, são aqueles em que também são realizadas operações matemáticas, mas o sujeito da operação é a própria variável em que foi chamado o operador. Sendo assim, a atribuição do resultado é imediata. Estes operadores estão listados na tabela 2.4.

Nome	Operador	Exemplo	Resultado
Atribuição	=	num = 1;	num terá seu valor alterado para 1.
Atribuição por Adição	+=	num += 5;	num terá seu valor aumentado em 5.
Atribuição por Subtração	-=	num -= 5;	num terá seu valor subtraído em 5.
Atribuição por Multiplicação	*=	num *= 5;	num terá seu valor multiplicado por 5.
Atribuição por Divisão	/=	num /= 5;	num terá seu valor dividido por 5.

Tabela 2.4: Operadores de atribuição no Processing.

Sempre que se faz uma operação de atribuição é importante ter em mente que o valor da variável irá mudar, afetando qualquer operação futura. Veja o exemplo abaixo:

```
int num1 = 2;  
println(num1 += 3); // num1 = (2 + 3) = 5  
println(num1 *= 4); // num1 = (5 * 4) = 20  
println(num1 /= 10); // num1 = (20/10) = 2  
println(num1 -= 2); // num1 = (2 - 2) = 0
```

## 2.2.3 | Relação

Os operadores de relação são os responsáveis por comparar variáveis. Ao utilizá-los, você poderá dizer se uma variável é maior, menor, igual ou diferente de outra. O resultado de uma operação de relação será sempre verdadeiro (*true*) ou falso (*false*), ou seja, uma variável do tipo booleana. A lista completa está na tabela 2.5.

Nome	Operador	Exemplo	Resultado
Maior	>	println(4 > 5);	False. O número 4 não é maior que o número 5.
Menor	<	println(4 < 5);	True. O número 4 é menor que o número 5.
Igual a	==	println(4 == 4);	True. O número 4 é igual ao número 4.
Diferente de	!=	println(4 != 4);	False. O número 4 não é diferente do número 4.
Maior ou Igual	>=	println(4 >= 4);	True. O número 4 não é maior que 4, mas é igual a 4 então a comparação é verdadeira.
Menor ou Igual	<=	println(5 <= 4);	False. O número 5 não é menor que 4 nem igual a 4, portanto a comparação é falsa.

Tabela 2.5: Operadores de relação no Processing.

## 2.2.4 | Lógicos

Os últimos, mas não menos importantes, são chamados de operadores lógicos, que realizam operações sobre variáveis booleanas. Na prática eles serão utilizados para concatenar diversas operações relacionais. Isso será muito bem vindo nos condicionais e estruturas de repetição conforme você verá mais adiante. Estes operadores podem ser consultados na tabela 2.6.

Um exemplo contendo múltiplas comparações:

```
boolean comp = ((4 > 5) || (5 <= 6)) && !(10 < 20);
```

Nome	Operador	Exemplo	Resultado
E (AND)	&&	<code>println((4&lt;5) &amp;&amp; (5&gt;1));</code>	True. O operador E retorna verdadeiro se, e somente se, todas as condições forem simultaneamente verdadeiras.
OU (OR)		<code>println((4&lt;5)    (5&gt;6));</code>	True. O operador OU retorna verdadeiro se pelo menos uma de todas as condições forem verdadeiras.
NÃO (NOT)	!	<code>println(!(4&gt;5));</code>	True. O operador NÃO inverte o resultado booleano. Como 4 não é maior que 5, o resultado é falso, porém o operador NÃO inverte esse resultado tornando-o verdadeiro.

Tabela 2.6: Operadores lógicos no Processing.

```
// Por partes:
// (4 > 5) -> False
// (5 <= 6) -> True
// (False || True) -> True
// (10 < 20) -> True, mas temos uma negação, logo !(10 < 20) -> False
// Portanto:
// (True && False) -> False
println(comp);
```

## 2.2.5 | Operadores e cores

Um pequeno alerta sobre operadores e a variável `color`. Se você quiser aplicar operadores de qualquer natureza sobre esse tipo de variável é importante que você realize as operações individualmente por canal e posteriormente os transforme em argumentos de uma variável `color`. Algo do tipo:

```
int r = 2*30;
int g = 100;
int b = 255/5;
color cor = color(r,g,b);
```

Realizar as operações diretamente sobre uma variável do tipo `color` não é uma violação na programação, mas não faz sentido do ponto de vista da matemática uma vez que ela é composta por outras variáveis (até 4 canais). Além disso, você deve ter em mente que os valores mínimos e máximos por canal são, respectivamente, 0 e 255. Qualquer número fora desse intervalo será reajustado para dentro do mesmo no ato da definição da variável.

## 2.2.6 | Funções matemáticas

---

Os operadores são a base de uma quantidade infinita de algoritmos e operações matemáticas e serão usados em quase todo código que você escrever. No entanto eles não cobrem diretamente algumas funções matemáticas úteis que são recorrentes em algoritmos. O Processing possui as mais comuns programadas e cabe a você apenas chamá-las. Resta a observação que elas são funções e não operadores, mas foram colocados nesta seção devido à abordagem matemática da mesma. Parte delas podem ser vistas na tabela 2.7 e as demais consultadas na referência.

Função	Descrição
round(num)	Arredonda um número do tipo float, que possui casas decimais, para um número inteiro. Se a parte decimal for menor que 0.5, o número é arredondado para baixo, e se for maior ou igual a 0.5, para cima.
ceil(num)	Arredonda o número entre parênteses para cima independente de sua parte decimal.
floor(num)	Arredonda o número entre parênteses para baixo independente de sua parte decimal.
max(num1,num2)	Retorna o maior número entre os dois fornecidos.
min(num1,num2)	Retorna o menor número entre os dois fornecidos.
pow(num1,num2)	Realiza a exponenciação de num1 por num2.
sqrt(num1)	Calcula a raiz quadrada do número fornecido.

Tabela 2.7: Funções matemáticas implementadas no Processing.

Existe ainda uma função capaz de converter o valor de uma variável que possua sua escala entre um mínimo e máximo original, para um novo mínimo e máximo. Sua chamada é feita da seguinte maneira:

```
float resultado = map(valor,originalMin,originalMax,novoMin,novoMax);
```

A função `map()` é particularmente útil, pois ela permite mapear, ou transformar, um valor em outro, veja a figura 2.11 para uma explicação visual. Em uma aplicação concreta, suponha que você queira que, ao movimentar o mouse horizontalmente pela janela de saída, a sua figura mude da cor preta para a vermelha. Neste caso você quer que uma cor seja o resultado de um mapeamento da posição do mouse. A posição horizontal do mouse (variável `mouseX`) possui valores entre zero e o tamanho da janela (variável `width`). Simultaneamente, o canal referente a cor vermelha, está contido entre 0 e 255, indicando a completa ausência do vermelho (restando o preto) até o vermelho puro. Sabendo dos máximos e mínimos da escala da variável mapeada (`mouseX`) e da variável de saída (canal vermelho), é possível usar a função `map()` para causar uma transição de cores na sua figura. Veja o código a seguir:

```
void draw() {  
    background(255);  
    // Posição do mouse (mouseX) vai de 0 ao tamanho da janela (width).  
    // Cor vermelha vai de 0 (preto) a 255 (vermelho).  
    float r = map(mouseX,width,0,255);
```

```

color cor = color(r,0,0);
fill(cor);
ellipse(50,50,50,50);
}

```

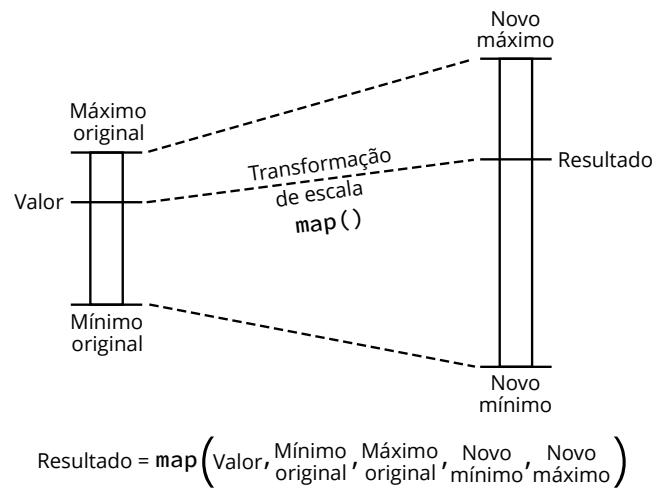


Figura 2.11: Representação visual da função `map()`.

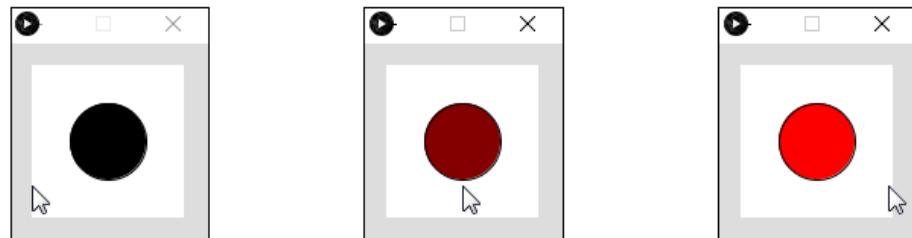


Figura 2.12: Resultado do mapeamento do mouse para uma cor.

## 2.3 | Funções

---

O conceito de função foi explicado superficialmente na seção 1.4 e agora será abordada a parte técnica sobre como você pode declarar e usar suas próprias funções. Um função, *script* ou sub-rotina pode ser definida como um conjunto de instruções agrupadas em um bloco, executadas sempre que você referenciá-la pelo seu nome. Suas vantagens incluem organização e redução de linhas de código, assim como reutilização em programas futuros. Vamos justificar e apreciar o poder das funções através de um estudo de caso. Você usou as linhas de código 2.5, repetidas aqui por conveniência, para desenhar um único quadrado:

```
float posX = 25;
float posY = 25;
float ladoQ = 50;

line(posX, posY, posX + ladoQ, posY);
line(posX + ladoQ, posY, posX + ladoQ, posY + ladoQ);
line(posX + ladoQ, posY + ladoQ, posX, posY + ladoQ);
line(posX, posY + ladoQ, posX, posY);
```

E se você quisesse desenhar um número maior, na ordem de dezenas? Seria necessário repetir essas mesmas linhas dezenas de vezes, tornando seu código denso, e sem agregar nenhum valor. Conforme estudado o quadrado é formado por quatro linhas desenhadas de maneira parametrizada caso você possua o ponto de um vértice desse quadrado e o tamanho de seu lado. Seria muito conveniente se você pudesse fornecer apenas esses três dados, posição x, posição y e lado, e o computador desenhasse essa figura para você. As funções servem exatamente para isso, para executar uma série de instruções que são repetidas ao longo do código. Assim como uma variável, uma função precisa ser declarada antes que possa ser usada e isso deverá ser feito fora do `setup()`, `draw()` ou de qualquer outra função. A declaração de uma função genérica é feita seguindo o modelo abaixo:

### Código 2.27 Estrutura de uma função genérica

```
tipoDaVariavelDeRetorno nomeDaFuncao(tipoVar1 var1, tipoVar2 var2, ..., tipoVarN varN) {
    // Corpo da função;
    return variavelDeRetorno;
}
```

Elas podem ser usadas no código ao escrevermos o nome da rotina seguido de parênteses contendo todos os argumentos de entrada da mesma:

```
nomeDaFuncao(arg1, arg2, ..., argN);
```

A estrutura completa de uma função pode ser intimidadora em um primeiro momento, mas ela sempre segue essa mesma forma. Uma vez que você aprender sobre cada um dos elementos que a compõe, você

estará pronto para criar as suas próprias funções. Abaixo você pode consultar, com mais detalhes, cada um deles:

- **Tipo da variável de retorno:** Se uma função realiza tarefas e devolve uma variável qualquer, o tipo de retorno deverá igual ao tipo dessa variável. Por exemplo, se devolver um número inteiro, o tipo de retorno será `int`, ponto flutuante, ele será `float`, e assim por diante de acordo com as variáveis que você estudou (tabela 2.1). A função `map()` é um exemplo cujo tipo de retorno é `float`, já que ela retorna um valor transformado que é o resultado de um cálculo. Por outro lado, se a função não retornar nada, o tipo dela será `void`. Isso é comum em funções que só exibem algo na tela ou no console, como `rect()` e `println()`.
- **Nome:** É o nome que você usará para chamar a função no programa. Utiliza-se das mesmas regras de nomeação de variáveis.
- **Parâmetros:** Muitas vezes as instruções executadas pelas funções dependem de valores que são externos a elas. No ato da declaração da função, as variáveis que fazem a interface entre a parte interna dela e o código principal do programa são chamadas de parâmetros. No ato de chamada da função no código o nome do termo muda e o correto é falar que passamos argumentos para a função. Alguns padrões de escrita de código sugerem que os nomes dos parâmetros sejam precedidos pelo subtraço (`_`), mas isso não é obrigatório.
- **Corpo:** É o código que compõe a função em si. Ele pode ser composto por uma quantidade arbitrária de linhas de código contendo funções pré declaradas (suas e do Processing), variáveis, condicionais, repetições e demais estruturas de programação. Em geral, tudo que você pode usar fora de uma função você pode usar dentro dela, com uma diferença importante, variáveis declaradas dentro de uma função são perdidas após a execução da mesma.
- **Retorno:** O campo `return` permite que a função devolva uma variável ao programa principal. Ele deve estar na última linha do corpo de uma função e é obrigatório apenas se a mesma retornar alguma variável. Neste caso, a variável de retorno deverá ser colocada na frente da palavra `return` no corpo da função e seguido de um ponto e vírgula. Funções precedidas da palavra `void` não precisam ter esse campo visto que não retornam nenhuma variável.

Você pode usar essas informações para construir qualquer função personalizada. Um bom exercício para fixar esse conhecimento é migrar um código que escrevemos previamente, como o 2.5, referente ao desenho de um quadrado. O fluxo ideal para projetar uma função é analisar as ações que você quer que ela realize e listá-las em tópicos para um entendimento mais claro dos objetivos. Veja o exemplo:

#### Projeto de uma função para desenhar um quadrado:

- **A função deve ser autoexplicativa:** A função deve ter um nome esclarecedor, que remete ao seu propósito, tal como `quadrado()`.
- **A função apenas exibirá um quadrado:** O desenho da figura será feito através de quatro instruções `line()`. Ela não irá retornar nenhum valor ou variável e, portanto, é precedida pelo tipo `void` e não possuirá o campo `return`.

- **Sua posição é determinada por um vértice:** O desenho do quadrado será a partir do seu canto superior esquerdo, com coordenadas x e y informadas pelo usuário, implicando em dois argumentos de entrada da função.
- **Seus lados não serão fixos:** O tamanho dos lados do quadrado também será definido pelo usuário. Como um quadrado possui todos os lados com o mesmo tamanho, será necessário informar apenas um valor, sendo este o terceiro argumento de entrada da função.

Com isso definimos o *nome* da sub-rotina, suas *funcionalidades* (*corpo*) assim como *tipo* e o *retorno* e, por fim, os seus *argumentos de entrada* (ou parâmetros na declaração). Uma vez que todos os pré-requisitos foram satisfeitos, você pode escrevê-la usando modelo apresentado anteriormente. Para chamá-la no programa, basta que você utilize o nome dela e passe a posição e tamanho do lado do quadrado que deseja desenhar:

```
void quadrado(float _posX, float _posY, float _lado) {
    // Desenha um quadrado composto por quatro arestas:
    line(_posX, _posY, _posX + _lado, _posY);
    line(_posX + _lado, _posY, _posX + _lado, _posY + _lado);
    line(_posX + _lado, _posY + _lado, _posX, _posY + _lado);
    line(_posX, _posY + _lado, _posX, _posY);
}

void setup() {
    // Chamada da função:
    quadrado(25,25,50);
}
```

Finalmente, é pertinente citar que você é livre para declarar duas ou mais funções com o mesmo nome<sup>5</sup>, desde que o número dos argumentos de entrada delas sejam diferentes entre si. O código abaixo demonstra que não ocorrem erros ao executar o programa:

```
void quadrado(float _posX, float _posY, float _lado) {
    // Desenha um quadrado composto por quatro arestas:
    line(_posX, _posY, _posX + _lado, _posY);
    line(_posX + _lado, _posY, _posX + _lado, _posY + _lado);
    line(_posX + _lado, _posY + _lado, _posX, _posY + _lado);
    line(_posX, _posY + _lado, _posX, _posY);
}
```

<sup>5</sup>Isso é chamado de sobrecarga ou *overload* de métodos na programação.

```

void quadrado(float _posX, float _ posY) {
    // Exibe as variáveis passadas para a função no console:
    println("Variável 1:", _posX, "Variável 2 Y", _ posY);
}

void quadrado() {
    // Desenha um quadrado de lado e posição fixa:
    rect(0,0,50,50);
}

void setup() {
    quadrado(25,25,50);
    quadrado(25,25);
    quadrado();
}

```

### 2.3.1 | Interrupções

---

O Processing possui uma grande base de funções que podem ser usadas durante a execução de seus programas. A maioria delas deve ser chamada explicitamente e sequencialmente, significando que elas seguem a ordem de execução em que elas foram escritas no código. Em contrapartida, existe uma outra classe de funções, nomeadas de interrupções, que são invocadas assincronamente sempre que um evento específico ocorrer, disparando sua execução. Elas estão usualmente relacionadas a acontecimentos externos ao programa, como intervenções através do clique do mouse ou do pressionar de uma tecla. As interrupções são habituais quando se espera algum tipo de interação com o usuário do programa. No Processing, elas se comportam como funções pré-declaradas, mas não executam nenhuma ação. Você pode customizá-las e adicionar instruções que você quer que elas realizem através do incremento do corpo da mesma, assim como você faz com `setup()` ou `draw()`. Elas podem ser consultadas na tabela 2.8.

Função	Descrição
<code>mousePressed()</code>	Executa a interrupção sempre que algum botão do mouse for pressionado.
<code>mouseReleased()</code>	Executa a interrupção sempre que algum botão do mouse for solto.
<code>mouseWheel()</code>	Executa a interrupção sempre que a roda do mouse ( <i>scroll</i> ) for girada.
<code>mouseDragged()</code>	Executa a interrupção sempre que o mouse for arrastado e algum botão dele estiver pressionado.
<code>keyPressed()</code>	Executa a interrupção sempre que alguma tecla for pressionada.
<code>keyReleased()</code>	Executa a interrupção sempre que alguma tecla for solta.

Tabela 2.8: Interrupções nativas do Processing.

O exemplo abaixo mostra como usar interrupções para desenhar figuras na tela sempre que o mouse for arrastado, o resultado é mostrado na figura 2.13.

```

void setup() {
    size(400,200);
    background(255);
}

void draw() {}

// Interrupção - Mouse pressionado e arrastado:
void mouseDragged() {
    float raio = dist(mouseX,mouseY,pmouseX,pmouseY);
    ellipse(mouseX,mouseY,raio,raio);
}

```

A linha de código que contém a função `dist()` serve para calcular a distância euclidiana entre dois pontos. Ao usarmos a posição do mouse atual e a do frame passado como argumentos de entrada temos como resultado uma medida do quanto rápido o mouse foi movimentado entre um frame e outro. Se a velocidade de arraste é elevada, então são desenhados círculos grandes, se não, círculos pequenos. O único ponto que merece maior destaque no código é o fato de ter sido incluída a função `draw()` vazia. Isso se faz necessário para que o programa entre em modo de animação, caso contrário seria gerado apenas um quadro, congelando qualquer interação com o usuário.

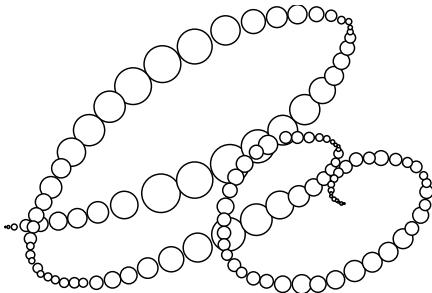


Figura 2.13: Desenho na janela ao arrastar o mouse.

## 2.4 | Condicionais

---

A próxima ferramenta que você irá aprender são os condicionais. Eles são definidos como estruturas de controle de fluxo de um programa que verificam se uma determinada condição é verdadeira ou falsa, e executam linhas de código baseadas nesse julgamento. Sua forma geral é dada por:

```
se (condição) {  
    // Se a condição for verdadeira, execute estas linhas de código.  
}  
caso contrário {  
    // Se a condição for falsa, execute estas linhas de código.  
}
```

No Processing temos o par de palavras chave `if/else`

```
if (condicao) {  
    // Se “condicao” for verdadeiro (true), execute estas linhas de código.  
}  
else {  
    // Se “condicao” for falso (false), execute estas linhas de código  
}
```

O campo da condição neste tipo de estrutura deve ser, obrigatoriamente, verdadeiro ou falso. Nada impede que ele seja uma variável ou uma série de operações sobre variáveis contanto que, na avaliação final, o campo tenha um resultado do tipo booleano. Você já estudou sobre esse tipo de variável nas seções passadas, então vamos brevemente relembrar algumas condições que são e não válidas para esse campo:

- `true` ou `false`: Válido, essas são variáveis booleanas que indicam verdadeiro ou falso.
- `(2 + 3)`: Não válido, o resultado de uma soma é um número da mesma natureza, por exemplo `float` ou `int`.
- `(2 + 3) > 1`: Válido, o resultado da comparação `5 > 1` é verdadeiro (`true`) que é uma variável booleana.
- `(2 + 3) > 7`: Válido, o resultado da comparação `5 > 7` é falso (`false`) que é uma variável booleana.

Essa estrutura é bem vinda quando se deseja alterar o comportamento do programa baseado em condições atuais da execução. Considere o caso em que você queira preencher de preto a respectiva metade da janela de exibição de acordo com a posição do seu cursor. Esse cenário é ideal para o uso de uma estrutura `if/else` já que é necessária uma condição (mouse do lado direito ou esquerdo da tela) para realizar uma ação (preencher a metade de preto), veja a figura 2.14. Utilizando condicionais você poderia escrever o código abaixo:

```
void setup() {  
    size(300,150);  
}
```

```

void draw() {
    background(210);
    fill(0);

    // Se o mouse estiver na parte esquerda da janela ela é preenchida com um retângulo
    // preto. Caso contrário, o mouse estará na parte direita, e ela é preenchida.
    if(mouseX < width/2) {
        rect(0,0,width/2,height);
    }
    else {
        rect(width/2,0,width/2,height);
    }
}

```

A condição `if(mouseX < width/2)` verifica se a posição horizontal do mouse está na metade esquerda da janela. Veja a figura 2.15 como exemplo.



Figura 2.14: Posição do mouse e preenchimento da janela de exibição.

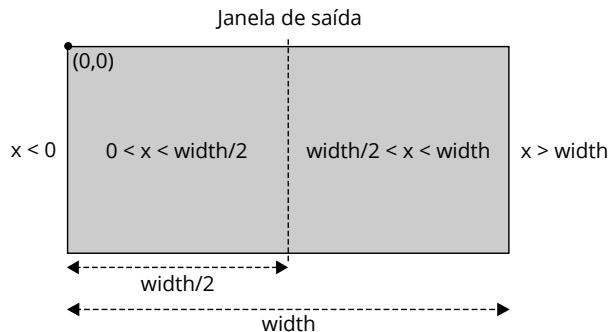


Figura 2.15: Divisão da janela de saída e coordenadas do eixo x.

Sempre que você quiser verificar uma condição na direção horizontal da janela (direito/esquerdo) você precisará testar apenas o eixo coordenado na direção x, como foi o caso. Se você quisesse verificar uma condição na direção vertical (cima/baixo), bastaria fazer os testes com o eixo y. Focando na estrutura

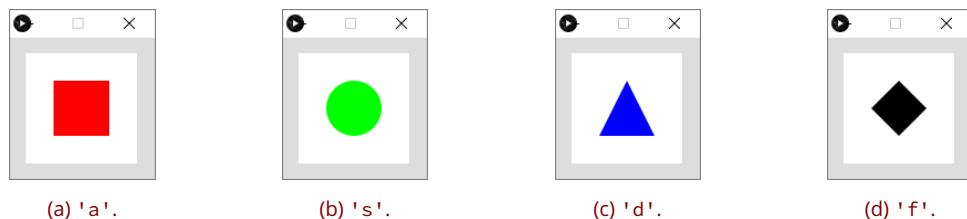
do condicional, note que só é necessário testar se o cursor está no lado esquerdo da janela pois, caso contrário (`else`), ele estará do lado direito. Cabe evidenciar que o computador só avaliará as condições do seu programa dentro dos limites dele, isto é, posições do cursor fora da janela de exibição serão ignoradas. Neste caso, o valor de `mouseX` permanecerá congelado na sua última posição conhecida.

### 2.4.1 | Múltiplos condicionais

Na seção passada você escreveu um código que testava apenas duas condições, atendidas completamente por uma única estrutura `if/else`. De maneira equivalente, é possível fazer múltiplos testes inserindo mais condicionais dentro de estruturas desse próprio tipo. Isto é chamado de aninhamento de condicionais ou *nested conditionals*. Considere o código em que foi criada uma interrupção que testa qual letra foi pressionada pelo usuário (variável do sistema `key`) e desenha uma figura baseada nela. Pressionar a tecla '`a`' desenha um quadrado, '`s`' um círculo, '`d`' um triângulo, '`f`' um losango e qualquer outra tecla limpa a tela, veja a figura 2.16.

**Código 2.35 Aninhamento de condicionais completo**

```
void draw() {  
}  
  
void keyPressed() {  
    background(255);  
    noStroke();  
  
    if(key == 'a') {  
        fill(255,0,0);  
        rect(25,25,50,50);  
    }  
    else {  
        if(key == 's') {  
            fill(0,255,0);  
            circle(25,25,50);  
        }  
        else {  
            if(key == 'd') {  
                fill(0,0,255);  
                triangle(25,25,40,40,40,10);  
            }  
            else {  
                if(key == 'f') {  
                    fill(0,0,0);  
                    diamond(25,25,50,50);  
                }  
            }  
        }  
    }  
}
```



**Figura 2.16:** Comportamento do programa baseado nas teclas pressionadas.

```

        ellipse(50,50,50,50);
    }
    else {
        if(key == 'd') {
            fill(0,0,255);
            triangle(50,25,25,75,75,75);
        }
        else {
            if(key == 'f') {
                fill(0,0,0);
                quad(50,25,25,50,50,50,75,75,50);
            }
        }
    }
}

```

Note que dentro da cada estrutura `else` é criada uma outra `if/else` para continuar as comparações da tecla pressionada com uma das letras específicas para o desenho. Esse código também pode ser visto em forma de fluxograma como mostrado na figura 2.17.

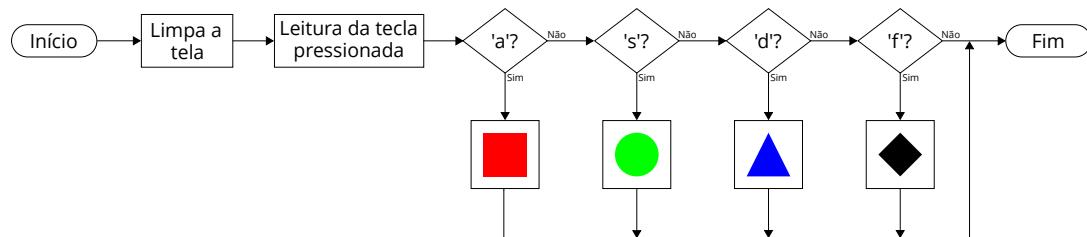


Figura 2.17: Fluxograma referente ao código 2.35.

Infelizmente o código 2.35 sofre de um problema típico de aninhamentos, a perda da legibilidade. Quanto mais comparações forem feitas, mais confusa ficará a escrita do código, principalmente se as instruções a serem executadas forem compostas por várias linhas. É possível contornar esse inconveniente reescrevendo o código de forma a separar completamente as comparações. Considere o exemplo:

#### Código 2.36 Aninhamento de condicionais separados

```

void draw() {
}

void keyPressed() {
    background(255);
}

```

```

noStroke();

if(key == 'a') {
    fill(255,0,0);
    rect(25,25,50,50);
}

if(key == 's') {
    fill(0,255,0);
    ellipse(50,50,50,50);
}

if(key == 'd') {
    fill(0,0,255);
    triangle(50,25,25,75,75,75);
}

if(key == 'f') {
    fill(0,0,0);
    quad(50,25,25,50,50,75,75,50);
}

```

É bem mais agradável analisar o código acima, mas veja que ele é acometido por um outro contratempo. Imagine que você pressione a tecla 'a', que é o primeiro teste de ambos os códigos, [2.35](#) e [2.36](#). Em [2.35](#) o teste será realizado e dado como verdadeiro, desta forma nenhum dos outros condicionais serão processados, pois eles se encontram dentro da estrutura `else`. Já em [2.36](#) a primeira condição será dada como verdadeira e será desenhado um quadrado, no entanto os outros condicionais não estão contidos dentro de um `else` e, por isso, também serão testados, ilustrado no fluxograma [2.18](#). Neste caso, enquanto o código [2.35](#) realizou apenas um teste, o código [2.36](#) realizou quatro, sendo bem menos eficiente do ponto de vista computacional. Claro que cabe ao programador balancear esses pontos, pois algumas vezes esse impacto será tão insignificante que é mais vantajoso manter um código limpo do que implementar micro otimizações de baixo ganho.

Uma alternativa para quem deseja manter a eficiência e uma interface limpa é utilizar a estrutura `switch` cuja forma é mostrada abaixo:

```

switch(variavel) {

    case teste1:
        // Código a ser executado se a comparação: (variavel == teste1) for verdadeira.
        break;

    //...
}

```

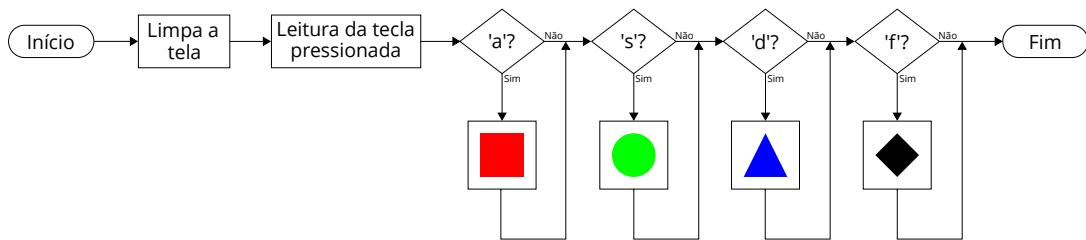


Figura 2.18: Fluxograma referente ao código 2.36.

```

case testeN:
    // Código a ser executado se a comparação: (variavel == testeN) for verdadeira.
    break;
}

```

Os códigos 2.35 e 2.36 convertidos para esse novo modelo seriam dados por:

```

void draw() {
}

void keyPressed() {
    background(255);
    noStroke();

    switch(key) {

        case 'a':
            fill(255,0,0);
            rect(25,25,50,50);
            break;

        case 's':
            fill(0,255,0);
            ellipse(50,50,50,50);
            break;

        case 'd':
            fill(0,0,255);
            triangle(50,25,25,75,75,75);
            break;

        case 'f':
            fill(0,0,0);
            break;
    }
}

```

```
        quad(50,25,25,50,50,75,75,50);  
        break;  
  
    }  
}
```

Uma grande limitação do elemento `switch` é que seus testes requerem as chamadas *expressões constantes*, ou seja, apenas comparações de igualdade (`==`) são válidas. Testes que usem de outros operadores (`>`, `<`, `>=`, `<=`, `!=`) ou cujas variáveis de teste não sejam `int`, `String`, `char` ou `enum` são considerados como erros durante a compilação.

Esta subseção, mais do que apresentar sobre outras formas de escrever condicionais, foi importante do ponto de vista de desenvolvimento de código. Ela foi desdobrada com o intuito de mostrar a você como pequenas escolhas na escrita de um bloco de instruções podem impactar antes mesmo da execução do programa, em sua legibilidade e, ao mesmo tempo, em sua performance durante a execução, relativo ao tempo de processamento. Apesar dessas informações serem mais voltadas para a parte técnica da programação, você deve ficar atento a esses tipos de detalhes, pois podem ocorrer não só em condicionais, mas em qualquer outro tipo de estrutura de código.

## 2.5 | Repetições

---

O último tópico que compõe a formação básica de um programador são as repetições ou *loops*. Basicamente esse tipo de estrutura permite que parte de seu código seja repetido um número arbitrário de vezes em uma fração das linhas que seriam necessárias. Até agora, se você quisesse desenhar um número elevado de figuras, digamos mil, seria necessário repetir mil vezes as linhas para desenhar essas figuras. Imagine quanto tempo você demoraria para chamar mil funções `rect()` e depois percebesse que esse número era excessivo e que, na verdade, você gostaria de 500 retângulos. Você iria ficar no mínimo frustrado por ter escrito o dobro do que realmente precisava, além de ter perdido um tempo precioso. As repetições foram criadas para evitar essa e outras situações semelhantes e ainda permitir um engrandecimento de seu programa por possibilitar criar mais conteúdo com menos linhas de código. Estruturas de repetição têm a seguinte forma:

```
enquanto (condição) {  
    // Código a ser repetido.  
}
```

que no Processing se reflete com o uso do `while`

```
while(condicao) {  
    // Código a ser repetido.  
}
```

O campo de condição do `while` funciona de maneira muito semelhante ao da estrutura `if/else`, de tal forma que se a condição dentro dos parênteses for verdadeira (`true`) todo código dentro das chaves será executado. A diferença é que *enquanto* essa condição for verdadeira, o código continuará a ser repetido. É necessário tomar um pouco de cuidado com qual condição se coloca nessa estrutura, pois caso contrário pode ser que a execução do seu código fique presa para sempre em uma repetição. Ela deve ser projetada para executar apenas um número finito de iterações. É usual criar uma variável de controle que é verificada no condicional do `while` e incrementada no corpo da repetição. Por exemplo, se você quisesse desenhar 50 quadrados centralizados na tela, como na figura 2.19, poderia escrever o seguinte código:

#### Código 2.41 Figuras concêntricas

```
void setup() {  
    size(300,300);  
    background(255);  
    noFill();  
  
    float x = 0, y = 0, lado = 0;  
  
    int contador = 0;  
  
    while(contador < 50) {  
        lado = lado + 5;  
        x = width/2 - lado/2;  
        y = height/2 - lado/2;  
  
        rect(x,y,lado,lado);  
  
        contador++;  
    }  
}
```

Vamos analisar com mais detalhes alguns pontos desse código:

```
float x = 0, y = 0, lado = 0;
```

As variáveis de posicionamento e tamanho do quadrado são criadas e inicializadas. A estratégia é fazer o quadrado começar com um tamanho de lado desprezível e aumentá-lo gradativamente.

```
int contador = 0;
```

Neste ponto é definida a variável que irá armazenar quantas vezes a repetição foi executada. É ela quem faz o controle do *loop* e provocará uma condição de parada da estrutura de repetição que, caso contrário, continuaria para sempre.

```
while(contador < 50)
```

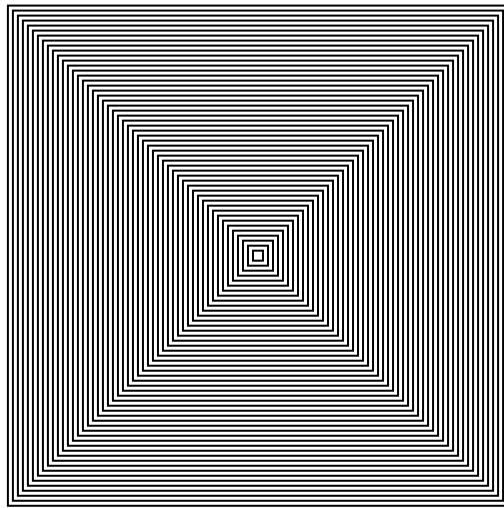


Figura 2.19: Desenho de quadrados usando repetições.

Aqui é declarado o início de uma repetição e sua condição de parada. Note que o código interno (dentro das chaves do `while`) será executado enquanto a condição (`contador < 50`) for verdadeira.

```
lado = lado + 5;  
x = width/2 - lado/2;  
y = height/2 - lado/2;
```

A primeira tarefa que é feita dentro do *loop* é aumentar o tamanho do lado do quadrado. Assim, cada vez que uma iteração da repetição for executada, o quadrado se torna maior criando um padrão formado por figuras concêntricas. Logo em seguida são calculadas as novas posições `x` e `y` que serão passadas para a função `rect()`. É necessário atualizar essas posições uma vez que essa função usa o canto superior esquerdo para desenhar a figura. A centralização é feita com base no centro da janela de exibição, ponto `(width/2,height/2)`, subtraída da metade do lado do quadrado. Essa é a mesma filosofia aplicada na seção 1.5. É de suma importância que essas atualizações sejam feitas dentro da estrutura de repetição para que o quadrado cresça a cada iteração.

```
rect(x,y,lado,lado);
```

Uma vez que as coordenadas foram atualizadas, a figura é desenhada.

```
contador++;
```

Nesta linha a variável contador é incrementada em um. Isso significa que após a primeira repetição o valor de contador será 1, após a segunda ele será 2 e assim sucessivamente até atingir 50. Quando isso acontecer, o condicional da estrutura de repetição se tornará falso e o código interno ao while deixará de ser executado.

Veja o poder incrível dessa nova ferramenta. Em vez de escrever mais de 70 linhas de código, você conseguiu reduzir para 22. E se você quisesse agora mais ou menos quadrados? Seria necessário alterar apenas a condição da repetição de (`contador < 50`) para outra como (`contador < 5000`), sem adicionar nenhuma linha de código a mais. A grande vantagem dessa estrutura é que você pode focar na construção de apenas um elemento e depois repeti-lo para formar um padrão complexo. Na figura 2.20 foi criada uma única linha e depois repetida ao longo da janela de exibição.

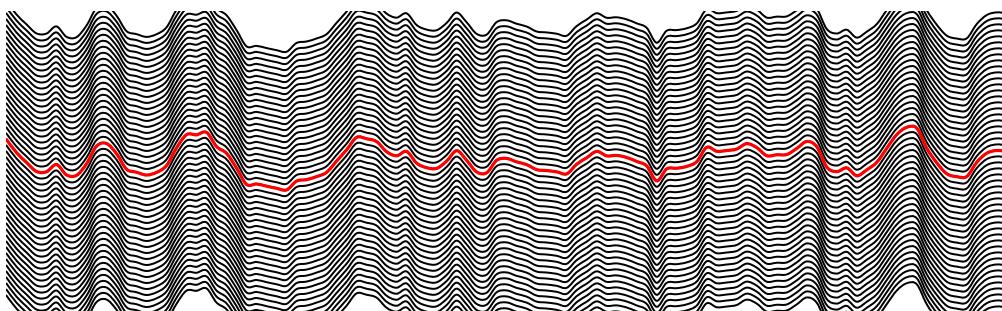


Figura 2.20: Múltiplas repetições da curva em vermelho.

### 2.5.1 | Repetições - `for`

A estrutura `while` é conveniente quando você está aprendendo repetições, necessita de mais flexibilidade ou quando opta por conjunções mais elaboradas. Ela é acometida pelo inconvenientes da necessidade de declarar variáveis controle fora do `loop` e manualmente incrementá-las. Uma estrutura correspondente, compacta e mais popular é a `for`. Veja sua forma abaixo:

```
for(variavel; condicao; incremento) {  
    // Código a ser repetido.  
}
```

Esta estrutura possuí em uma única linha, respectivamente, a declaração e inicialização da variável de controle, a condição de parada da repetição e o incremento a cada iteração. A seguir é mostrada uma comparação entre essas duas formas de repetição:

```
int contador = 0;
while(contador < 50) {
    // Código a ser repetido.
    contador++;
}
```

O for equivalente é dado por:

```
for(int contador = 0; contador < 50; contador++) {
    // Código a ser repetido.
}
```

A grande vantagem dessa notação é que você pode se dedicar diretamente ao código a ser repetido e não precisa alterar a variável de controle (contador) manualmente em cada iteração. Neste livro é dado foco a estrutura for justamente por esses fatores.

## 2.5.2 | Múltiplas repetições

---

As estruturas de repetição são ideais para economizar linhas de códigos ou transcorrer um grande volume de dados, mas na arte computacional elas também são usadas para criar ilusões e padrões. Para exemplificar esse conceito vamos criar um padrão de Truchet, cuja base é formada pelo revestimento quadriculado de um espaço. Imagine a janela de saída como uma tela dividida em blocos e considere, a princípio, apenas uma linha dessa janela, ilustrada na figura 2.21.



Figura 2.21: Janela de saída como uma fileira de blocos.

A quantidade de blocos necessários para preencher toda a janela horizontalmente pode ser calculado utilizando a informação do tamanho dela e do bloco, que é arbitrariamente definido:

```
void setup() {
    size(600,200);

    // Tamanho do bloco, em pixels, da divisão da janela de saída:
    float lado = 50;
    // Número de blocos necessários para preencher a janela horizontalmente:
    float numBlocosHoriz = width/lado;

    println(numBlocosHoriz);
}
```

Dependendo do tamanho do bloco que você escolher, como por exemplo 70 pixels, os cálculos acima podem indicar que você precisa de um número fracionário de figuras (8.57 blocos) para preencher a tela. Isso não é um impedimento, mas pode deixar o padrão irregular, com aparência de cortado. Essa situação é evitada se você escolher um tamanho de bloco que seja um divisor de resto zero do comprimento e altura da janela de saída. Agora que você possui o número e tamanho dos blocos é preciso descobrir como desenhá-los lado a lado, de maneira sequencial. Esta será uma operação padronizada com diversos elementos, sendo uma boa candidata para se utilizar de repetições. Considere como os blocos se ajustam um ao lado do outro, figura 2.22.

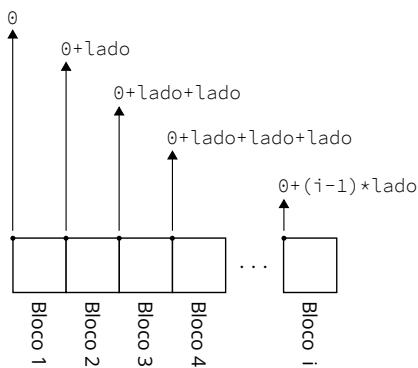


Figura 2.22: Coordenadas dos blocos que dividem a tela.

Os blocos serão desenhados pela função nativa do Processing, `rect()`, com o intuito de simplificar o código. É preciso lembrar que essa função recebe a posição do canto superior esquerdo de cada retângulo na sequência, assim como seu tamanho. Pela figura 2.22 você pode perceber que a posição do primeiro bloco é a origem, ou seja zero. A do segundo é igual a posição horizontal do primeiro, somada do lado do primeiro bloco, `lado`. A do terceiro é igual a posição inicial do segundo somado de `lado`, totalizando  $2 \times \text{lado}$  a partir do ponto inicial. Se você continuar com esse raciocínio para obter a posição de todos os outros blocos perceberá que cada posição horizontal será igual ao número do bloco atual subtraído de um<sup>6</sup>, que multiplica o tamanho do lado do bloco, variável `lado`. Sendo assim, poderia escrever o código abaixo para desenhar uma linha de quadrados:

---

<sup>6</sup>É possível evitar essa subtração se começarmos com um bloco de número zero em vez de um.

```

void setup() {
    size(600,200);
    background(255);

    // Tamanho do bloco da divisão da janela de saída - Em pixels:
    float lado = 20;

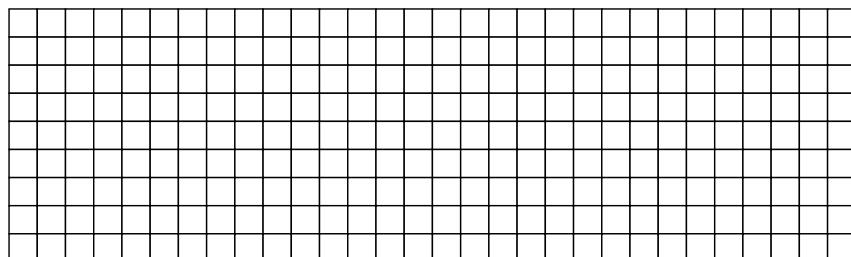
    // Calcula o número de blocos na horizontal:
    float numBlocosH = width/lado;

    for(int i = 0; i < numBlocosH; i++) {
        rect(i*lado,height/2 - lado/2,lado,lado);
    }
}

```

Neste exemplo, a posição vertical dos blocos é uma constante apenas para centralizá-los na janela de exibição e permitir uma visualização clara do que está ocorrendo. Quanto à estrutura `for`, veja que a parametrização da posição horizontal dos blocos foi feita pela variável `i`, contida no intervalo de zero até o número total de blocos para preencher a tela. Você pode ter estranhado o uso da variável `i`, sem um significado concreto, como índice da repetição, mas isso é uma prática habitual para compactação do código. A variável `i` é apenas um contador e seu nome é insignificante no escopo do programa. Execute o código e veja como é gerada uma linha de blocos que preenche toda extensão horizontal da tela.

Você pode utilizar uma abordagem similar para preencher toda extensão vertical da tela. A única diferença é que desta vez a posição vertical de cada um dos blocos é atualizada em vez de permanecer uma constante. Os padrões de Truchet são ainda mais elaborados, formados pela divisão total da tela em blocos. Esta é uma tarefa repetitiva na qual se preenche completamente uma linha horizontal, em seguida toma-se um passo na vertical e novamente preenche-se uma linha na horizontal. Esse ritmo é mantido até que se termine de estampar a tela em sua totalidade, veja a figura 2.23.



**Figura 2.23: Preenchimento horizontal e vertical da janela.**

O preenchimento sequencial em linhas e colunas requer um tipo especial de *loop*, em que uma estrutura de repetição está contida dentro de outra. Essa estratégia é chamada de aninhamento de repetições, ou

*nested loops*, e é evidenciada no código abaixo:

#### Código 2.53 Janela quadriculada

```
void setup() {  
    size(600,180);  
    background(255);  
  
    // Tamanho do bloco da divisão da janela de saída - Em pixels:  
    float lado = 20;  
  
    // Calcula o número de blocos na horizontal:  
    float numBlocosH = width/lado;  
    // Calcula o número de blocos na vertical:  
    float numBlocosV = height/lado;  
  
    for(int i = 0; i < numBlocosH; i++) {  
        for(int j = 0; j < numBlocosV; j++) {  
            rect(i*lado,j*lado,lado,lado);  
        }  
    }  
}
```

Em estruturas aninhadas, as repetições internas serão reiniciadas sempre que houver um passo na repetição externa. Por exemplo, no código 2.53, o for mais interno desenhará uma quantidade de retângulos igual ao valor da variável numBlocosV. Neste momento será dado um “passo” na estrutura do for externo, alterando *i* de zero para um e causando a repetição das linhas internas dele, novamente a estrutura for de índice *j*. O ciclo termina quando *i* atingir numBlocosH e *j* atingir numBlocosV, o que terá feito com que o Processing desenhasse um número de retângulos igual a multiplicação<sup>7</sup> dos limites das repetições, um total de (numBlocosH\*numBlocosV).

Finalizado o preenchimento completo da tela com quadrados, você deve estar se perguntando o *porquê* de tudo isso. Realmente, até agora não há nada de interessante do ponto de vista artístico, mas a divisão da tela é importante em uma série de padrões visuais. Por exemplo, e se em vez de desenhar um bloco, você desenhasse apenas uma das diagonais dele? Você pode implementar isso substituindo a função rect() pela função line(). Após alterar o código sua janela será dividida igual a figura 2.24.

#### Código 2.54 Preenchimento da tela com diagonais

<sup>7</sup>Visto que o incremento de ambas as variáveis de controle, *i* e *j*, é igual a um.

```

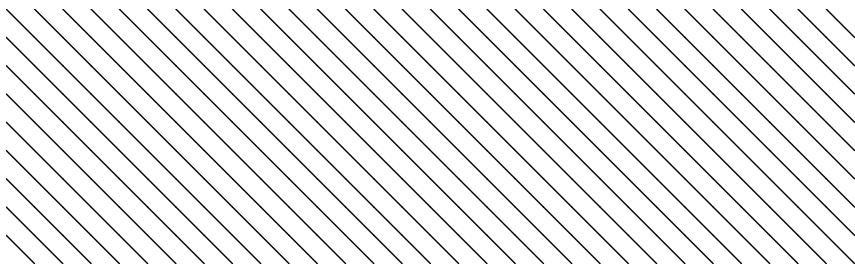
void setup() {
    size(600,180);
    background(255);

    // Tamanho do bloco da divisão da janela de saída - Em pixels:
    float lado = 30;

    // Calcula o número de blocos na horizontal:
    float numBlocosH = width/lado;
    // Calcula o número de blocos na vertical:
    float numBlocosV = height/lado;

    for(int i = 0; i < numBlocosH; i++) {
        for(int j = 0; j < numBlocosV; j++) {
            line(i*lado,j*lado,i*lado + lado,j*lado + lado);
        }
    }
}

```



**Figura 2.24: Padrões de linhas em vez de blocos.**

É possível ainda adicionar condicionais para criar padrões visuais variados. Você pode alternar entre desenhar uma ou outra diagonal de cada bloco dependendo se os contadores das repetições forem pares ou ímpares. Substitua a linha:

```
line(i*lado,j*lado,i*lado + lado,j*lado + lado);
```

por este condicional:

```

if(i%2 == 0) { //Se i é par, desenha uma diagonal:
    line(i*lado,j*lado,i*lado + lado,j*lado + lado);
}
else { //Se i é ímpar, desenha a outra diagonal:
    line(i*lado + lado,j*lado,i*lado,j*lado + lado);
}

```

e você terá imagens como a figura 2.25. No código acima o operador módulo (%) foi empregado pois ele retorna o resto da divisão de um número pelo seu dividendo. O resto de qualquer número par dividido por dois será zero e de qualquer número ímpar será um. Sendo assim, este operador permite identificar as duas classes de números (par ou ímpar) que podem ser usadas no condicional para alternar o padrão de desenho.

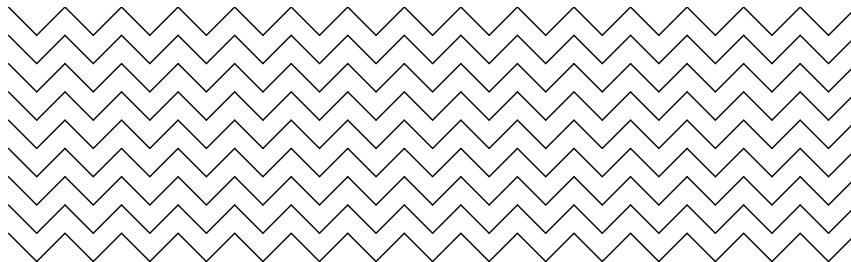


Figura 2.25: Padrões de linhas alternadas.

Experimente trocar os testes do condicional e verá que muitos outros padrões surgem das disposições variadas das diagonais. Altere a condição:

```
if(i%2 == 0)
```

por outras do tipo:

```
if(i*j%2 == 0)          // ou
if(i*j%3 == 0)          // ou
if((i+j)%3 == 0)        // ou
if(((i%2)+(j%3))%2 == 0)
```

e obterá resultados como os da figura 2.26.

Esse gênero de padrões foi inspirado no trabalho do fraude dominicano Sébastien Truchet, que desenvolveu um método<sup>8</sup> capaz de ordenar gravuras simples em disposições complexas. Os chamados padrões de Truchet costumam ser usados em uma série de aplicações estéticas e, em sua forma primordial, são o resultado de uma análise combinatória de quatro matrizes compostas por um triângulo retângulo inscrito em um quadrado, figura 2.27.

<sup>8</sup>Douat, D. (1722). *Méthode pour faire une infinité de dessins différens avec des carreaux mi-partis de deux couleurs par une ligne diagonale*. Chez Florentin de Laulne.

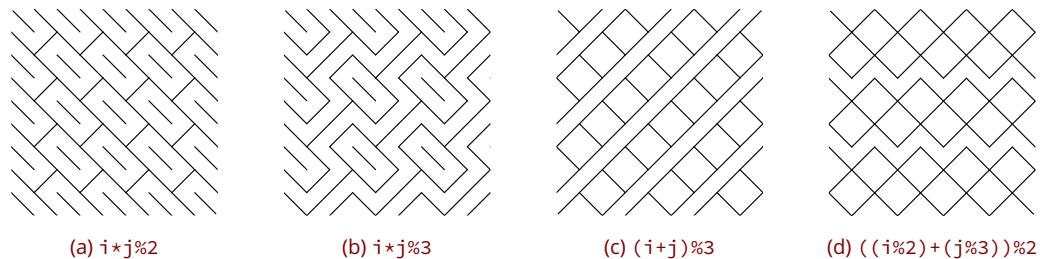


Figura 2.26: Padrões variados formados pelas diagonais dos blocos.



Figura 2.27: Matrizes de Truchet.

Dispostos sequencialmente, lado a lado, elas assumem uma forma unitária, sintetizando imagens como a figura 2.28.

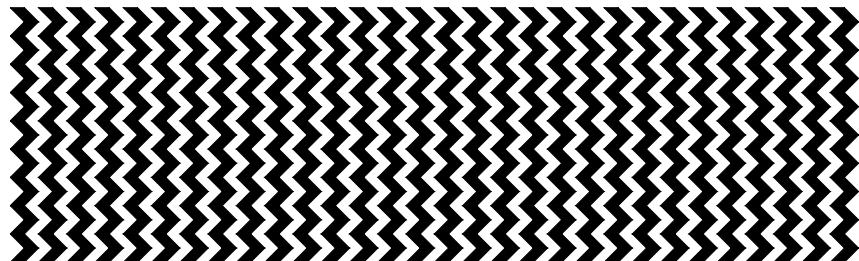


Figura 2.28: Padrões formados por triângulos.

Para criar esse efeito em seu código, localize as linhas que desenham as diagonais em 2.54:

```
line(i*lado,j*lado,i*lado + lado,j*lado + lado);
```

e altere por estas, que desenham triângulos:

```
fill(0);
if(i%2 == 0 && j%2 == 0) {
    triangle(i*lado,j*lado,i*lado + lado,j*lado,i*lado + lado,j*lado + lado);
```

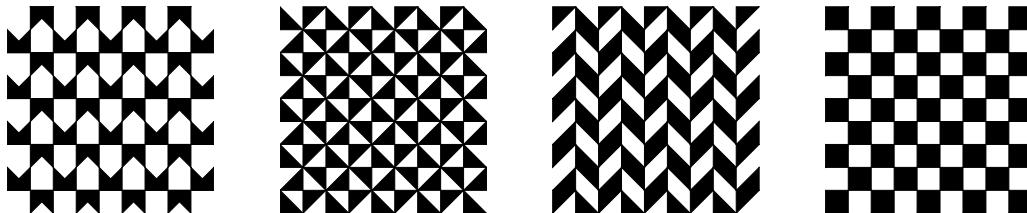


Figura 2.29: Padrões de Truchet.

```
}

if(i%2 != 0 && j%2 == 0) {
    triangle(i*lado,j*lado,i*lado,j*lado + lado,i*lado + lado,j*lado + lado);
}
if(i%2 == 0 && j%2 != 0) {
    triangle(i*lado + lado,j*lado,i*lado + lado,j*lado + lado,i*lado, j*lado + lado);
}
if(i%2 != 0 && j%2 != 0) {
    triangle(i*lado + lado,j*lado,i*lado,j*lado,i*lado,j*lado + lado);
}
```

Usando esse mesmo código é possível criar dezenas de padrões apenas modificando os condicionais, veja a figura 2.29. Ao longo dos anos o método de Truchet vem sendo usado como guia para experimentação com regras similares, usualmente formadas por matrizes na qual uma figura não radialmente simétrica é inscrita em quadrados, triângulos ou hexágonos, veja 2.30.

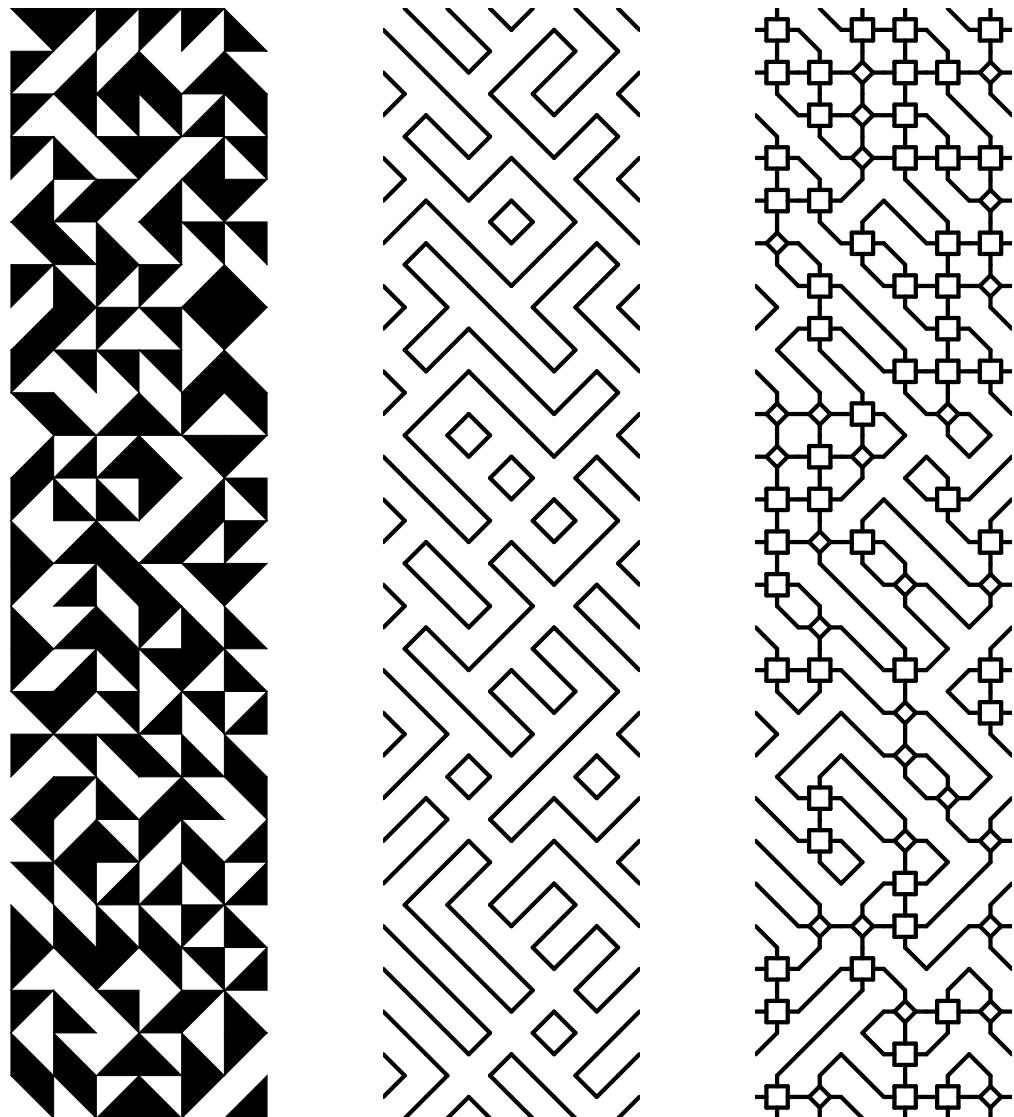


Figura 2.30: Padrões de Truchet.

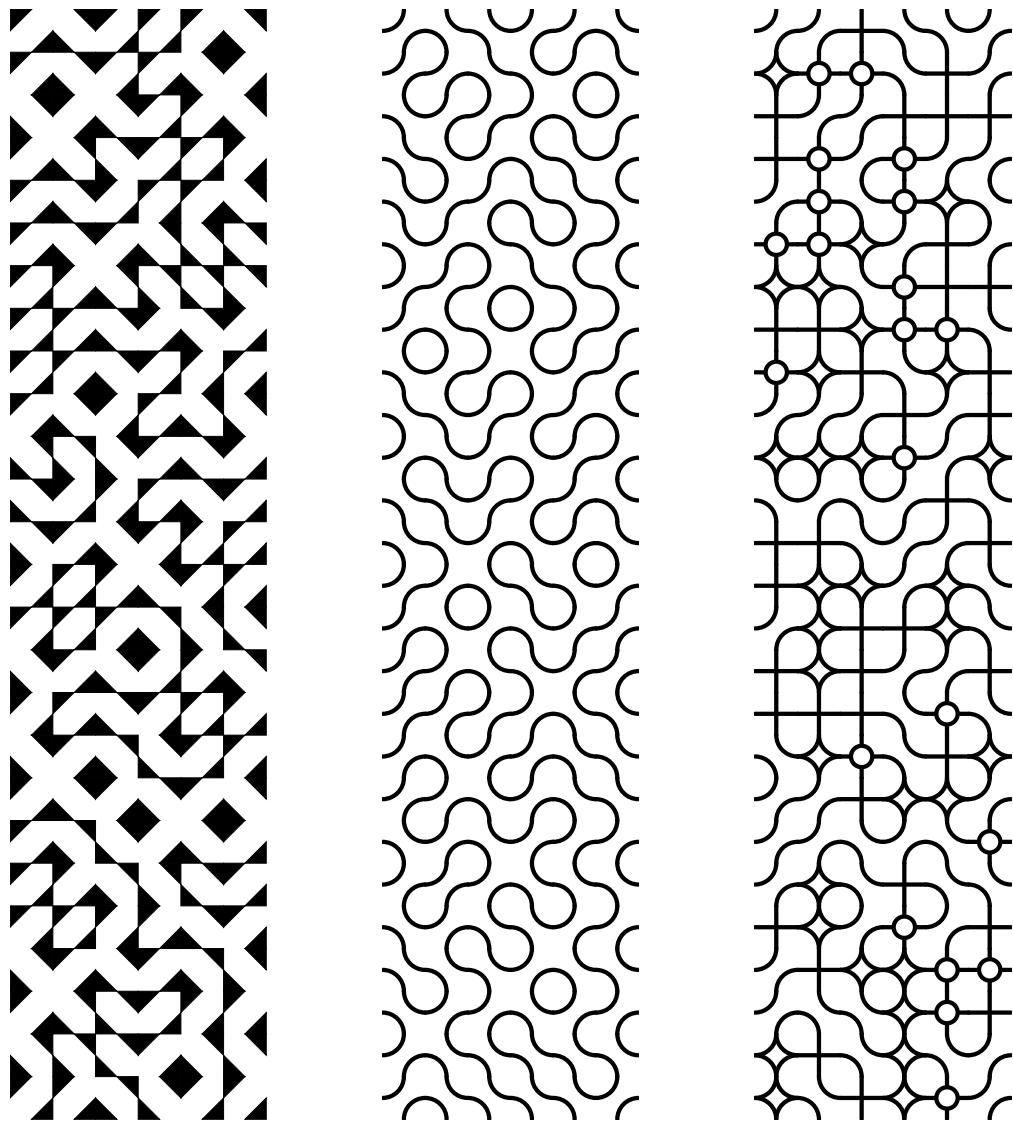


Figura 2.31: Padrões de Truchet.

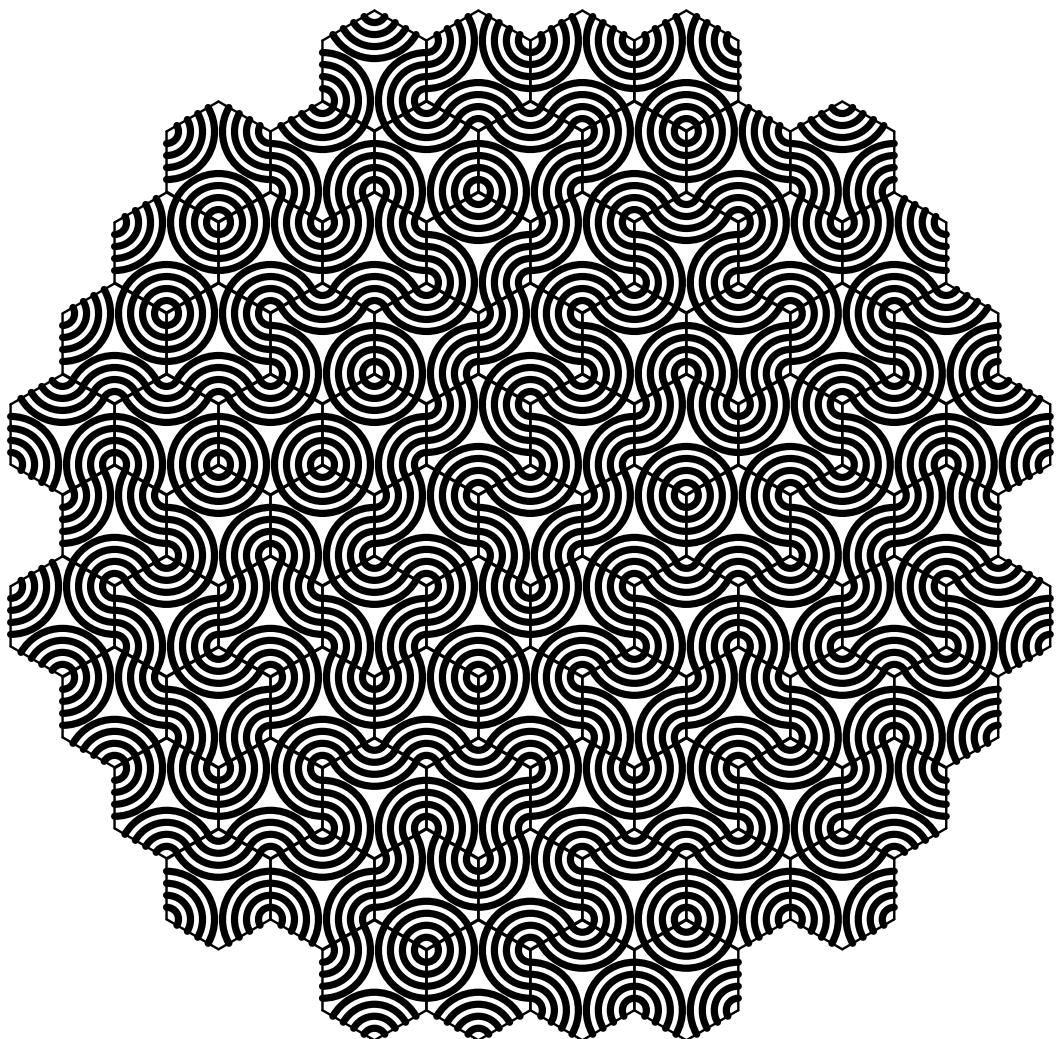


Figura 2.32: Padrões de Truchet em uma grade hexagonal.

## 2.6 | Tópicos avançados - classes e objetos

---

A seção final deste capítulo é dedicada à apresentação de um tópico extremamente popular na programação moderna, a Programação Orientada a Objetos (do inglês, *Object Oriented Programming* ou OOP). Na terceira parte deste livro existe um capítulo destinado exclusivamente ao uso deste tipo de abordagem, com um exemplo construído passo a passo. Se desejar, você pode postergar esta seção até lá, mas eu aconselho fortemente que você a siga e comece a se familiarizar com esta incrível metodologia de pensamento.

A Programação Orientada a Objetos é uma maneira de conduzir o design de softwares. Ela visa agrupar dados e código em uma abstração única, cujo objetivo é modelar um evento, um objeto ou ser, em fim, um elemento que faça parte do seu dia a dia. Muito mais do que uma série de regras, a OOP é uma abstração de conceitos e uma maneira diferente de abordar problemas, escrever códigos e conceber programas. No ponto central da OOP reside a classe. Você pode imaginar uma classe como sendo uma estrutura de dados altamente customizável que contém suas próprias variáveis e funções, também chamadas, respectivamente, de atributos e métodos. Isso facilita o seu entendimento uma vez que você estudou esses dois tópicos separadamente.

Você deve estar se perguntando o porquê das classes serem tão revolucionárias, então vamos para um cenário concreto. Suponha que você precisasse desenvolver um programa para modelar um animal de estimação, um cachorro por exemplo, qual tipo de variável usaria? Um cão seria uma `String`, identificado pelo seu nome? Ou um `int` que guardaria um registro no sistema? E se em um segundo momento você também quisesse que esse cão realizasse ações, como correr? Seus comportamentos seriam descritos por uma série de instruções e não variáveis. Onde você iria declarar a função `correr()`, no corpo do programa? De fato isto não é o ideal, pois você estaria misturando dados relativos ao objeto (cachorro) com a execução do programa principal que pode ser composto por muitos outros objetos e funções. As classes surgem para sistematizar, e de certa forma padronizar, essas questões levantadas.

Uma cão possui uma lista de características além do nome ou registro, como altura, peso, cor dos olhos e raça dentre outros, que formam os seus atributos. Ele também pode realizar uma série de ações próprias como correr, latir ou brincar, que seriam seus métodos. A classe é altamente versátil, pois ela permite agrupar essas variáveis e funções em uma única estrutura de dados. Um ponto importante é que a classe é uma espécie de “molde” geral, por exemplo, o animal cachorro, enquanto um membro específico da classe é chamado de objeto (ou instância da classe). Uma cachorrinha chamada Mia, com 14cm de altura, 5kg de peso, cor de olhos preto e raça Pinscher seria uma instância da classe geral cachorro. A comparação é mostrada na figura 2.33.

No Processing, uma classe é definida da seguinte maneira:

**Código 2.61 Estrutura padrão de uma classe**

```
class NomeDaClasse {
```

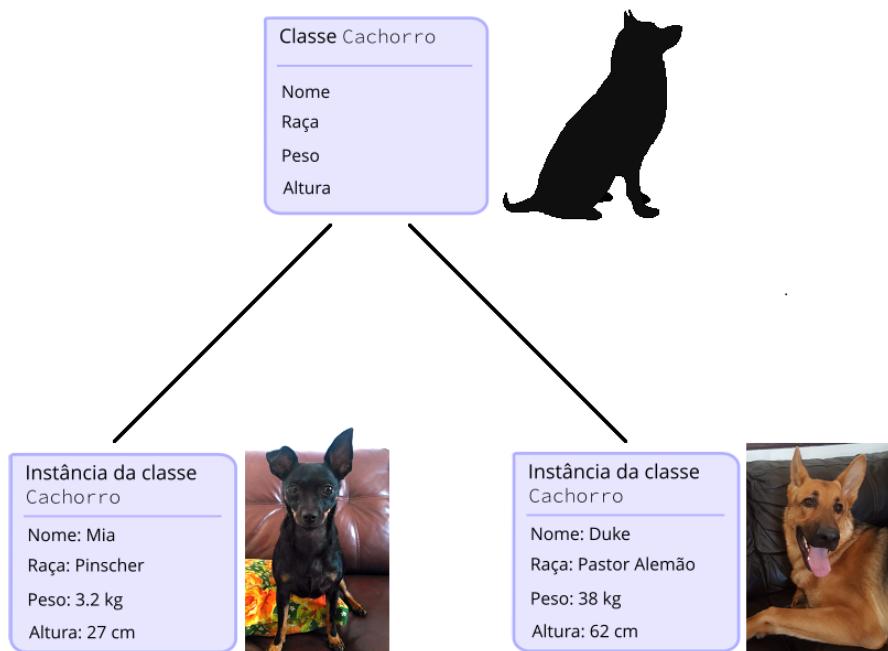


Figura 2.33: Classe Cachorro e Instâncias Mia e Duke<sup>a</sup>.

<sup>a</sup>Todos os direitos reservados pelo fotógrafo Rafael Girundi.

```
// Variáveis ou atributos da classe.

// Construtor
NomeDaClasse(parametrosDoConstrutor) {
    // Código do construtor.
}

// Funções ou métodos da classe.
}
```

Observe que a classe é uma estrutura complexa no sentido que é composta por variáveis e funções. Em um programa do Processing, antes mesmo de declarar um objeto de uma classe, é necessário que a mesma esteja programada, que é basicamente um bloco como o do código 2.61. Mais detalhadamente, as classes possuem os seguintes pontos importantes:

- **Declaração:** Uma classe é precedida pela palavra `class`, e seu nome segue as mesmas restrições da nomenclatura de variáveis e funções. Devem ser declaradas sempre fora do `setup()`, `draw()`

ou qualquer outra função. Uma boa prática é declará-las com a primeira letra maiúscula. As chaves que seguem após o nome delimitam todo o código referente à classe.

- **Construtores:** São funções executadas quando é feita a instanciação do objeto da classe. Eles podem ser entendidos como uma função que é chamada sempre que você cria um novo objeto da classe. O propósito do construtor é servir como uma interface para passar informações pertinentes ou essenciais para a existência de uma instância. Sua restrição é que construtores devem ter exatamente o mesmo nome da classe e não possuir nenhum tipo de retorno (nem mesmo void).
- **Instanciação:** A instanciação de um objeto da classe é feita de maneira muito semelhante à declaração e inicialização de uma nova variável, contudo é precedida pela palavra new seguida do nome da classe e dos argumentos do construtor, tal como:

```
NomeDaClasse nomeDaInstancia = new NomeDaClasse(argumentosDoConstrutor);
```

- **Variáveis:** São chamadas de atributos de uma classe e podem ser qualquer tipo de variável estudada até o momento, inclusive outras classes, sendo sujeitas as mesmas regras de nomenclatura e declaração. Essas variáveis são internas às classes e não é possível usá-las em um programa sem referenciar explicitamente o objeto ou classe de origem. Para acessar os atributos de um objeto no programa principal utiliza-se da chamada sintaxe por ponto:

```
nomeDaInstancia.nomeDoAtributo;
```

- **Funções:** Ou métodos de uma classe também seguem as mesmas regras das funções especificadas na seção 2.3. Assim como os atributos, elas são internas a classe e suas invocações também exigem a sintaxe por ponto:

```
nomeDaInstancia.nomeDaFuncao(argumentosDaFuncao);
```

Confira no código abaixo como a classe genérica, citada anteriormente, modelaria a classe cachorro:

```
// Declaração da classe genérica cachorro.  
// Agrupa atributos e métodos comuns a todos os cachorros:  
  
class Cachorro {  
    // Atributos:  
    String nome, raca;  
    float peso, altura;  
  
    // Construtor:  
    // Inicializa os atributos essenciais para a instanciação do objeto:  
    Cachorro(String _nome, String _raca, float _peso, float _altura) {  
        nome = _nome;  
        raca = _raca;  
        peso = _peso;
```

```

        altura = _altura;
    }

    // Métodos
    void latir() {
        println("Au Au!");
    }
}

// Programa principal:

void setup() {
    // Declara um objeto da classe cachorro:
    Cachorro cao1 = new Cachorro("Mia","Pinscher",3.2,27);

    // Exibe atributos do objeto cao1:
    println("Nome:", cao1.nome);
    println("Raça:", cao1.raca);

    // Invoca um método do objeto cao1:
    cao1.latir();
}

```

Isso conclui a parte *técnica* sobre classes. Os conceitos apresentados aqui permitem a principal aplicação das classes, exemplificada através de um estudo de caso na próxima seção.

### 2.6.1 | Uma classe na prática

---

Professores de desenho costumam dizer que metade da habilidade de criar uma boa obra está na destreza da mão e a outra metade em uma mudança na forma de observar o mundo. Com a OOP é exatamente igual. Metade do necessário você acabou de aprender, relativo a como declarar uma classe, seus construtores, métodos e atributos. A outra metade consiste na mudança de como *enxergar* a separação e organização de seus programas. O entendimento dessa mudança de paradigma pode ser melhor ilustrado através de um exemplo. Considere o código abaixo que desenha círculos dispostos como a figura 2.34. Neste momento não estamos interessados na construção do padrão em si, o foco deve ser direcionado para a estrutura global da escrita do programa.

```

void setup() {
    size(700,200);
    background(255);

    // Desenho do padrão:
    for(int i = 0; i < 31; i++) {

```

```

        for(int j = 0; j < 10; j++) {
            color corCirc = color(180-50*(j-2),50*(j-3),j*25);
            fill(corCirc);
            ellipse(i*22.3+15,j*25+12.5*(i%2),25-abs(i-15),25-abs(i-15));
        }
    }
}

```

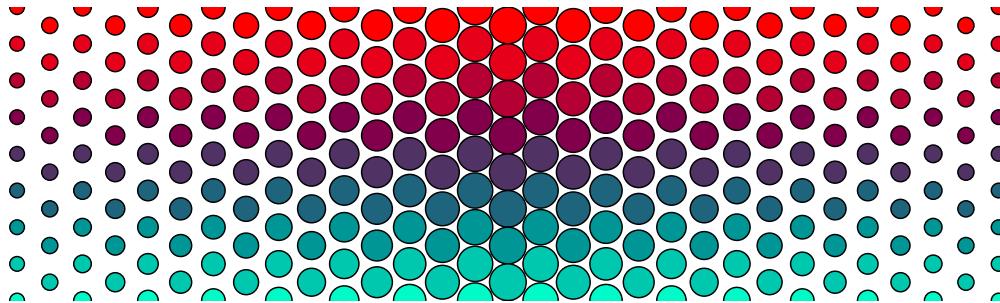


Figura 2.34: Padrão formado por círculos.

Em relação as funcionalidades, não há nenhum mistério até agora visto que você conhece todos os comandos que formam esse programa; mas e em relação ao código? E se você quisesse saber qual a posição, o tamanho do raio ou até mesmo as cores do primeiro círculo desenhado, ou do círculo número 42, seria possível? Da maneira que o código acima foi escrito não é possível acessar nenhuma dessas informações simplesmente porque elas não foram armazenadas. Guardar essas informações implica em criar variáveis para conter os atributos dos círculos, tais como posição x, posição y, valor do raio e cor. Vamos reescrever o código e aproveitar para utilizar de vetores como a maneira de armazenar esses dados, reduzindo assim o tamanho final do texto. Além disso, optamos por criar uma função separada para exibir esses círculos coloridos de forma a organizar o código.

#### Código 2.67 Círculos - Programação Procedural

```

// Variáveis para armazenamento de dados dos círculos:
float[] posX;
float[] posY;
float[] raio;
color[] corC;

void setup() {
    size(700,200);
    background(255);
}

```

```

// Total de círculos: 31 (horiz) * 10 (vert):
posX = new float[310];
posY = new float[310];
raio = new float[310];
corC = new color[310];

// Índice identificador do número do círculo:
int indice = 0;

// Etapa de construção dos círculos:
for(int i = 0; i < 31; i++) {
    for(int j = 0; j < 10; j++) {
        color corCirc = color(180-50*(j-2),50*(j-3),j*25);

        // Guarda dados dos círculos para consultas posteriores:
        posX[indice] = i*22.3 + 15;
        posY[indice] = j*25+12.5*(i%2);
        raio[indice] = 25-abs(i-15);
        corC[indice] = corCirc;

        indice++;
    }
}

// Etapa de exibição dos círculos:
for(int i = 0; i < indice - 1; i++) {
    exibirCirculo(i);
}

println("Posição X,Y do círculo número 1:", posX[0], ",",
       posY[0]);
println("Tamanho do raio do círculo número 42 ", raio[41]);
}

void exibirCirculo(int _c) {
    fill(corC[_c]);
    ellipse(posX[_c],posY[_c],raio[_c],raio[_c]);
}

```

Veja como foi necessário declarar variáveis adicionais e preenchê-las de maneira cuidadosa de modo que correspondessem especificamente a cada círculo. Se mais informações sobre esses círculos forem adicionadas (diâmetro, área, perímetro, exibição como um arco, etc.), serão necessárias cada vez mais variáveis e funções para obter esses dados. Isso gera inconvenientes, como o aumento do número de variáveis que não tem relação com o programa principal (e sim com os círculos), poluição do código principal com funções específicas dos círculos e atenção para preencher e consultar os dados com os índices corretos. Grande parte desses problemas podem ser contornados se abordarmos o programa com a mentalidade da OOP.

Retorne ao código 2.67 e analise-o com calma. Você deve ter percebido que todos os círculos possuem variáveis comuns como posição do centro, valor de raio e cor, assim como uma função comum, a de exibição (todos eles são mostrados na tela). Portanto é possível considerar a forma “círculo” como uma estrutura de dados abstrata que agrupa as variáveis e funções comuns a todas essas figuras, sendo estas a base para a construção da classe Círculo. A representação individual de cada círculo será alcançada através da criação de uma instância (ou objeto) dessa classe. O código seguir demonstra a unificação das variáveis e funções comuns aos objetos na classe:

#### Código 2.68 Círculos - OOP

```
// Declaração da classe:  
  
class Circulo {  
    // Atributos:  
    float posX, posY;  
    float raio;  
    color cor;  
  
    // Construtor:  
    Circulo(float _posX, float _posY, float _raio, color _cor) {  
        posX = _posX;  
        posY = _posY;  
        raio = _raio;  
        cor = _cor;  
    }  
  
    // Métodos:  
    void exibir() {  
        fill(cor);  
        ellipse(posX,posY,raio,raio);  
    }  
}  
  
// Programa Principal:  
  
// Declara um vetor de objetos da classe "Circulo".  
// Total de círculos: 31 (horiz) * 10 (vert):  
Circulo[] circ = new Circulo[310];  
  
void setup() {  
    size(700,200);  
    background(255);  
  
    int indice = 0;
```

```

// Instanciação dos objetos da classe:
for(int i = 0; i < 31; i++) {
    for(int j = 0; j < 10; j++) {
        color corCirc = color(180-50*(j-2),50*(j-3),j*25);
        circ[indice] = new Circulo(i*22.3+15,j*25+12.5*(i%2),25-abs(i-15),corCirc);
        indice++;
    }
}

// Etapa de exibição dos círculos:
for(int i = 0; i < circ.length; i++) {
    circ[i].exibir();
}

println("Posição X,Y do círculo número 1:", circ[0]. posX, ", " , circ[0]. posY);
println("Tamanho do raio do círculo número 42:", circ[41].raio);
}

```

Note como esse código ficou muito mais limpo e organizado que o anterior. Agora a classe `Circulo` contém todas as informações referentes a essa figura e, se necessário consultá-las, basta acessar o respectivo atributo do objeto através da sintaxe por ponto. Essa característica tornam as classes uma estrutura de dados autocontida. No que diz respeito ao código em si, perceba que agora há uma distinção notável entre o programa e a estrutura de dados que ele utiliza. No programa principal o foco foi direcionado apenas para a criação de objetos do tipo `Circulo` e, posteriormente, sua exibição. Todas as peculiaridades relativas aos detalhes técnicos da declaração e atribuição minuciosa das variáveis ou da implementação da função de exibição foram abstraídas pela classe, que agora faz isso automaticamente para você. Um esquema visual holístico das diferenças entre os códigos [2.67](#) e [2.68](#) pode ser vista na figura [2.35](#).

Os tópicos apresentados nesta seção são uma introdução mínima as classes, mas suficiente para que você aplique esse conhecimento em quase a totalidade de seus programas. Se você quiser aprofundar mais na OOP, você pode procurar recursos que expliquem conceitos como polimorfismo, herança e encapsulamento que aumentam o reaproveitamento do código e a segurança de acesso as classes. Independente de suas outras funcionalidades, lembre-se que elas são uma de suas maiores aliadas na abordagem de problemas e no planejamento de soluções. A medida que você aumentar a complexidade de seus programas você irá querer organizá-los em componentes e subcomponentes, e nenhuma metodologia fornece um maneira mais natural e pragmática para atingir esse objetivo do que a OOP .

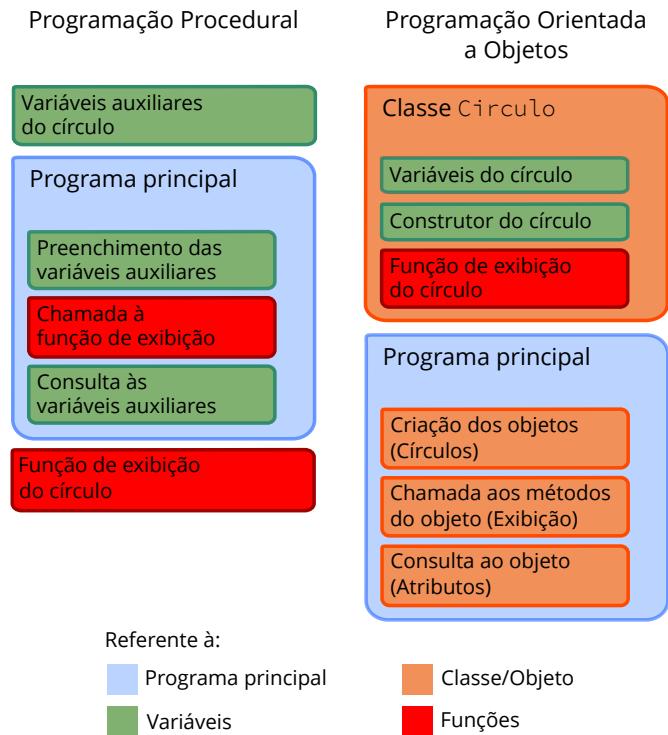


Figura 2.35: Diferenças entre abordagens Procedural e OOP.

## 2.7 | Sumário

---

Neste primeiro capítulo focado puramente no código você aprendeu sobre os componentes que formam a base de todos os programas que você irá escrever. Brevemente, você estudou sobre:

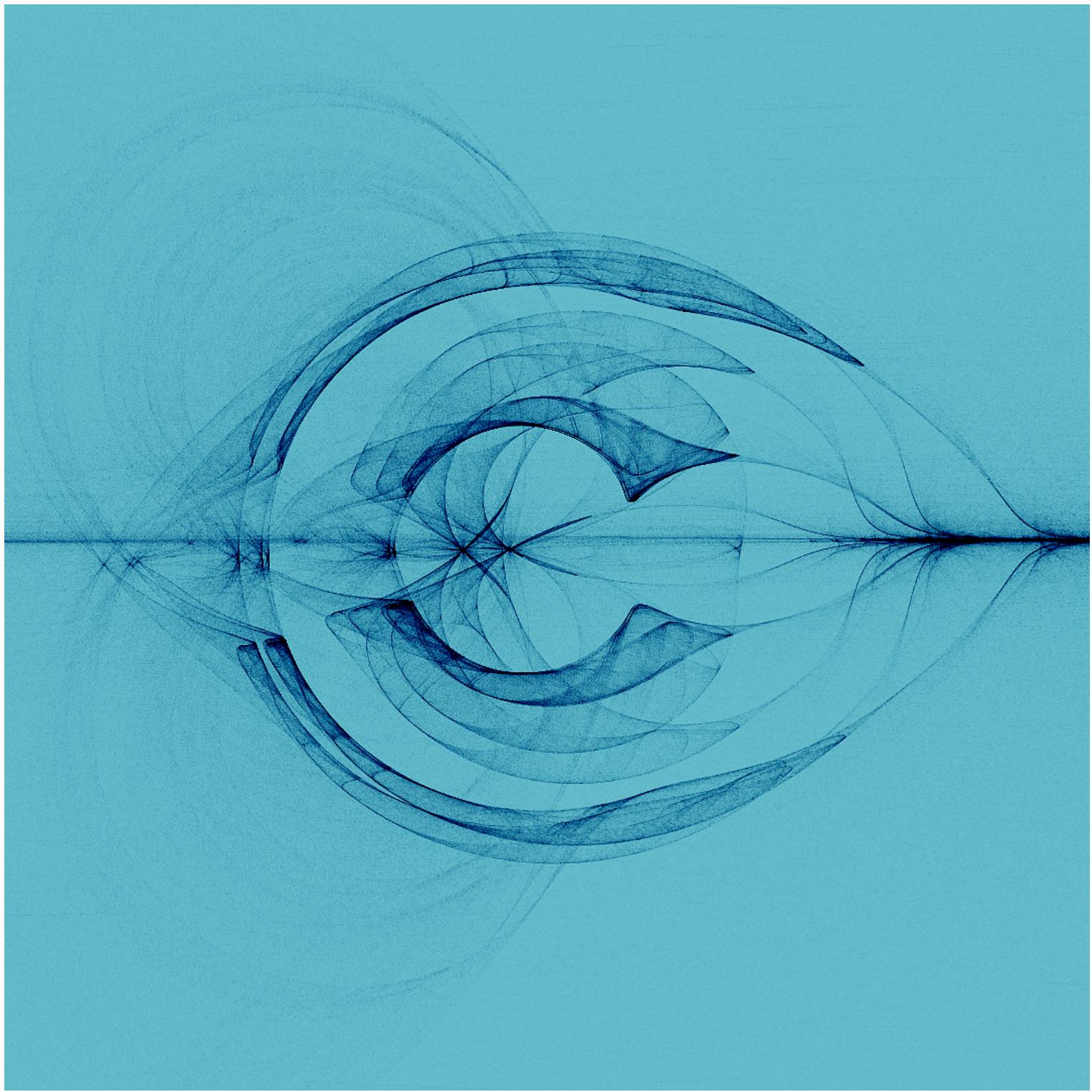
- Variáveis:** São estruturas de dados que armazenam informações que podem ser acessadas com um simples referenciar ao seu nome.
- Operadores:** Manipulam e estabelecem relações entre as variáveis. São elementos chave em todas as demais estruturas de programação.
- Funções:** Permite compactar uma infinidade de instruções em um único bloco ou *script*. São executadas sempre que conveniente através da invocação da função.

- **Condicionais:** Principais responsáveis por direcionar o fluxo de um programa de acordo com condições pré-estabelecidas ou em execução.
- **Repetições:** Possibilitam repetir blocos de código em uma composição reduzida, potencialmente economizando centenas de linhas além do precioso tempo do programador.
- **Classes:** É a mais customizável e poderosa estrutura de dados na programação. Agrupa variáveis e métodos referentes a uma mesma abstração ou modelo. Permite acessar os dados de uma maneira organizada e segregá-los do programa principal.

Com isso você finaliza a primeira parte da fundação de um programador. Símbolos e nomes que antes não faziam sentido são prontamente decodificados por você. Mas calma, não deixe o poder subir à cabeça por que, na verdade, você é somente parte de um programador. Um programador não é alguém que memoriza todos os comandos, funções e sintaxes, e sim alguém que sabe usar o código a seu favor, criando e programando algoritmos para atingir um propósito. Saber a sintaxe e não ter um objetivo é como decorar um dicionário e não saber usar as palavras para formar frases.

Na segunda parte dessa jornada você será convidado a se aprofundar em tópicos que compõem a carteira criativa de um artista programador (ou programador artista, fica a seu critério). Você aprenderá um pouco mais sobre como usar propriedades e resultados matemáticos para estampar estéticas singulares nos seus trabalhos. Mais à frente você testemunhará como é importante saber aplicar conhecimentos que não se restringem a linguagens de programação, abrangendo conceitos universais da matemática e da natureza.







# EVOLUÇÃO

Em todo caos há um cosmos, em toda desordem, uma ordem secreta.

- CARL JUNG



[ CAPÍTULO 3 ]:

# Caos

Aprendendo a abandonar o controle

Na filosofia existe um princípio denominado determinismo que defende que todos os eventos que já aconteceram e irão suceder são resultados de estados e ações passadas. De certa forma, o que você realizou nos capítulos anteriores se encaixa nesse pensamento. Você aprendeu a posicionar, com precisão cirúrgica, figuras geométricas, assim como definir repetições e condicionais cujas consequências eram completamente conhecidas antes mesmo da execução do programa. Apesar de isso ser um feito incrível, sempre que você pressionar o botão de executar o resultado será o mesmo. Em pouco tempo o previsível se torna desinteressante.

No espectro oposto desse raciocínio está o caos, em que é impossível prever qualquer consequência, pois elas são independentes e descorrelacionadas de todas e quaisquer ações. Aqui reina o imprevisível e a completa ausência de lógica. Um pandemônio que termina em indiferença.

No centro dessa dualidade nasce o núcleo do que faz o mundo interessante. É o familiar, mas cheio de surpresas, o caos controlado. Na natureza é isso que faz as árvores terem forma de árvores, e nunca serem absolutamente iguais umas as outras. O mesmo vale para as nuvens, para os animais e as pessoas. Sucintamente, a *forma* é mantida, o conteúdo não. Neste capítulo você entenderá como aplicar esse conceitos para transformar a arte computacional na intrigante arte gerativa.

## 3.1 | Aleatoriedade

---

A arte gerativa está intrinsecamente ligada à falta de uniformidade e à capacidade de criar infinitas figuras e padrões usando microvariações presentes durante a execução do código. Uma maneira de forçar essas variações é parametrizando elementos do código tais como variáveis, condicionais e repetições e, de alguma forma, alterá-los. Existem dois grandes fatores que contribuem profundamente para uma boa arte gerativa:

São eles o “*o que*” deve ser parametrizado no código e o “*como*” deve ser parametrizado.

O *o que* deve ser parametrizado é usualmente definido na etapa de concepção da arte para criar uma forma, molde ou configuração distingível na escala macro. No exemplo citado da árvore, consiste em sa-

ber que ela sempre possuirá um tronco, galhos, folhas e flores, mas não é estabelecido o número, tamanho ou cores desses elementos.

O *como* deve ser parametrizado é normalmente definido na etapa de simulação para criar uma execução marcante na escala micro, evidenciada nos detalhes. Para a árvore, isso implicaria em definir que, quando ela tem entre 6 e 10 metros de altura, copa ampla de 3 a 6 metros e flores de 5 a 8 cm com diversos tons de amarelo ela possuirá uma estética deslumbrante (3.1). Na computação artística é ideal experimentar com esses valores durante ou após a execução do código, pois assim você pode ajustá-los sucessivamente até obter um resultado que goste.



Figura 3.1: Foto do majestoso Ipê Amarelo durante sua floração.

Um modo de alterar o valor dos elementos parametrizados é utilizando a aleatoriedade para gerar números diferentes toda vez que o programa for executado. No entanto humanos não são muito bons nessa

tarefa<sup>1,2</sup> e diversas vezes tendem a se repetir, produzindo resultados similares. Por outro lado, máquinas são excelentes em gerar números imprevisíveis<sup>3</sup> e por isso você usará o computador para automatizar as variações desejadas.

O primeiro método para obter números aleatórios que você irá estudar é através da função `random()`, que gera um número qualquer, baseado em uma distribuição uniforme<sup>4</sup>, entre os argumentos fornecidos. Por exemplo:

- `random(5,10)`: Gera um número aleatório entre 5 e 10.
- `random(0,10)` ou `random(10)`: Gera um número aleatório entre 0 e 10.
- `random(height)`: Gera um número aleatório entre 0 e a altura da janela de exibição.

Note que a função `random(10)` é numericamente igual a função `10*random(1)` uma vez que `random(1)` gera um número entre 0 e 1 que, multiplicado por 10, resulta em número entre 0 e 10. Em alguns casos é mais interessante utilizar este segundo tipo de notação, uma vez que remete a funções de distribuição probabilística na matemática. Algumas funções equivalentes são:

- `random(30)  $\iff$  30*random(1)`.
- `random(5,20)  $\iff$  (5 + random(15))  $\iff$  (5 + 15*random(1))`.
- Genericamente, `random(min,max)` é escrito como `(min + (max-min)*random(1))`.

Se você quiser visualizar os números gerados pode escrever um código para exibi-los no console:

```
// Exibe no console cem números entre 0 e 1000:  
for(int i = 0; i < 100; i++) {  
    println(random(1000));  
}
```

Transferindo para a tabela 3.1 teríamos:

<sup>1</sup>Wagenaar, W. A. (1972). "Generation of random sequences by human subjects: a critical survey of the literature". Psychological Bulletin 77: pags.65-72.

<sup>2</sup>Brugger, P. (1997). "Variables that influence the generation of random sequences: An update". Perceptual and Motor Skills 84(2): pags.627-661.

<sup>3</sup>Na verdade, um computador não consegue gerar números verdadeiramente aleatórios e sim pseudo-aleatórios. Ênfase no pseudo, pois eles são gerados de acordo com parâmetros do computador como `clock`, ou através de algoritmos como o Gerador Congruente Linear. O problema da pseudo-aleatoriedade é que uma vez descoberto como os números são gerados eles se tornam determinísticos. Mas não se preocupe, isso é muito mais significativo para a área de criptografia do que para a arte computacional.

<sup>4</sup>Uma distribuição uniforme, ou retangular, é aquela contida entre um máximo e um mínimo de tal forma que a chance de qualquer número ser escolhido dentro dela é a mesma.

645.9	594.3	071.9	647.0	743.5	394.5	357.1	260.1	699.1	229.4
422.8	438.6	764.0	918.5	814.5	317.7	362.5	355.9	966.2	567.8
231.0	066.8	399.8	315.1	800.0	474.6	687.1	242.1	317.1	160.6
521.6	056.0	513.2	530.1	742.6	925.0	145.2	536.8	159.2	033.1
212.3	286.7	553.5	830.8	682.9	148.8	394.0	113.6	052.1	302.0
004.5	757.0	278.3	681.7	192.4	810.1	636.1	947.7	499.8	998.8
740.6	144.9	194.2	320.8	607.3	085.8	113.0	917.6	741.3	460.2
329.6	369.8	222.0	211.5	965.7	844.2	839.0	651.5	060.8	501.7
032.2	096.8	268.4	394.1	617.8	859.1	924.7	894.5	691.6	229.5
169.3	937.7	267.0	552.2	839.3	941.3	943.2	632.3	667.4	805.5

Tabela 3.1: Números gerados pela função `random()`.

Em forma de texto, como a tabela 3.1, é difícil ver o quanto significativo são ou não esses números. Felizmente o Processing é uma linguagem voltada para visualizações e é possível ver uma representação gráfica da função `random()`. Inicialmente vamos desenvolver um programa que, para cada pixel do eixo x (largura) dessa janela, seja gerado um ponto y aleatório e traçado uma reta ligando o ponto anterior gerado ao atual. Esta é uma análise unidimensional de uma distribuição aleatória, já que os pontos do eixo y são a única incógnita no ato da execução do programa. O código abaixo mostra como realizar isso e o resultado pode ser visto na figura 3.2.

### Código 3.2 Distribuição aleatória

```
void setup() {
    size(400,200);
    background(255);

    float xAnterior = 0;
    float yAnterior = height*random(1);

    for(int i = 1; i < width; i++) {
        float y = height*random(1);
        line(xAnterior,yAnterior,i,y);
        yAnterior = y;
        xAnterior = i;
    }
}
```

Imediatamente, o que você pode perceber é uma distribuição muito errática, uma curva abundante em picos e vales. Na prática, isso significa que cada vez que você chama a função `random()` ela retorna um número que não tem nenhuma relação com qualquer outro que foi ou será gerado por ela. Consequentemente nada impede que ocorram diferenças gigantescas entre números de iterações sequenciais. Por exemplo, se você chamar a função `random(1000)` duas vezes seguidas, na primeira pode obter 0.00 e na



Figura 3.2: Linhas ligando pontos aleatórios gerados por `random()`.

segunda 999.99, obtendo a maior amplitude possível do intervalo.

Números puramente aleatórios podem ser usados de diversas formas para influenciar o comportamento de seu programa. Se usados para movimentar figuras, podem implicar em variações bruscas e descontínuas ou grandes alterações nas direções de objetos. Ao serem aplicados para colorir figuras, as cores raramente irão manter um tom harmônico, que pode ser bom ou ruim, depende do efeito que você deseja causar. Em geral, usar números fornecidos pela função `random()`, sem nenhum tipo de ajuste, pode resultar em uma impressão de *sintético* e *não naturalidade*, seja do movimento ou variação. Tal fato pode ser exemplificado pelo código abaixo, que dispõe figuras aleatoriamente na tela e as colore com a função `random()`. Veja a imagem gerada na figura 3.3.

### Código 3.3 Cores aleatórias

```
void setup() {
    size(400,150);
    noStroke();

    for(int i = 0; i < 500; i++) {
        // Posição aleatória na janela:
        float x = random(width);
        float y = random(height);

        // Cor aleatória:
        float r = random(255);
        float g = random(255);
        float b = random(255);
        fill(r,g,b);
        rect(x,y,30,30);
    }
}
```

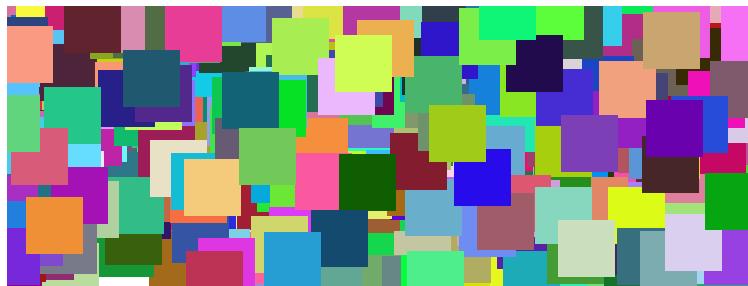


Figura 3.3: Variando a cor de desenho de acordo com a função `random()`.

## 3.2 | Distúrbios

---

A aleatoriedade pura, introduzida na seção passada, tem um papel importante ao gerar números descorrelacionados. No entanto ela desaponta ao ser muito brusca e, ironicamente, muito imprevisível, podendo criar visuais mecânicos e frios. Foi devido a esse fenômeno que em 1983 o Dr. Ken Perlin, um professor de ciências da computação, inventou o ruído Perlin<sup>5</sup>, um algoritmo capaz de gerar uma sequência naturalmente ordenada de números pseudo-aleatórios. Calma, essa frase complicada significa apenas que o Dr. Perlin conseguiu criar um algoritmo que gera números aleatórios com uma distribuição *suave, natural* e, ultimamente, *orgânica*. A linguagem Processing possuí esse algoritmo<sup>6</sup> convenientemente programado através da função `noise()`.

A chamada da função `noise()` não é tão direta quanto a `random()` e existem alguns detalhes que podem ser melhor consultados em sua referência. Os mais importantes para o seu funcionamento correto são:

- A função `noise()` sempre devolve um número entre 0 e 1.
- A função deve receber argumentos de entrada que estejam constantemente variando. Quanto menores as variações dos parâmetros entre as chamadas da função, menores serão as diferenças entre os números gerados. Um parâmetro fixo sempre fornecerá o mesmo valor de ruído.

A forma mais básica de sua chamada é:

```
float ruido = noise(arg1);
```

---

<sup>5</sup>Do inglês, *Perlin noise*

<sup>6</sup>A versão do Processing é uma simplificação do verdadeiro ruído Perlin.

O truque para entender essas peculiaridades está na maneira de enxergar essa função. Você deve imaginar a função `noise()` como um “observador” de uma curva aleatória. Essa curva é completamente construída no início da execução do seu programa e, quando você passa um argumento para a função `noise()`, você simplesmente realiza uma consulta a um ponto dessa curva. As figuras 3.4 e 3.5 exemplificam essa ideia.

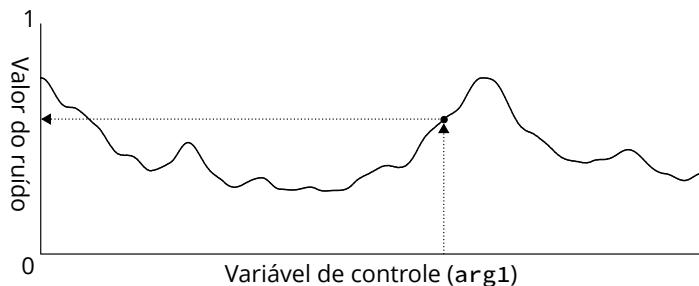


Figura 3.4: Exemplo de curva do ruído (`noise`).

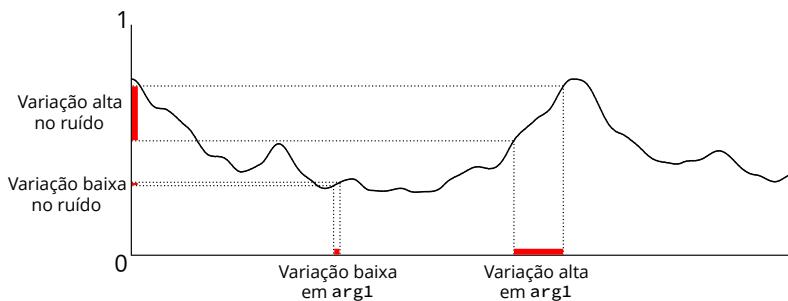


Figura 3.5: Variações no argumento e o resultado do valor de ruído.

Essa interpretação também explica a necessidade dos parâmetros dessa função estarem sempre variando, caso contrário você estaria consultando sempre o mesmo “ponto” da curva e recebendo um valor estático.

Para fazer a comparação com a função `random()`, iremos traçar a curva da figura 3.2 usando a função `noise()`. Será necessário criar uma variável auxiliar e incrementá-la antes de cada invocação da função para que a mesma retorne valores diferentes. Adaptando o código 3.2 você pode escrever:

#### Código 3.5 Distribuição ruidosa

```
void setup() {
```

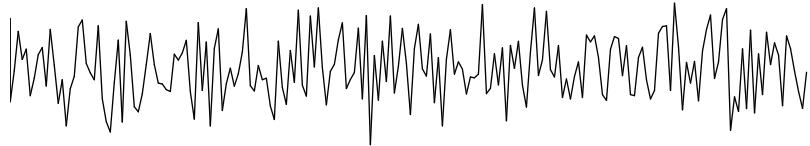
```

size(400,200);
background(255);

float ruido = 0;
float amplitudeRuido = 1;
float xAnterior = 0;
float yAnterior = height*noise(ruido);

for(int i = 1; i < width; i++) {
    float y = height*noise(ruido);
    line(xAnterior,yAnterior,i,y);
    yAnterior = y;
    xAnterior = i;
    ruido += amplitudeRuido;
}
}

```



**Figura 3.6:** Linhas ligando pontos aleatórios gerados por `noise()`.

Uma análise rápida da distribuição resultante, figura 3.6, nos leva à conclusão de que a função `noise()` é ineficaz dado que comportamento da curva não mudou. A amplitude entre amostras adjacentes continua extremamente alta, sendo indesejado para criar sequências orgânicas. No entanto devemos ter cuidado com inferências precipitadas e, em caso de dúvidas, retornar à página de ajuda da linguagem.

Lembre-se que a função `noise()` terá um comportamento que dependerá diretamente da variação em seus argumentos a cada vez que ela for chamada. Se o passo (ou diferença) entre o argumento anterior e o próximo for muito alto você terá grandes amplitudes, e se for muito baixo, terá uma saída quase que constante. O segredo é experimentar com diversos valores até encontrar um se adapte com o tipo de resposta que você deseja obter. A referência do Processing sugere que variações contidas no intervalo de 0.005 a 0.03 entre chamadas da função `noise()` atendem a grande parte das aplicações. No código 3.5 o passo foi parametrizado pela variável `amplitudeRuido`. Na figura 3.7 é mostrado exatamente o mesmo programa com valores diferentes para essa variável.

Observe como o comportamento da curva foi alterado significativamente de acordo com o argumento da função. Perceba também uma amenização na turbulência das curvas em comparação com as da função `random()`. Quando esses valores forem usados para direcionar um movimento ou alterar uma cor, a ilusão



(a) `amplitudeRuido = 0.3`



(b) `amplitudeRuido = 0.05`



(c) `amplitudeRuido = 0.01`



(d) `amplitudeRuido = 0.003`

Figura 3.7: Curvas para os diferentes valores do passo da função `noise()`.

criada será de uma variação orgânica e natural. O código abaixo é uma adaptação do 3.3, e cria cores com transições suaves, quase que contínuas, exibidas na figura 3.8.

#### Código 3.6 Cores ruidosas

```
void setup() {
    size(400,150);
    noStroke();

    // Variáveis que controlam o ruído das cores:
    float nr = 0, ng = 100, nb = 200;

    for(int i = 0; i < 500; i++) {
        float x = random(width);
        float y = random(height);
        float r = 255*noise(nr);
        float g = 255*noise(ng);
        float b = 255*noise(nb);
        fill(r,g,b);
        rect(x,y,30,30);

        // Incremento das variáveis para produzir ruídos distintos:
        nr += 0.01;
        ng += 0.01;
        nb += 0.01;
    }
}
```



Figura 3.8: Variando a cor de desenho de acordo com a função noise().

### 3.3 | Ruídos multidimensionais

---

Nas duas primeiras seções deste capítulo você observou como as funções `random()` e `noise()` são peças fundamentais para incluir o caos em seus programas. Nos dois códigos, [3.2](#) e [3.5](#), foram geradas visualizações bidimensionais de uma variação unidimensional, referente à análise da diferença entre um número aleatório e o seu sucessor. O objetivo de tais figuras foi facilitar a comparação do aspecto brusco ou suave proveniente dessas funções.

Uma segunda maneira de criar um padrão visual e simultaneamente comparar as funções `random()` e `noise()` é através de uma distribuição e figura verdadeiramente bidimensional, formada por um plano de duas dimensões: a janela de saída. Definindo uma janela do tipo `size(300,150)`, existirão um total de 45000 pixels que podem ser coloridos conforme nossa vontade. Você pode automatizar a tarefa de referenciar os pixels sequencialmente se criar duas estruturas de repetição aninhadas (seção [2.5.2](#)) projetadas para percorrer a largura e a altura da tela. Assim, para cada índice gerado pelas repetições, você pode colorir o respectivo pixel usando a escala de cinza cuja cor será fornecida tanto pela função `random()` quanto pela `noise()`. O aninhamento de repetições para o primeiro caso pode ser escrito como:

```
void setup() {  
    size(300,150);  
    background(255);  
  
    // Varre todos os pixels da janela e os preenche com uma cor aleatória:  
    for(int i = 0; i < width; i++) {  
        for(int j = 0; j < height; j++) {  
            stroke(color(random(255)));  
            point(i,j);  
        }  
    }  
}
```

A função `noise()` requer sua versão correspondente de dois argumentos, com elementos declarados e incrementados individualmente. Também deve-se atentar ao cuidado de “reiniciar” uma das dimensões do ruído quando for dado um passo na repetição mais externa. O preenchimento da janela é feito a seguir:

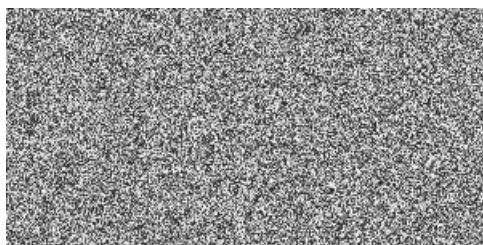
```
void setup() {  
    size(300,150);  
    background(255);  
  
    float amplitudeRuido = 0.05;  
    float ruidoX = 0;  
    float ruidoY = 100;  
  
    // Varre todos os pixels da janela e os preenche com uma cor ruidosa:  
    for(int i = 0; i < width; i++) {
```

```

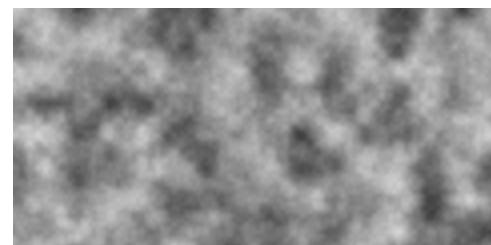
        for(int j = 0; j < height; j++) {
            ruidoY += amplitudeRuido;
            stroke(color(255*noise(ruidoX,ruidoY)));
            point(i,j);
        }
        // Reinicia a variável ruidoY para seu valor inicial, e incrementa ruidoX:
        ruidoY = 100;
        ruidoX += amplitudeRuido;
    }
}

```

A figura 3.9a mostra o resultado para a função `random()` e a figura 3.9b para a função `noise()`. Vamos analisar as imagens de acordo com o que foi explicado nas seções anteriores. A imagem produzida pela função `random()` se assimila a uma estática de televisão. Isso era esperado uma vez que essa função gera elementos aleatórios independentes um do outro, sendo muito provável que a amplitude entre o máximo e o mínimo de uma cor seja alta, produzindo pixels claros (cores próximas a 255) e escuros (cores próximas a 0) lado a lado, correspondendo ao efeito visual de estática. Por sua vez a função `noise()` também gera valores aleatórios, no entanto eles não variam muito entre si (dependendo do passo) gerando uma figura muito mais suave, algo parecido com uma neblina, nuvem, mármore ou floresta. Essa classe de ruído permite criar texturas gerativas e foi justamente por esse tipo de efeito visual que o Dr. Perlin recebeu o *Academy Award for Technical Achievement* em 1997, uma espécie de Oscar técnico da indústria cinematográfica.



(a) Utilizando a função `random()`.



(b) Utilizando a função `noise()`.

Figura 3.9: Janela de saída completamente preenchida.

Você pode confirmar que essas imagens bidimensionais não passam de uma extensão das amostras exemplificadas nas figuras e 3.2 e 3.7 se traçar a amplitude da cor dos pixels em forma de curva. Veja as figuras 3.10 e 3.11.

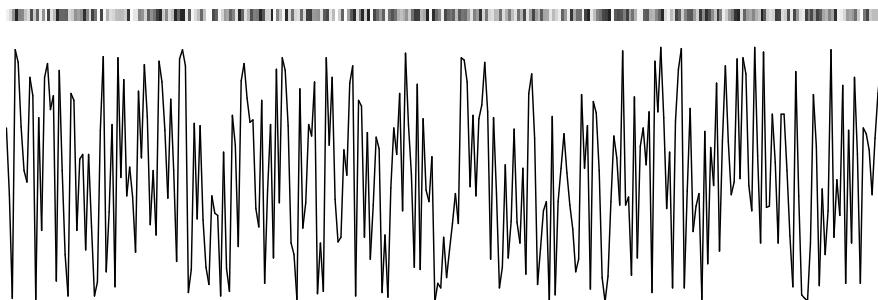


Figura 3.10: Análise de uma linha da imagem 3.9a, formada pela função `random()`.

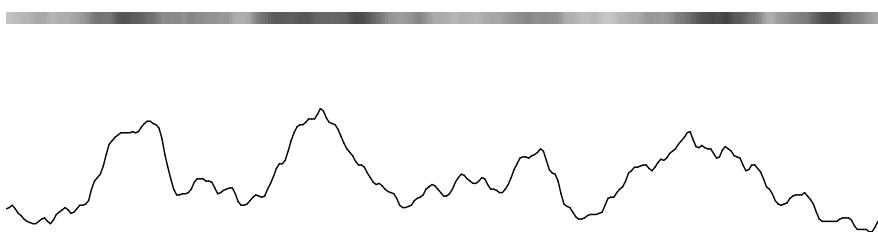


Figura 3.11: Análise de uma linha da imagem 3.9b, formada pela função `noise()`.

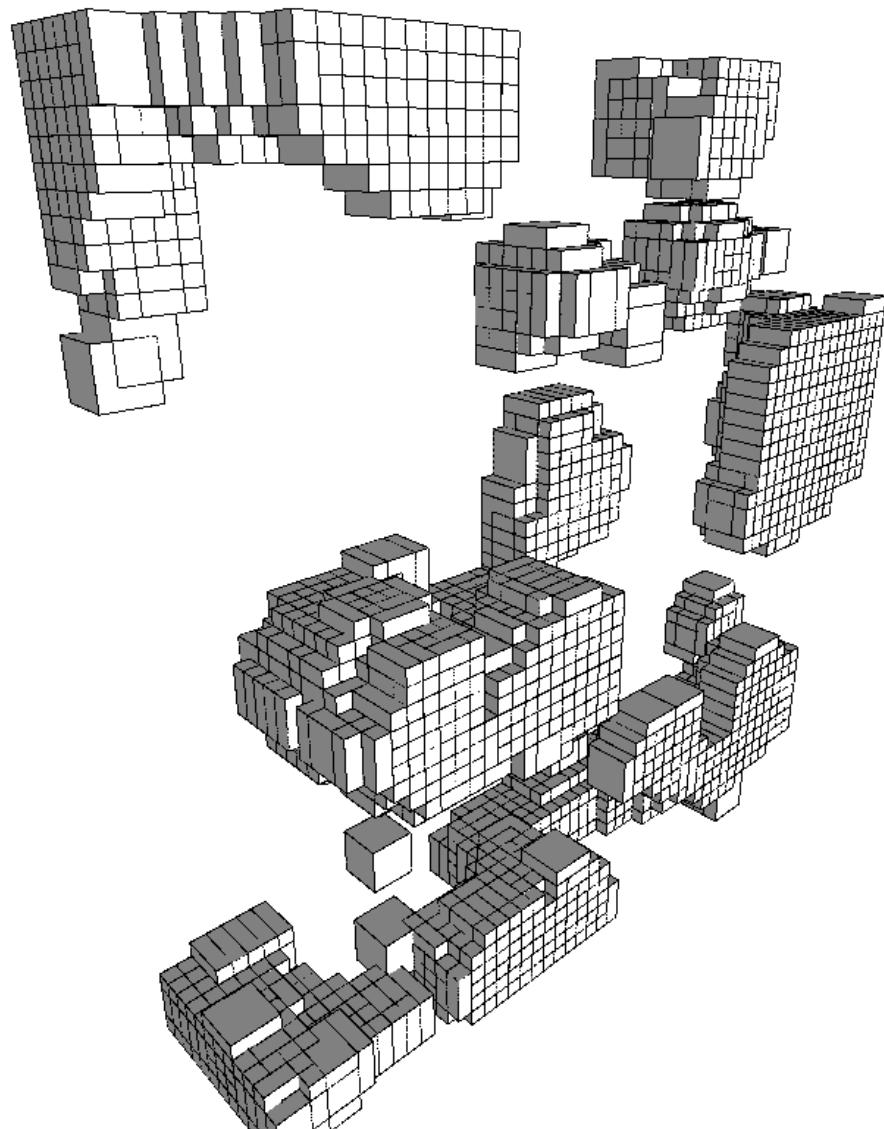


Figura 3.12: Ruído representado em três dimensões.

## 3.4 | Sob o controle do acaso

---

Um conceito que costuma ser referenciado paralelamente quando se trata de acaso é o de probabilidade, definida como a chance de um evento ocorrer, em um experimento aleatório, dentre todas as possibilidades. Matematicamente a probabilidade pode ser escrita como uma fórmula

$$\text{Probabilidade}(\%) = 100 * \frac{\text{Chance de ocorrer o evento}}{\text{Todas as possibilidades}}$$

Por exemplo, se você jogar uma moeda para cima e aguardar ela cair só existem duas opções<sup>7</sup>, cara ou coroa, portanto o universo de possibilidades é dois. Quanto as probabilidades, temos 50% (ou metade, 1/2) de chance da “cara” estar voltada para cima e 50% de chance da “coroa” estar voltada para cima. Mentalize um segundo cenário em você joga um dado de seis lados, ele sempre cairá em uma das seis faces, ou seja, existe 16.667% (ou 1/6) de chance dele cair no número 1, ou no 2 e assim por diante. E qual a chance de um número ser menor que 3? Bem, o dado possui 6 lados (todas as possibilidades), e dois desses números (1 e 2) são estritamente menores que três. Usando a definição de probabilidade, a chance é de 2/6, ou 0.333... que equivale a 33.3%.

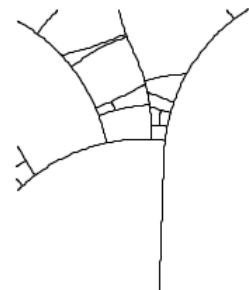
Na arte computacional a probabilidade tem um papel semelhante ao da aleatoriedade: adicionar o elemento da surpresa, mutação e variedade. Em um exemplo concreto, suponha que você desenvolva um algoritmo que crie um ramo central que, aleatoriamente, se derive em ramos secundários e terciários e assim sucessivamente. O que ocorreria se a probabilidade de derivar esses ramos fosse alterada? A figura seria diferente? Em 3.13 está mostrado, visualmente, a resposta dessa pergunta através de um algoritmo inspirado na venação foliar<sup>8</sup>. As diferenças são claramente explícitas mesmo estando vinculadas a poucos coeficientes que controlam as chances de ocorrerem ramificações.

No Processing a maneira mais fácil de se criar uma verificação de probabilidade é através de um condicional usando a função `random()`. As probabilidades podem ser balanceadas ou não, e isso costuma ser usado para priorizar um evento em relação a outro. O código abaixo mostra a implementação de uma probabilidade qualquer no Processing:

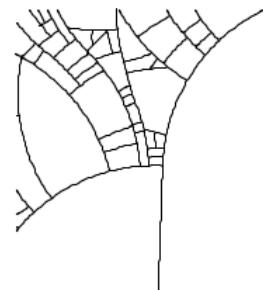
---

<sup>7</sup>É muito improvável da moeda cair exatamente “em pé”, por isso esse evento foi desconsiderado.

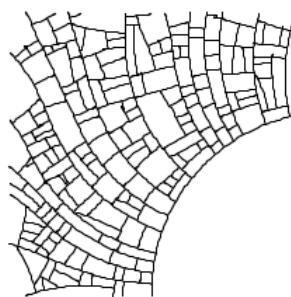
<sup>8</sup>Algoritmo desenvolvido com base no artigo Reunions A. et al (2005). *Modeling and visualization of leaf venation patterns*. ACM Transactions on Graphics 24(3): pags. 702-711



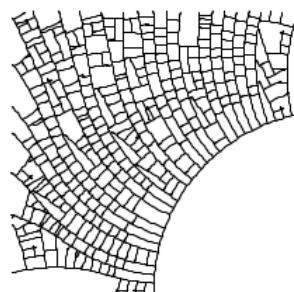
(a) 5%.



(b) 10%.



(c) 25%.



(d) 50%.

Figura 3.13: Curvas para as diferentes probabilidades de ramificações.

```

float probabilidade = 0.8;
if(random(1) < probabilidade) {
    // O código aqui tem 80% de chance de ser executado.
}
else {
    // O código aqui tem 20% de chance de ser executado.
}

```

O ponto chave da probabilidade está na expressão:

```
random(1) < probabilidade
```

A função `random(1)` fornece uma distribuição uniforme entre zero e um, logo a chance de gerar qualquer número menor que 0.8 é de 80% uma vez que todo número de 0.0 até 0.7999 é considerado menor que 0.8. Este caso é similar ao exemplo relativo ao jogar do dado do início deste capítulo. Se você quiser adicionar múltiplos testes envolvendo probabilidades também é possível, mas terá de usar o mesmo número gerado para todos os testes. Neste caso os condicionais deverão ser feitos da menor probabilidade para a maior, além de estarem contidos dentro de estruturas `else`. Isto é ilustrado no código a seguir, que preenche a tela, figura 3.14, com cores baseadas em diferentes probabilidades. Em seguida, para confirmar nosso algoritmo, foi escrito no console, 3.15, o percentual de quadrados de cada cor em relação ao número total.

```

// Contador de número de quadrados da respectiva cor:
int vermelho = 0, verde = 0, azul = 0;

void setup() {
    size(800,300);
    background(255);
    noStroke();

    for(int i = 0; i < width/5; i++) {
        for(int j = 0; j < height/5; j++) {

            // Número qualquer entre 0 e 1
            float prob = random(1);

            if(prob < 0.2) {    // 20% de chance do quadrado ser pintado de vermelho.
                fill(255,0,0);
                vermelho++;
            }
            else {
                if(prob < 0.5) {    // 30% de chance do quadrado ser pintado de verde.
                    fill(0,255,0);
                    verde++;
                }
                else {    // 50% de chance do quadrado ser pintado de azul.

```

```

        fill(0,0,255);
        azul++;
    }
}
// Desenho do quadrado em si:
rect(i*5,j*5,5,5);
}
}

int total = vermelho+verde+azul;
println("Número total de quadrados: " + total);
println("Percentual de quadrados vermelhos:", 100.0*vermelho/total, "%");
println("Percentual de quadrados verdes:", 100.0*verde/total, "%");
println("Percentual de quadrados azuis:", 100.0*azul/total, "%");
}

```

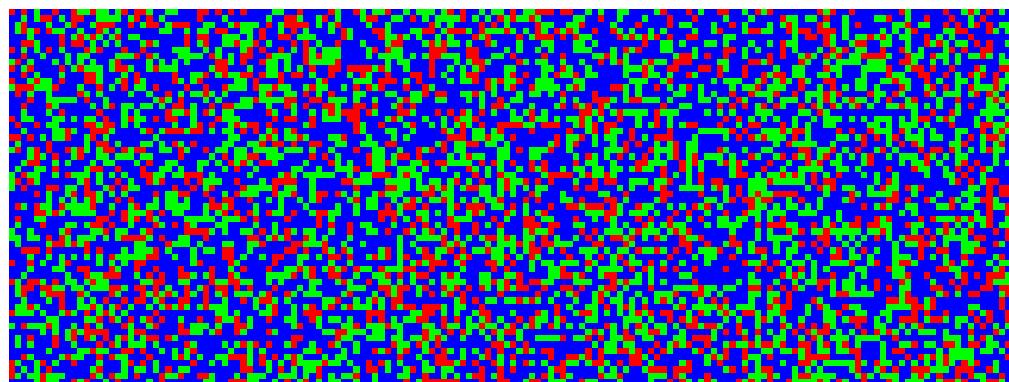


Figura 3.14: Figuras coloridas com diferentes probabilidades.

The screenshot shows the Processing IDE's console window with the following text:

```

Número total de quadrados: 9600
Percentual de quadrados vermelhos: 19.833334%
Percentual de quadrados verdes: 30.125%
Percentual de quadrados azuis: 50.041668%

```

Figura 3.15: Percentual de quadrados de cada cor.

No caso específico de probabilidades não é vantajoso usar a função `noise()` dada que ela não fornece números verdadeiramente descorrelacionados e introduziria um viés na possibilidade de um evento ocor-

rer. Existem outros tipos de distribuições probabilísticas além da uniforme que você acabou de aprender, mas no final todas têm o objetivo de diminuir a monotonia através da manipulação de ocorrências.

## 3.5 | Sumário

---

Neste capítulo foi explicado como utilizar o Processing para gerar números aleatórios, tanto independentes quanto correlacionados. Quando usados para alterar o valor de variáveis parametrizadas no código elas originam uma variedade infinita de padrões a cada execução do seu programa. Brevemente, você aprendeu sobre:

- **Função random()**: Responsável por devolver números aleatórios totalmente descorrelacionados dos números gerados anteriormente. Pode ser usado para criar variações e efeitos bruscos ou repentinios.
- **Função noise()**: Permite gerar números aleatórios baseados em uma variação numérica. Quanto maior essa variação, maior a semelhança dessa função com a aleatoriedade pura. Quanto menor essa variação, menor a amplitude entre amostras, gerando distribuições suaves e naturais.
- **Probabilidades**: Importante para direcionar e moldar o fluxo ou característica do programa. Enfatiza elementos que devem ser repetidos com mais frequência e introduz eventos ocasionais capazes de impactar significativamente no resultado final da obra.

O principal ponto desta seção é que o caos, na arte gerativa, deve ser moldado e direcionado até o ponto que você deseje que ele atue. Uma vez que esse ponto é atingido você deve aceitá-lo e deixar que ele se expanda através de uma dança anárquica que florescerá em imagens, padrões ou figuras verdadeiramente únicas e belas.

No próximo capítulo você descobrirá como o oposto do que você acabou de estudar também pode ser surpreendentemente cativante. A matemática, personificação das leis que regem o universo, está muitas vezes oculta sobre mantos densos de fórmulas, letras e números, mas uma vez desvelada você compreenderá porque ela sempre fascinou a humanidade.



[ CAPÍTULO 4 ]:

# Ordem

A essência da realidade

A obsessão da humanidade em ordenar o mundo e responder a perguntas sobre a vida, o universo e tudo mais, a impulsionou a inventar<sup>1</sup> a mãe de todas as invenções, a matemática. Ela é tão absurdamente necessária na civilização moderna que está integralmente presente em todas as áreas que compõem nossa sociedade. Na física ela é a base para formalizar conceitos que se estenderem desde as menores partículas da mecânica quântica até as mais colossais estrelas do cosmos. Na química ela auxilia a entender a formação de elementos da tabela periódica assim como prever taxas em mecanismos de reações. Na biologia ela é responsável por modelos capazes de predizer ou descrever ocorrências naturais, como comportamentos e padrões de organismos ou mudanças populacionais. Muito abstrato para você? Então lembre-se que ela é a base do nosso sistema financeiro e usada sempre que você precisa seguir uma receita ou dizer as horas. Resumindo, é ela quem permite modelar os segredos e as belezas do universo através das ciências.

Curiosamente, a matemática por si própria, pura, contém muitas belezas que costumam passar despercebidas por nós. Existem aquelas relacionadas com o método e que remetem à simplicidade ou multiplicidade de provas, como é o caso do célebre Teorema de Pitágoras. Outras derivam de conexões entre diferentes áreas da matemática, tal qual a Equação de Euler, considerada a equação mais notável da matemática pelo físico Richard Feynman. Há, ainda, aquelas ligadas diretamente aos números, como a perpétua constante  $\pi$  ou a sequência de Fibonacci.

A estética, na programação artística, se apoia vigorosamente em um tipo de beleza mais direta e visual, proveniente de imagens, quase que místicas, nascidas de um amálgama de números e fórmulas. Felizmente, os artifícios empregados para essa tarefa passam longe do reino do esoterismo, sendo fortemente enraizados em diversas áreas da matemática. Neste capítulo você será convidado a mudar um pouco a sua

---

<sup>1</sup>Como o próprio nome sugere, a matemática é um estudo, um aprendizado. A questão principal sobre a origem da matemática, se ela é um constructo humano ou uma verdade universal, é um debate filosoficamente interminável que facilmente incorpora a tríade composta pelas ciências naturais, sociais e humanas. A palavra “inventar” foi escolhida apenas por conveniência.

visão sobre a matemática e entender como você pode domá-la, reinterpretando conceitos para aplicações relacionadas à arte gerativa. Você aprenderá como algumas fórmulas simples possuem vastas aplicações na computação e em sua vertente artística.

## 4.1 | Trigonometria

---

Vamos começar com um conceito cuja origem pode ser traçada há cerca de quatro mil<sup>2</sup> anos atrás com duas civilizações à frente de seu tempo. Os egípcios, fascinados pela forma piramidal, construiram tumbas monolíticas que, a olhos desatentos, parecem apenas um aglomerado de pedras empilhadas. No entanto a descoberta de documentos<sup>3</sup> datados da época do império médio dos egípcios revelaram fórmulas matemáticas, sugerindo que os mesmos cortavam essas grandes estruturas em triângulos e calculavam múltiplas propriedades geométricas, em especial o *Seked* um análogo à moderna cotangente de ângulos. A segunda civilização, a Babilônica, registrou, em uma coleção de tábuas de argila, conceitos relativos à matemática, sendo parte destinada a geometria e cálculos de áreas de triângulos e volumes de prismas. Astrônomos babilônicos documentavam, com detalhes, informações acerca de eclipses lunares, solares, movimentações planetárias e de estrelas, incluindo previsões sobre as mesmas<sup>4</sup>. Esses estudos antigos, em grande parte empíricos e voltados para um objetivo prático, seriam futuramente aperfeiçoados pelos gregos, formalizando a universal *trigonometria*, uma área da matemática que investiga relações envolvendo comprimentos e ângulos em triângulos. Ela possui aplicações na arquitetura, acústica, arte, música, eletrônica, estatística, astronomia, cinema e inúmeros outros segmentos. Nesta seção você será direcionado a focar em uma pequenina parte do todo, relacionada ao estudo de triângulos e desenhos de círculos e demais figuras geométricas.

Você deve se lembrar que no Processing existe a função `ellipse()` para desenhar círculos por completo, mas nós não estamos interessados *exatamente* nele. Nossa atenção deve ser voltada para os infinitos pontos que compõem a figura. As informações relativas as coordenadas x e y desses pontos podem ser obtidas através de relações trigonométricas e usadas para tarefas como criação de padrões visuais. Não se preocupe se você não se lembrar desses conceitos, vamos aplicar apenas a trigonometria básica, e quando digo básica, significa do ensino médio, não mais que isso. Começaremos pelo objeto de estudo, o círculo, que é uma figura fechada da geometria Euclidiana, composto pelas coordenadas cartesianas de seu centro (c) e o tamanho de seu raio (r). Para desenhar um círculo basta imaginar uma reta iniciada no

---

<sup>2</sup>Reves, G. E. (1954). "Outline of the history of trigonometry". School Science and Mathematics 53(2): pags.87-171.

<sup>3</sup>Papiro de Rhind e Papiro de Moscou

<sup>4</sup>Maor, Eli (1998). "Trigonometric Delights". Princeton University Press, New Jersey. ISBN-13: 978-0-69-115820-4

ponto central, cujo comprimento é igual ao raio, e depois girar<sup>5</sup> essa reta 360° em torno desse ponto. Ele é ilustrado na figura 4.1.

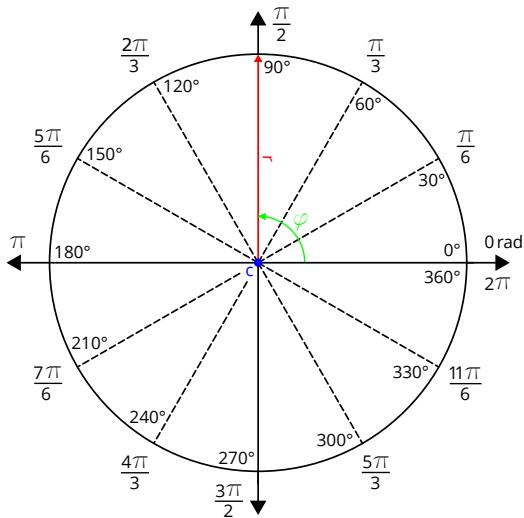


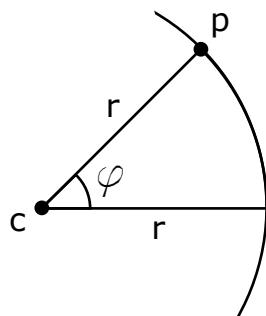
Figura 4.1: Um círculo e suas divisões.

Por outro lado, se em vez de completar 360° para gerar um círculo, você girar um ângulo qualquer, representado pela letra grega  $\phi$  ( $\phi$ ), você terá um arco dele, como o da figura 4.2a. Nosso principal objetivo é encontrar as coordenadas dos infinitos pontos localizados na superfície ou borda do círculo. Isso pode ser feito se traçarmos triângulos dentro do arco formado pelos pontos de interesse da figura, veja a imagem 4.2b.

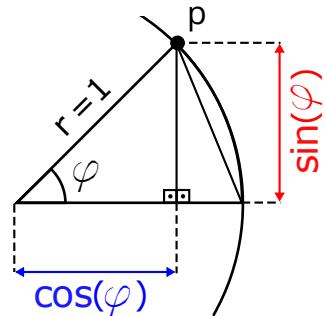
Após manipulações matemáticas usando as funções seno e cosseno, obtemos as equações<sup>6</sup> que fornecem as posições x e y de qualquer ponto localizado na superfície de um círculo.

<sup>5</sup>Na matemática convencional os ângulos crescem de valor no sentido de giro anti-horário, enquanto no Processing isso se sucede no sentido horário.

<sup>6</sup>Estas talvez sejam as fórmulas mais importantes apresentadas neste livro. Elas são definidas como equações do círculo, mas podem ser usadas para modelar eventos periódicos, simular movimentos suaves e são a base de grande parte dos cálculos relacionados ao trabalho com visualizações bidimensionais.



(a) Arco e ponto no círculo.



(b) Propriedades trigonométricas.

Figura 4.2: Setores em um círculo.

#### Código 4.1 Equações do Círculo

```
// Equações para obter as coordenadas de um ponto localizado na
// extremidade de um arco de ângulo φ graus de um círculo de
// centro (centroX,centroY) e raio r.
x = centroX + raio*cos(angulo);
y = centroY + raio*sin(angulo);
```

Estas equações, apesar de matematicamente simples, são muito poderosas como você irá ver. Antes de colocá-las em prática existe um último ponto que você deve saber. As funções `sin()` e `cos()` do Processing trabalham com ângulos em radianos em vez de graus. A transformação entre essas duas grandezas é feita através da fórmula abaixo:

$$\text{Ângulo (radianos)} = \text{Ângulo (graus)} * \frac{\pi}{180}$$

Felizmente o Processing nos ajuda com a função `radians()` que faz a conversão de graus para radianos. Finalmente, unindo o que foi explicado, você pode desenvolver um código para desenhar um círculo parametrizado, usando fórmulas em vez de uma função. Na teoria, um círculo é composto de infinitos pontos, mas na prática isso é inviável dado que o computador é um sistema digital discreto. Contornaremos essa limitação dividindo o círculo em um número arbitrário de pontos que serão conectados através de retas, código 4.2.

#### Código 4.2 Círculo Paramétrico

```
void setup() {
```

```

size(200,200);
background(255);

// Coordenadas do centro do círculo e valor do raio:
float centroX = 100, centroY = 100;
float raio = 50;

// O círculo será desenhado através de linhas, precisamos dos pontos anteriores:
float xAnterior = centroX + raio*cos(0);
float yAnterior = centroY + raio*sin(0);

// Número de pontos que irão compor o círculo:
int div = 5;          // Pentágono
//int div = 10;        // Decágono
//int div = 50;        // Pentacontágono

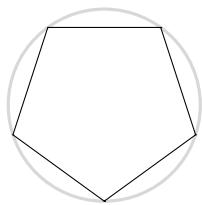
// Círculo parametrizado:
for(int i = 0; i < div + 1; i++) {
    float x = centroX + raio*cos(radians(i*360.0/div));
    float y = centroY + raio*sin(radians(i*360.0/div));

    line(x,y,xAnterior,yAnterior);

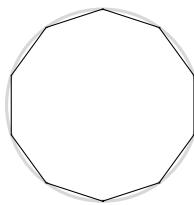
    xAnterior = x;
    yAnterior = y;
}

// Círculo pelo Processing:
noFill();
stroke(0,40);
strokeWeight(2);
ellipse(centroX,centroY,2*raio,2*raio);
}

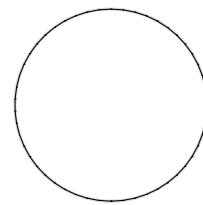
```



(a) 5 pontos.



(b) 10 pontos.



(c) 50 pontos.

Figura 4.3: Círculos desenhados com diferentes densidades.

Neste código usamos a estratégia de dividir o desenho do círculo em setores, cada um com um tamanho de arco que depende da variável `div`, sendo marcado por um ponto na superfície da figura. Perceba que quanto maior da divisão do círculo, figura 4.3, que equivale a um passo menor no ângulo dos arcos, mais ele se parece com um círculo em vez de outro polígono regular. Cinco divisões geram um pentágono, enquanto cinquenta divisões criam um pentacontágono (50 lados) que, na prática, é uma ilusão de um círculo perfeito.

## 4.2 | Beleza matemática

---

Desenhar círculos através de fórmulas matemáticas é algo incrível, pois mostra como é possível transcrever uma forma natural em uma estrutura rigorosamente numérica. Por outro lado, do ponto de vista artístico, essas figuras não trazem nada de inovador... ainda. A mágica acontece quando conceitos simples são utilizados de uma maneira não convencional e inesperada, e é assim que começamos de fato a mergulhar na programação artística. Para ilustrar essa ideia vamos fazer um pequeno exercício mental ou, se você preferir, faça no papel:

Imagine um círculo que é desenhado ponto a ponto até completar os  $360^\circ$ , conforme visto no código 4.2, mas apenas um ponto por frame de animação. Agora imagine um segundo círculo desenhado dessa mesma forma, mas a partir de um ponto defasado de  $180^\circ$  do primeiro e com a metade do tamanho do passo dele. Por último, visualize uma reta ligando esses dois pontos desenhados quadro a quadro. Esse processo é demonstrado na imagem 4.4. Que figura final você formará quando ligar todos os pontos desses dois círculos?

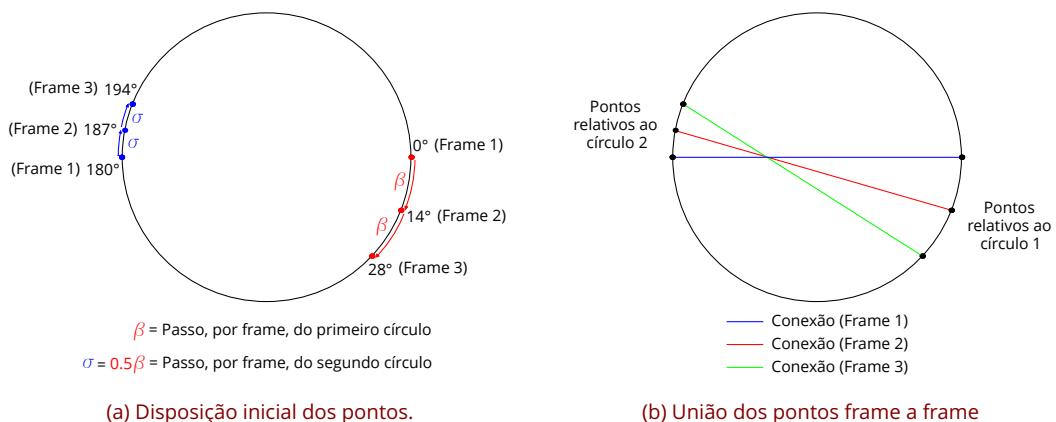


Figura 4.4: Exercício de conexão entre pontos de círculos.

Você deve ter percebido que é muito difícil imaginar o resultado e isso ocorre simplesmente porque somos humanos. Este exercício exige uma capacidade de abstração imensa, assim como de memória para gravar o que acontece a cada quadro ou a cada novo ponto que é desenhado. Mas veja que, conceitualmente, este é um exercício simplório e que retas ligando círculos não devem produzir nada de fantástico, apenas retas, correto? Pois bem, você pode tirar a dúvida escrevendo um programa que desenha exatamente a atividade proposta. A janela de saída está mostrada na figura 4.5.

#### Código 4.3 Conectando dois círculos

```
void setup() {  
    size(400,400);  
    background(255);  
    frameRate(20);  
}  
  
void draw() {  
    float centroX = 200, centroY = 200;  
    float raio = 160, vel = 0.5;  
  
    int div = 50;  
    // Passo, ou quantos graus serão "percorridos" por frame do Processing:  
    float passo = 360.0/div;  
  
    // Primeiro Círculo:  
    float x1 = centroX + raio*cos(radians(frameCount*passo));  
    float y1 = centroY + raio*sin(radians(frameCount*passo));  
  
    // Segundo Círculo - Defasado 180° e animado com metade do passo do primeiro:  
    float x2 = centroX + raio*cos(radians(180 + vel*frameCount*passo));  
    float y2 = centroY + raio*sin(radians(180 + vel*frameCount*passo));  
  
    strokeWeight(2);  
    // Pontos vermelhos: Círculo 1.  
    stroke(255,0,0);  
    point(x1,y1);  
    // Pontos azuis: Círculo 2.  
    stroke(0,0,255);  
    point(x2,y2);  
  
    // Conexão entre os pontos dos dois círculos:  
    strokeWeight(0);  
    stroke(0,255,0);  
    line(x1,y1,x2,y2);  
}
```

Neste código, a função `draw()` é responsável por desenhar apenas um ponto de cada círculo por frame. O grande auxiliador nesse processo é a variável `frameCount`, que é automaticamente e continuamente

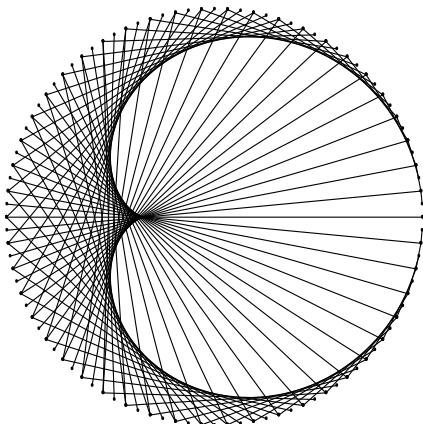
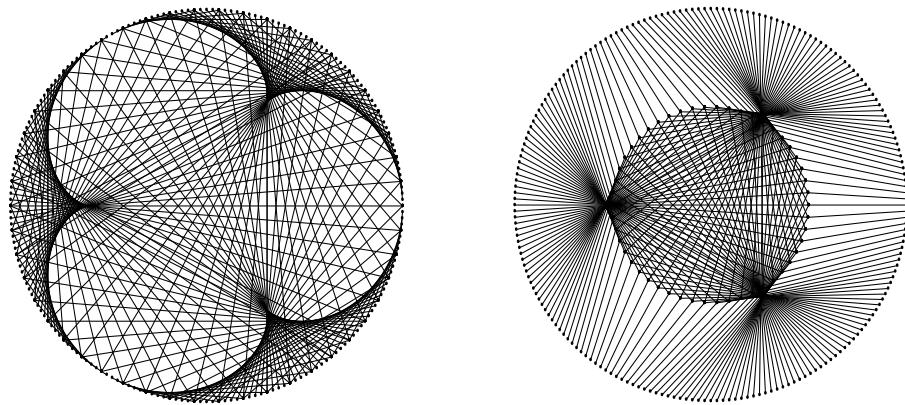


Figura 4.5: Pontos de dois círculos conectados através de retas.

incrementada uma vez que contém o número de frames decorridos desde o início da execução do programa. Outros padrões são prontamente construídos se alterarmos o valor da velocidade de rotação ou raio de um dos círculos, exemplificados na figura 4.6.

Com base nas figuras geradas podemos concluir que a combinação de conceitos simples pode resultar em algo muito mais complexo do que aplicá-los de maneira separada. Em nosso estudo de caso em específico isso é ilustrado pelas imagens hipnotizantes em forma de mandala geradas pelas regras que estabelecemos. A beleza matemática desse e de outros padrões desenhados dificilmente poderia ser prevista e, se fosse desenhada à mão, levaria um longo tempo para ser completada. São necessários um total de 150 pontos e 100 retas para desenhar a figura 4.5, sem contar o tempo para marcar os pontos de divisão do círculo. O computador é capaz de completar esses passos em décimos de segundo se não for necessário animar a tela. Esta é a grande vantagem de usar a computação para experimentar com a arte. Os padrões podem ser gerados, repetidos e alterados de maneira rápida e com um custo insignificante.



(a) Velocidade de rotação diminuída para 0.25.

(b) Raio externo com metade do valor do interno.

Figura 4.6: Padrões formados através de alterações na velocidade e raio.

#### 4.2.1 | Pequenos acidentes favoráveis

O famoso pintor Bob Ross, adorado por fãs dentro e fora da internet por sua calma e seu estilo de pintura prática, casualmente pronunciava a frase “*happy little accidents*”<sup>7</sup> sempre que pinçelava de uma maneira “errada”. Nesse momento pessoas que assistiam a seus vídeos se contorciam em desespero, acreditando que pintura estaria arruinada. Com uma calma digna de um mestre Zen, Bob subvertia esses acidentes e inventava uma maneira de corrigir o erro, pintando um outro objeto no lugar ou aplicando um efeito inesperado na paisagem.

Na programação artística podemos traçar um paralelo dessa mentalidade no sentido que *erros*, nem sempre são erros, e sim uma maneira de explorar novas ideias que antes não haviam sido pensadas. Por exemplo, os padrões gerados na seção passada são altamente precisos no contexto em que você programou uma reta que conecta a coordenada x do círculo 1 com a do círculo 2 e a coordenada y do círculo 1 com a do círculo 2. Mas e se você tivesse cometido um *erro* e na hora de digitar você, equivocadamente, trocasse os pontos ou índices, sua arte estaria arruinada? Novamente, uma vantagem da programação é que você pode testar essas ideias de maneira ágil e com recursos e custos de implantação materialmente irrisórios: o seu tempo e criatividade. Se você não gostar, basta alterar uma única linha de código para

<sup>7</sup>Aproximadamente traduzido para o título desta seção.

desfazer o que você fez. Para ilustrar esse conceito, localize as linhas abaixo no código 4.3:

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

E altere para um dos casos abaixo:

```
// Inversão dos índices dos círculos:
point(x1,y2);
point(x2,y1);
line(x1,y2,x2,y1);
```

ou

```
// Inversão das coordenadas x e y:
point(x1,x2);
point(y1,y2);
line(x1,x2,y1,y2);
```

Os padrões gerados estão mostrados nas figuras 4.7 e 4.8. Perceba como mudanças muito pequenas, como a simples troca de duas variáveis, causam impactos significativos no comportamento do programa. No caso deste exemplo são geradas novas variantes de padrões impossíveis de serem reproduzidas apenas alterando a posição inicial dos círculo ou o tamanho do passo, por frame, do ângulo. Esta é uma lição que demonstra perfeitamente que mesmo *erros* podem ser usados como fonte de aprendizado, aperfeiçoamento e inspiração.

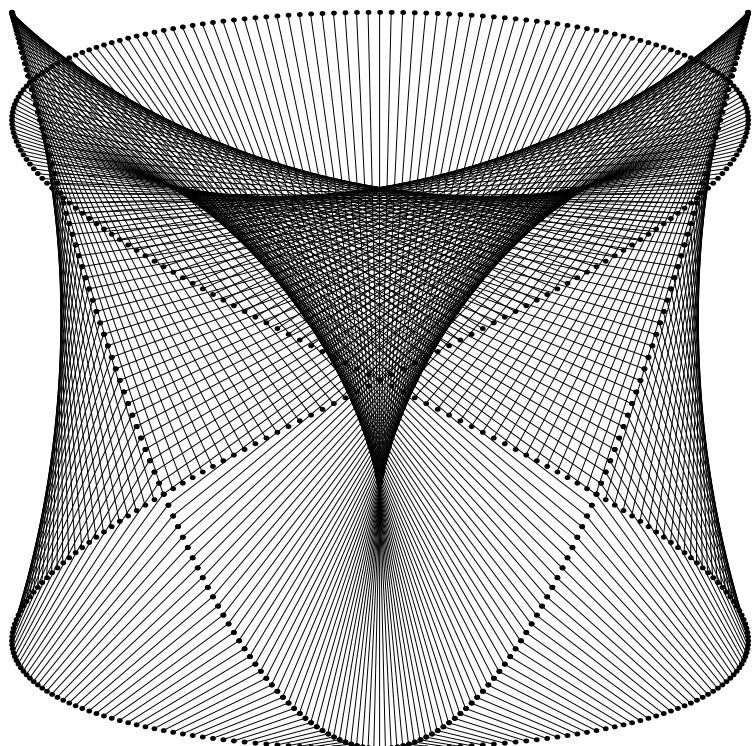


Figura 4.7: Conexão `line(x1,y2,x2,y1)`,  
girado em 90°.

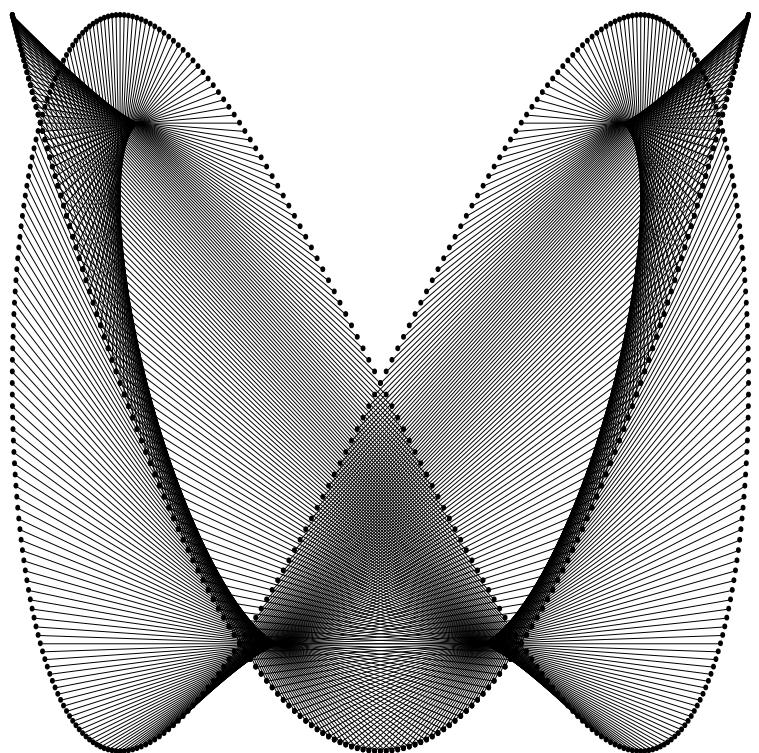


Figura 4.8: Conexão `line(x1,x2,y1,y2)`,  
girado em 90°.

## 4.3 | Simetria radial

---

Uma das classes de figuras provenientes da trigonometria são aquelas que apresentam a elegante simetria radial. Tal característica indica que um corpo pode ser cortado por um ou mais planos longitudinais que, quando atravessam seu centro, o dividem em partes iguais. Ao girar essa parcela em torno de um ponto central, ela é capaz de reconstruir o objeto original. Essas formas são abundantemente encontradas na natureza em animais, flores e frutas, veja a seção de uma laranja na figura 4.9.

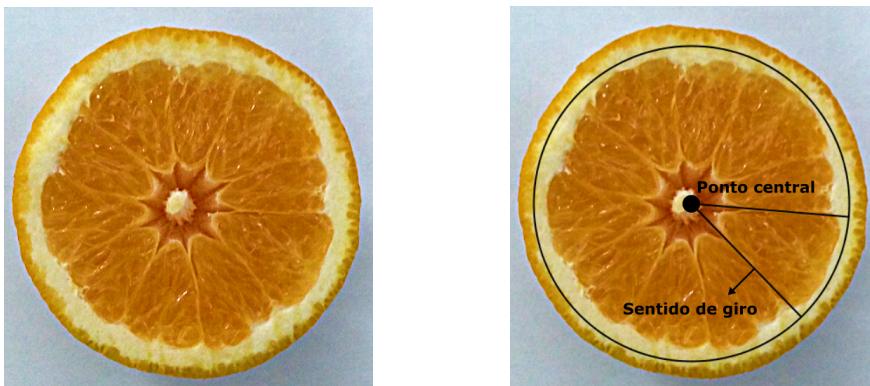


Figura 4.9: Seção de uma Laranja, apresentando decamerismo ou uma simetria de ordem 10.

As equações 4.1 foram batizadas como as equações do círculo, mas elas também podem ser usadas para desenhar uma variedade de figuras que apresentam esse tipo de simetria. Isso se deve ao fato delas serem desenhadas nos  $360^\circ$  em torno de um ponto central, semelhante a um círculo. Uma estrela por exemplo, imagem 4.10, é uma figura radialmente simétrica.

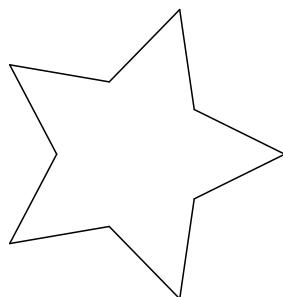
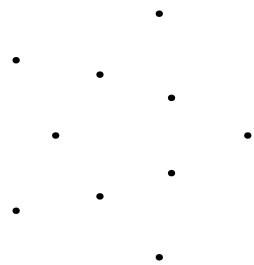
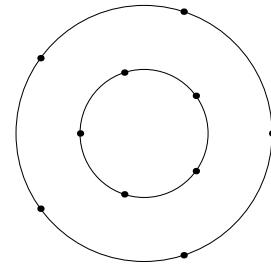


Figura 4.10: Estrela de cinco pontas.



(a) Apenas pontos que compõem a estrela.



(b) Pontos circunscritos em círculos.

Figura 4.11: Figura de uma estrela decomposta.

Observe que um estrela é composta por pontos e linhas contidas dentro de um círculo, sugerindo que ela pode ser gerada pelas fórmulas 4.1. É mais fácil entender o processo por trás desse tipo de figura se removermos as linhas de contorno como na figura 4.11.

A estrela é formada por dez pontos, dos quais alguns estão na borda de um círculo externo de raio  $r_e$  e outros estão na parte interna, em um círculo cujo raio  $r_i$  é a metade do valor do primeiro. Você pode escrever um código para desenhar essa disposição de pontos utilizando uma estrutura de repetição semelhante a usada no início deste capítulo para desenhar um círculo. No entanto será preciso incluir um condicional para alternar o desenho dos pontos entre o círculo externo e interno. Sempre que um ponto for desenhado com raio  $r_e$ , o seguinte será desenhado com  $r_i$ . O código a seguir dispõe os pontos como na figura 4.11 e depois os conecta.

```
void setup() {
    size(300,300);
    background(255);

    // Coordenadas dos pontos que compõem a estrela de 5 vértices:
    float[] x = new float[10];
    float[] y = new float[10];

    for(int i = 0; i < 10; i++) {
        float r = 0;
        // Se o ponto for par temos o raio externo, se ímpar, o interno:
        if(i%2 == 0) {
            r = 100;
        }
        else {
            r = 50;
        }
        x[i] = r * sin((i * 2 * PI) / 5);
        y[i] = r * cos((i * 2 * PI) / 5);
    }
}
```

```

    }

    x[i] = width/2 + r*cos(radians(i*36));
    y[i] = height/2 + r*sin(radians(i*36));
}

// Desenho da estrela - Conecta, sequencialmente, os pontos gerados:
for(int i = 1; i < 10; i++) {
    line(x[i-1],y[i-1],x[i],y[i]);
}
// A última ligação deve ser feita fora da repetição,
// unindo o primeiro com o último ponto:
line(x[0],y[0],x[9],y[9]);
}

```

O condicional que testa se a variável *i* é par ou ímpar já foi explicado na seção 2.5.2, e é reutilizado aqui, pois sequencialmente sempre existirá um número ímpar depois de um par. Como queremos que seja desenhado, alternadamente, um ponto com raio *re* e outro com raio *ri*, é conveniente utilizar essa estratégia para posicionar os pontos.

Agora que o conceito de figuras radiais foi desmistificado, é possível incorporar algumas funcionalidades para deixá-las menos triviais. O primeiro passo é a parametrização das variáveis que mais impactam na forma da figura, como o número de vértices da estrela e os raios internos e externos dos círculos que a compõem. Isso permite adicionar variedade as formas com um simples mudar de valores, de uma maneira rápida e simplificada. O segundo ponto é animar a exibição da estrela através de um movimento de giro que, em uma figura radial, consiste em incrementar, quadro a quadro, o ângulo de todos os pontos que são desenhados. A velocidade de giro será diretamente proporcional ao incremento desse ângulo e, durante a execução, todos os pontos serão afetados, criando a ilusão de movimento da figura. Veja a implementação dessas funcionalidades no código abaixo e os resultados na figura 4.12.

#### Código 4.8 Estrela animada

```

// Número de pontas e pontos da estrela:
int pontas = 6, nbr = 2*pontas;
// Raio externo e interno:
float re = 100, ri = 0.5*re;
// Giro da estrela (animação):
float incremento = 0;

void setup() {
    size(300,300);
    background(255);
}

void draw() {

```

```

background(255);

// Coordenadas dos pontos que compõem a estrela:
float[] x = new float[nbr];
float[] y = new float[nbr];

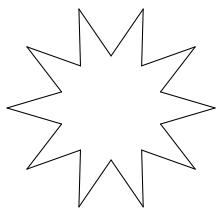
for(int i = 0; i < nbr; i++) {
    float r = 0;
    // Se o ponto for par temos o raio externo, se ímpar, o interno:
    if(i%2 == 0) {
        r = re;
    }
    else {
        r = ri;
    }

    x[i] = width/2 + r*cos(incremento + radians(i*360.0/nbr));
    y[i] = height/2 + r*sin(incremento + radians(i*360.0/nbr));
}

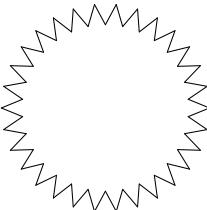
// Desenho da estrela - Conecta, sequencialmente, os pontos gerados:
for(int i = 1; i < nbr; i++) {
    line(x[i-1],y[i-1],x[i],y[i]);
}
// A última ligação deve ser feita fora da repetição,
// unindo o primeiro com o último ponto:
line(x[0],y[0],x[nbr-1],y[nbr-1]);

// Aumenta o ângulo de incremento para criar a ilusão de giro da figura:
incremento += 0.01;
}

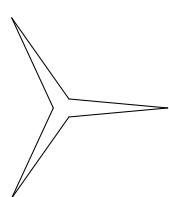
```



(a) 10 Pontas e  $ri = 0.5 * re$



(b) 30 Pontas e  $ri = 0.8 * re$



(c) 3 Pontas e  $ri = 0.1 * re$

Figura 4.12: Figuras das estrelas parametrizadas.

A animação pode se tornar mais dinâmica se você alterar de modo desproporcional características que mudem significativamente a forma da figura. Experimente incrementar gradativamente o ângulo de de-

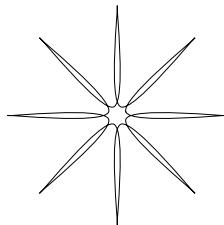
senho de todos os pontos internos *ou* externos da estrela. Realizar essa soma apenas quando  $i$  é par ou ímpar é uma maneira de se obter este efeito. Substitua as linhas:

```
x[i] = width/2 + r*cos(incremento + radians(i*360.0/nbr));  
y[i] = height/2 + r*sin(incremento + radians(i*360.0/nbr));
```

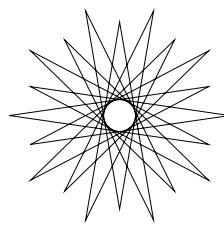
por:

```
x[i] = width/2 + r*cos(incremento + radians(i*360.0/nbr + (i%2)*frameCount));  
y[i] = height/2 + r*sin(incremento + radians(i*360.0/nbr + (i%2)*frameCount));
```

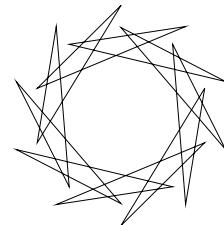
Usando somente derivações do código 4.8 você é capaz de criar uma quantidade surpreendente de figuras cuja variedade pode ser multiplicada se você utilizar números negativos em raios e defasagens em ângulos ou conectar os pontos através de curvas em vez de retas. Os resultados de parte dessas ideias são mostrados na figura 4.13. Combinando todas essas estratégias e sobrepondo figuras radiais você pode criar padrões similares à mandalas, veja a figura 4.14.



(a) Utilizando curvas.



(b) Raio interno negativo.



(c) Desfasagem entre pontos.

Figura 4.13: Variedade de formas de estrelas.

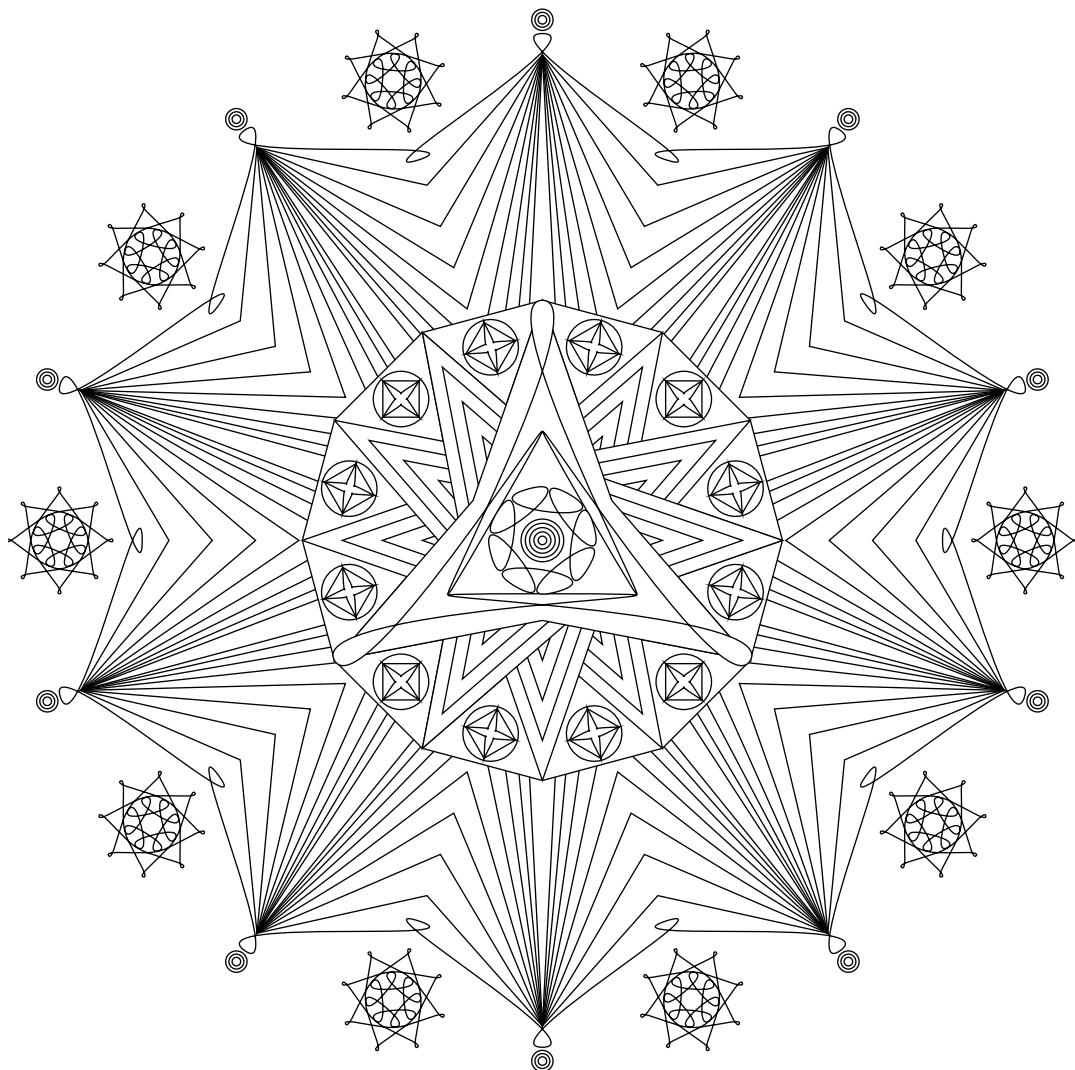


Figura 4.14: Mandala formada por figuras radialmente simétricas.

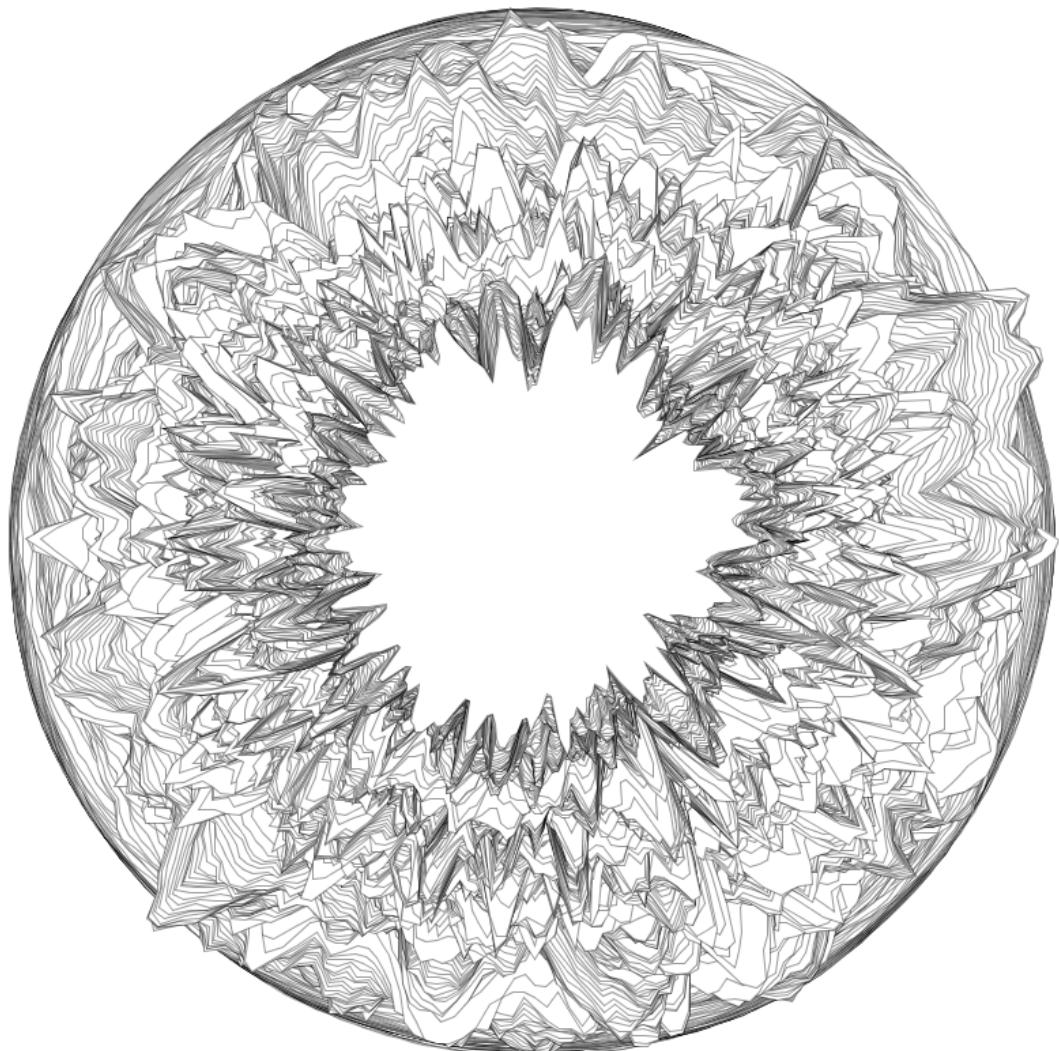


Figura 4.15: Sobreposição de figuras radiais com vértices aleatórios.

## 4.4 | Espiral

---

Aproveitando a caráter “circular” deste capítulo, existe mais uma forma inspirada na natureza que devemos estudar, a espiral. Diferentemente do que se possa imaginar, o principal propósito da espiral não é a estética, é a eficiência<sup>8</sup>. Essa estrutura é responsável por minimizar custos de voo de caça em falcões<sup>9</sup> ou maximizar a densidade de sementes em plantas<sup>10</sup>. Entretanto nada nos impede de apreciar os belíssimos espécies vegetais e animais que a tomam emprestada, figura 4.16.



(a) Filotaxia espiralada em uma planta.



(b) Concha de um caracol<sup>a</sup>, cortesia do fotógrafo Jon Mace.

<sup>a</sup>Todos os direitos reservados pelo fotógrafo original.

Figura 4.16: Espirais na natureza.

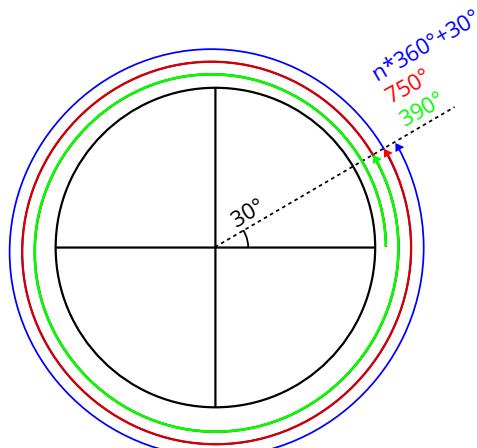
Analisando esse padrão tecnicamente percebemos que ele se difere de um círculo uma vez que é composto por um serpenteamento duradouro em torno de um ponto central. Isso nos leva a concluir dois fatos, que o “raio” de uma espiral não é fixo, continuamente crescendo (ou decrescendo) a cada volta da figura, e que ela requer múltiplas revoluções em torno do ponto central, ultrapassando o intervalo de  $0^\circ$  a  $360^\circ$  do círculo. Este último ponto é facilmente contornado, já que as funções seno e cosseno são periódicas e se repetem a cada  $360^\circ$ . Matematicamente temos que o valor do seno ou do cosseno de um ângulo qualquer

<sup>8</sup>Provavelmente, porém não provado. A natureza raramente é uma ciência exata e muito menos escrita em fórmulas explícitas disponíveis para consulta. O que existe, de fato, são hipóteses que revelam que quando certas estruturas matemáticas são usadas, vantagens evolutivas seriam asseguradas.

<sup>9</sup>Tucker, V. A. et al (2000). Curved flight paths and sideways vision in peregrine falcons (*Falco peregrinus*). *The Journal of Experimental Biology* 203(24): pags.3755-3763.

<sup>10</sup>Ridley, J. N. (1982). Packing efficiency in sunflower heads. *Mathematical Biosciences* 58(1): pags.129-139.

será absolutamente igual ao desse ângulo somado a um múltiplo inteiro de  $360^\circ$ . Por exemplo, o seno de  $30^\circ$  é igual ao de  $390^\circ$  ( $30^\circ + 360^\circ$ ) ou  $750^\circ$  ( $30^\circ + 720^\circ$ ) ou, generalizando,  $30^\circ + n \cdot 360^\circ$  (onde  $n$  é um número inteiro), veja a figura 4.17.



$$\begin{aligned}\sin(30^\circ) &= \sin(390^\circ) = \sin(750^\circ) = \sin(n \cdot 360^\circ + 30^\circ) = 0.5 \\ \cos(30^\circ) &= \cos(390^\circ) = \cos(750^\circ) = \cos(n \cdot 360^\circ + 30^\circ) = 0.866 \\ n &= 1, 2, 3, 4...\end{aligned}$$

Figura 4.17: Múltiplas voltas em um círculo.

Quando desenhamos um círculo evitamos ultrapassar esse limite, pois estariamos revisitando uma área desenhada anteriormente no intervalo de  $0^\circ$  a  $360^\circ$ . Entretanto, essa propriedade é muito poderosa e permite que contornemos infinitamente um ponto central, justamente uma das necessidades para transformar o desenho de um círculo em uma espiral. Para completá-la, impomos a condição de aumentar progressivamente o raio original do círculo, impedindo permanentemente o fechamento da forma. O código abaixo gera a figura 4.18.

```
void setup() {
    size(300,300);
    background(255);

    // Raio inicial e ângulo girado a cada iteração:
    float r = 0, passo = 5, numVoltas = 15;

    // Número de repetições:
    float numRep = 360*(numVoltas/passo);
```

```

// Ponto atual e anterior:
float x, y, xa = width/2, ya = height/2;

for(int i = 0; i < numRep; i++) {
    x = width/2 + r*cos(radians(i*passo));
    y = height/2 + r*sin(radians(i*passo));

    line(xa,ya,x,y);

    xa = x;
    ya = y;
    r += 0.1;
}
}

```

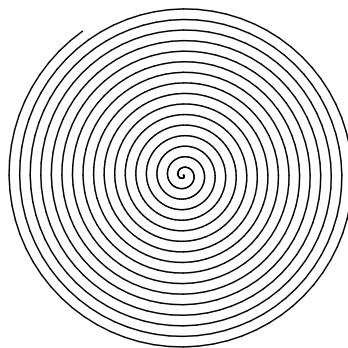
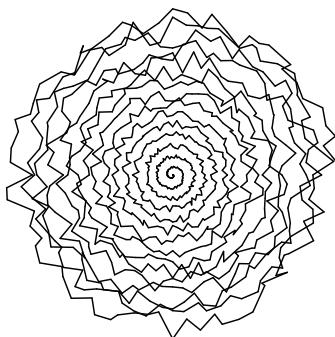


Figura 4.18: Espiral.

Vamos finalizar esta seção unindo nossas equações matemáticas determinísticas com o acaso não determinístico do capítulo passado. A escolha do “onde”<sup>11</sup> exatamente inserir o aleatório não é padronizável, mas por certo podemos analisar nossas equações e experimentar aplicá-lo em parâmetros que causarão grande impacto na imagem final. Para o caso de figuras radiais, uma possibilidade é decidirmos que queremos uma imagem sempre centrada e com passos regulares de ângulo para manter uma consistência na visualização. Segundo as equações trigonométricas 4.1, isso significa manter as variáveis centroX, centroY e angulo fixas. O único fator restante para atuar é no raio do círculo. Utilizando o código anterior

---

<sup>11</sup>No capítulo 3 é equivalente a questão do *o que* parametrizar.



(a) Função `random()`.



(b) Função `noise()`.

Figura 4.19: Espirais sob o efeito do aleatório.

como base, nossa espiral pode ser transformada para uma configuração mais rebelde, como as das figuras 4.19a e 4.19b. Altere as linhas:

```
x = width/2 + r*cos(radians(i*passo));  
y = height/2 + r*sin(radians(i*passo));
```

usando a função `random()` para gerar um efeito turbulentó:

```
x = width/2 + random(0.9,1.1)*r*cos(radians(i*passo));  
y = height/2 + random(0.9,1.1)*r*sin(radians(i*passo));
```

ou `noise()` para um mais natural:

```
x = width/2 + 1.5*noise(i/30.0)*r*cos(radians(i*passo));  
y = height/2 + 1.5*noise(i/30.0)*r*sin(radians(i*passo));
```

Perceba como entrelaçar da ordem e do caos é simbótico ao se alimentar da complementariedade de cada uma das partes. As equações matemáticas fornecem a base, ou a *forma* geral da figura, e a aleatoriedade contribui ao infiltrar o gerativo naquilo que seria predeterminado. O resultado é uma estrutura modular com incontáveis detalhes que se alteram cada vez que ela é gerada.



Figura 4.20: Baseado em: "Autorretrato" de Rembrandt van Rijn, cortesia<sup>b</sup> da National Gallery of Art, Washington D.C..

<sup>b</sup>Todos os direitos reservados pela instituição.

## 4.5 | Sumário

---

A trigonometria pode ser aplicada em uma série de tarefas na computação. O foco deste capítulo foi apresentar as poderosas equações paramétricas do círculo que, dentre outras aplicações, são usadas para desenhar polígonos regulares, figuras radiais e ainda espirais. É importante que você se familiarize e entenda, principalmente do ponto de vista geométrico, o que é possível realizar com as funções seno e cosseno. Adicionando a aleatoriedade aos códigos, geramos imagens erráticas, mas estruturadas, reforçando o papel do acaso no conjunto da obra.

No próximo capítulo é mostrado como empregar a formatação de maneira inteligente para enriquecer as exibições criadas por você em seus programas. Também será exemplificado como fundir todos os conceitos explicados até o momento para criar um projeto gerativo completo.



[ CAPÍTULO 5 ]:

# Ilusões

Valor nos detalhes

O processo artístico por trás de uma pintura, ilustração ou qualquer outro tipo de produção composta por múltiplas fases, normalmente começa com o planejamento. Grandes artistas usam e abusam dos rascunhos para delinear a forma inicial de uma ideia e, posteriormente, aprimorá-la. Nos quatro capítulos anteriores você foi direcionado para o criar dos esboços de sua obra através do posicionamento de figuras, controle do fluxo do desenho, repetições de padrões e inclusão de variedade através de elementos como a incerteza. Esses são os aspectos técnicos da composição artística, aqueles que precedem a arte.

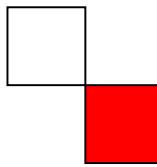
O próximo passo é a etapa de construção de estilo e refinamento da forma que consiste em adicionar riqueza visual, seja por meio de cores, profundidade de sombreamento, transparência ou complexidade dos traços. Neste capítulo você irá aprender como utilizar as funções do Processing para transformar rascunhos tediosos em produções intrigantes que atraem olhares.

É tentador procurar padronizar o máximo de tarefas na computação, mas isso dificilmente pode ser estendido para a arte. A interpretação de uma obra é algo muito pessoal e depende tanto de nossa vivência pessoal quanto do contexto histórico e social da época. Consequentemente leve as informações apresentadas neste capítulo exatamente como elas são: apenas sugestões baseadas em práticas comuns na arte tradicional e computacional. Extrapole esse pensamento, entenda que toda regra tem uma exceção e, às vezes, quebrá-las causa mais efeito do que segui-las.

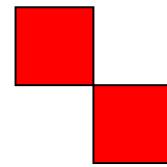
## 5.1 | Linhas etéreas

As formatações em seus programas são como um paralelo dos diversos materiais artísticos disponíveis para a customização do traçado de seu desenho. Espessuras de ponta de lápis e cores vibrantes de *crayon* podem ser emuladas diretamente com as funções nativas do Processing, enquanto pinceladas e depósitos irregulares de tinta costumam ser programados em aplicações mais elaboradas. Ao longo do livro você informalmente usou funções de estilo como:

- **fill()** Preenche a figura com uma cor qualquer.
- **noFill()**: A figura não possuirá preenchimento, ela será transparente.



(a) Primeiro frame da execução.



(b) Demais frames da execução.

Figura 5.1: Formatação como uma máquina de estados.

- **stroke()**: Preenche a borda da figura (ou uma linha/ponto) com uma cor qualquer.
- **noStroke()**: A borda da figura não possuirá preenchimento, ela será transparente.
- **strokeWeight()**: Define a espessura da borda da figura (ou de uma linha/ponto).

Rigorosamente, as ferramentas de formatação funcionam como máquinas de estado, isto é, a formatação que você estabelecer permanecerá como vigente até que você a mude de novo. Por exemplo, se você definiu um `fill(255,0,0)` no início do seu programa, todas as figuras que você desenhar serão preenchidas de vermelho até que você mude de cor chamando outra função `fill()`. A formatação é persistente ao ponto de não voltar para a original mesmo no início de um novo frame da função `draw()`. Você pode fazer uma analogia com o processo de desenhar no papel, uma vez que você decide usar um lápis vermelho, tudo que você desenhar será vermelho até que você decida usar outro lápis. O código abaixo e a figura 5.1 ilustram esse fenômeno:

```
void setup() {
    background(255);
    // O Processing irá gerar um quadro a cada 2 segundos,
    // desta forma você terá tempo para ver a formatação como uma máquina de estados.
    frameRate(0.5);
}

void draw() {
    // Exibe o número de quadros decorridos no console:
    println("Frame:", frameCount);
    rect(10,10,40,40);
    fill(255,0,0);
    rect(50,50,40,40);
}
```

Ao executar o código você deve ter observado no primeiro frame são exibidos dois quadrados, um branco e um vermelho e, no segundo frame, dois vermelhos. No quadro inicial não havia sido estabelecida uma formatação para o preenchimento das formas, `fill()`, e o Processing usou o padrão, que é a cor branca. Porém, no meio da execução do primeiro quadro a cor de preenchimento é definida como vermelho e logo

em seguida é desenhada uma forma com essa cor. Como o Processing usa de uma máquina de estados, quando ele parte para o segundo quadro a formatação de preenchimento já está definida e ele não usará mais o padrão original. Desta forma ambos os quadrados são preenchidos com a cor vermelha. Fique atento a essa particularidade visto que, se você desejar múltiplos estilos de formatação no seu programa, deverá colocar o respectivo código imediatamente antes do desenho das figuras.

Um dos fatores emulados pelas funções de estilo que contribui significativamente na aparência final de uma imagem é a suavidade dos traços usados nos desenhos. Essa formatação pode ser evidenciada através de um panorama saturado de formas, tais como círculos de posições e raios distintos. No código abaixo, no qual o ruído Perlin é o protagonista na disposição irregular das figuras, exploramos a suavidade dos traços:

```
// Ruídos:  
float nx = 0, ny = 100, nr = 200;  
  
void setup() {  
    size(1200, 500);  
    background(255);  
    noFill();  
}  
  
void draw() {  
    float posX = map(noise(nx),0,1,0,width);  
    float posY = map(noise(ny),0,1,0,height);  
    float r = 100 + 200*noise(nr);  
  
    // Círculo de posição e raio variados:  
    ellipse(posX,posY,r,r);  
  
    // Os incrementos de ruídos foram alterados  
    // inúmeras vezes até achar um movimento satisfatório.  
    nx += 0.008;  
    ny += 0.005;  
    nr += 0.005;  
}
```

A janela de saída se parecerá com algo similar a figura 5.2. Você deve ter notado que não há variação no traço do círculo o que torna a figura densa e sobrecarregada. Se o programa continuasse a executar por um pouco mais de tempo, obteríamos apenas um borrão preto proveniente da aglomeração de traços sólidos. Experimente deixar o programa rodando por um minuto e depois volte para ver os resultados.

Seguindo a linha de suavidade dos traços, podemos usar a função `strokeWeight()` com o intuito de reduzir a espessura dos mesmos. É possível aumentar ainda mais profundidade visual se adicionarmos transparência a cor preta, firmando que apenas o perseverante acúmulo de figuras através dos frames produza uma cor mais intensa. Adicione a linha abaixo logo antes do desenho do círculo.

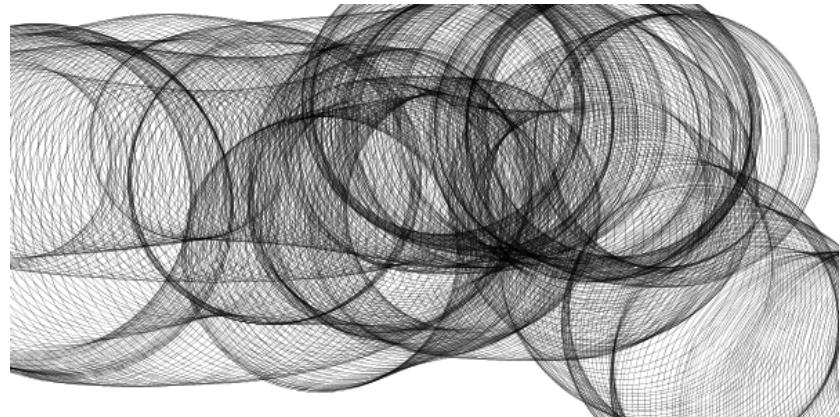


Figura 5.2: Desenhos de múltiplos círculos na janela.

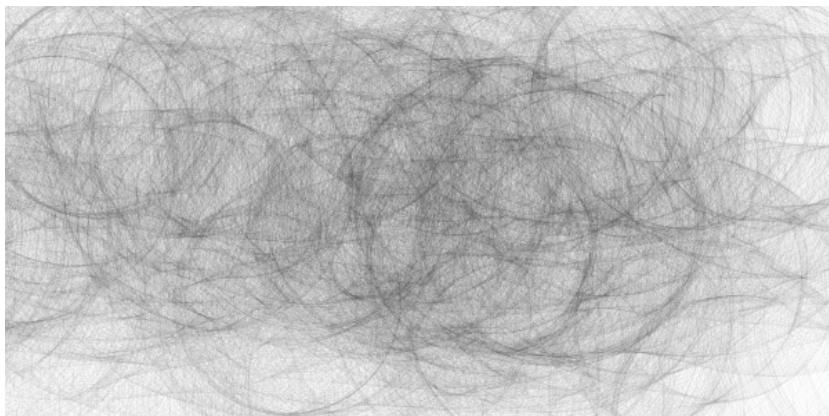
```
stroke(map(r,100,300,100,0),20);
```

Veja a diferença ao deixar os dois programas executarem exatamente o mesmo número de frames (8000) na figura 5.3.

Essa técnica leva a um aumento no tempo total de execução para que a figura seja “concluída”, situação facilmente relevada dada a sutileza das linhas que fabricarão uma imagem menos agressiva ao olhar.



(a) Formatação padrão do Processing.



(b) Formatação customizada.

Figura 5.3: Comparação da imagem gerada após 8000 frames.

## 5.2 | Faça-se a cor

---

As cores são outro grupo de elementos que fornecem um aspecto completamente novo as suas imagens. Na arte tradicional e na ilustração moderna elas costumam ser agrupadas em paletas de cores, que são um conjunto discreto e finito de tons como os da figura 5.4, idealizados para manter a consistência em seus projetos.

As paletas são a representação física de um esquema de cores que, por sua vez, é definido como uma escolha de cores a serem usadas no design de um determinado tipo de mídia. Eles são empregados para criar estilo e aumentar o atrativo e apelo estético de um trabalho. Existem diversas técnicas para a escolha de um esquema de cores, como complementares, triádicas, tetraédricas e análogas dentre outras, mostradas na figura 5.5.

A definição da paleta pode seguir regras estritas como as técnicas citadas acima ou então através de combinações não convencionais puramente decididas por sentimento. O instinto, nesse ponto, tem um papel substancial ao implicitamente externar como as cores influenciam nas emoções humanas, uma das áreas que é hoje estudada pela psicologia<sup>1</sup>. No Processing é possível escrever um código que emula uma paleta usando vetores:

```
// Cores definidas no sistema RGB:  
color[] c = { color(88,24,64), color(144,12,58), color(199,0,52), color(255,87,46) };
```

que é equivalente a:

```
// Cores definidas através de seu código hexadecimal:  
color[] c = { #581840, #900C3A, #C70034, #FF572E };
```

Esta segunda maneira é mais usual e utiliza de uma codificação hexadecimal das cores. A equivalência entre a descrição RGB e hexadecimal segue uma transformação entre bases numéricas e a ferramenta de seleção de cores, explicada na seção 2.1.3, automaticamente realiza esse cálculo para você. Veja a figura 5.6.

Do ponto de vista de código, é interessante criar uma paleta de cores como um vetor para acessá-las diretamente pelos seus índices. Tal abordagem segmenta o esquema de cores do código, aumentando a organização e permitindo sua troca sem que seja necessário reescrever o código. Também possibilita que seja utilizado da aleatoriedade para escolher as cores de maneira inesperada, veja o código abaixo:

---

<sup>1</sup>Valdez, P. e Mehrabian, A. (1994). "Effects of Color on Emotions". Journal of Experimental Psychology General 123(4): pags.394-409.

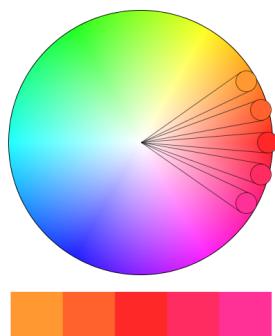


(a) Na arte.

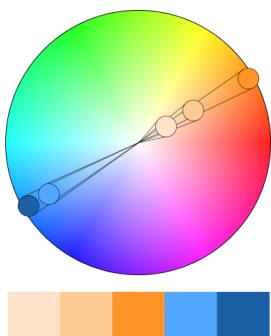


(b) Na computação.

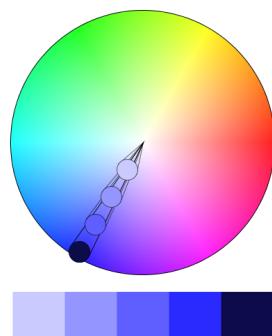
Figura 5.4: Exemplo de paletas de cores em meios distintos.



(a) Análogas.



(b) Complementares.



(c) Monocromáticas.

Figura 5.5: Esquema de cores.

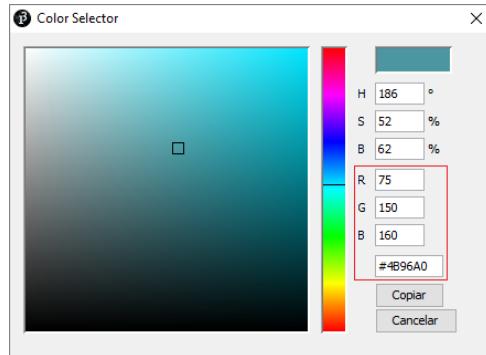


Figura 5.6: Equivalência RGB e Hexadecimal das cores.

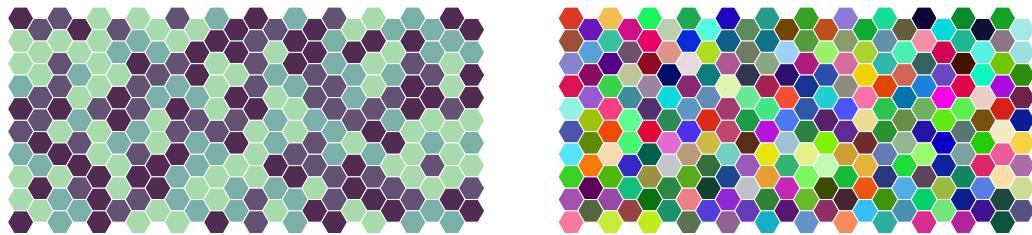


Figura 5.7: Paleta customizada e aleatória.

```
color[] c = { #581840, #9000C3A, #C70034, #FF572E };
// Cor escolhida aleatoriamente dentre todas as possibilidades da paleta:
int indice = round(random(3));
background(c[indice]);
```

Cabe ressaltar que a escolha aleatória dentro de um espaço finito de cores, como a de uma paleta, é fundamentalmente diferente de uma paleta completamente arbitrária uma vez que, no primeiro caso, o esquema é mantido e no segundo não. Veja as distinções na figura 5.7. A melhor maneira de expor o impacto causado por uma paleta de cores é através de uma comparação do tipo “antes e depois”. As figuras 5.8, 5.9 e 5.10 ilustram o mesmo programa executado, inicialmente através da formatação padrão e depois usando uma paleta de cores. A figura 5.11 é a colorização da mandala do capítulo anterior.

As diferenças são notáveis como você pode ter percebido. Os dois principais responsáveis pelo sucesso das paletas sobre um esquema de cores inteiramente aleatório ou a ausência delas são a limitação do espectro de cores e escolha cuidadosa das mesmas para permitir uma harmonização de tons e contrastes.

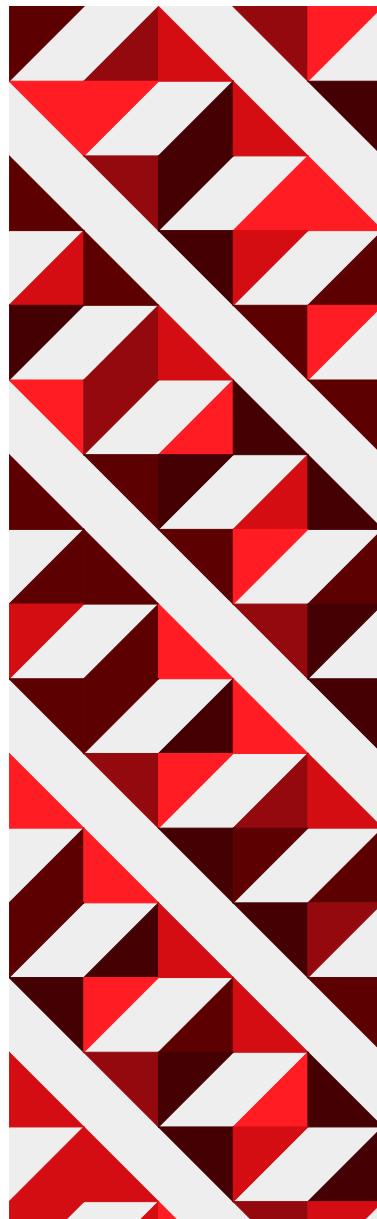
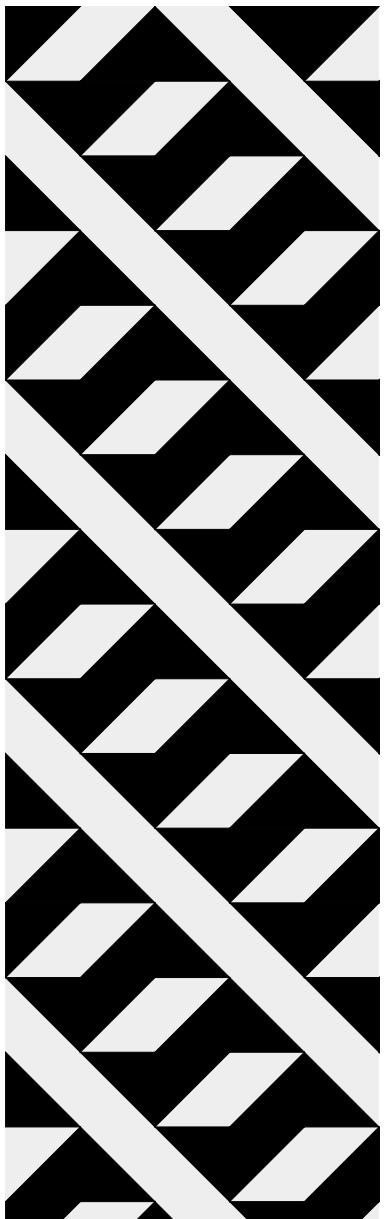


Figura 5.8: Padrão com e sem cores.

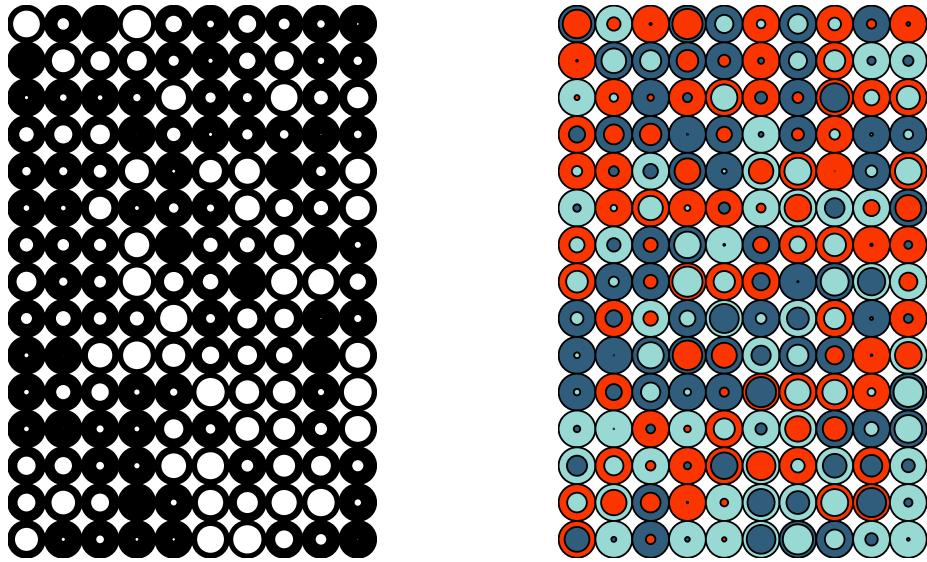


Figura 5.9: Padrão com e sem cores.

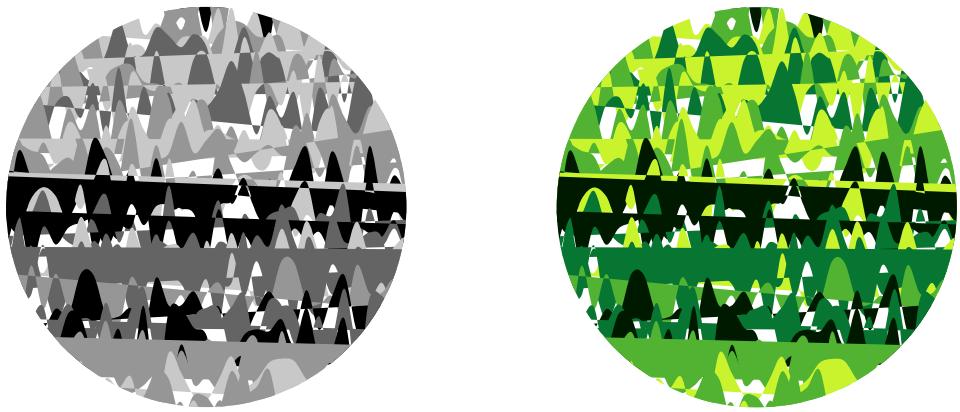


Figura 5.10: Padrão com e sem cores.

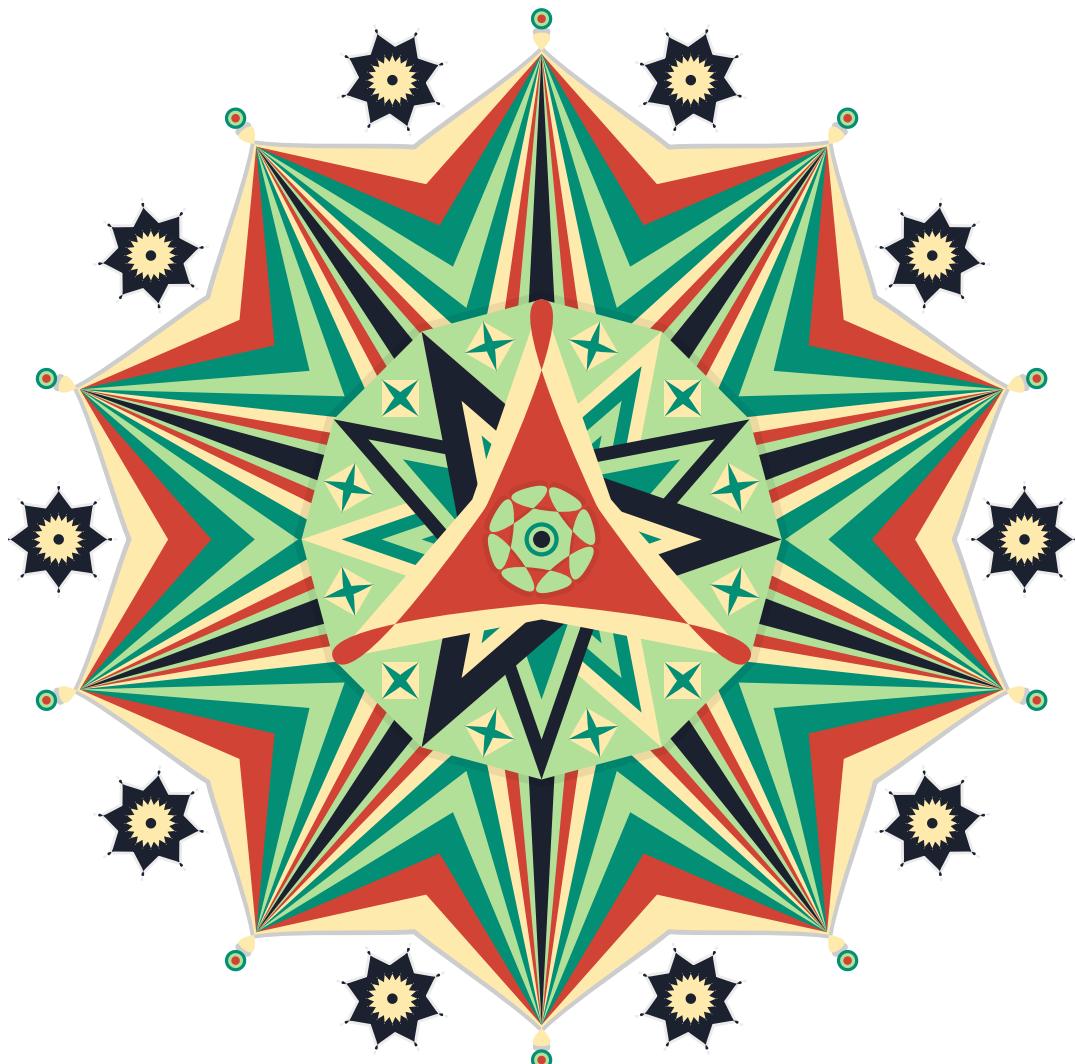


Figura 5.11: Mandala sob uma paleta de cores.

## 5.3 | Através do espelho

A simetria é uma propriedade excepcionalmente bela que reina em absoluto em nossa sociedade e no mundo natural. Sua presença é tão constante que a consideramos banal, muitas vezes esquecendo de admirar sua singela beleza e a sensação de equilíbrio e harmonia que ela proporciona. Na arte computacional a simetria pode ser usada para criar figuras caleidoscópicas, invariantes e espelhadas.

O conceito técnico por trás da simetria é simples: a forma é mantida mesmo sob presença de operações geométricas como reflexão, translação, rotação, ampliação e redução (figura 5.12). A simetria radial, como aquela mostrada na seção 4.3, é obtida a partir de rotações apoiadas na trigonometria, cujo posicionamento das figuras utilizará das funções seno e cosseno. O segundo tipo, a simetria de escala, relativa à contração ou expansão da figura, é representada pelos fractais e será apresentada mais a frente neste mesmo livro. Finalmente, a simetria reflexiva, estudada nesta seção, é caracterizada pela ação de cortar um objeto por um ou mais planos, dividindo-o em partes iguais, porém espelhadas. Na presença de movimento, a seção complementar se moverá na direção oposta da original. Se uma forma se move para a esquerda, sua reflexão se moverá para a direita. O mesmo vale para a orientação vertical. Esse ponto é ilustrado conceitualmente em 5.13 e computacionalmente em 5.14.

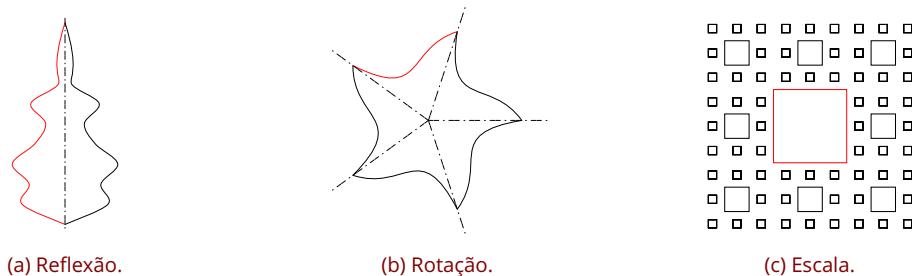


Figura 5.12: Exemplos de figuras simétricas.

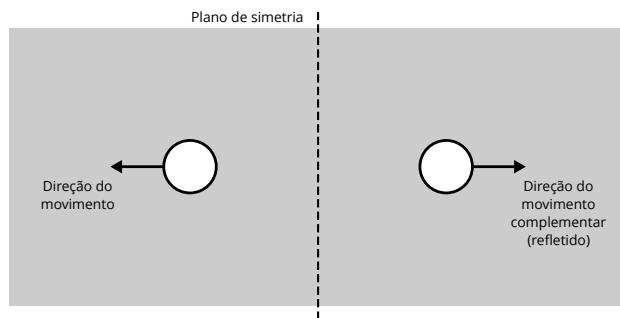


Figura 5.13: Movimento em simetria reflexiva.

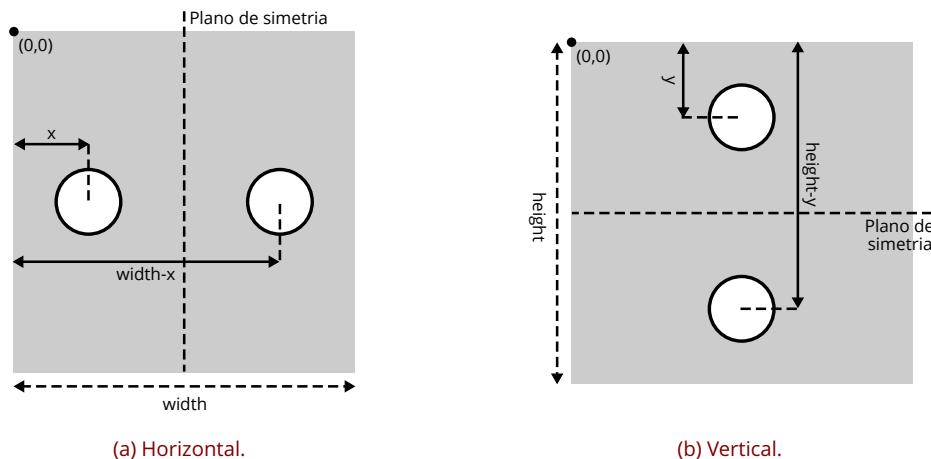


Figura 5.14: Simetria reflexiva.

No capítulo destinado a funções, na seção de interrupções, 2.3.1, você desenvolveu um código que desenhava círculos ao movimentar o mouse. Você pode experimentar adicionar a simetria reflexiva ao código para transformar o tipo de figuras que serão geradas. Uma possível adaptação é mostrada no código a seguir, que contempla simultaneamente as simetrias horizontais (eixo x) e verticais (eixo y), desenhando quatro círculos no lugar de apenas um. Veja o resultado na figura 5.15.

```
// Paleta de cores:  
color[] c = { #581840, #900C3A, #C70034, #FF572E };  
  
void setup() {  
    size(800,800);  
    background(255);  
    noStroke();  
}  
  
void draw() {}  
  
// Mouse pressionado e arrastado:  
void mouseDragged() {  
    int indice = round(random(3));  
    fill(c[indice]);  
    float raio = dist(mouseX,mouseY,pmouseX,pmouseY);  
  
    ellipse(mouseX,mouseY,raio,raio);  
    ellipse(mouseX,height-mouseY,raio,raio);  
    ellipse(width-mouseX,mouseY,raio,raio);  
    ellipse(width-mouseX,height-mouseY,raio,raio);  
}
```

A computação, aliada à simetria, também pode ser aplicada quando se deseja explorar padrões artísticos repetitivos. A figura 5.16 é um máximo da simetria (aumente o zoom na imagem) que demonstra todas as 32768 formas de se preencher um quadrado, simétrico em relação ao seu eixo y central, dividido em vinte e cinco partes iguais<sup>2</sup>.

<sup>2</sup>Baseado na obra *Invader Fractal* de Jared Tarbell

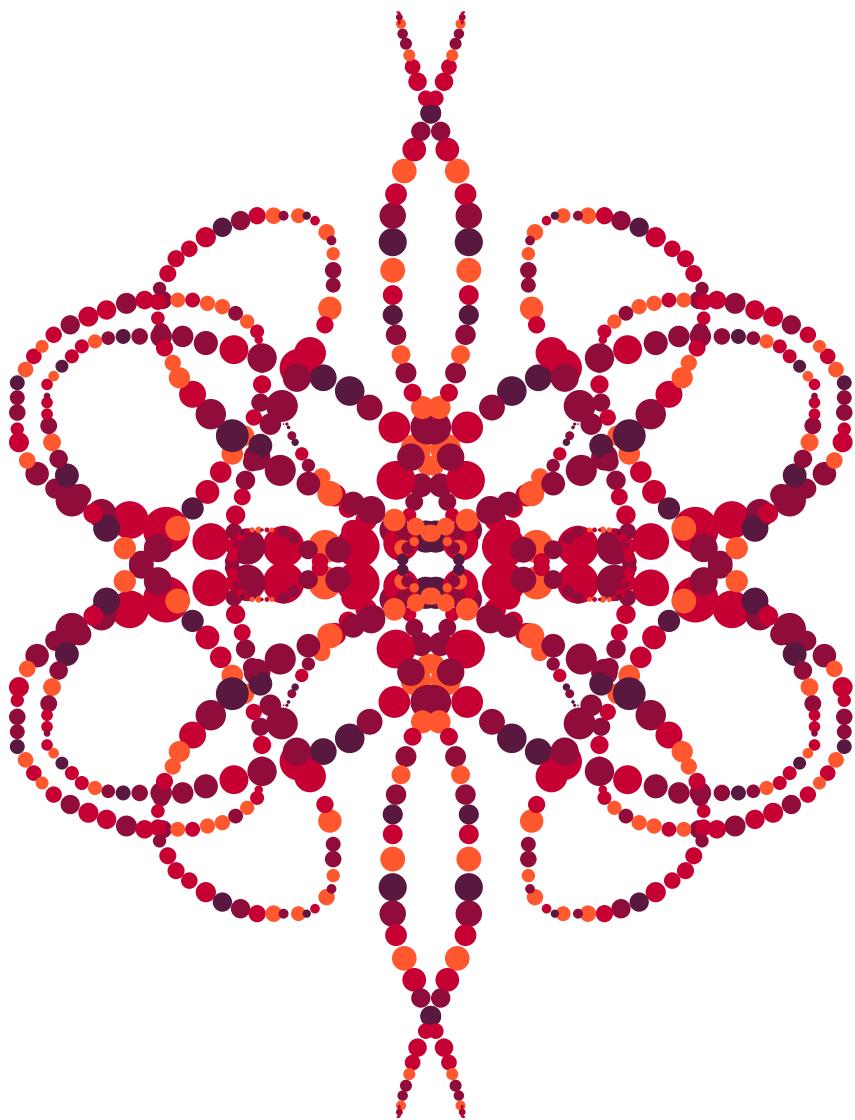


Figura 5.15: Desenho simétrico na janela ao arrastar o mouse.

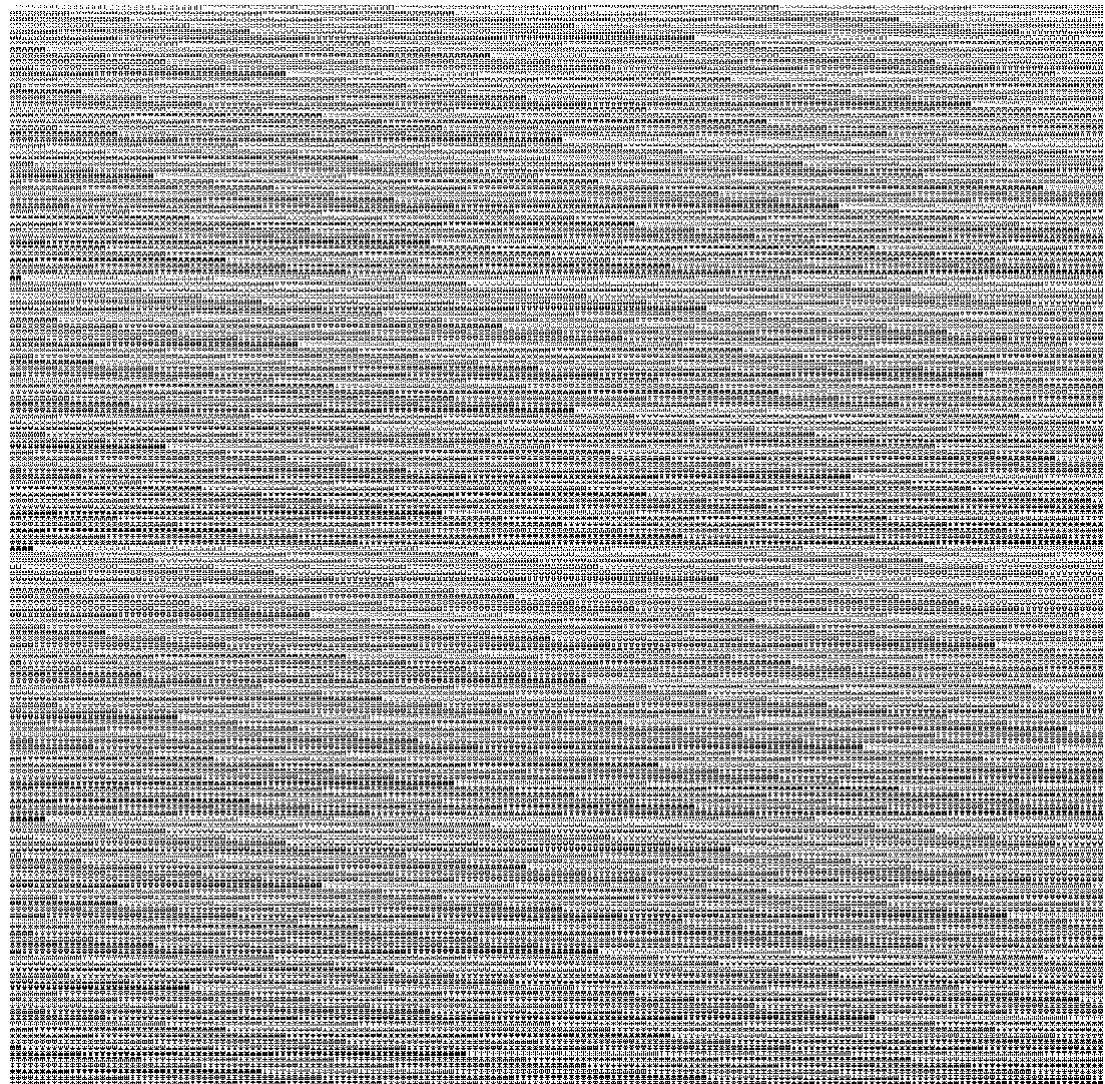


Figura 5.16: Total de 32768 possibilidades.

## 5.4 | Um exemplo para reinar sobre todos

---

A segunda parte deste livro, e este capítulo, serão encerrados com a proposta de que você use tudo que aprendeu até o momento para desenvolver um programa que crie imagens gerativas. O principal objetivo será a visualização, de uma maneira mais elaborada, da função `noise()`.

Na seção 4.1, dominada por operações trigonométricas, você aprendeu que é possível determinar qualquer ponto na superfície de um círculo através de duas fórmulas que dependem de seu centro, raio e ângulo do setor. Elas fórmulas também podem ser usadas para traçar uma reta qualquer, cujo primeiro ponto é o centro do círculo e o segundo é dado pelo ponto em sua superfície, que pode ser facilmente determinado quando se tem o raio e o ângulo. Estes dois últimos parâmetros são arbitrários e você é livre para escolher qual será o comprimento dessa reta (raio do círculo) e qual será sua orientação (ângulo do setor). Por exemplo, se desejar traçar uma reta com início no ponto (50,50), com 40 pixels de comprimento e uma inclinação de 30°, basta utilizar as fórmulas 4.1 para obter os pontos finais dela:

```
float xf = 50 + 40*cos(radians(30)); // O resultado é xf = 84.64
float yf = 50 + 40*sin(radians(30)); // O resultado é yf = 70.00
```

e ela seria traçada como:

```
line(50,50,xf,yf).
```

Isto será muito útil por que em vez de usar pontos para visualizar o ruído Perlin, você pode utilizar retas anguladas. Para obter um efeito uniforme é interessante distribuir essas retas homogeneamente em toda a janela de saída. Você pode usar um raciocínio semelhante ao do posicionamento de blocos no padrão de Truchet, seção 2.5.2, e construir uma estrutura dupla de repetições. Vamos concretizar essas ideias através de um programa, veja o código abaixo. Algumas variáveis foram parametrizadas para auxiliar nas etapas posteriores. O resultado é mostrado na figura 5.17.

```
void setup() {
    size(700,200);
    frameRate(30);
    background(255);
}

void draw() {
    background(255);

    // Separação entre uma linha e outra (em pixels):
    int sepLinhas = 20;
    // Tamanho das linhas desenhadas (em pixels):
    float tamLinha = 20;

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

```

        for (int x = 0; x <= width; x += sepLinhas) {
            // Define a variável que irá determinar a inclinação das retas:
            float angulo = 30;

            float xf = x + tamLinha*cos(radians(angulo));
            float yf = y + tamLinha*sin(radians(angulo));
            line(x,y,xf,yf);
        }
    }
}

```

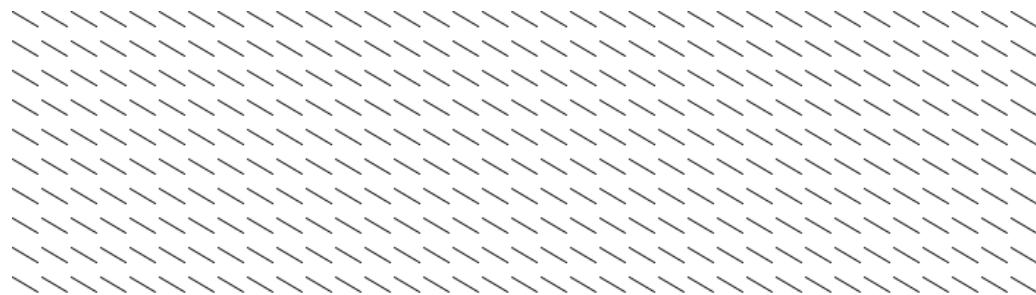


Figura 5.17: Retas anguladas distribuídas homogeneamente na tela.

Visto que as retas estão distribuídas de maneira adequada, você pode adicionar o ruído Perlin para diversificar a angulação das mesmas. Altere as estruturas de repetição de acordo com o código abaixo para criar padrões semelhantes ao da figura 5.18.

```

// Valores iniciais dos ruídos:
float nxi = random(100);
float nyi = random(100);

void setup() {
    size(700,200);
    frameRate(30);
    background(255);
}

void draw() {
    background(255);

    // Separação entre uma linha e outra (em pixels):
    int sepLinhas = 20;
    // Tamanho das linhas desenhadas (em pixels):

```

```

float tamLinha = 20;

// Incrementa o valor inicial dos ruídos:
nxi += 0.01;
nyi += 0.01;

// Define os ruídos locais a serem usados na angulação das retas:
float ny = nyi;
float nx = nxi;

for (int y = 0; y <= height; y += sepLinhas) {
    // Incrementa e reinicia o ruído local:
    ny += 0.05;
    nx = nxi;

    for (int x = 0; x <= width; x += sepLinhas) {
        // Incrementa o ruído local:
        nx += 0.05;

        // Define a variável que irá determinar a inclinação das retas:
        float angulo = 360*noise(nx,ny);

        float xf = x + tamLinha*cos(radians(angulo));
        float yf = y + tamLinha*sin(radians(angulo));
        line(x,y,xf,yf);
    }
}
}

```

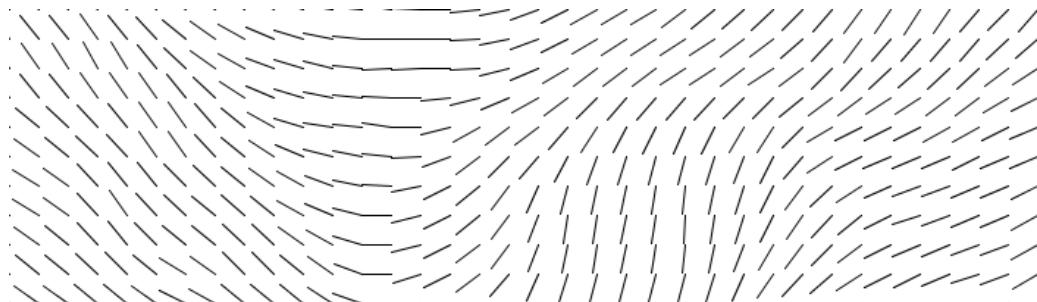


Figura 5.18: Retas anguladas através da função `noise()`.

Ao executar o programa e a animação, você deve ter observado que a função `noise()` cria uma variação suave no ângulo das retas, expressando um efeito de fluxo na sua imagem. Mais uma vez isso reforça a natureza orgânica do ruído Perlin. A figura 5.19 mostra o que aconteceria se fosse utilizada a função

`random()` em seu lugar.

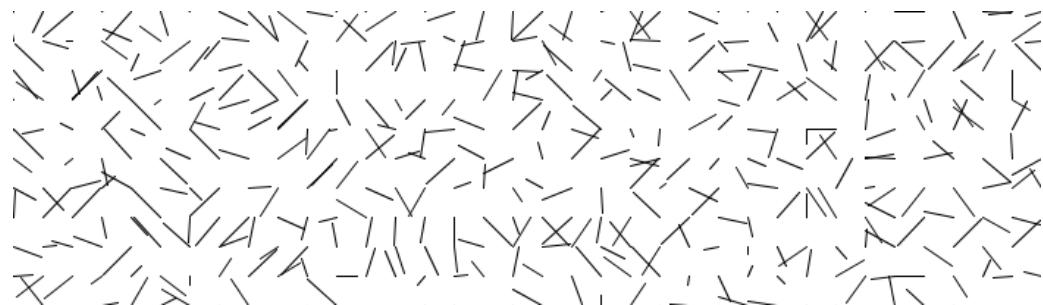


Figura 5.19: Retas anguladas através da função `random()`.

Neste ponto podemos considerar que a etapa de rascunho, o alicerce do programa, está terminada e que nos resta canalizar nossa atenção para a formatação. A primeira alteração que você pode fazer é experimentar com diversos valores para o tamanho e quantidade de retas. Esses valores foram parametrizados na concepção do código, atere-os para obter padrões como os da figura 5.20.

O segundo ponto é enriquecer visualmente a figura usando tons e cores. As alternativas aqui são muitas, sendo uma delas baseada na disposição espacial das retas. Nesse tipo de abordagem utiliza-se da informação da posição dos objetos, no caso as retas, para definir a formatação. Um exemplo seria colocar tons mais claros em retas à direita da tela e gradativamente escurece-las até atingir o canto esquerdo da janela, criando uma espécie de gradiente como mostrado na figura 5.21. A única alteração que deve ser feita é adicionar uma formatação, logo antes do desenho, interna na segunda estrutura de repetição:

#### Código 5.12 Formatação baseada em posição

```
for (int x = 0; x <= width; x += sepLinhas) {  
    // Incrementa o ruído local:  
    nx += 0.05;  
  
    // Define a variável que irá determinar a inclinação das retas:  
    float angulo = 360*noise(nx,ny);  
  
    // Utiliza as coordenadas espaciais (x,y) para criar um gradiente:  
    float cor = map(x+y,0,width+height,0,255);  
    stroke(cor);  
  
    float xf = x + tamLinha*cos(radians(angulo));  
    float yf = y + tamLinha*sin(radians(angulo));  
    line(x,y,xf,yf);  
}
```

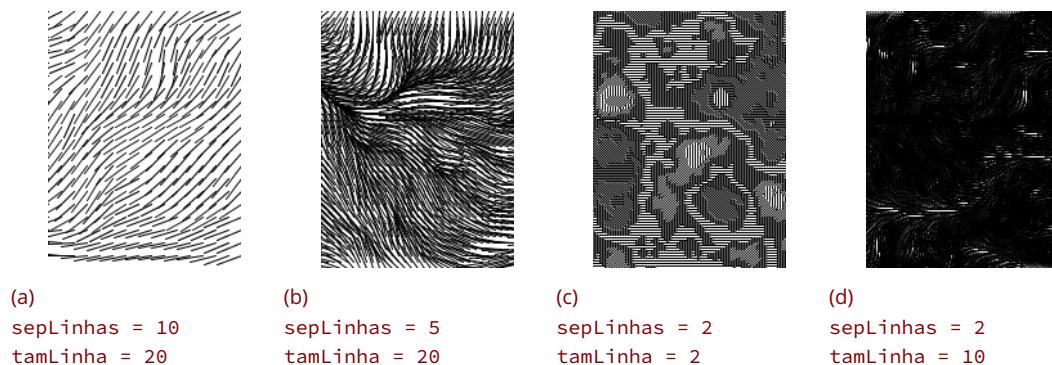


Figura 5.20: Padrões formados por outras parametrizações.

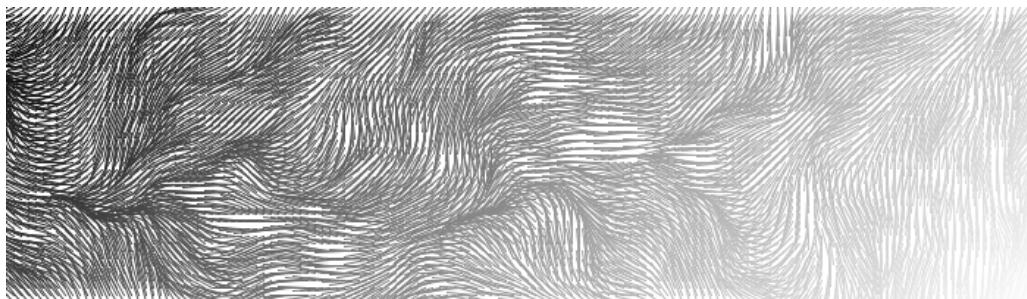


Figura 5.21: Formatação em estilo gradiente.

A formatação no estilo gradiente é interessante em alguns casos, mas ela é simplória e previsível. Nela, sempre existirão um ou mais pontos de convergência de cores e algum tipo de interpolação (no caso acima, linear). Uma boa estratégia para cativar olhares é abusar de contrastes inesperados em um mesmo padrão. Então, qual variável pode ser responsável pela formatação sem ser a posição das retas? Um ótimo candidato é um parâmetro que varia inesperadamente com o desenho de cada uma delas como, por exemplo, seu ângulo cujos valores estão entre  $0^\circ$  e  $360^\circ$ . O código abaixo é uma sugestão de como fazer isso, e uma das possíveis imagens geradas está mostrada na figura 5.22.

#### Código 5.13 Formatação baseada em ângulo

```
for (int x = 0; x <= width; x += sepLinhas) {  
    // Incrementa o ruído local:  
    nx += 0.05;
```

```

// Define a variável que irá determinar a inclinação das retas:
float angulo = 360*noise(nx,ny);

// Utiliza a angulação das retas para definição da cor:
float cor = map(angulo,0,360,0,255);
stroke(cor);

float xf = x + tamLinha*cos(radians(angulo));
float yf = y + tamLinha*sin(radians(angulo));
line(x,y,xf,yf);
}

```

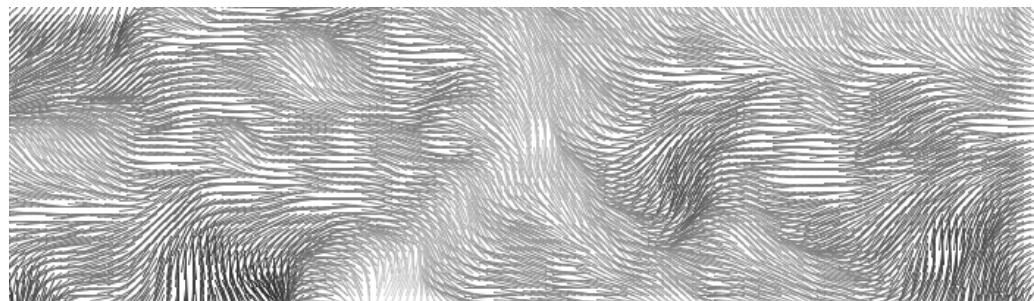


Figura 5.22: Retas coloridas de acordo com suas angulações.

Você pode perceber que agora as retas formam um padrão menos homogêneo. Essa formatação ainda tem uma consequência oculta bem mais importante do que somente a inclusão de cores. Note que não há mais problema se as retas se sobrepor em uma vez que serão criados contrastes proveniente das diferentes angulações. Isso permite que a quantidade de retas seja aumentada consideravelmente. Para conseguir imagens como a [5.23](#) altere os valores das variáveis para:

```

int sepLinhas = 2;
float tamLinha = 10;

```

e comente a limpeza de fundo dentro de `draw()`:

```
//background(255);
```

A finalização da obra pode ser feita aumentando os incrementos dos ruídos que provocará uma crescente oscilação nas inclinações das retas, entalhando a imagem com uma textura mais visível, progressivamente mais agressiva. Mudando o foco da composição, podemos adicionar cores ao invés de apenas tons de cinza. Optou-se por uma paleta monocromática para manter a ilusão de uma peça de constituição única. Por último, você pode eternizar sua peça salvando-a diretamente com a função `saveFrame()` que recebe

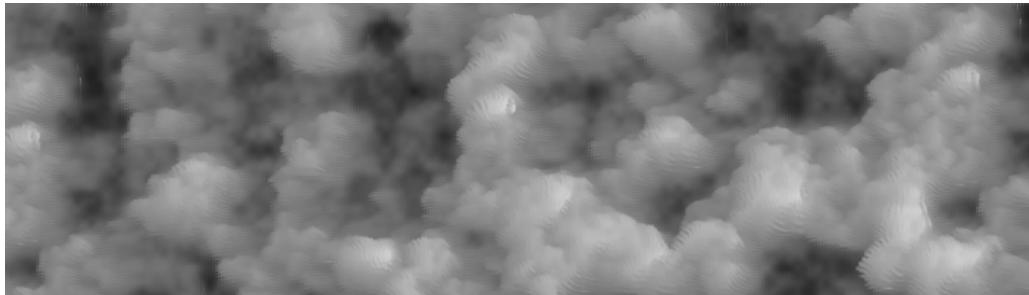


Figura 5.23: Retas sobrepostas com formatação dependente de ângulo.

o caminho<sup>3</sup>, o nome e o tipo do arquivo de imagem a ser gravada. A extensão .png é interessante por possuir uma qualidade satisfatória mantendo um tamanho reduzido. Uma boa prática ao salvar imagens (ou qualquer outro arquivo) é definir uma condição específica para essa ação, caso contrário o Processing criará arquivos na velocidade de execução do programa. O código completo, incluindo essas últimas alterações, se encontra abaixo e, a imagem, na figura 5.24:

```
// Valores iniciais dos ruídos:  
float nxi = random(100);  
float nyi = random(100);  
  
void setup() {  
    size(700,200);  
    frameRate(30);  
    background(255);  
}  
  
void draw() {  
    //background(255);  
    strokeWeight(2);  
  
    // Separação entre uma linha e outra (em pixels):  
    int sepLinhas = 2;  
    // Tamanho das linhas desenhadas (em pixels):  
    float tamLinha = 10;
```

<sup>3</sup>Utilizar essa função sem incluir um caminho salva a imagem no próprio diretório do programa.

```

// Incrementa o valor inicial dos ruídos:
nxi += 0.01;
nyi += 0.01;

// Define os ruídos locais a serem usados na angulação das retas:
float ny = nyi;
float nx = nxi;

for (int y = 0; y <= height; y += sepLinhas) {
    // Incrementa e reinicia o ruído local:
    ny += 0.1;
    nx = nxi;

    for (int x = 0; x <= width; x += sepLinhas) {
        // Incrementa o ruído local:
        nx += 0.1;

        // Define a variável que irá determinar a inclinação das retas:
        float angulo = 360*noise(nx,ny);

        // Cores baseadas na inclinação das retas:
        float verm = 60 + map(angulo,0,360,0,255);
        stroke(verm,0,0);

        // Cálculo do ponto final da reta angulada:
        float xf = x + tamLinha*cos(radians(angulo));
        float yf = y + tamLinha*sin(radians(angulo));
        line(x,y,xf,yf);
    }
}

if(frameCount == 5) {
    saveFrame("Imagen.png");
}
}
}

```

A comparação, lado a lado entre as diversas etapas desse projeto, figura 5.25, nos ajuda a entender que empilhar sucessivamente camadas de formatação impulsiona uma evolução na riqueza e complexidade do padrão.

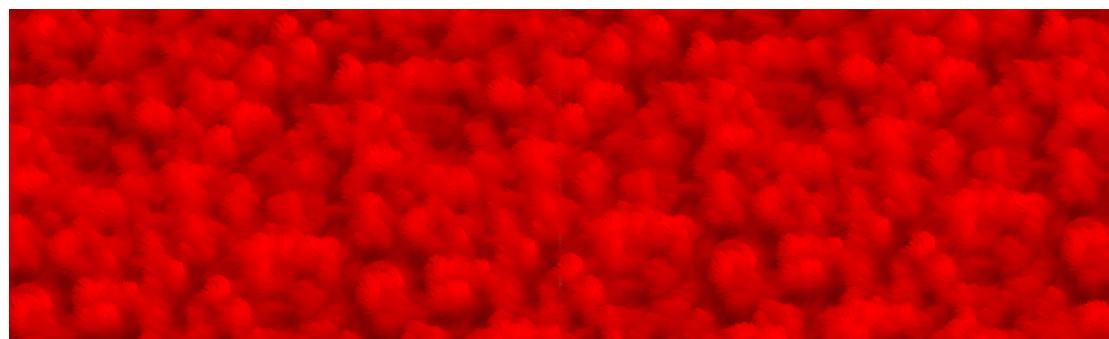


Figura 5.24: Imagem final após alterações na formatação.

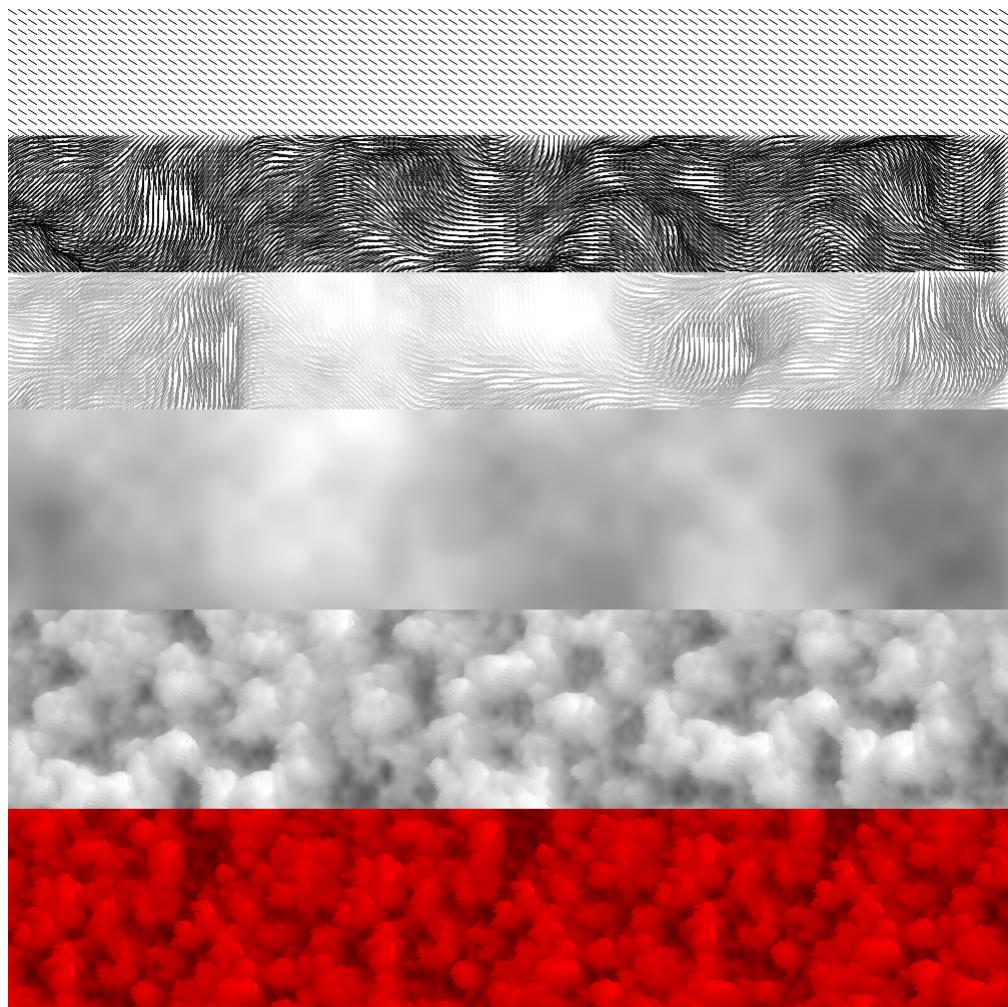


Figura 5.25: Evolução da visualização do ruído bidimensional.

## 5.5 | Sumário

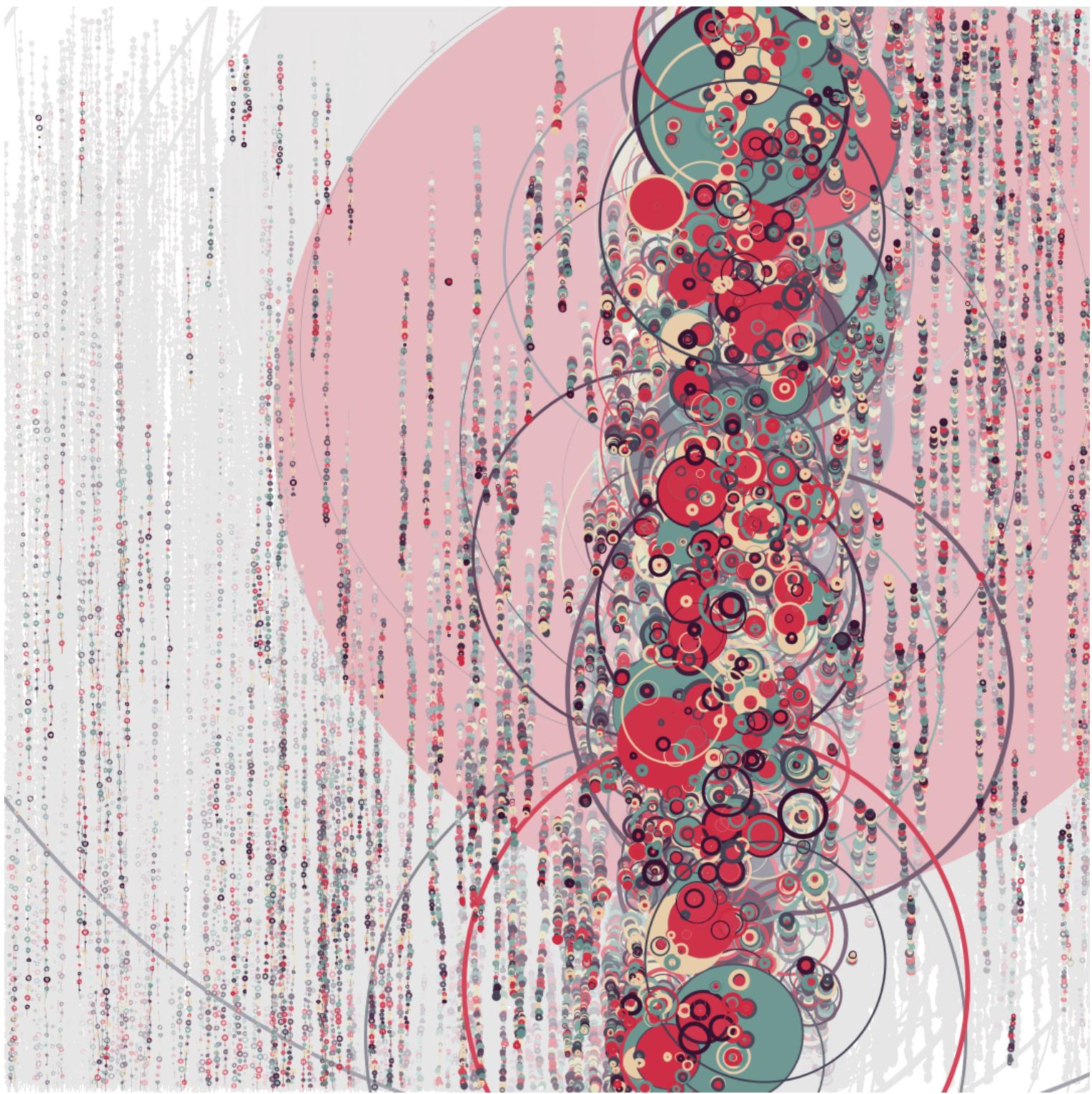
---

Os diferentes estilos de formatação, que algumas vezes são vistas por programadores como irrisórias no contexto do programa como um todo, não devem ser menosprezadas quando se trabalha com a computação artística. Você aprendeu diversas maneiras de enriquecer visualmente suas imagens e criar uma estética marcante através de técnicas de formatação que parecem simples, mas juntas são responsáveis por modificar profundamente a imagem. Você também foi capaz de escrever um código que utiliza de tudo<sup>4</sup> que você aprendeu até o momento, como condicionais, repetições, trigonometria, aleatoriedade e, por fim, a própria formatação. Isso finaliza a segunda parte do livro, indicando que você agora possui um versátil conhecimento matemático e poderosas ferramentas de programação para criar verdadeiras obras primas nascidas da sua criatividade.

Na terceira parte deste livro você investigará melhor como aplicar conceitos matemáticos aliados aos recursos que você vem estudando. Serão mostradas algumas dicas e estratégias de programação, mas o verdadeiro valor nestes últimos capítulos está no processo da projeção da criatividade e na transformação do conceitual no físico.

---

<sup>4</sup>A programação orientada a objetos não foi utilizada por ser considerada como um tópico avançado. Ela possuí uma aplicação própria em um capítulo posterior.





# TRANSCENDÊNCIA



Progresso consiste, não em melhorar o que existe, mas em avançar na direção do que existirá.

- KHALIL GIBRAN



[ CAPÍTULO 6 ]:

# Fractais

Catástase computacional

Em 1989 o pai da geometria fractal, o matemático Benoit B. Mandelbrot, publicou um artigo intitulado Geometria Fractal: o que é, e o que ela faz?<sup>1</sup> em que a definiu em um pequeno parágrafo:

*"Geometria fractal é um meio termo geométrico trabalhável entre a excessiva ordem geométrica Euclidiana e o caos geométrico da matemática geral. É baseado em uma forma de simetria que foi previamente subutilizada, chamada de invariância na presença de contração e dilatação. A geometria fractal é convenientemente vista como uma linguagem que prova seu valor pelos seus usos. É usada na arte e na matemática pura, sem aplicações "práticas", podendo ser dita como sendo poética."*

Em um contexto puramente técnico, pode parecer estranho que palavras como “arte” e “poética” apareçam ao lado de termos matemáticos, mas esta é apenas mais uma prova da natureza singular dessas complexas estruturas geométricas. Fractais talvez sejam uma das formas mais famosas de arte matemática existentes. Muitas vezes associados a estruturas caóticas, psicodélicas e enlouquecedoras, os fractais são fonte de admiração de um público cativado pela sua beleza hipnotizante ou pelas regras simples que dão vida a padrões complexos. Sem mais demoras, iremos agora entrar em um capítulo dedicado a essas formas intrigantes e ao espaço que elas conquistaram na arte computacional.

---

<sup>1</sup> Mandelbrot, B. B. (1989). *"Fractal Geometry: What is it, and What Does it do?"*. Proceedings of The Royal Society A Mathematical Physical and Engineering Sciences 423(1864).

## 6.1 | Autossimilaridade e recursividade

---

Fractais (do latim *fractus*: fração, quebrado) são padrões autossimilares ao longo de uma escala de ampliação infinita. Isso significa que fractais podem ser divididos em inúmeros outros pedaços que, se observados de perto, seriam idênticos ao fractal original. Curiosamente, fractais são encontrados em múltiplos fenômenos naturais, como flocos de neve (figura 6.1a), raios, linhas costeiras, plantas (figura 6.1b), montanhas, veias e padrões em animais dentre outros.



(a) Foco de neve ampliado<sup>a</sup>, cortesia dos fotógrafos  
Olga Sytina e Alexey Kljatov.

<sup>a</sup>Todos os direitos reservados pelos artistas originais.



(b) Brócolis Romanesco<sup>b</sup>, cortesia do fotógrafo  
Scott Atwood.

<sup>b</sup>Todos os direitos reservados pelo artista original.

Figura 6.1: Fractais encontrados na natureza.

No segmento computacional, o flocos de neve de Koch é um dos fractais em que se pode observar o comportamento autossimilar de forma mais evidente. Na imagem 6.2 é mostrada sua representação com sete níveis de subdivisões. Ele foi concebido como uma figura do tipo *Scalable Vector Graphics* e você pode aplicar um fator de ampliação relativamente alto a ela (mais de 5000%) para ver como o fractal se repete em escala macro e micro.

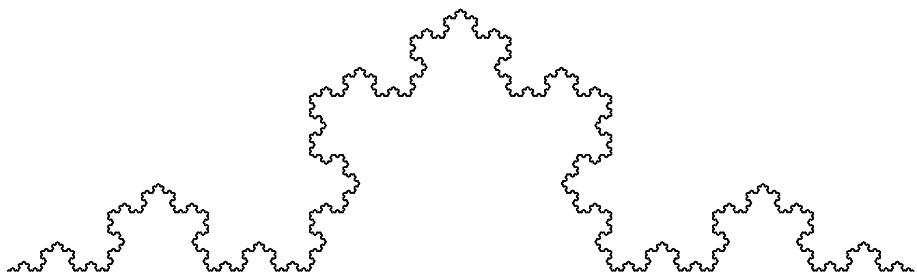
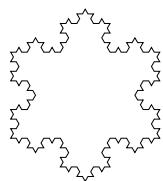
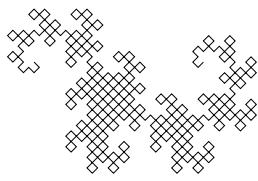


Figura 6.2: Floco de neve de Koch, 7 divisões.

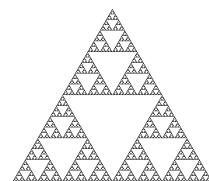
Os fractais mais notáveis são divididos em três grandes grupos. Os fractais geométricos, figura 6.3, são normalmente criados através de sucessivas repetições de um processo geométrico simples ao longo de um espaço indeterminado. O floco de neve de Koch é um fractal geométrico nascido a partir de uma regra de subdivisão infinita das retas que o compõe. Outros exemplos seriam os Triângulos de Sierpiński, a Árvore Fractal e a Curva Dragão.



(a) Floco de neve de Koch.



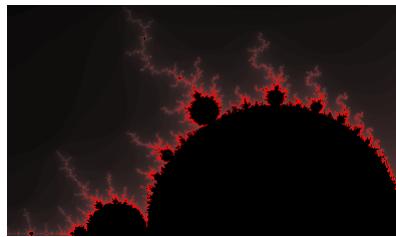
(b) Dragão de Harter-Heighway.



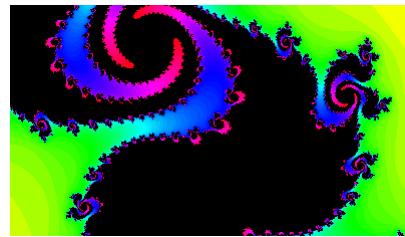
(c) Triângulo de Sierpinski.

Figura 6.3: Fractais geométricos.

Por outro lado, os denominados fractais abstratos, figura 6.4, como os criados por Mandelbrot, podem ser gerados através de computadores que calculam repetidamente uma equação matemática ou uma relação de recorrência. O Conjunto de Mandelbrot e o de Julia são os mais famosos deste tipo.



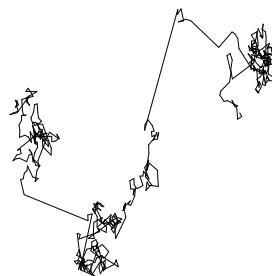
(a) Conjunto de Mandelbrot.



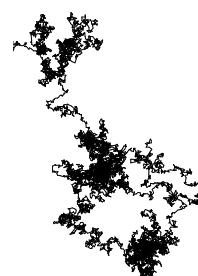
(b) Conjunto de Julia.

Figura 6.4: Fractais abstratos.

Por último, estão os fractais aleatórios, figura 6.5, formados através de processos estocásticos. Neste grupo estão o Voo de Lévy e as Trajetórias do Movimento Browniano.



(a) Voo de Lévy.



(b) Movimento Browniano.

Figura 6.5: Fractais aleatórios.

Um fractal é criado pela repetição infinita, no âmbito espacial e temporal, de um padrão sobre si mesmo. Esse incessante empilhamento de operações e desenhos o torna ideal para ser materializado por um computador. As palavras chave quando nos referimos a essas figuras são *recursividade* e *regras*. A recursividade faz analogia a funções que realizam chamadas a elas mesmas. Por exemplo, uma linha de ação lógica de programação seria projetar uma função que desenhasse parte do padrão e, em seguida, fazer com que essa sub-rotina invocasse a si mesma, repetindo a forma e criando a autossimilaridade. É importante observar que enquanto essa repetição infinita é inofensiva na teoria, ela é muito perigosa na prática. Na maioria das vezes a recursividade multiplica o número de operações realizadas por iteração ou ciclo de desenho da figura. Quando programado, um fractal deve possuir uma condição de parada, caso contrário

ele irá explodir no numero de operações e travar a simulação por falta de memória. As condições de parada da recursividade para um fractal normalmente se resumem a um tamanho visual mínimo, afinal, não faz sentido possuir um número infinito de detalhes se o computador não for capaz de exibi-los. Baseado nessa explicação simplificada, você pode ter concluído que a forma básica de uma função recursiva que desenha um fractal é algo do tipo:

```
void desenhaFractal(argumentos) {  
    // 1) Desenha o padrão.  
  
    // 2) Gera novosArgumentos baseados em regras.  
  
    // 3) Função chama a si mesma com os novos argumentos.  
    if(condicaoDeParada == false) {  
        desenhaFractal(argumentos e/ou novosArgumentos);  
    }  
}
```

Igualmente importante são as regras ou os processos que dão vida a esses padrões. Os fractais geométricos são profundamente enraizados em regras simples repetidas ao longo de muitas iterações, gerando imagens de uma complexidade visual mesmerizante. Em geral essas regras se limitam a desenhar elementos geométricos (círculos, retas, retângulos, etc.), rotados e transladados, respeitando pré-condições de projeto. Um exemplo que podemos citar é do próprio floco de neve de Koch, cujas leis de construção, mostradas abaixo e ilustradas em 6.6, são baseadas na subdivisão recursiva de retas.

1. Comece com um segmento de reta.
2. Divida a reta em outros três segmentos de comprimentos idênticos.
3. Desenhe um triângulo equilátero que possua o segmento central do passo 2 como base.
4. Remova o segmento que é a base do triângulo desenhado no passo 3
5. Repita os passos anteriores para cada segmento de reta existente.

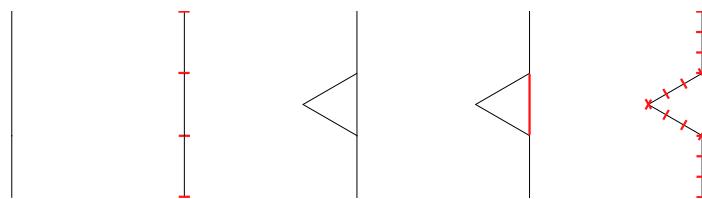


Figura 6.6: Passos para a construção do Floco de neve de Koch.

Os detalhes em um fractal são revelados quando as regras de construção são repetidas para cada subelemento (passo 5 nas instruções citadas anteriormente), figura 6.7.

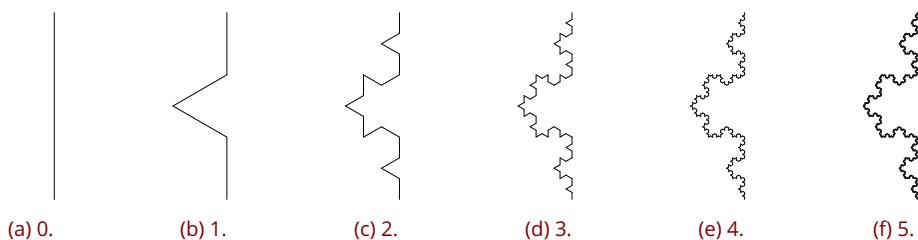


Figura 6.7: Níveis de subdivisões no Floco de neve de Koch.

Você entenderá melhor como essas regras locais são capazes de gerar um resultado global quando programar um fractal nas próximas seções.

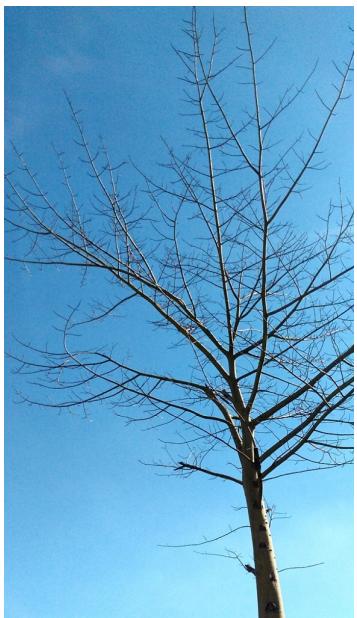
## 6.2 | Fractal árvore

Um estudo de caso clássico, simples e belo é o fractal planta ou árvore. Ele recebe esse nome em homenagem a sua forma, que remete a de uma árvore composta por um tronco central que se desdobra em diversos galhos ou ramos a medida que ela cresce, figura 6.8.

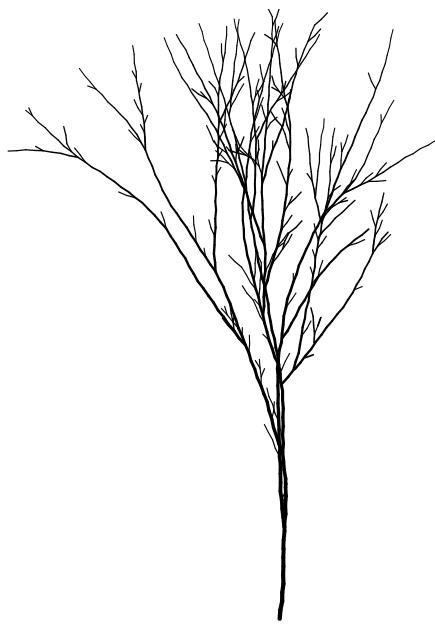
Ao final deste capítulo você será capaz de reproduzir formas como esta, mas primeiramente você deve começar pelo básico e entender como um fractal é transmutado do conceito para o código. Concentre-se na forma mais rudimentar de um fractal regular<sup>2</sup> desta categoria e como ele se desenvolve a cada iteração da recursividade, mostrados na figura 6.9.

A análise do fractal através de etapas é mais naturalmente entendida se analisarmos apenas um único ramo da árvore, figura 6.10a, uma vez que os demais são gerados por repetições simétricas da forma principal. Este ramo também será proposidadamente distorcido de modo que a cada iteração o tamanho do próximo galho seja apenas ligeiramente menor que o anterior, como mostrado na figura 6.10b. O objetivo é propiciar uma explanação mais clara da etapa geométrica de projeto do fractal.

<sup>2</sup>Não perturbado por funções do tipo `noise()` ou `random()`.



(a) Silhueta de uma árvore.



(b) Fractal árvore deformado usando a função `noise()`.

Figura 6.8: Comparação árvore e fractal

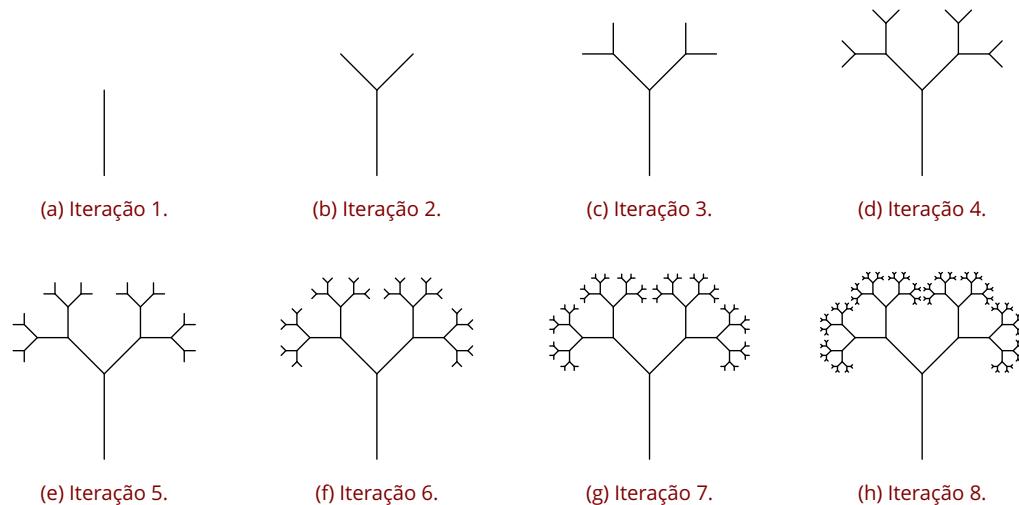


Figura 6.9: Etapas do desenvolvimento do fractal árvore.

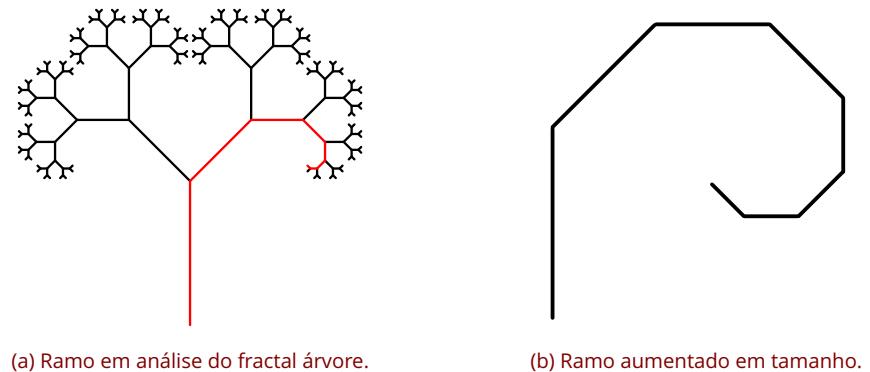


Figura 6.10: Objeto de estudo do fractal árvore.

Se você dividir o ramo em cada uma de suas iterações, perceberá que ele consiste em três operações primordiais:

1. Desenhar uma reta cujo ponto inicial é igual ao ponto final da reta da iteração passada.
2. Inclinar a reta desenhada em 1, para que ela tenha um certo ângulo em relação à reta desenhada na iteração passada.
3. Reduzir o tamanho da reta desenhada em 1, de forma que ela seja menor que a reta desenhada na iteração passada.

Para fins de exemplo, considere que o ângulo de inclinação da segunda operação é fixo em  $45^\circ$ . Estes três procedimentos estão evidenciados nas figuras 6.11a e 6.11b.

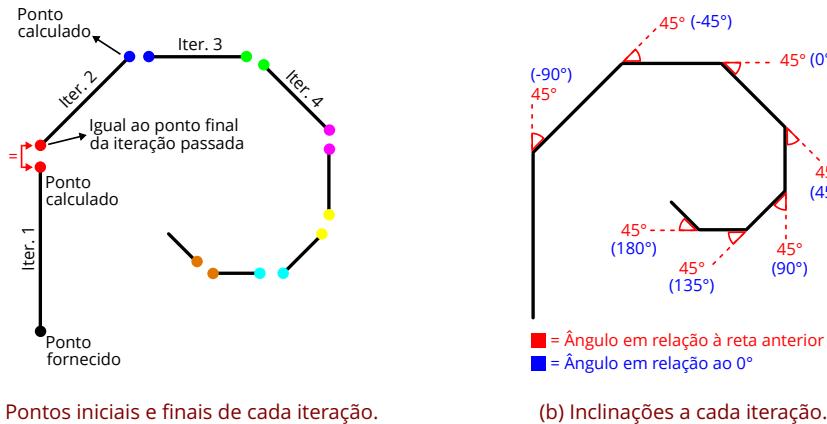


Figura 6.11: Etapas do desenho de um ramo do fractal árvore.

Com base nessas proposições podemos inferir que a função para desenhar a reta que compõe cada iteração desse ramo possuirá as seguintes premissas:

- **Parâmetros da função:** Deverão ser fornecidos, para cada iteração, um ponto inicial, um comprimento da reta e o ângulo de inclinação dessa reta, totalizando três argumentos de entrada.
- **Corpo da função:** Essa função calculará o ponto final da reta usando os argumentos de entrada e as equações trigonométricas do círculo (4.1). A reta que representa o ramo da árvore será traçada através da função `line()` usando as coordenadas do ponto inicial (fornecido) e final (calculado).
- **Retorno da função:** A função deverá devolver as coordenadas  $x$  e  $y$  do ponto final da reta para que essa informação seja usada na próxima chamada da sub-rotina (próxima iteração). Isto implica em mais de uma variável de retorno sendo necessário usar vetores para agrupar os dados.

Seguindo esse raciocínio você pode prontamente escrever a função para desenhar um ramo dessa árvore e depois chamá-la no programa principal:

### Código 6.2 Ramo simples

```
void setup() {  
    size(500,500);  
    background(255);  
  
    float[] pontos = new float[2];  
  
    // Utilizado -90° apenas para que a árvore cresça para cima da tela.  
    pontos = arvore(width/2,height,100,-90);  
}  
  
float[] arvore(float _xi, float _yi, float _comp, float _angulo) {  
    // Fórmula básica mostrada na seção 4.1  
    float xf = _xi + cos(radians(_angulo))*_comp;  
    float yf = _yi + sin(radians(_angulo))*_comp;  
    line(_xi,_yi,xf,yf);  
  
    // Esta função retorna os pontos final do ramo:  
    float[] posFinal = {xf,yf};  
  
    return posFinal;  
}
```

Ao executar esse código você verá exatamente a primeira e única iteração da árvore, mostrada anteriormente na figura 6.9a. O restante do ramo será construído com as chamadas subsequentes à função `arvore()`. Complemente o código do `setup()` com as linhas:

```
// Gera um dos ramos da figura 6.9d:  
  
// Iteração 1.  
pontos = arvore(width/2,height,100,-90);  
  
// Iteração 2: 60% menor e inclinado 45° em relação à Iteração 1.  
pontos = arvore(pontos[0],pontos[1],100*0.6,-90 + 45);  
  
// Iteração 3: 60% menor e inclinado 45° em relação à Iteração 2.  
pontos = arvore(pontos[0],pontos[1],100*0.6*0.6,-90 + 45 + 45);  
  
// Iteração 4: 60% menor e inclinado 45° em relação à Iteração 3.  
pontos = arvore(pontos[0],pontos[1],100*0.6*0.6*0.6,-90 + 45 + 45 + 45);
```

Observe que essa abordagem não é elegante, uma vez que é necessário que a função `arvore()` retorne

variáveis que serão usadas em chamadas posteriores dessa mesma função, além de ser requerida a atualização manual dos argumentos de entrada. Uma solução para esse problema é usar a *recursividade*. Perceba que a função `arvore()`, escrita dessa maneira, não é recursiva, ou seja, ela não faz uma invocação a si mesma. No entanto isso seria muito interessante dado que ela possuí todos os argumentos necessários para a execução da próxima iteração: o ponto final da reta anterior, o novo ângulo de inclinação e o novo comprimento. Vamos atualizar nossa função com essas sugestões, mas **não execute esse novo código** ainda!

#### Código 6.4 Ramo recursivo

```
void arvore(float _xi, float _yi, float _comp, float _angulo) {  
    // Fórmula básica mostrada na seção 4.1  
    float xf = _xi + cos(radians(_angulo))*_comp;  
    float yf = _yi + sin(radians(_angulo))*_comp;  
  
    line(_xi,_yi,xf,yf);  
  
    // Cada novo ramo será 60% do tamanho do original:  
    float novoComp = 0.6*_comp;  
  
    // Cada novo ramo será inclinado 45° do ramo anterior:  
    float novoAngulo = _angulo + 45;  
  
    // Recursividade: Esta função faz uma chamada  
    // a ela própria (com novos argumentos) para desenhar  
    // a próxima reta do ramo.  
    arvore(xf,yf,novoComp,novoAngulo);  
}
```

A rotina acima irá desenhar uma figura como a 6.10b, mas se você executá-la dessa maneira o Processing irá travar, com uma mensagem de erro igual a mostrada na figura 6.12, indicando um estouro de memória em virtude da *recursividade infinita*. Essa falha ocorreu porque em 6.4 existe um erro inerente de projeto visto que não foi adicionada uma condição de parada para impedir a repetição descontrolada da chamada da função. Isto é de suma importância no fractal árvore, pois ele cresce quadráticamente com o número de iterações. Em outras palavras, uma reta gera duas, que geram quatro, que geram oito e assim por diante, sempre dobrando o número de retas a cada iteração.

Um bom candidato para condição de parada é a resolução visual. Como cada reta que compõe o fractal diminui o seu tamanho a cada iteração, se esse tamanho for menor que certo valor, tal como 2 pixels, a função não será mais chamada recursivamente. Vamos aproveitar e fazer uma última alteração. Conforme mostrado na figura 6.9, cada iteração consiste em desenhar dois ramos (ou retas) com inclinações simétricas. Por exemplo, se um é desenhado com +45° o outro deverá ser desenhado com -45°. Podemos incluir essa simples condição e, em vez de fazer uma única chamada recursiva, fazer duas para que sejam desenhados dois ramos no final de cada iteração. Veja o código completo a seguir:

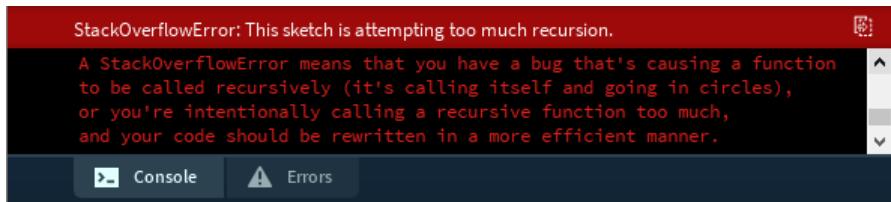


Figura 6.12: Erro de recursividade infinita.

### Código 6.5 Árvore recursiva

```
void setup() {
    size(500,500);
    background(255);

    arvore(width/2,height,100,-90);
}

void arvore(float _xi, float _yi, float _comp, float _angulo) {
    // Condição de parada da recursividade:
    // Comprimento do ramo menos que 2 pixels.
    if(_comp > 2) {
        // Cálculo do ponto final do ramo e seu desenho:
        float xf = _xi + cos(radians(_angulo))*_comp;
        float yf = _yi + sin(radians(_angulo))*_comp;
        line(_xi,_yi,xf,yf);

        // Cada novo ramo será 60% do tamanho do original:
        float novoComp = 0.6*_comp;

        // São duas retas, uma inclinada a +45°da reta original
        // e outra a -45°da reta original:
        float novoAnguloR1 = _angulo + 45;
        float novoAnguloR2 = _angulo - 45;

        // Recursividade: Esta função faz duas chamadas
        // a ela própria (com novos argumentos) para desenhar
        // outras 2 retas.
        arvore(xf,yf,novoComp,novoAnguloR1);
        arvore(xf,yf,novoComp,novoAnguloR2);
    }
}
```

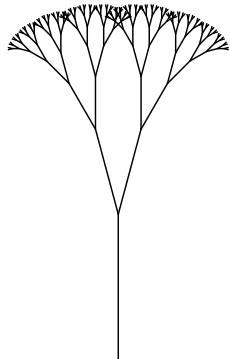
Você pode perceber que não houve nenhuma mudança quanto a chamada da função em `setup()`, visto que todos os passos para desenhar o fractal estão contidos em `arvore()`. O fluxo de desenho também é

mantido e conhecido: a primeira reta será desenhada no ponto central da parte inferior da tela, com um comprimento de cem pixels e uma angulação de -90°. Como dito anteriormente, esse ângulo foi definido de maneira que a árvore crescesse para cima. Se você alterar esse ângulo a árvore passará a ter uma inclinação diferente, mas seus ramos continuarão com uma inclinação de 45° em relação ao ramo originário e com 60% do tamanho do mesmo, já que tais características foram fixadas no corpo da função.

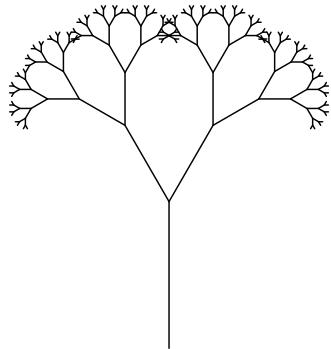
Claro que você deve estar imaginando como seria essa árvore se você alterasse os 45° para algum outro valor. Você pode investigar o resultado mudando essa grandeza diretamente no corpo da função, ou pode reescrevê-la para que ela conte com mais um argumento, sendo este o ângulo de inclinação dos ramos. O código abaixo produz resultados como os da figura 6.13.

#### Código 6.6 Árvores fractais

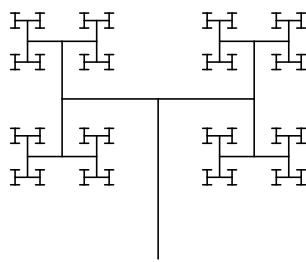
```
void setup() {  
    size(500,500);  
    background(255);  
}  
  
void draw() {  
    background(255);  
  
    // Angulo do ramo vai de 0° a 180°  
    // de acordo com a posição horizontal do mouse:  
    float aRamo = map(mouseX,0,width,0,180);  
  
    arvore(width/2,height,100,-90,aRamo);  
}  
  
void arvore(float _xi, float _yi, float _comp, float _angulo, float _anguloRamo) {  
    if(_comp > 2) {  
        float xf = _xi + cos(radians(_angulo))*_comp;  
        float yf = _yi + sin(radians(_angulo))*_comp;  
  
        line(_xi,_yi,xf,yf);  
  
        float novoComp = 0.6*_comp;  
  
        float novoAnguloR1 = _angulo + _anguloRamo;  
        float novoAnguloR2 = _angulo - _anguloRamo;  
  
        arvore(xf,yf,novoComp,novoAnguloR1,_anguloRamo);  
        arvore(xf,yf,novoComp,novoAnguloR2,_anguloRamo);  
    }  
}
```



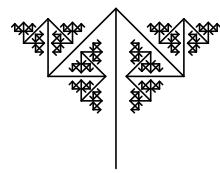
(a)  $15^\circ$ .



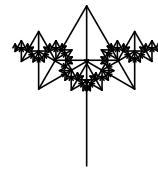
(b)  $30^\circ$ .



(c)  $90^\circ$ .



(d)  $135^\circ$ .



(e)  $150^\circ$ .

Figura 6.13: Fractais com ramificações anguladas.

## 6.2.1 | Subespécies

---

Finalizada a implementação da forma do fractal, podemos experimentar com variações sobre a função `arvore()` para criar efeitos artísticos. Muitos deles se resumem a modificar as variáveis da função original, mas irão ecoar em estéticas significativamente diferentes. No entanto, deve-se ter cuidado ao alterar esses coeficientes, pois isso pode acarretar em um aumento colossal de operações e travar a simulação. Em geral, uma boa prática é ajustar os coeficientes *individualmente* e alterar *levemente* seus valores, sempre observando quais são os efeitos gerados no fractal.

A primeira alteração que você pode fazer é reduzir o quanto um ramo será encurtado a cada iteração que, no código original, é de 40% em relação ao comprimento do ramo originário. Reduzir menos esse tamanho implica em mais repetições para que a condição de parada seja atingida, diretamente aumentando o número de ramos. Para realizar essa alteração basta modificar a linha que define essa redução. Os efeitos são mostrados na figura 6.14.

```
// Reduz em 30% o comprimento do ramo por iteração
float novoComp = 0.70*_comp;
```

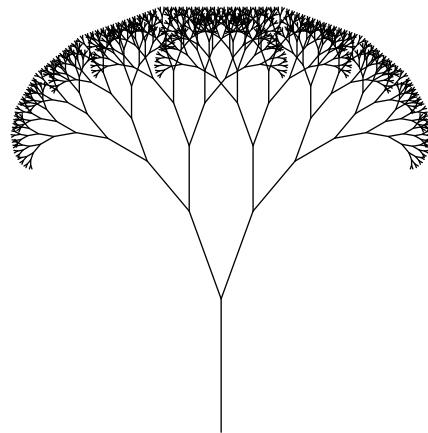
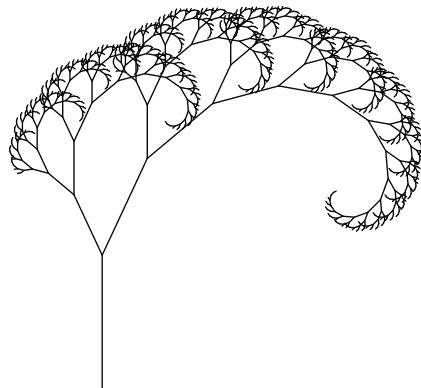
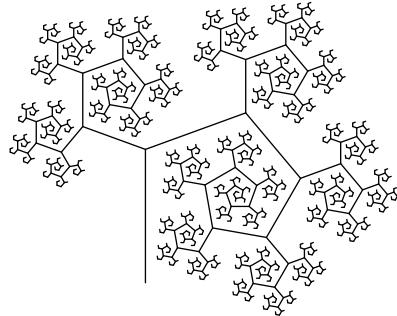


Figura 6.14: Ramos reduzidos 30% a cada iteração.

Outra característica marcante que pode ser modificada é a simetria do fractal. Podemos forçar um crescimento assimétrico ao criar dois fatores distintos de redução de tamanho do ramo, um para cada função recursiva. Veja na figura 6.15 que o fractal assume forma bem diferente da original. Retorne ao código 6.6 e altere as linhas internas a função `arvore()` de:



(a) Inclinação de 25°.



(b) Inclinação de 70°.

Figura 6.15: Fractal com quebra de simetria.

```

float novoComp = 0.6*_comp;

arvore(xf,yf,novoComp,novoAnguloR1,_anguloRamo);
arvore(xf,yf,novoComp,novoAnguloR2,_anguloRamo);
    
```

para:

```

// Um ramo diminui 20% do comprimento originário, enquanto o outro diminui 50%:
float novoCompR1 = 0.8*_comp;
float novoCompR2 = 0.5*_comp;

arvore(xf,yf,novoCompR1,novoAnguloR1,_anguloRamo);
arvore(xf,yf,novoCompR2,novoAnguloR2,_anguloRamo);
    
```

E se você quiser que o fractal se pareça realmente com uma árvore? Então seria preciso colocar “folhas” nas terminações de seus “galhos”. Neste caso você deve lembrar que o fractal só atinge o último ramo após passar pela condição de parada. Consequentemente você pode adicionar um `else` no condicional para fazer com que ela desenhe um círculo que imitará uma folha. Veja a figura 6.16.

```

void arvore(float _xi, float _yi, float _comp, float _angulo, float _anguloRamo) {
    if(_comp > 2) {
        stroke(50);
        strokeWeight(1);
        float xf = _xi + cos(radians(_angulo))*_comp;
        float yf = _yi + sin(radians(_angulo))*_comp;
    }
}
    
```

```

    line(_xi,_yi,xf,yf);

    float novoComp = 0.6*_comp;

    float novoAnguloR1 = _angulo + _anguloRamo;
    float novoAnguloR2 = _angulo - _anguloRamo;

    arvore(xf,yf,novoComp,novoAnguloR1,_anguloRamo);
    arvore(xf,yf,novoComp,novoAnguloR2,_anguloRamo);
}
else {
    fill(random(100,120),random(165,185),random(130,150),10);
    stroke(random(100,120),random(165,185),random(130,150),150);
    strokeWeight(0);
    float raio = random(25,50);
    ellipse(_xi,_yi,raio,raio);
}
}

```

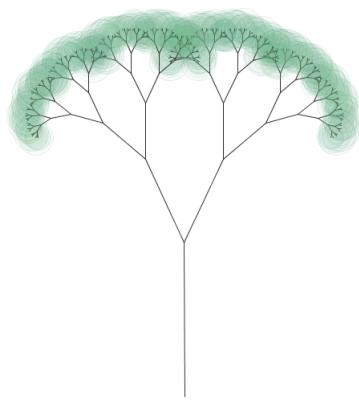


Figura 6.16: Fractal árvore com “folhas” em suas extremidades.

Obviamente que poderíamos aumentar ainda mais a semelhança com uma árvore se a figura possuísse um tronco e galhos que se tornassem cada vez mais finos de acordo com seu tamanho. Essa alteração requer uma variável que esteja ligada as iterações do fractal, uma vez que mais iterações significam ramos mais distantes do ramo originário e, portanto, mais finos. Sabemos que a própria função `arvore()` possui um argumento de entrada relativo ao tamanho do ramo, que diminui conforme o fractal se expande. Sendo assim podemos vincular esse argumento à formatação da espessura da linha desenhada, simulando um afinamento da estrutura a cada ciclo. Os resultados podem ser vistos na figura 6.17.

```

void arvore(float _xi, float _yi, float _comp, float _angulo, float _anguloRamo) {
    if(_comp > 2) {
        // O comprimento vai de 100 a 2, e a espessura de 10 a 0.
        // É necessário que a transformação seja decrescente
        // para que ocorra o afinamento dos ramos.
        float espessura = map(_comp,100,2,10,0);
        strokeWeight(espessura);

        float xf = _xi + cos(radians(_angulo))*_comp;
        float yf = _yi + sin(radians(_angulo))*_comp;

        line(_xi,_yi,xf,yf);

        float novoComp = 0.65*_comp;

        float novoAnguloR1 = _angulo + _anguloRamo;
        float novoAnguloR2 = _angulo - _anguloRamo;

        arvore(xf,yf,novoComp,novoAnguloR1,_anguloRamo);
        arvore(xf,yf,novoComp,novoAnguloR2,_anguloRamo);
    }
}

```

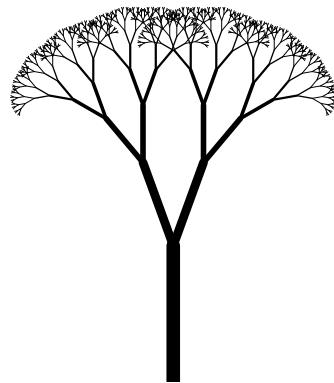


Figura 6.17: Fractal árvore com espessura nos “galhos”.

Todas as formatações apresentadas contribuem para uma exibição que emula o natural, mas a forma principal ainda pode ser retraçada a sua origem estruturalmente mecânica. O recurso final para combater esse excesso de rigidez é aplicar os conceitos do capítulo 4 relativos a aleatoriedade. O caos pode ser usado para quebrar os alicerces simétricos e regulares dos fractais, criando formas mais orgânicas. A autossimilaridade perfeita será perdida, mas a definição é abrangente o suficiente para incluir aproximações das formas. O procedimento para aplicar a incerteza é idêntico ao das seções passadas: identificar um ponto

que influencie consideravelmente o desenho de nossa figura e torná-lo aleatório. Em nosso exemplo em específico, a angulação dos ramos é a candidata perfeita. Simplesmente edite a parte abaixo do código:

```
float novoAnguloR1 = _angulo + _anguloRamo;  
float novoAnguloR2 = _angulo - _anguloRamo;
```

para:

```
float novoAnguloR1 = _angulo + random(_anguloRamo);  
float novoAnguloR2 = _angulo - random(_anguloRamo);
```

Antes de executar seu novo programa, tenha certeza de mover a chamada da função `arvore()` para dentro de `setup()` (ou seja, remova-a de dentro do `draw()`). Isso é especialmente importante, caso contrário a cada frame será gerado um número aleatório e a sua árvore vai parecer que está se movimentando muito rápida e irregularmente. A figura 6.18 apresenta um fractal sob o efeito de perturbações caóticas.



Figura 6.18: Fractal árvore aleatório.

Diversas outras ideias podem ser usadas para incrementar os fractais, como múltiplas exibições simétricas, irregularidades na angulação dos ramos, inclinações exacerbadas ou irrisórias, dentre outras. Combinando-as com as algumas das técnicas apresentadas nesta seção, você pode gerar padrões realmente complexos e surreais que não parecem ter sido desenhados por regras simples, como é o caso do fractal árvore. Essa filosofia foi empregada para gerar figuras como a 6.19 ou 6.20. Note que, no sentido fiel da palavra, essas figuras não são fractais e sim obras gerativas nascidas de instruções recursivas.



Figura 6.19: “Fractal” árvore unindo todas as estratégias de formatação.

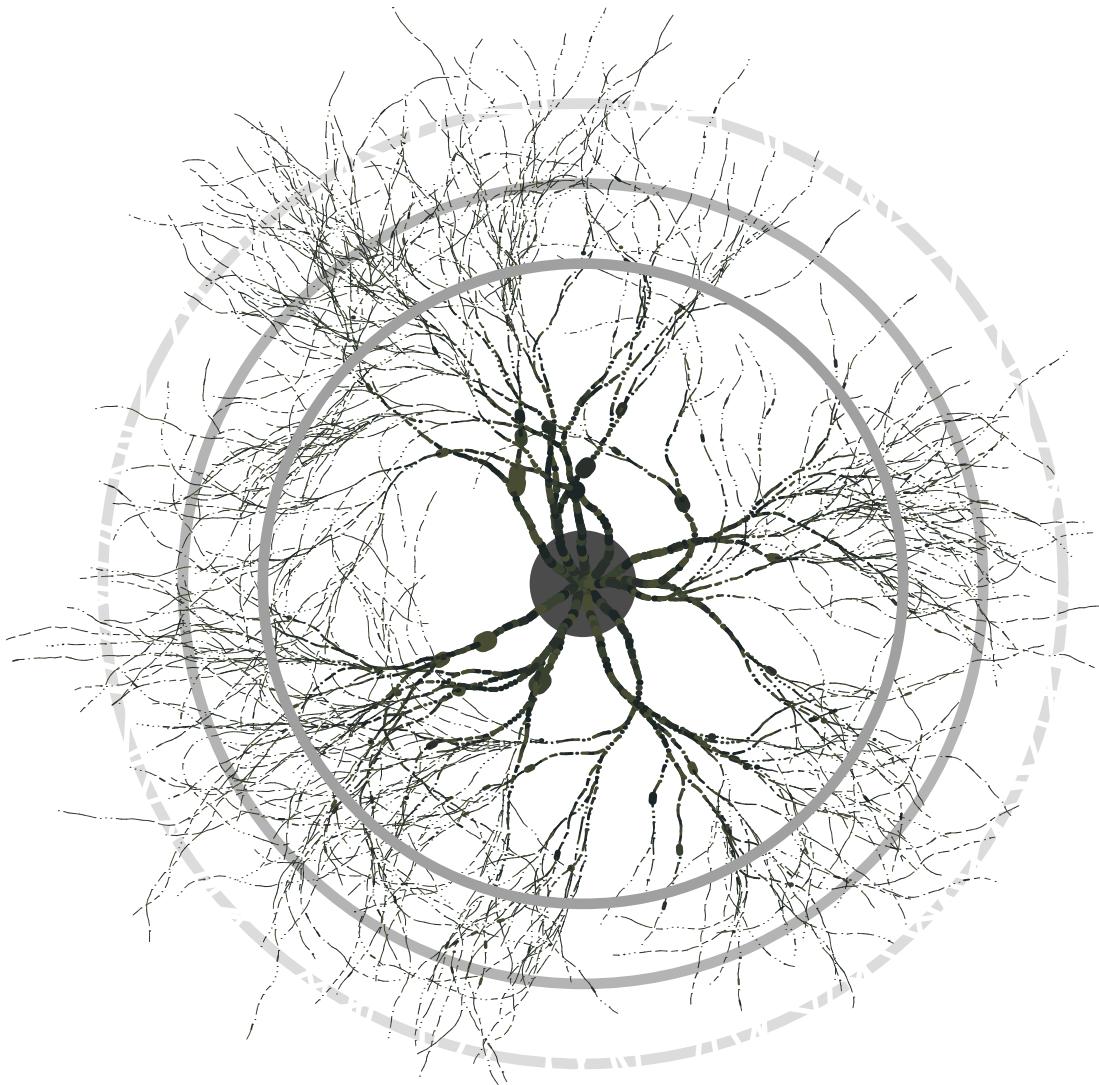


Figura 6.20: Recursividade gerativa.

## 6.3 | Sumário

---

Neste capítulo você aprendeu um pouco mais sobre o que são as curiosas estruturas autossimilares denominadas fractais. Sua beleza hipnotizante é tanto derivada da figura macro do fractal quanto da repetição infinita de regras e frações simples. Na parte que aludiu à programação você viu que a recursividade computacional consiste na chamada de uma função dentro si mesma, sendo a chave para criar a autossimilaridade dos fractais.

No quesito prático, você acompanhou um passo a passo do projeto de um fractal, incluindo como derivar as regras de desenho e condições de parada da recursividade. Por último, foram mostradas múltiplas maneiras de se utilizar formatações para conceder um estilo único em sua imagem, ou adicionar variedade a sua arte.

[ CAPÍTULO 7 ]:

# Emergência

Causa e efeito

Uma das primeiras séries de peças artísticas concebidas com o Processing foi fruto de um de seus fundadores, Casey Reas, que a batizou como *Process*. Cada uma dessas obras foi individualmente numerada com um código que representava a visualização de um processo diferente. Casey definiu um *processo* como um ambiente construído para elementos, que determina a maneira da qual suas relações e interações serão visualizadas. Por sua vez, elementos, ou *agentes*, são sistemas computacionais compostos por uma forma, como um círculo ou ponto, e um ou mais comportamentos, como se movimentar aleatoriamente pela janela de exibição. A intenção de Casey foi mostrar que o comportamento descrito individualmente pelos elementos era simples e desinteressante, mas a interação entre um número elevado de processos e elementos tornava a imagem final altamente imprevisível e única. A isso se dá o nome de *emergência*, um movimento inesperado para o além daquilo que você pudesse prever que regras simples seriam capazes de criar. Esta filosofia, de que a totalidade do resultado final é mais valorizado do que a contribuição individual, é um dos pontos centrais da linguagem Processing.

Uma maneira de entender mais claramente o processo emergente, e como a ação local resulta em uma consequência global, é examinando modelos encontrados na natureza. Imagine um cardume composto por milhares de peixes que se movem como um grande e único ser. O cardume é o comportamento emergente que surge, não de uma coordenação central dos peixes, que neste caso são os agentes, mas sim das regras individuais que cada um deles segue. Podemos rapidamente imaginar que alguns dos objetivos de um peixe são a alimentação, a reprodução e a sobrevivência (escapar de predadores), todos esses tornam-se mais fáceis quando em um grupo. Simultaneamente, os peixes usam suas percepções (audição, visão, olfato e paladar) para se manter perto de outros peixes, localizar alimentos próximos e evitar colisões. O cardume é formado pela interação de cada um dos elementos que o compõe.

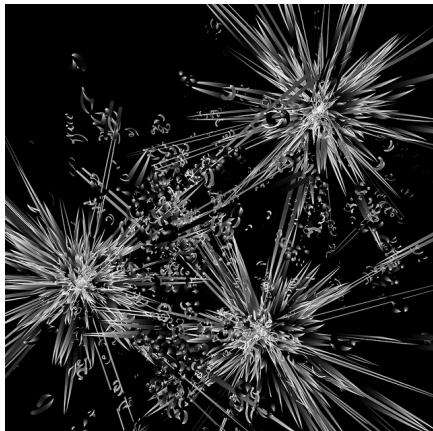
Na computação existem diversas simulações clássicas que abraçam a ideia de emergência no sentido de que o pontual explode em complexidade. Algumas das mais comuns são o jogo da vida de John Conway, a colônia de formigas, a floresta de Reeves e os *boids* de Reynolds. Todos esses algoritmos são casos clássicos para ilustrar a emergência, seja visando imitar a natureza ou formando um processo puramente sintético. Infelizmente, a maioria dessas simulações pode ser encaixada em duas categorias indesejadas para aqueles iniciantes na computação, seja programador ou artista. Em um extremo elas são fáceis de



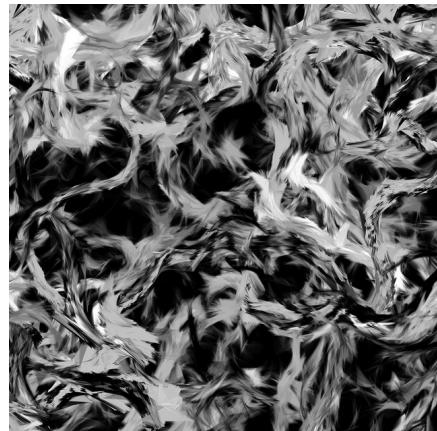
Figura 7.1: Cardume de sardinhas<sup>a</sup>, cortesia do fotógrafo Mark Harris.

<sup>a</sup>Todos os direitos reservados pelo artista original.

programar, mas difíceis de se entender o propósito ou a origem do raciocínio, como por exemplo o jogo da vida de Conway. E no espectro oposto elas são fáceis de entender e difíceis de programar, como os *boids* de Reynolds que imitam a murmuração de pássaros, mas requerem um uso exacerbado de vetores, condicionais, repetições e otimizações locais. Todavia nem tudo está perdido, existe um terceiro método que iremos examinar, mais brando na programação e no entendimento, baseado em *Process*, figura 7.2, o trabalho citado de Casey Reas.



(a) Process 9 (A)<sup>b</sup>.



(b) Process 18 (B)<sup>b</sup>.

Figura 7.2: Representações gráficas de processos emergentes: *Process*. Cortesia do próprio artista, Casey Reas.

<sup>b</sup>Todos os direitos reservados pelo artista original.

## 7.1 | Agente autônomo

Os principais autores da obra final são formalmente nomeados de agentes autônomos, reforçando a afirmação que eles agem localmente e de acordo com suas próprias intenções. Eles são constituídos por três<sup>1</sup> características principais:

- **Forma:** É a representação física do agente. Normalmente é um ponto, linha, círculo ou qualquer outra figura, geométrica ou não.
- **Percepções:** É o processo de reconhecimento e interpretação das informações que os agentes têm acesso para tomada de decisões. Por exemplo, um elemento pode verificar se está próximo de outro, ou se está contido dentro de uma região específica do espaço (como um círculo ou quadrado).
- **Comportamentos:** São as ações que os agentes são capazes de expressar. Algumas delas podem ser se movimentar pelo espaço, mudar sua direção ou interferir com a trajetória de outros elementos.

<sup>1</sup>Em *Process Reas* define apenas duas características: forma e comportamento. No entanto iremos trabalhar com três, pois a separação entre percepções e comportamentos é mais natural.

tos. Aqui também estão contidos os objetivos globais dos agentes, tais como se movimentar aleatoriamente pela janela de exibição ou sempre se aproximar ou afastar do elemento mais próximo a ele.

Do ponto de vista da programação, um agente é composto por múltiplos componentes heterogêneos. Suas formas são representadas por atributos, como posição e tamanho, e suas percepções ou comportamentos particulares podem ser descritos através de funções internas. É interessante que essas características estejam organizadas e acessíveis por agente, tornando-o uma unidade complexa. Felizmente você estudou na seção 2.6 que as classes permitem satisfazer todas essas necessidades ao agrupar diferentes estruturas de programação e produzir objetos completamente individualizados e autocontidos.

Nesta seção você dará vida a uma série de agentes através das instanciações de uma classe geradora. Portanto vamos começar definindo, arbitrariamente, o que iremos programar. Suponha um elemento simplificado cujo único propósito é dar um passo em linha reta, com uma determinada direção, a cada instante de tempo. Simultaneamente ele deve ser capaz de perceber seu ambiente e reajustar sua direção para se manter dentro da janela de exibição. Com base nessa descrição podemos inferir que cada agente possuirá uma posição no espaço, um ângulo de movimentação e um tamanho de passo. Finalmente, iremos adicionar dois métodos de visualização que concederão forma ao agente, um para exibi-lo simplesmente como um ponto e o outro como uma reta, pois assim será possível acompanhar melhor sua trajetória ao longo do tempo. A classe abaixo, que descreve o agente, foi escrita com base nessas premissas e sua explicação é feita no próximo parágrafo.

### Código 7.1 Classe Agente

```
class Agente {  
    // Atributos:  
    float posXA, posYA, posX, posY;  
    float angulo;  
    float passo;  
  
    // Construtor:  
    Agente(float _posX, float _posY, float _angulo, float _passo) {  
        posX = _posX;  
        posXA = posX;  
        posY = _posY;  
        posYA = posY;  
        angulo = _angulo;  
        passo = _passo;  
    }  
  
    // Métodos:  
  
    // Atualiza a posição espacial do agente:  
    void atualizar() {
```

```

// Grava a posição anterior:
posXA = posX;
posYA = posY;

// Atualiza a posição atual:
posX = posX + passo*cos(radians(angulo));
posY = posY + passo*sin(radians(angulo));
}

// Forma do agente - Ponto
void exibirComoPonto() {
    stroke(0);
    strokeWeight(2);
    point(posX,posY);
}

// Forma do agente - Reta
void exibirComoReta() {
    stroke(0);
    strokeWeight(1);
    line(posXA,posYA,posX,posY);
}
}

```

O construtor do agente recebe as informações essenciais para a definição mínima do mesmo: uma posição inicial na janela de saída, um ângulo inicial, que representa a direção de sua trajetória, e o tamanho do passo, que especifica o quanto efetivamente de espaço o agente irá percorrer ao se movimentar a cada quadro.

O deslocamento do agente é feito através da função `atualizar()` que recalcula sua posição atual de acordo com sua direção e tamanho de passo. O movimento de elementos na computação é derivado da física e formalizado através de vetores<sup>2</sup>, mas este assunto é extenso e relativamente técnico, não sendo detalhado neste livro. Em vez disso existe uma maneira mais intuitiva de entender o movimento dos agentes usando as versáteis equações do círculo, [4.1](#), obtidas na seção de trigonometria. Para isso imagine o agente como sendo o ponto central de um círculo. O raio desse círculo é justamente a variável passo do agente, e a direção dele é o ângulo do arco, variável `angulo`. Desta forma, o ponto que está na borda do círculo é a posição futura do agente após ele dar um “passo”, e é obtida usando as fórmulas citadas. Toda vez que a posição é atualizada, o ponto futuro se torna o ponto presente e o círculo é deslocado para esse novo ponto, permitindo obter a próxima posição da mesma maneira. As figuras [7.3](#) e [7.4](#) ilustram essa

---

<sup>2</sup>Os vetores citados aqui são aqueles descritos na geometria analítica.

abordagem.

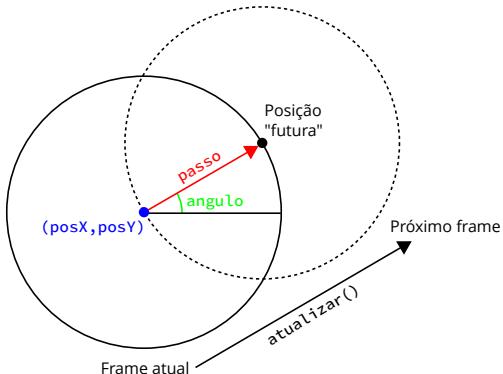


Figura 7.3: Movimentação do agente no espaço.

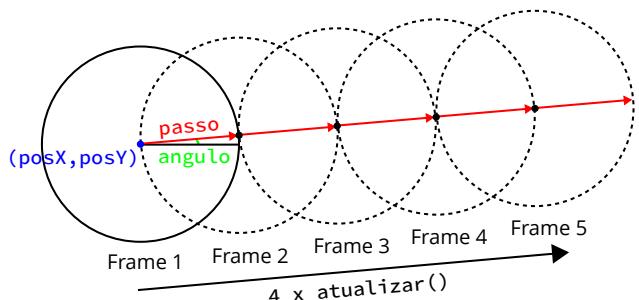


Figura 7.4: Movimentação do agente ao longo dos quadros.

As outras duas funções da classe Agente, `exibirComoPonto()` e `exibirComoReta()`, são responsáveis pelas suas formas. A exibição através de um ponto necessita apenas da posição atual (`posX` e `posY`), enquanto a que desenha uma reta necessita da posição atual e também da anterior (`posXA` e `posYA`). Não se esqueça de copiar todo o código 7.1 da classe para dentro do editor de texto do Processing.

Finalizada a análise da classe, podemos nos dedicar ao programa principal. Para popular o seu processo você pode criar um vetor do tipo Agente e adicionar um número qualquer de elementos a ele. Consequentemente o que resta escrever no código são as chamadas às rotinas para exibir cada um dos agentes e atualizar suas posições a cada quadro. Ambas devem ser feitas dentro da função `draw()` para que o movimento seja animado, veja o código abaixo:

### Código 7.2 Agentes autônomos na janela de exibição

```
Agente[] ags;

void setup() {
    size(300,200);
    background(255);

    int numAgentes = 10;
    ags = new Agente[numAgentes];

    for(int i = 0; i < numAgentes; i++) {
        // Cria e distribui os agentes aleatoriamente pela tela:
        ags[i] = new Agente(random(width),random(height),random(360),2);
    }
}

void draw() {
    for(int i = 0; i < ags.length; i++) {
        // Para cada agente declarado, atualiza e exibe ele:
        ags[i].atualizar();
        ags[i].exibirComoReta();
    }
}
```

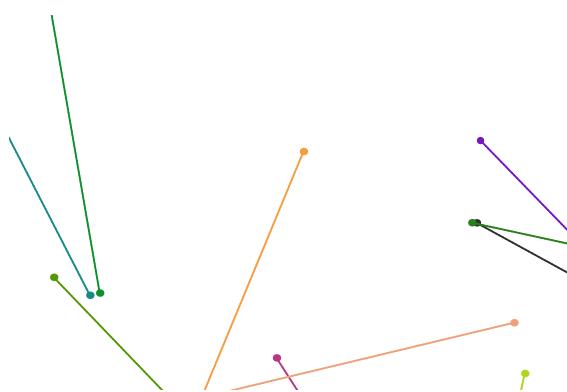


Figura 7.5: Movimentação de dez agentes.

Ao executar o código proposto você deve ter visto algo parecido com a figura 7.5<sup>3</sup>, confirmando que o agente está fazendo aquilo que você programou ele para fazer: andar em linha reta. Entretanto, perceba que o agente se locomove independentemente da janela de exibição e rapidamente sai do nosso campo de visão. Nossa próxima meta é impedir que isso aconteça.

Vamos começar entendendo melhor o espaço de restrição do agente que em nosso caso é a janela do Processing, ilustrada na figura 7.6. Lembre-se que é você quem define o tamanho da tela através da função `size()` e, portanto, possui exatamente as coordenadas e dimensões da mesma. Esses dados são suficientes para que você programe o agente de forma que ele detecte esses limites e não se mova além deles.

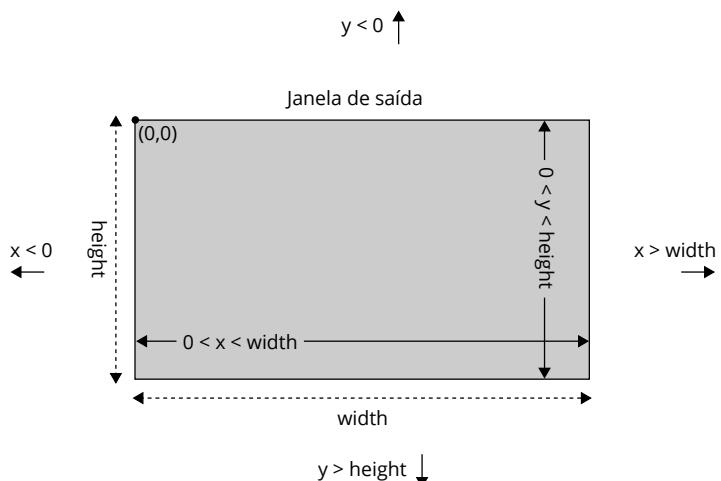


Figura 7.6: Janela de exibição e seus limites.

Note que o agente está em um plano bidimensional e sua posição é definida por uma parcela horizontal e uma vertical, sendo possível saber se ele está dentro ou fora da janela usando suas coordenadas. Se a posição horizontal do agente for menor que zero ou maior que o comprimento da tela (`width`), ou então se sua posição vertical for menor que zero ou maior que a altura da tela (`height`), o agente estará fora da janela de exibição e do nosso campo de visão, veja a figura 7.7.

<sup>3</sup>O processo ilustrado é o mesmo do código 7.2. Neste livro foram usadas cores apenas para evidenciar os agentes individualmente uma vez que a animação não é possível.

O próprio agente deve ser capaz de perceber o espaço em que se encontra, entender se sair dele e tomar as devidas ações para contornar esse evento. Ele pode, por exemplo, retornar para os limites da janela de exibição e alterar o ângulo da sua direção, de forma a evitar o movimento de saída da zona restrita. A maneira fisicamente correta de mudar o ângulo de um objeto qualquer quando ele atinge uma superfície é criando uma simetria de ângulo de incidência ( $\phi_i$ ) e reflexão ( $\phi_r$ ), mas para contribuir com a parcela gerativa de nossa obra iremos adicionar o caos e fazer com que esse ângulo varie aleatoriamente. Mesmo que tal fato não corresponda à realidade, ele faz parte do processo da experimentação e podemos criar as regras que quisermos. A figura 7.8 ilustra esse conceito.

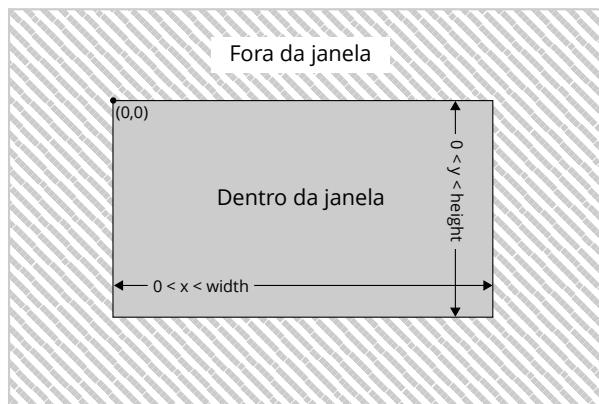


Figura 7.7: Posições fora do limite da janela.

Veja que você está adicionando mais um comportamento a esse agente, em que ele verifica se está fora da janela de exibição e, se estiver, retorna para uma posição dentro dela e depois altera sua direção. Isso corresponde a um novo método a ser adicionado na classe Agente, que pode ser visto no código a seguir:



Figura 7.8: Possíveis ângulos de reflexão.

```

void detectaBordas()
{
    // Se o agente sair da região de limite, altera sua direção de movimento:
    if(posX < 0 || posX > width || posY < 0 || posY > height) {
        angulo = random(360);
    }

    // Se o agente sair da região de limite, ele volta para a mesma:
    if(posX < 0) {
        posX = 0;
    }
    else {
        if(posX > width) {
            posX = width;
        }
    }
    if(posY < 0) {
        posY = 0;
    }
    else {
        if(posY > height) {
            posY = height;
        }
    }
}

```

Agora, além de chamar as funções `atualizar()` e `exibirComoReta()` você tem que fazer com que o agente teste sua posição chamado o método `detectaBordas()`.

```

for(int i = 0; i < ags.length; i++) {
    ags[i].detectaBordas();
    ags[i].atualizar();
    ags[i].exibirComoReta();
}

```

Ao executar o código após essas alterações, você verá que o agente se limita a ficar sempre dentro da janela de exibição, criando um padrão menos previsível, como o da figura 7.9. O seu agente autônomo está completo, possuindo uma *forma* de linha, e dotado de *percepções* e *comportamentos*, que o impele a sempre andar em linha reta e permanecer no campo de visão do programador.

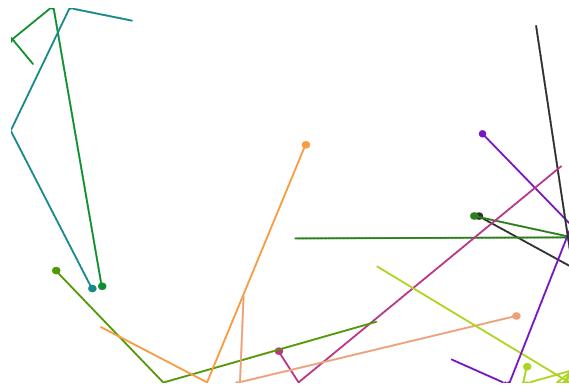


Figura 7.9: Agentes após as alterações nos seus comportamentos.

## 7.2 | Processo

---

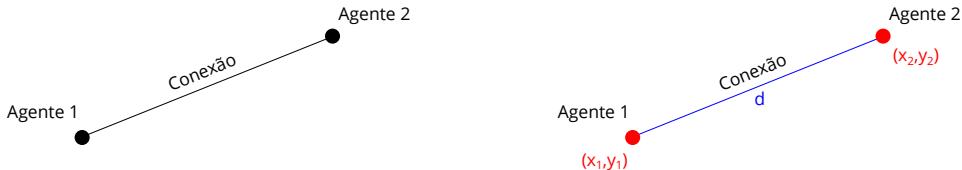
Muito bem, uma vez definidas as propriedades dos seus agentes, você pode focar na concepção de um processo. Isso naturalmente invocará uma pergunta fundamental:

*Mas afinal, o que exatamente é o “processo”?*

Mesmo que possa parecer muito peculiar, você é livre para responder essa pergunta como quiser, não há uma resposta certa ou errada. O que de fato existe é a sua capacidade criativa de inventar um processo utilizando os agentes, com o propósito singular de explorar os diversos padrões emergentes provenientes das interações entre eles mesmos e o ambiente em que estão inseridos. Esta é a parte subjetiva da computação artística. Por exemplo, você pode desejar visualizar o que acontece quando você desenha um círculo cada vez que dois agentes colidem; ou então desenhar triângulos com vértices nas posições dos agentes sempre que eles se aproximarem uns dos outros; ou quem sabe fazer com que um agente absorva o outro e aumente de tamanho. As possibilidades são infinitas, assim como a quantidade e diversidade de imagens que cada processo é capaz de gerar. Analisar com detalhes todos os cenários que podem ser idealizados como um processo é matéria para um livro por si só, portanto iremos dissecar apenas um estudo de caso.

Vamos começar com uma ideia simples e abandonar o desenho da trajetória dos agentes, optando por traçar uma conexão entre eles caso os mesmos se encontrem próximos. Perceba que neste exato ponto existe uma interface crucial entre o conceitual e o prático, entre o que você deseja fazer e como fazer isso. Você pode maximizar suas chances de sucesso nessa etapa, e em demais projetos, se focar em três aspectos centrais: estabelecer os dados que você possui, compreender como os mesmos podem ser usados para atingir seus objetivos e, se necessário, implementar novas funcionalidades. Esse fluxo de ação, para o processo de ligar os agentes através de retas, está ilustrado na figura 7.10.

Note que você tem acesso, ou pode encontrar, todas as variáveis que precisa, basta dividir o problema



$(x_1, y_1)$  = Coordenadas do primeiro agente no espaço  
 $(x_2, y_2)$  = Coordenadas do segundo agente no espaço  
 $d$  = Distância calculada entre os agentes

(a) O objetivo final é conectar os agentes.

(b) Informações disponíveis a serem usadas.

Figura 7.10: Estratificação do processo para dois agentes.

em etapas. Um último detalhe que deve ser esclarecido é que, apesar da figura 7.10 contemplar apenas dois agentes, na realidade existe um número qualquer de elementos simultaneamente próximos. Desta forma, para cada agente que estiver analisando, você deve comparar com todos os demais e calcular a distância entre eles. Se essa distância for menor que certo valor então você deverá traçar uma reta unindo eles, caso contrário não. Você pode cobrir todos esses casos se utilizar de uma estrutura de repetição aninhada, conforme visto nas seções anteriores. No código abaixo é mostrado como incrementar o *loop* de atualização dos agentes com uma nova funcionalidade que realiza as comparações de distância entre os agentes e os desenho das retas de conexão:

```

for(int i = 0; i < ags.length; i++) {
    ags[i].detectaBordas();
    ags[i].atualizar();
    ags[i].exibirComoReta();

    float distMaxima = 50;
    for(int j = i + 1; j < ags.length; j++) {
        // Calcula a distância entre dois agentes:
        float distEntreAgentes = dist(ags[i].posX, ags[i].posY, ags[j].posX, ags[j].posY);

        // Se a distância entre os agentes for menor que 50 pixels,
        // conectamos eles através de uma reta:
        if(distEntreAgentes < distMaxima) {
            line(ags[i].posX, ags[i].posY, ags[j].posX, ags[j].posY);
        }
    }
}
  
```

Essa solução se resume a ter um olhar puramente matemático, em que os agentes não passam de pon-

tos num espaço bidimensional cuja distância é calculada através da fórmula da distância Euclidiana. No Processing, você pode encontrar a separação entre dois pontos quaisquer através da função `dist()` que recebe como argumentos as coordenadas x e y de cada um deles.

Fique atento a um outro fator importante, a segunda estrutura de repetição não começa do zero. Isto é feito para otimizar o desenho das linhas que conectam os agentes, evitando desenhar conexões sobrepostas que apenas deixariam sua simulação mais lenta. Tal ajuste permite que o programa se abstenha de desenhar linhas entre os mesmos agentes, por exemplo  $i = 1$  e  $j = 1$ , e linhas entre agentes já conectados anteriormente, como  $i = 1$  e  $j = 2$  e, posteriormente,  $i = 2$  e  $j = 1$ . Execute o código e veja a concepção visual, figura 7.11, do seu processo.

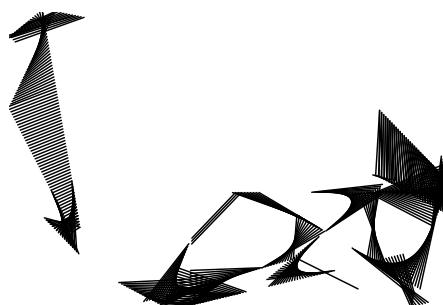


Figura 7.11: Processo emergente através da interação entre agentes.

A figura não é representativa no livro uma vez que ela é estática, mas na animação você deve ter percebido que sempre que um ponto se aproxima de outro surge uma linha conectando os dois. Essa é a prova que o seu processo está tecnicamente e funcionalmente perfeito. Observe que não programamos um *comportamento* emergente em si, já que as decisões tomadas por um agente não sofrem influências dos demais. Todavia, esta é uma *visualização* emergente, com uma estética particular, que se manifesta progressivamente, fruto das relações imprevisíveis entre os agentes.

## 7.3 | Extrapolação

---

Os agentes e o processo das seções anteriores naturalmente se fundem em um resultado emergente que é melhor observado ao longo do tempo em vez de instantaneamente. A sua representação visual é revelada quando aumentamos a escala da simulação, tanto referente ao número de elementos quanto ao tempo. Retorne à função `setup()` e altere os campos mostrados para os valores abaixo:

```
size(700,200);  
int numAgentes = 50;
```

aproveite e comente a exibição dos agentes por retas:

```
// ags[i].exibirComoReta();
```

Desta forma o acúmulo de desenhos gera uma imagem cuja complexidade não está somente ligada ao número de agentes, mas também ao tempo total de execução. Ironicamente, se você permitir que essa simulação execute por muito tempo, irá obter uma imagem que decepciona, algo parecido com um borrão preto, tal como a figura 7.12. A etapa de formatação é mais uma vez essencial para resolver esse problema. O código que você escreveu até agora considera apenas a parte técnica dos agentes e do processo, na qual são usadas as cores e transparências padrões do Processing que estão limitadas à visualização básica das formas através de linhas pretas. Dependendo da complexidade, do número de elementos da simulação e como eles interagem entre si, o resultado não fará jus ao que está realmente acontecendo.



Figura 7.12: Processo após executar 1000 frames.

Você já aprendeu a apreciar o poder de uma formatação bem pensada e poderá aplicar neste caso para agregar riqueza visual ao seu processo. O principal guia nesta etapa deve ser a criatividade do programador e artista, algo que novamente abre espaço para inúmeras possibilidades, além de ser muito difícil padronizar ou justificar tecnicamente. O que pode, e será feito, é identificar certos estilos de formatação e expor como programá-los usando o Processing.

O primeiro tipo de formatação, a dualidade, é um dos mais simples, e costuma funcionar relativamente bem em processos densos ou com muitos desenhos. Ela consiste em usar formatações que são opostas uma da outra para criar contrastes visuais. Por exemplo, desenhar linhas finas e grossas, pontos grandes e pequenos ou, neste caso em específico, a dualidade de cores. O preto e o branco são as cores extremas mais clássicas, mas você poderia escolher o azul e o vermelho, ou qualquer outra combinação contrastante. Você pode aplicá-la em seu código adicionando linhas que alterem a cor do desenho, internas à estrutura for aninhada responsável pelo desenho das conexões:

```
for(int j = i + 1; j < ags.length; j++) {  
    float distEntreAgentes = dist(ags[i].posX, ags[i].posY, ags[j].posX, ags[j].posY);
```

```

if(distEntreAgentes < distMaxima) {
    if(random(1) < 0.5) {
        stroke(255);
    }
    else {
        stroke(0);
    }
    line(ags[i]. posX, ags[i]. posY, ags[j]. posX, ags[j]. posY);
}
}

```

Atualmente nosso processo conta com um número relativamente elevado de agentes, determinando que seja usada uma proporção balanceada de cores para as conexões: metade das vezes ela deve ser preta e metade branca. Lembrando que se você quisesse uma maior proporção de linhas brancas (ou pretas), poderia alterar a condição que define a probabilidade desse evento acontecer.

```

(random(1) < 0.50) // 50% das linhas serão brancas.
(random(1) < 0.60) // 60% das linhas serão brancas.
(random(1) < 0.95) // 95% das linhas serão brancas. etc.

```

O resultado dessa formatação pode ser visto na figura 7.13. Perceba como a visualização do processo mudou, sendo possível ver reminiscências da posição e trajetória dos agentes. Nada mal, mas você pode deixar a formatação ainda melhor se em vez de usar cores completamente sólidas, você optar por certo grau de transparência e suavidade. Para isso inclua o campo de alfa ao colorir os traços. O resultado é mostrado na figura 7.14.

```

if(random(1) < 0.5) {
    stroke(255,50);
}
else {
    stroke(0,50);
}

```



Figura 7.13: Dualidade de cores sem transparência.

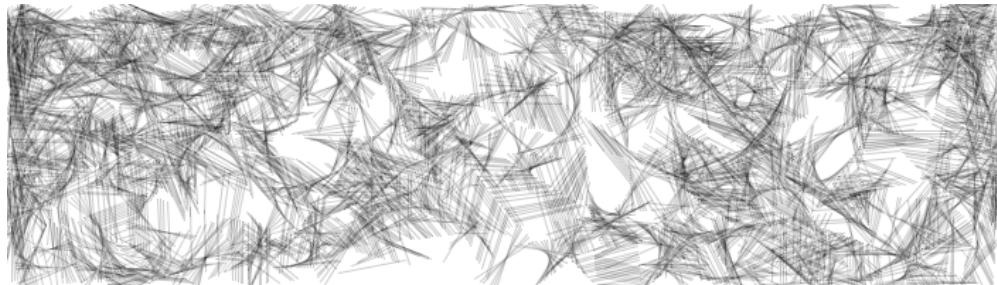


Figura 7.14: Dualidade de cores com transparência.

Agora, além do preto e do branco puros, existe o cinza, proveniente da combinação dessas cores. Isso reduz o contraste, mas adiciona uma maior profundidade visual de cores. Você ainda pode impor condições mais complexas de formatação. Por exemplo, quanto mais próximos dois agentes estiverem, mais opaca deve ser a cor das linhas de conexão e, quanto mais longe, mais transparente. A função map() é novamente a protagonista para transformar a variável que definirá a formatação, que é a distância entre os elementos, em um valor de transparência.

```
float alpha = map(distEntreAgentes,0,distMaxima,50,0);

if(random(1) < 0.5) {
    stroke(255,alpha);
}
else {
    stroke(0,alpha);
}
```

Ao escrever esse código, você está estabelecendo que a variável distEntreAgentes, que armazena a distância entre dois agentes e cujos valores podem estar entre 0 e distMaxima, agora será transformada em valores no intervalo 50 e 0 e passados para a variável alfa que representa a transparência da formatação. Execute o código modificado e sua janela de exibição se parecerá com a figura 7.15.

Formatações que não estão ligadas à forma também são válidas no ato de criar padrões visuais, em especial quando se trabalha com sistemas que possuem grandes números de elementos, sejam partículas ou um enxame de agentes autônomos. Uma delas é através do posicionamento inicial determinístico desses pontos no espaço. Em outras palavras, é você quem irá decidir onde os agentes começam em vez de delegar essa tarefa ao acaso. Esse raciocínio parece estar contra o princípio da arte gerativa, uma vez que a aleatoriedade e o ruído foram amplamente defendidos em seções anteriores devido a sua capacidade de emular o orgânico e criar imagens visualmente agradáveis. O grande dilema é que alguns processos podem ser fundamentalmente tão caóticos e carregados que dissolvem ou até mesmo eliminam o natural. Neste caso, possuir alguma figura familiar, como uma reta ou um círculo, contribui para atrair e direcio-

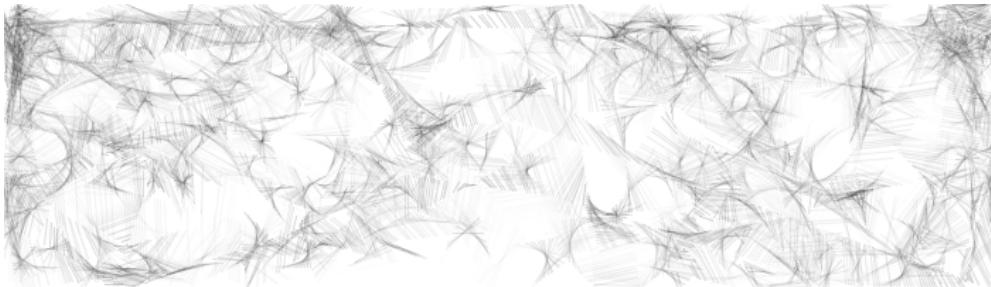


Figura 7.15: Dualidade de cores com transparência em função da distância.

nar o olhar, criando uma impressão de caos controlado, de união da forma e do processo, remetendo a padrões visualmente mais interessantes.

A segunda etapa da formatação de nosso processo será focada no posicionamento iniciais dos agentes. Uma distribuição homogênea de pontos ao longo de uma reta horizontal pode ser feita ao instanciar os elementos da classe Agente sempre com a mesma coordenada y. No entanto esta ação não é suficiente para diminuir o caos uma vez que o ângulo das direções dos agentes ainda será aleatório e, portanto, cabe a você também fixá-lo. Neste exemplo consideramos que o espaço de distribuição é uma reta completamente horizontal com uma angulação de 0°. Você pode criar um efeito dos agentes partindo dessa reta ao designar que eles tenham uma direção perpendicular a ela, o que implica que os ângulos dos agentes devem ser de +90° ou -90°, salvo alguns poucos graus aleatórios para não deixar o processo completamente mecânico. Compilando essa estratégia, você pode alterar a etapa de criação dos agentes no programa, atualizando a função setup() para o código mostrado abaixo. O processo final está ilustrado em 7.16.

```
void setup() {
    size(700,200);
    background(255);

    int numAgentes = 100;
    ags = new Agente[numAgentes];

    for(int i = 0; i < numAgentes; i++) {

        // 50% de chance do agente ir para cima ou para baixo.
        float angulo;
        if(random(1) < 0.5) {
            angulo = -90 + random(-10,10);
        }
        else {
            angulo = 90 + random(-10,10);
        }
    }
}
```

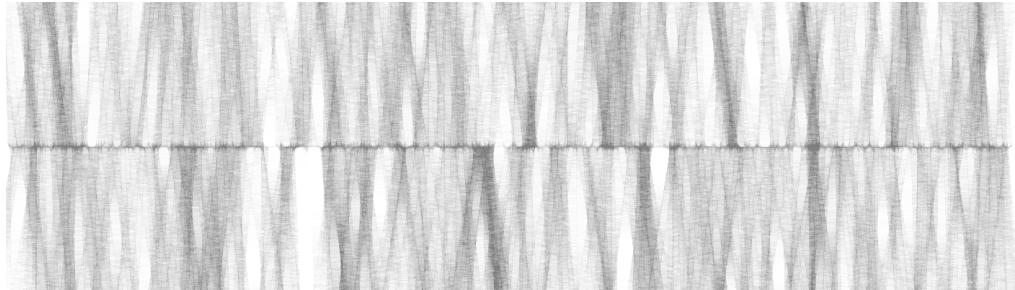
```

    }

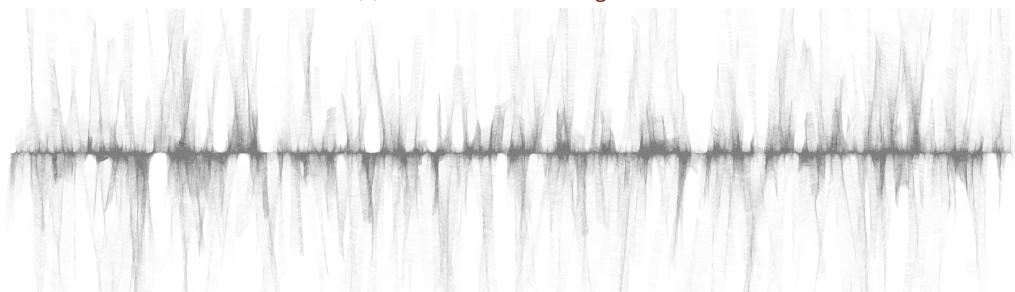
    ags[i] = new Agente(random(width),height/2,angulo,1);
}

}

```



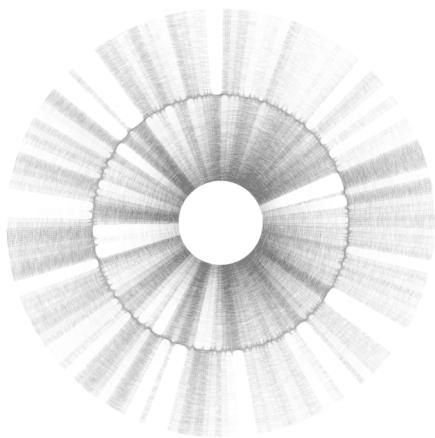
(a) Passo constante dos agentes.



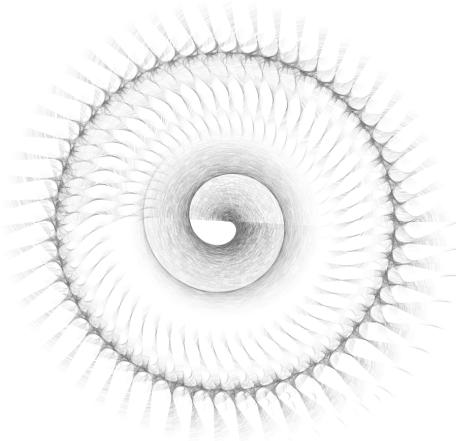
(b) Passo variável dos agentes.

**Figura 7.16: Distribuição inicial horizontal dos agentes.**

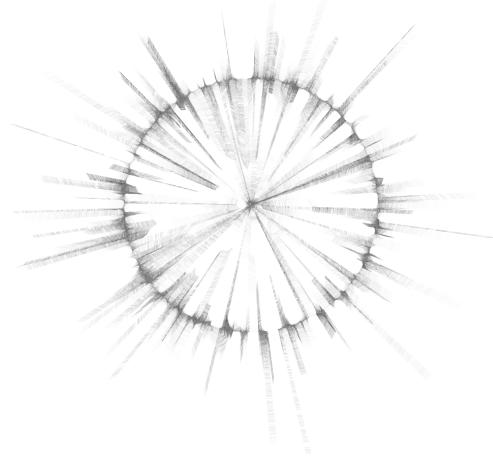
Você também pode experimentar importar uma distribuição circular inicial dos agentes, figura 7.17, obtida através das fórmulas do círculo, equações 4.1. A aplicação simultânea de todas essas técnicas é capaz de combinar os agentes, o processo e a visualização efetiva em incontáveis imagens que possuem o mesmo núcleo gerador, mas diferentes execuções. Na sua forma mais pura, você acabou de criar uma obra artística viva, gerativa e diferente a cada vez que você pressionar o botão para iniciar a simulação.



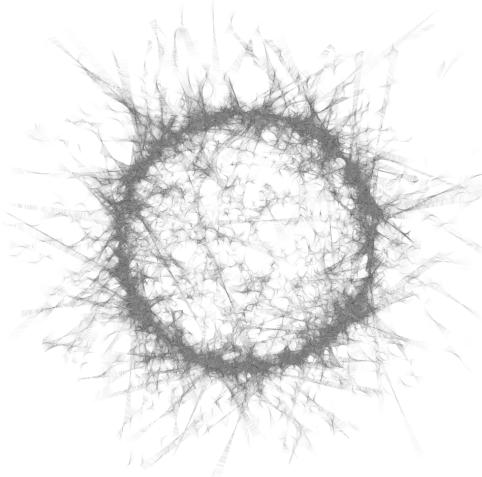
(a) Passo e posicionamento homogêneos.



(b) Posicionamento em espiral.



(c) Passo aleatório.



(d) Passo e posicionamento aleatórios.

Figura 7.17: Distribuição inicial radial dos agentes.

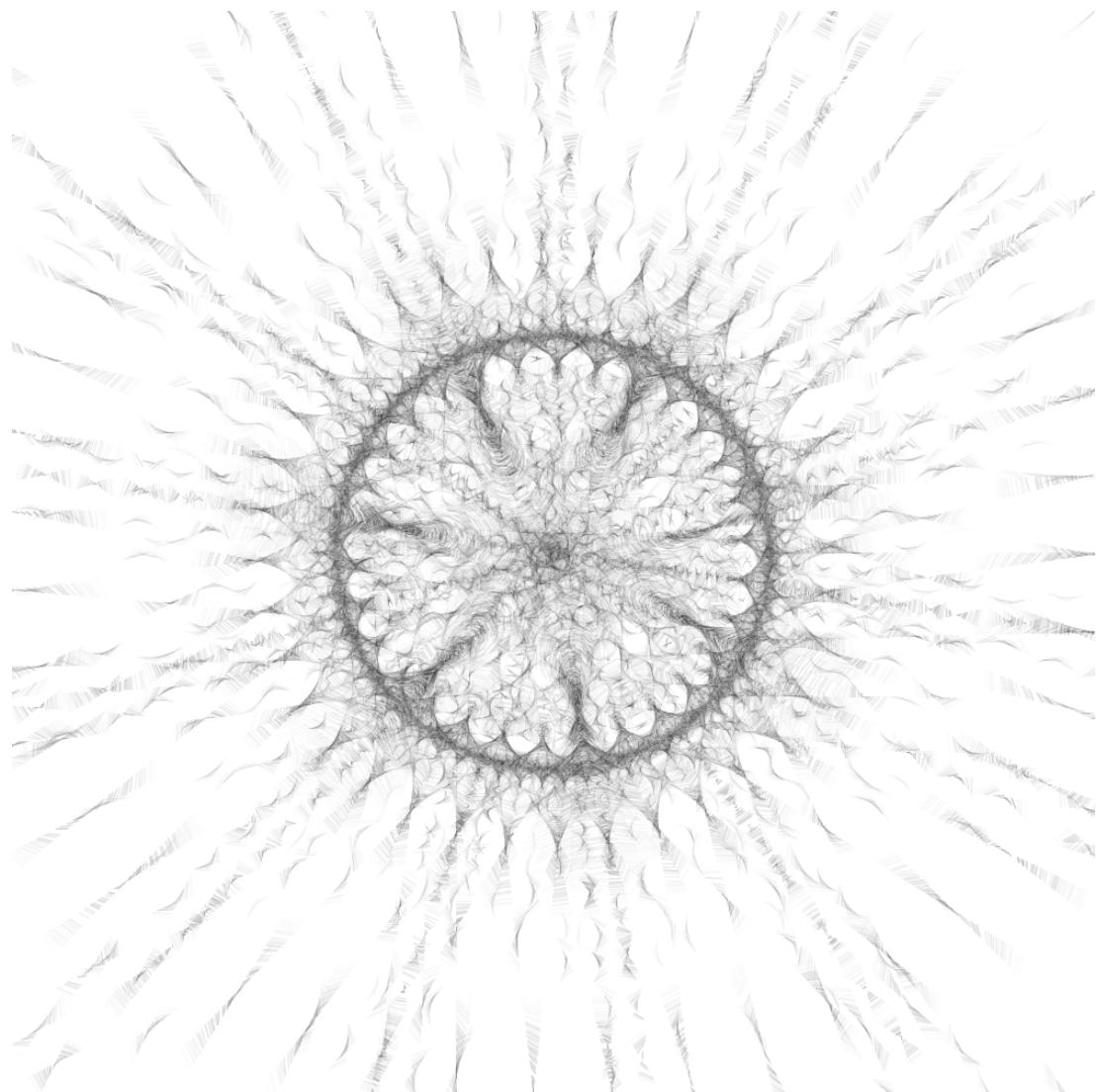


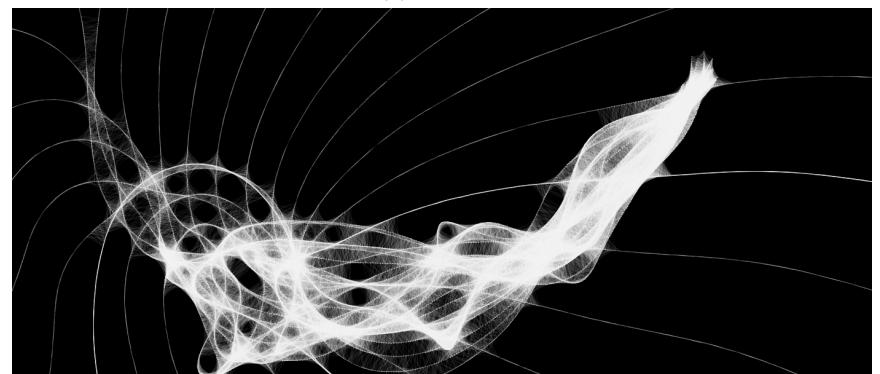
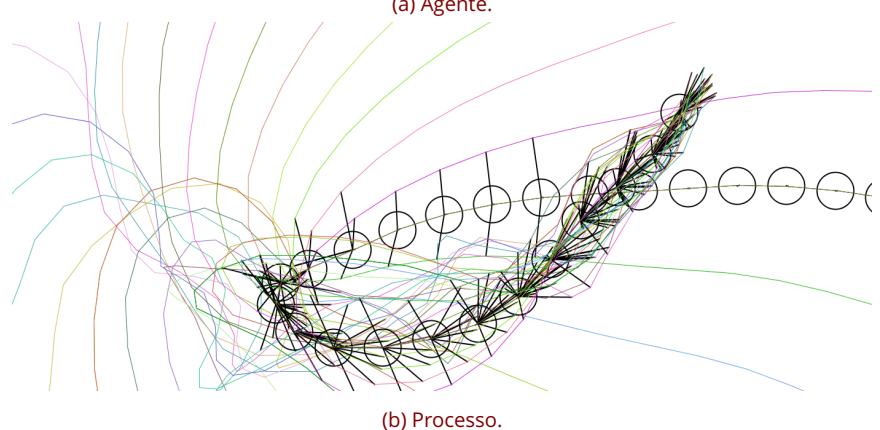
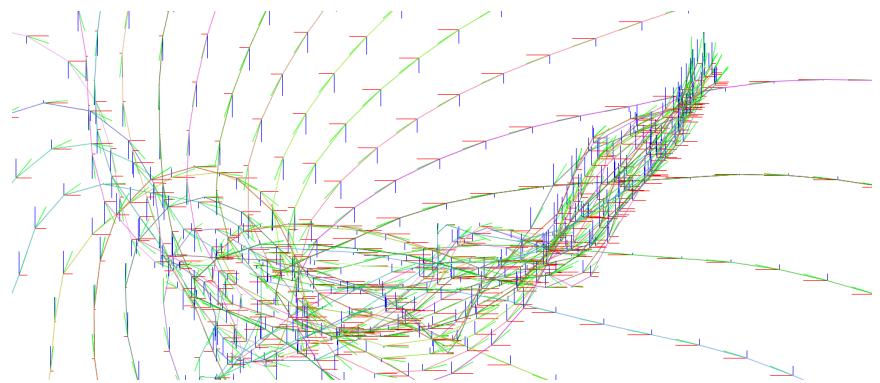
Figura 7.18: Distribuição circular com angulações desproporcionais.

## 7.4 | Sumário

---

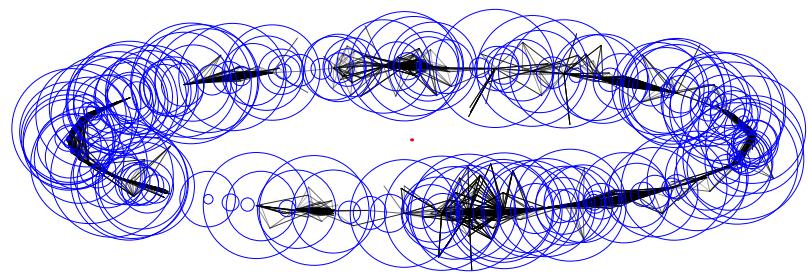
Neste capítulo foram explorados os conceitos de agentes autônomos e processos. Os agentes são uma estrutura de dados que emulam um ser responsivo dotado de forma, percepções e comportamentos. Eles tomam decisões locais baseadas em outros agentes ou de acordo com o próprio meio em que estão inseridos. Do ponto de vista técnico, você pôde perceber como as classes e objetos são importantes para modelar uma estrutura de dados complexa e unitária como é o caso dos agentes.

Em uma escala macro, as interações entre os agentes e o meio são capazes de gerar um resultado global sem coordenação central, mas organizado e único. A isso se dá o nome de emergência. Neste capítulo você aprendeu como programar, passo a passo, um sistema desse tipo. Você também acompanhou, com detalhes, como efetuar a transformação de ideias abstratas em interações concretas. Por último, você usou a seu favor o caos e as técnicas de formatação para estampar um estilo único ao seu processo.

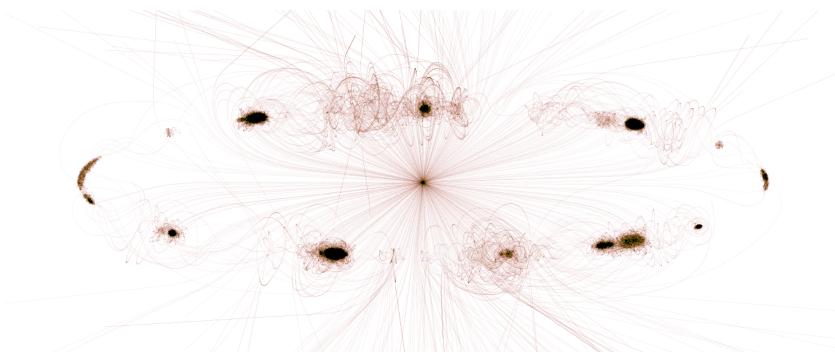


(c) Renderização.

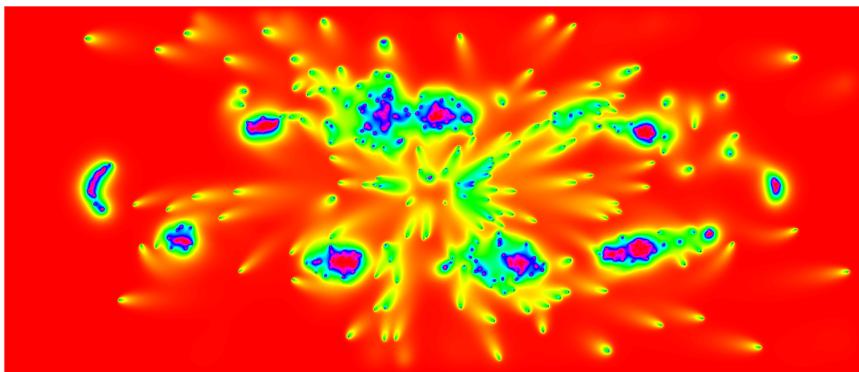
Figura 7.19: Etapas do processo 000164APR.



(a) Agente.



(b) Processo.



(c) Renderização.

Figura 7.20: Etapas do processo b74e10R.



[ CAPÍTULO 8 ]:

# O Código Transcendente

Libertação

O Caminho do Zen é um pequeno guia escrito pelo filósofo, escritor e orador Alan Watts, que pincela histórias<sup>1</sup> do Budismo e do Taoísmo até o desenvolvimento do Zen Budismo. Em uma de suas páginas podemos encontrar uma parábola em que um mestre, que toma chá com dois de seus estudantes, joga um leque para um deles e pergunta: "O que é isso?". Em vez de responder, o jovem abre o leque e se abana. O mestre responde - "Nada mal." - e passa para o outro aluno - "Agora você.". O estudante fecha o leque e coça o pescoço com ele. Em seguida, abre-o novamente, coloca um pedaço de bolo em cima e o oferece ao mestre. Isso é considerado ainda melhor, pois quando não há nomes, o mundo não é mais "classificado por limites ou barreiras". Veja que esse texto louva o pragmatismo ao expor que, quando deixamos de atribuir rótulos aos conceitos que aprendemos, estamos desbloqueando seu potencial, não ortodoxo, de uso. Mesmo sendo cuidadoso ao passar as lições, o fenômeno da restrição também acomete o livro que você está lendo, afinal foi utilizada a metodologia da teoria com foco na prática. Nesta última lição cabe a você combater essa mentalidade sendo como um aluno do Caminho Zen, imparcial, questionador e aberto, nutrindo-se da filosofia do *Gestalt*, da "visão do todo".

Agora que você chegou ao final de sua jornada você é capaz de associar os tópicos que aprendeu no livro com os tipos de imagens e padrões que eles podem gerar. Por exemplo, você pode usar relações trigonométricas para desenhar círculos e, mais genéricamente, polígonos, figuras radialmente simétricas e espirais. Linhas e retas, quando formatadas de maneira inteligente, criam imagens complexas, curiosas aos olhos de quem vê. A programação recursiva que é um ponto chave em muitas técnicas de soluções de problemas numéricos na matemática, na arte gerativa é empregada para formar figuras autossimilares, os fractais. Você presenciou como regras simples repetidas ao longo de uma escala espacial dão forma a

---

<sup>1</sup>Alan Watts, auto proclamado "animador espiritual", não é um autor clássico quando o assunto é Zen Budismo, mas o ponto aqui é ilustrar como a linguagem ou o próprio meio de ensino, qualquer que seja, pode limitar nossa capacidade de abstração se não formos críticos em nossas análises e interpretações.

essas estruturas altamente elaboradas. Ainda no tópico de que o simples é capaz de conceber o complexo, você estudou a emergência, amplamente encontrada na natureza e em nossa própria sociedade. Ela é a manifestação suprema de como pequenas ações descorrelacionadas e locais podem explodir em eventos globais entrelaçados. No ápice de todos esses tópicos, você pôde testemunhar como o caos penetra no cerne da programação criativa, liberando o não determinismo. Sua principal consequência no gerativo é a multiplicação ao infinito dos detalhes e formas em seus padrões, tornando-os virtualmente únicos cada vez que você executar seu código.

Mesmo sabendo de tudo isso, assim como na parábola de Watts, nenhum desses tópicos é *exclusivamente* autocontido, eles não são feitos apenas para as tarefas aqui citadas. Trigonometria não desenha só círculos, fractais são apenas uma perna da recursividade e assim por diante. Os exemplos apresentados neste livro são os casos clássicos do uso dessas técnicas e não devem ser os limitadores de suas ideias, e sim os impulsionadores. Este último capítulo é dedicado a mostrar outras maneiras de se empregar as técnicas aprendidas e, paralelamente, obras de programadores/artistas que não tiveram a pretensão de explicar como usar a programação, apenas a usam com o intuito singular de criar. É neste estágio que o *fator humano* realmente é revelado.

Absorva as informações deste capítulo de forma puramente conceitual, uma inspiração do que pode ser atingido com a programação (criativa ou não). Em alguns momentos serão usadas funções e lógicas que implorariam por explicações aprofundadas, mas que infelizmente não cobriremos dada a própria premissa do livro. Tal fato será contrabalançado por exemplos simples que servirão como fomentadores de projetos e faróis de guia em suas futuras jornadas.

## 8.1 | Tratamento de imagens

---

O tratamento de fotos e imagens é uma atividade inconscientemente consolidada em nosso cotidiano. Hoje ela não é mais restrita aos profissionais do mundo da fotografia que usavam softwares avançados e crípticos, ela é presente como funcionalidade em redes sociais e álbuns digitais. Tais programas contam com inúmeros filtros de correção de contraste, saturação e luminosidade, em adição aos filtros artísticos de realce de cores ou estilização da foto. Do ponto de vista computacional, esses programas são capazes de alterar elementos visuais através de informações que extraem das próprias imagens, tais como cores e posicionamento espacial do pixel. Estes, por sua vez, são usados como entradas para um algoritmo responsável por, a nível atômico, executar operações pixel a pixel respeitando uma lógica interna. A manipulação de imagens é, atualmente, assunto de vários artigos científicos que buscam melhorias na velocidade e qualidade do processamento.

A linguagem Processing conta com variáveis e funções destinadas à leitura e escrita de arquivos de imagens (.gif, .jpg e .png dentre outros) e é preparada para buscar propriedades das mesmas. Esta é uma grande vantagem uma vez que permite que desenvolvamos nossas próprias rotinas de experimentação, combinação ou alteração de imagens por meio de filtros customizados. No cenário da arte computacional

encontramos artistas que compartilham desse sentimento exploratório como o espanhol Sergio Albiac, que conta com uma expressiva coleção de peças que brilhantemente fundem a fotografia clássica com técnicas gerativas. O resultado, figura 8.1, são imagens de natureza elusiva, não delimitadas, reflexo de uma incubação enraizada no caos.

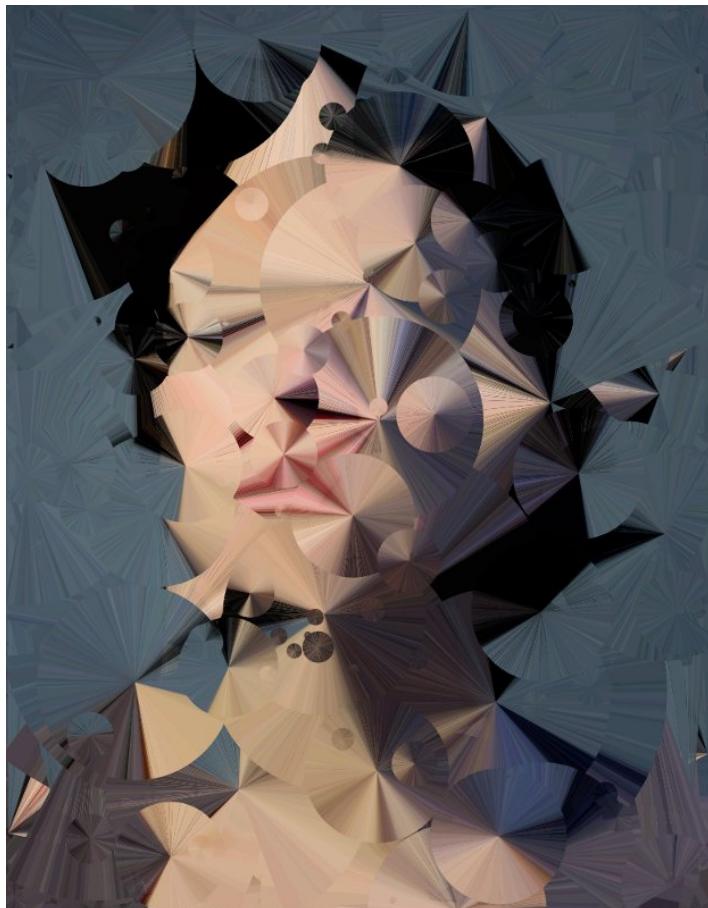


Figura 8.1: *Statistical beauty*<sup>a</sup>, cortesia do próprio artista, Sergio Albiac.

<sup>a</sup>Licenciada sobre Creative Commons BY-NC-ND.

Demonstraremos a capacidade do Processing replicando uma técnica simultaneamente simplória e mesmerizante, o pontilhismo, desenvolvida no final de 1880 por Georges Seurat e Paul Signac como uma vertente do Impressionismo. Ela é caracterizada pela reprodução de uma imagem através da deposição

de tinta por pontos, exercício incrivelmente repetitivo e demorado na arte tradicional em que, por outro lado, o computador se sobressai. No código abaixo é carregado um arquivo de imagem, intitulado “Imagem.png”<sup>2</sup>, através da função `loadImage()`, armazenando seu conteúdo na classe `PImage`<sup>3</sup>. Em seguida um pixel aleatório dentro dessa imagem tem sua cor amostrada através do método público `get()` e utilizada para preencher um ponto de densidade constante, representado por uma elipse, em sua mesma posição. O processo é completado ao repetirmos esses mesmos passos milhares de vezes, veja o resultado em [8.2](#).

```
// Estrutura de dados que armazena arquivos de imagem:  
PImage imagem;  
  
// Tamanho e número de pontos que compõem o pontilhismo:  
float r = 5, n = 100000;  
  
void setup() {  
    size(360,300);  
    background(255);  
    noStroke();  
  
    // Carrega o arquivo de foto que está dentro da pasta "data":  
    imagem = loadImage("Imagem.jpg");  
  
    for(int i = 0; i < n; i++) {  
        // Escolhe um pixel aleatório da imagem carregada e obtém sua cor:  
        int x = round(random(imagem.width-1));  
        int y = round(random(imagem.height-1));  
        color c = imagem.get(x,y);  
  
        // Colore e posiciona o ponto (representado por uma elipse)  
        // no local em que foi amostrado:  
        fill(c);  
        ellipse(x,y,r,r);  
    }  
}
```

<sup>2</sup>Este arquivo deve estar dentro da pasta “data” no diretório em que você salvou o programa. Isso pode ser feito acessando a barra de menus do PDE do Processing, depois em *Sketch e Adicionar Ficheiro....* Na janela que abrirá, escolha o seu arquivo de imagem. O procedimento foi detalhado na seção [1.3](#).

<sup>3</sup>A classe `PImage`, nativa do Processing, tem como propósito guardar imagens e facilitar o acesso aos atributos e pixels da mesma. Ela ainda conta com métodos de manipulação de imagens do tipo máscara, filtros, combinações e redimensionamento.



(a) Imagem original.



(b) Pontilhismo.

Figura 8.2: Baseado em: "Hipopótamo", presente de Edward S. Harkness, 1917, cortesia<sup>b</sup> do Metropolitan Museum of Art, New York.

<sup>b</sup>Todos os direitos reservados pela instituição.

## 8.2 | Áudio/Visual

O gerativo como uma peça estática foi amplamente explorado neste livro, mas ele também pode ser empregado em instalações áudio visuais vivas que incessantemente se regeneram e mutam, dando vida a peças de longa duração. Neste tipo de programa, áudio e vídeo trabalham sinergicamente, com papéis complementares, enriquecendo a obra de maneira singular. Trabalhos assim podem ser encontrado ecléticamente decorando shoppings, projetados em fachadas de prédios e até acompanhando músicas em casas noturnas. Um dos exemplos mais belos e bem executados dessa integração é do artista chinês Ruiwen Guo, conhecido como Raven Kwok. Ele desenvolveu uma série de algoritmos cujas saídas são visualizações, figura 8.3, que respondem a notas e batidas em músicas de um álbum<sup>4</sup> completo, efetivamente certificando a versatilidade de códigos gerativos.

O Processing possui módulos externos<sup>5</sup> para carregar arquivos de som, viabilizando a reprodução, sequen-

<sup>4</sup>New Age | Dark Age por Karma Fields

<sup>5</sup>O PDE nativo não inclui o processamento de arquivos de som, funcionalidades adicionadas pelas bibliotecas *Minim*, de Fede e Mills, ou *Sound* da Processing Foundation.



Figura 8.3: *Stickup Screenshot 09*<sup>c</sup>, cortesia do próprio artista, Raven Kwok.

<sup>c</sup>Todos os direitos reservados pelo artista original.

ciamento, mixagem e até mesmo análise de frequência sonora através da FFT<sup>6</sup>. Cabe ressaltar que, assim como o processamento de imagens, a análise de áudios é um tema contemporâneo, prestigiado por muitos pesquisadores e, por enquanto, ainda não é uma ciência integralmente desbravada. Notas musicais e batidas são conceitos complexos do ponto de vista de sinais de dados, com formas de ondas heterogêneas, que requerem o uso de princípios como detecção de energia e segmentação em tempo/frequência, além de sofrerem sobreposições e múltiplas interferências nas gravações<sup>7</sup>. No projeto citado de Kwok as imagens gerativas são produtos do Processing, mas o ato colossal de separar e escrever o sequencial sonoro das faixas foi, em sua maioria, manualmente levantado pelo artista. Somente com essa estratégia ele conseguiu garantir o sincronismo perfeito entre áudio e vídeo.

<sup>6</sup>Transformada Rápida de Fourier ou, do inglês, *Fast Fourier Transform*

<sup>7</sup>Müller, M. et al. (2011). *Signal Processing for Music Analysis*. IEEE Journal of Selected Topics in Signal Processing 5(6): pags.1088-1110.

O potencial para desenvolver peças completamente automatizadas na esfera do reativo ao áudio, ainda que limitado no Processing, é válido de ser demonstrado em sua forma básica. O primeiro passo é instalar a biblioteca de áudio seguindo o descriptivo mostrado na seção 1.3. Ao abrir a janela de importação, digite “Minim” no campo de filtro, clique em *Install* e, quando terminar, reinicie o PDE. Foi dada preferência a essa biblioteca no lugar da Sound por apresentar, em geral, uma melhor compatibilidade com os sistemas operacionais. As funções da Minim só serão habilitadas após sua inclusão explícita em um programa através do comando `import ddf.minim.*;`. O código abaixo é um dos mais simples que pode ser escrito para expor as operações com áudio. Nele carregamos<sup>8</sup> um arquivo de música, “Audio.mp3”<sup>9</sup>, e dinamicamente projetamos a amplitude RMS<sup>10</sup> do sinal na janela de exibição, figura 8.4.

```
// Biblioteca e variáveis Minim
import ddf.minim.*;
Minim minim;
AudioPlayer audio;

void setup() {
    size(300, 300);
    background(255);

    // Cria o objeto minim - obrigatório para o funcionamento correto:
    minim = new Minim(this);

    // Carrega o arquivo de áudio que está dentro da pasta "data" e faz o playback:
    audio = minim.loadFile("Audio.mp3", 1024);
    audio.loop();
}

void draw() {

    // O sinal de áudio em cada instante é dividido em 1024 amostras com
    // valores entre -1 e 1 para cada canal (direito e esquerdo).
    // Aqui fazemos uma média dos canais e armazenamos o valor para
    // posteriormente desenhar o padrão.
    float[] sinalAudio = new float[1024];
    for(int i = 0; i < sinalAudio.length - 1; i++) {
        float mediaAtual = (audio.left.get(i) + audio.right.get(i))/2.0;
    }
}
```

<sup>8</sup>Este arquivo deve ser importado para o seu programa usando o mesmo procedimento citado na seção anterior.

<sup>9</sup>Faixa “Horizon Ending”, cortesia do artista Soft and Furious. Licenciada sobre [Creative Commons CC0 1.0 Universal](#).

<sup>10</sup>Raiz do Valor Quadrático Médio ou, do inglês, *Root Mean Square*

```

        sinalAudio[i] = mediaAtual;
    }

strokeWeight(2);
stroke(random(255),random(255),random(255));

// Transforma a amplitude RMS do sinal para pontos em um círculo:
float divAngulo = 360.0/1024.0;
for(int i = 0; i < sinalAudio.length - 1; i++) {
    float x1 = width/2 + (80 + 80*sinalAudio[i])*cos(i*radians(divAngulo));
    float y1 = height/2 + (80 + 80*sinalAudio[i])*sin(i*radians(divAngulo));
    point(x1,y1);
}
}

```

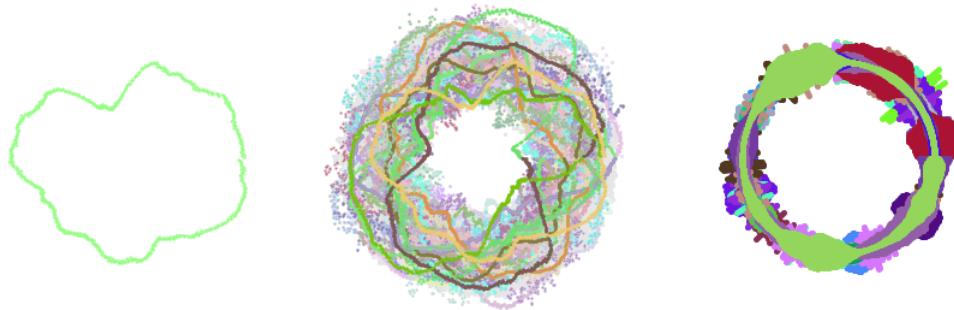


Figura 8.4: Pontos em um círculo reagindo ao som.

## 8.3 | Visualização de dados

---

O amanhecer da era da informação trouxe consigo uma série de facilidades para nossa sociedade. Um leve movimento de nossos dedos nos faz receber notícias de qualquer lugar do mundo ou compartilhar partes de nossas vidas por meios digitais. Lamentavelmente ela também acarreta em um inconveniente, vivemos em um ciclo no qual somos bombardeados e produzimos uma avalanche de informações cujo volume é tão monumental que a torna impossível de ser processada por completa. A construção da informação ocorre através de interpretações e associações de sua unidade básica, o dado, que assume a forma de números, letras, palavras, símbolos, imagens, sons e assim por diante. Entender como apresentá-lo de maneira correta para produzir uma mensagem de valor é uma missão desafiadora que requer sua própria especialidade. A visualização de dados é uma vertente das ciências da informação incumbida do tratamento, processamento e exposição efetiva de dados. Atualmente é uma das mais cobiçadas áreas em

nossa era da informação. Um dos principais motivos de sua importância é que virtualmente tudo que fazemos hoje pode ser convertido para um tipo de dado e utilizado para gerar conhecimento ou tomar uma decisão.

O elemento primário de trabalho, o dado, é, em geral, derivado de uma medição ou amostragem de um episódio, evento ou objeto, caracterizando sua natureza determinística. Felizmente isso não é um contratempo uma vez que podemos aplicar a dimensão do gerativo caracterizada pelo mapeamento entre domínios<sup>11</sup> ou, em outras palavras, pela transformação de uma grandeza em outra<sup>12</sup>. Esses elementos podem ser explorados ainda mais profundamente se organizarmos ou propositadamente desorganizarmos suas relações, originando resultados visuais elaborados. Ilustraremos brevemente nesta subseção como a escolha da preparação de um dado impacta em sua exibição<sup>13</sup> em vez de abordarmos diretamente o gerativo. No exemplo a seguir são mostradas quatro maneiras de se interpretar<sup>14</sup> os dez mil primeiros algarismos que compõem as casas decimais do número  $\pi$ . A figura 8.6a representa um histograma, em forma de arco, da densidade dos números 0 a 9 que constituem a constante. Em 8.6b, esses mesmos números correspondem ao ângulo de direção de um agente que se desloca no espaço. A visualização 8.6c exibe uma colorização de qual par numérico é mais recorrente ao selecionarmos como amostra dois números sequenciais de  $\pi$ . Por último, 8.6d, demonstra conexões em forma de curva entre os sucessivos algarismos dos decimais de  $\pi$ .

3.141592653589793238462643383279502884197169399...

Figura 8.5: Parte da constante Pi.

Veja que nenhuma dessas visualizações, com exceção da primeira, são particularmente “úteis” ou efetivas, podendo ser classificadas mais como artísticas do que científicas. Também é curioso que as imagens geradas são excepcionalmente imprevisíveis, mesmo fazendo o uso de dados 100% determinísticos, fato justificado pela nossa capacidade limitada de interpretar ou abstrair uma grande quantidade de variáveis e interações.

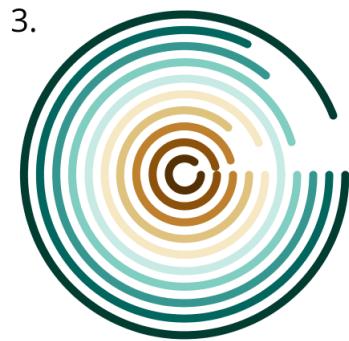
Esse fenômeno pode ser confirmado ao observarmos tanto obras de natureza artística quanto científica, como a de Martin Krzywinski que combina dados, visualizações criativas e estratégias inteligentes ao invés de se apoiar nas funções `random()` e `noise()`. Em seu projeto, *The Universe: Superclusters & Voids*, ele

<sup>11</sup>Galanter, P. (2006). *Generative art and rules-based art*. Vague Terrain (3).

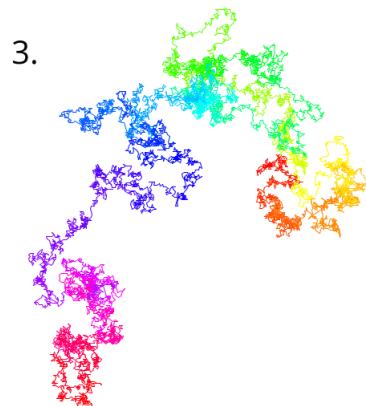
<sup>12</sup>Por exemplo, valor de temperatura para uma cor, ou dados do mercado de ações em sons.

<sup>13</sup>inclusive, esse é um dos principais fatores dos quais temos que estar atentos durante a leitura de informações provenientes de meios que se estendem desde revistas e jornais, até livros e artigos científicos

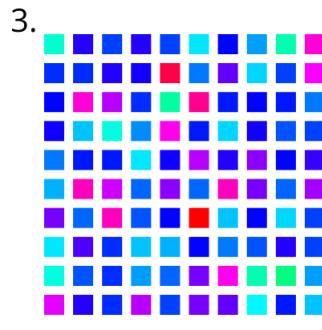
<sup>14</sup>Inspirado em trabalhos como os de Martin Krzywinski e Cristian Ilies Vasile.



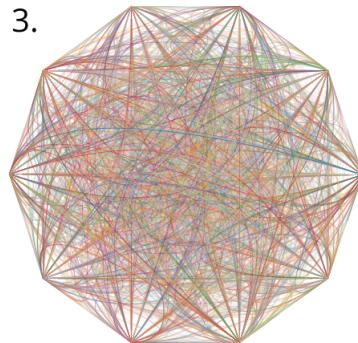
(a) Histograma em arcos.



(b) Deslocamento espacial.



(c) Densidade de sequências numéricas.



(d) Conexões entre dígitos.

Figura 8.6: Visualização de dez mil casas decimais do número  $\pi$ .

transformou uma lista dos mais conhecidos superaglomerados de galáxias de nosso universo em um mapa que contempla mais de dois bilhões de anos luz no sistema supergaláctico de coordenadas. O trabalho dele se mostra impressionante ao incluir constelações, estrelas, vazios e supervazios, explodindo em uma paisagem espetacular, figura 8.7, em que a ordem expira o caos.



Figura 8.7: *The Universe: Superclusters & Voids*<sup>d</sup>, cortesia dos próprios autores, Martin Krzywinski do Centro de Ciências do Genoma, Vancouver, Canada, e Viorica Hrincu.

<sup>d</sup>Todos os direitos reservados pelos autores originais.

Em programas cujo objetivo é a visualização de informações, tem-se como premissa a disponibilidade dos dados. Eles podem ser de natureza on-line, medidos e atualizados constantemente, ou off-line, provenientes de uma base histórica. A leitura de um arquivo se encaixa nessa segunda modalidade e usaremos um texto como a fonte de sinais de entrada. No Processing podemos consultar um arquivo com extensão

.txt (.csv, .ini, .dat) através da função `loadStrings()`<sup>15</sup>, que retorna um vetor multidimensional do tipo `String` com o seu conteúdo. O primeiro índice desse vetor armazena uma linha completa do arquivo e o segundo índice o carácter na posição dessa respectiva linha. A variável `íntegra`, neste estado, é crua, composta pela aglomeração de letras, números e símbolos, usualmente necessitando de pós-processamentos que incluem a substituição de caracteres indesejados e a quebra da linha em palavras ou letras através de outras funções. No exemplo abaixo carregamos um arquivo nomeado "Texto.txt"<sup>16</sup> e exibimos seu conteúdo<sup>17</sup> letra a letra, ciclicamente, na janela de exibição.

```
// Arquivo de texto:  
String conteudoArquivo[];  
char linhaArquivo[];  
int indiceLinha = 0, indiceLetra = 0;  
  
float x = 0, y = 25;  
  
void setup() {  
    size(800,200);  
    background(255);  
  
    // Carrega o arquivo de texto que está dentro da pasta "data":  
    conteudoArquivo = loadStrings("Texto.txt");  
  
    // Transfere a linha do arquivo da posição "indiceLinha" para um vetor  
    // contendo os caracteres dela:  
    linhaArquivo = conteudoArquivo[indiceLinha].toCharArray();  
}  
  
void draw() {  
    // Escreve a letra na tela:  
    fill(0);  
    text(linhaArquivo[indiceLetra],x,y);  
  
    // Atualiza as variáveis de desenho:  
    x += 7;  
    y += 10*(0.5 - noise(x));
```

<sup>15</sup>A função `loadStrings()` é desaconselhada na leitura de textos muito extensos, pois ele será carregado integralmente na memória do computador. Uma opção mais adequada seria a classe `BufferedReader` para ler os arquivos aos poucos e não sobrecarregar o sistema.

<sup>16</sup>Este arquivo deve ser importado para o seu programa usando o mesmo procedimento descritos nas seções anteriores.

<sup>17</sup>Excerto de vinte mil léguas submarinas de Jules Verne, domínio público, parte do [Projeto Gutenberg](#)

```

// Se chegar ao final da janela, volta para o início:
if(x >= width) {
    x = 0;
}

// Atualiza a posição da próxima letra da linha do arquivo:
indiceLetra++;

// Se chegar ao final da linha do arquivo, pula para a próxima:
if(indiceLetra >= linhaArquivo.length) {
    indiceLetra = 0;
    indiceLinha++;
    linhaArquivo = conteudoArquivo[indiceLinha].toCharArray();
}
}

```

Chapter 8 Mobiis in Mibili. THIS BRUTALLY EXECUTED capture was carried out with lightning speed. My companions and I had no time to collect ourselves. I don't know how they felt about being shoved inside this aquatic prison, but as for me, I was shivering all over. With whom were we dealing? Surely with some new breed of pirates, exploiting the sea after their own fashion. The narrow hatch had barely closed over me when I was surrounded by profound darkness. Saturated with the outside light, my eyes couldn't make out a thing. I felt my naked feet clinging to the steps of an iron ladder. Forcibly seized, Ned Land and Conseil were behind me. At the foot of the ladder, a door opened and instantly closed behind us with a loud clang. We were alone. Where? I couldn't say, could barely even imagine.

Figura 8.8: Arquivo de texto sendo exibido na janela de saída.

## 8.4 | Design computacional

O advento da simplificação de novas tecnologias permitiu cada vez mais que elas se tornassem amigáveis para os usuários, passando a ser moldadas em ferramentas de auxílio a áreas técnicas e artísticas. O design auxiliado por computador (CAD, do inglês, *Computer Aided Design*) é um domínio que continuamente vem se beneficiando de softwares de desenho, projeto, simulação e análise que aceleram a velocidade de trabalho de profissionais das mais abrangentes especialidades. O espectro coberto pela palavra CAD em si é muito amplo, e inclui o computador como aliado na execução de tarefas que englobam desde esquematizações arquitetônicas a otimizações em paletas de turbinas eólicas.

O gerativo pode ser empregado como um colaborador no CAD, aumentando exponencialmente a quantidade e variedade de protótipos e cenários. Uma indústria que se favorece imensamente com essa prática é a dos jogos digitais, em que é possível aplicar sinais caóticos e, paralelamente, formular leis que alimentam

um algoritmo capaz de estruturar mundos de maneira inteiramente automática, tornando cada experiência única. Um exemplo concreto do design gerativo são os diversos produtos oferecidos pelo estúdio de design *Nervous System*. Em sua linha de iluminação encontramos a *Hypae*, figura 8.9, uma luminária desenhada completamente por um software dotado de funções que emulam uma expansão orgânica. A peça literalmente cresce a partir de uma base ao seguir um algoritmo de venação foliar guiado por superfícies paramétricas e costurado por triangulações tridimensionais. O resultado é uma obra gerativa fascinante com contornos naturais e detalhes exclusivos entalhados em cada unidade.

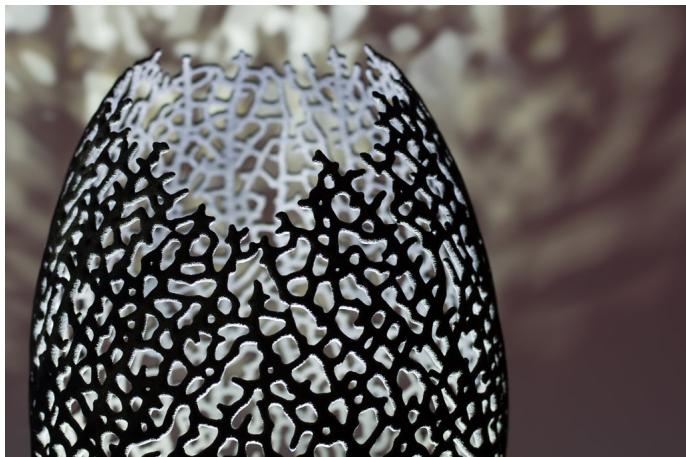


Figura 8.9: *Hyphae Lamp*<sup>e</sup>, cortesia da própria empresa que fabrica as peças, *Nervous System*.

<sup>e</sup>Todos os direitos reservados pelo artista original.

Citando especificamente softwares CAD, como aqueles desenvolvidos pela Autodesk, temos uma recente comercialização<sup>18</sup> que inclui módulos gerativos. Essa interface costuma receber um modelo modular do projetista e progressivamente o reforma por meio de técnicas que incorporam agentes autônomos, algoritmos evolucionários e redes neurais artificiais. O resultado são dezenas a milhares de alternativas que satisfazem as restrições de projeto. Uma abordagem similar que caminha lado a lado com o gerativo é a do design paramétrico, no qual se parametrizam certas propriedades do projeto através de variáveis e regras bem definidas, contidas em um universo finito de possibilidades. O computador, então, itera sobre todas as combinações existentes, formando uma população de modelos candidatos.

<sup>18</sup>Alternativas gratuitas incluem o Generative Components e o Rhino-Grasshopper.

Talvez seja natural que os próprios profissionais dos setores criativos da indústria questionem sobre práticas que se baseiam na automação, argumentando que não envolveria o recurso humano, tornando a peça resultante menos completa do que ela realmente poderia ser. No entanto, utilizar de técnicas gerativas ou paramétricas para auxiliar no design de um produto, ou qualquer que seja a obra final, não implica em obrigatoriamente escolher o item gerado pelo computador. O gerativo é um grande aliado ao fornecer inúmeros modelos microvariados que podem ser utilizados para expandir a visão do profissional. Sob este foco, o gerativo abraça o papel de inspirador em vez do substituto.

O processo do design, intrinsecamente falando, é uma tarefa com uma elevada dose de fator humano. Parte dessa necessidade vem das perguntas que devem ser respondidas antes mesmo de se começar: Qual o objetivo do design, é um produto, uma mensagem, uma obra, etc. Como será idealizado o projeto, qual o público, o meio de divulgação, etc. São incontáveis as “variáveis” que esculpem os limites da fabricação final, sendo quase impossível exemplificar um caso concreto sem dissertar extensamente sobre ele. Não obstante, é de grande valor podermos analisar como usar o código para auxiliar na visualização de possibilidades de um design e, para tal propósito, iremos usar um caso hipotético e extremamente simplificado. Considere uma empresa que deseje firmar sua identidade visual através de um símbolo<sup>19</sup> formado por três triângulos inscritos dentro de um círculo<sup>20</sup>. A cliente não sabe *exatamente* como esses triângulos devem estar dispostos, o que leva a infinitas combinações nas posição dos vértices e, consequentemente, no número de possibilidades. Aplicar uma tática puramente paramétrica, que itera sobre todos os casos, não é vantajoso pelo motivo citado acima. Iremos tratar o problema usando um plano híbrido, em que os vértices dos triângulos serão parametrizados, mas assumirão valores aleatórios dentro do conjunto válido que é a borda do círculo. O código abaixo desenha o símbolo contemplando as regras definidas pela empresa e o seu resultado é ilustrado na figura 8.10:

```
void setup() {  
    size(300,300);  
    background(255);  
    desenhaSimbolo(100);  
}  
  
// Função para desenhar o símbolo.  
// Regra: Três triângulos inscritos em um círculo:  
void desenhaSimbolo(float _raio) {  
    noFill();
```

<sup>19</sup>Representação gráfica de uma marca de uma empresa.

<sup>20</sup>O motivo do porquê dessa forma específica pode ser justificado por respostas de uma série de perguntas conforme mostrado anteriormente, mas o escopo deste livro é programação e não teoria de design e, portanto, isto não será discutido aqui.

```

ellipse(width/2,height/2,2*_raio,2*_raio);

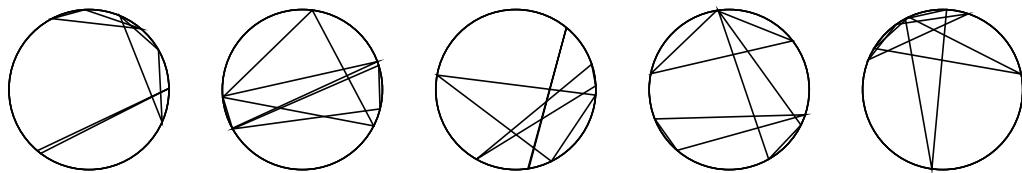
// Como são três triângulos, são necessárias três repetições:
for(int j = 0; j < 3; j++) {

    // Armazena os ângulos dos pontos dos triângulos:
    float[] a = new float[3];

    // Cada triângulo é composto por três pontos, posicionados na
    // superfície de um círculo com ângulos aleatórios (0 a 360 graus):
    for(int i = 0; i < 3; i++) {
        a[i] = radians(random(360));
    }

    // Desenha o triângulo:
    triangle(width/2 + _raio*cos(a[0]), // Ponto 1, coord. x
              height/2 + _raio*sin(a[0]), // Ponto 1, coord. y
              width/2 + _raio*cos(a[1]), // Ponto 2, coord. x
              height/2 + _raio*sin(a[1]), // Ponto 2, coord. y
              width/2 + _raio*cos(a[2]), // Ponto 3, coord. x
              height/2 + _raio*sin(a[2]) // Ponto 3, coord. y
            );
    }
}

```



**Figura 8.10:** Três triângulos inscritos em um círculo.

Veja que ele foi escrito prontamente a partir de conceitos explicados nas seções passadas, nenhum particularmente complicado, resultando em protótipos brutos, mas cujos objetivos foram estritamente cumpridos. Isso significa que a forma pode ser posteriormente trabalhada e aperfeiçoada até o produto final. Claro que, mesmo do ponto de vista do design paramétrico, este exemplo foi reduzido ao máximo: uma única regra. Veja a figura 8.11 em que se empilharam múltiplas leis distintas, ampliando a utilidade deste tipo de estratégia.

A versão gerativa deste mesmo estudo de caso é formulada através da concepção de um agente autônomo

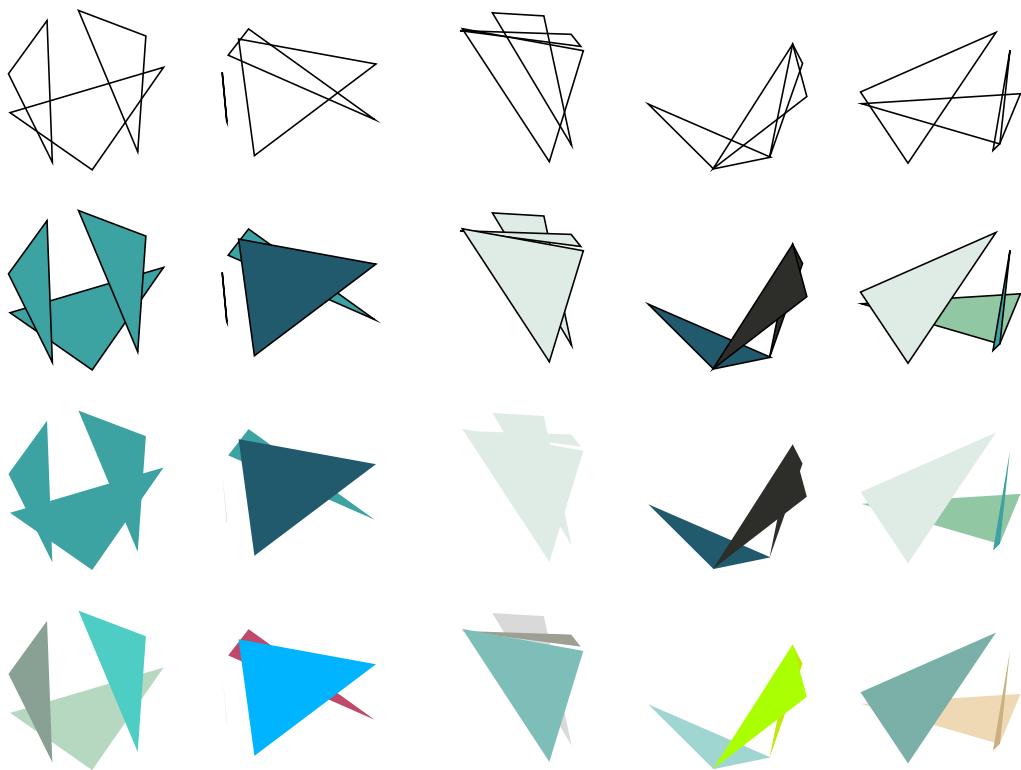


Figura 8.11: Etapas do refinamento das regras paramétricas.

análogo aquele do capítulo 7, com a diferença de que a região limite em que ele deve se manter dentro será um círculo em vez de um retângulo. Com o auxílio de formatações condicionais conseguimos resultados, figura 8.12, muito parecidos com os do paramétrico, apesar de serem técnicas fundamentalmente distintas. A figura 8.13 mostra uma extração da segunda forma de 8.12 em que foram realizadas mudanças de cores, espelhamentos e rotações, todas respeitando as condições de projeto, para criar o

logotipo da empresa *Drakein*<sup>21</sup>. Este caso fortifica o papel do gerativo como um avalista na idealização de suas próprias produções.



Figura 8.12: Família de símbolos gerados através de agentes autônomos.



Figura 8.13: Logotipo da empresa *Drakein*.

Para finalizar esta seção é relevante evidenciar o papel do código como aquele que de fato talvez seja esperado dele, o de ferramenta técnica e objetiva. Assim como qualquer outra linguagem de programação, o Processing pode ser utilizado para suprir necessidades ou metas específicas. Por exemplo, no código 8.5, resultado na figura 8.14a, foi desenvolvido um contador em que, cada vez que o usuário clicar no centro da janela do programa, o número de contagens incrementa em um. Não há nada de artístico ou criativo nisso,

<sup>21</sup>O nome e o símbolo representados na figura 8.13 são completamente fictícios e foram inventados pelo autor com o objetivo único de proporcionar um exemplo para este livro. O autor se dedicou a pesquisar o nome e, até a data de publicação do texto, não encontrou nenhuma referência comercial a ele.

ele existe apenas para efetuar uma tarefa: contar. Em 8.14b, usamos a linguagem para desenvolver um software bem mais complexo do ponto de vista técnico<sup>22</sup>. Ele constrói um modelo tridimensional de um objeto ou equipamento cilindricamente simétrico e, a partir de interpolações de leituras de temperaturas de instrumentos<sup>23</sup> em campo, traça um perfil térmico de fácil visualização. Tal solução poderia ser usada em aplicações que incluem desde altos fornos da indústria siderúrgica até em silos de grãos da alimentícia. Em ambos os casos o Processing foi configurado como uma aplicação de engenharia, com métricas de expectativas, conformação e resultados. Porém, note um ponto interessante, projetar um *framework* para trabalhos desse tipo se beneficia de um experiência prévia com o código criativo visto que existe a manipulação da exposição da informação (cores, formas e demais elementos que fazem parte da formatação e visualização efetiva dos dados) e posicionamentos espaciais (auxiliado pela distribuição de figuras geométricas). A emergência do *gestalt*, de uma visão holística da diversidade e associações entre áreas do conhecimento, é fundamental para aprimorar suas obras independente de seu propósito: arte, ciência, ou talvez como os sábios renascentistas faziam, uma mistura dos dois.

#### Código 8.5 Programa contador

```
int contador = 0;

void setup() {
    size(310, 640);
    textAlign(CENTER);
}

void draw() {
    background(204);

    // Desenha os botões de interface:
    fill(255);
    rect(10, 160, 290, 150);
    rect(10, 320, 290, 150);
    rect(10, 480, 290, 150);

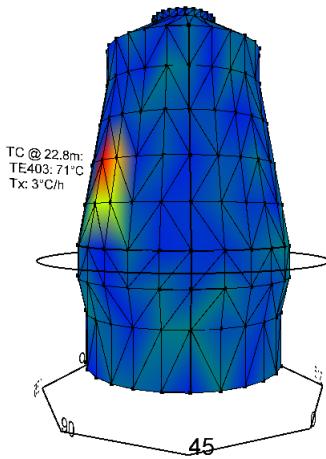
    // Escreve os textos do aplicativo:
    fill(0);
    textSize(40);
    text("Contador: ", 155, 64);
```

<sup>22</sup>O software desenvolvido contempla funções como triangulamentos, conexões com banco de dados e ajustes de câmera para visão tridimensional.

<sup>23</sup>Termopares, termistores e termorresistências são alguns deles.

```
    text(contador, 155, 125);
    textSize(100);
    text("+",155,260);
    text("-",155,425);
    text("0",155,590);
}

// Trata os cliques sobre os botões (eixo x desconsiderado):
void mousePressed() {
    if(mouseY > 160 && mouseY < 310) {
        contador++;
    }
    else {
        if(mouseY > 320 && mouseY < 470) {
            contador--;
        }
        else {
            if(mouseY > 480 && mouseY < 630) {
                contador = 0;
            }
        }
    }
}
```



(a) Contador manual sendo executado em um dispositivo Android. (b) Distribuição térmica em um modelo de alto-forno.

Figura 8.14: Processing utilizado como software de engenharia.

## 8.5 | Sumário

Este capítulo foi dedicado à explorar uma visão pragmática da programação, na qual os tópicos previamente estudados foram reinterpretados e moldados em composições não ortodoxas. As subseções ilustram áreas em que a programação criativa encontra seu espaço, protagonizando soluções que impressionam com a versatilidade de suas aplicações. É um capítulo desafiador de ser exemplificado sem entrar em um labirinto de explicações e considerações subjetivas. Cada ideia apresentada facilmente evolui para iniciativas que envolvem múltiplas áreas de conhecimento e sofrem sucessivas alterações até suas formas, questionavelmente, finalizadas; como atribuído a Leonardo da Vinci, “A arte nunca é terminada, apenas abandonada”.



# O Processo Final

Caro leitor, após todo caminho de aprendizado que você trilhou neste livro, você finalmente se encontra em seu destino. Fico honrado por sua companhia durante este tempo, mas agora devo me afastar e voltar para o início, onde guiarei outra alma que busca a beleza, estética ou sistêmica, da arte computacional. Quanto a você, saiba que o mais importante não é ter chegado até aqui, nestas últimas páginas, e sim toda a sua jornada de experimentação artística e programação técnica. Espero que você tenha apreciado como a dualidade do natural com sintético é capaz de produzir frutos que são no mínimo intrigantes.

Por fim, gostaria de fazer uma reflexão e convidá-lo a meditar sobre um segredo que está escondido nas entrelinhas deste texto. O Processing permite que você use da programação para explorar seu lado matemático e artístico, criar imagens magníficas e programas que reagem a outras pessoas, seres ou ao ambiente. Ele facilita a manipulação e a visualização efetiva, ou simplesmente criativa, de dados, músicas, imagens e vídeos. Pode ser empregado em softwares usados diariamente por milhares de pessoas, ou apenas por você, em uma sessão casual de arte computacional. O Processing viabiliza uma infinidade de aplicações com os mais variados propósitos, limitadas apenas pelos platôs de sua imaginação. Entretanto, lembre-se da máxima: ele não passa de uma ferramenta. O verdadeiro valor de tudo que você criar está em você mesmo, o seu código *transcende* a escrita, as regras, a linguagem e os resultados. Apenas você é capaz de abstrair conceitos de um meio ou solidificar ideias das incertezas. O seu ser é a personificação de sua própria computação, uma amalgama de instinto, inteligência, paixão, razão, emoção, ordem e caos. Em última instância, só existe um *processo* cuja natureza fundamentalmente emergente e complexa supera todos os outros: você.

Obrigado pela leitura.



# Índice

## A

**AdobeFlash**, 15

**Agente, classe**, 196

atualizar(), método, 197

construtor, 197

exibição, 198

exibirComoPonto(), método, 198

exibirComoReta(), método, 198

**agentes**, 193

comportamentos, 195

forma, 195

na programação, 196

percepções, 195

**Alan Watts**, 217

**Albiac, Sergio**, 219

**aleatoriedade**, 96

visualizada através de

cores, 99

curva, 98

espiral, 137

retas, 160

texto, 98

**animação**, 59

**arte gerativa**, xix

origens, 5

parametrização, 95

vs. arte computacional, 6

**autossimilaridade**, 172

## C

**círculo**

arco de, 117

desenho paramétrico do, 118

equações paramétricas do, 117

**Caminho do Zen**, 217

**caos**, 95

**Cinder**, 14

**class**

palavra-chave, 82

**classe**, 81

componentes de uma, 82

estrutura básica, 81

vs. objeto, 81

**color, variável**, 45

argumentos, 46

transparência, 48

**comentários, código**, 23

**computação**, xvii

na arte, xviii

**condicionais**

aninhamento de, 62

estrutura básica, 60

introdução a, 60

**contador, programa**, 234

**Cordeiro, Waldemar**, 6

**cores**, 45

canais de, 45

combinação de, 46

## D

**dados**, 224

fontes de, 225

on-line e off-line, 227

**design auxiliado por computador**, 229

**design gerativo**, 230

**design paramétrico**, 230

**determinismo**, 95

**dist()**, função, 59

**distribuição uniforme, 97**  
**draw(), função, 59**  
**draw(), função, 34**  
**Duchamp, Marcel, 7**

## E

---

**Edmonds, Ernest, 7**  
**ellipse(), função, 116**  
**emergência, 193**  
**erros na programação artística, 123**  
**espiral, 134**  
**esquema de cores, 146**  
**estrela, 127**  
    giro de uma, 129  
**estrutura genérica, 55**  
**estruturas de dados**  
    boolean, 40  
    char, 40  
    color, *veja* color, variável  
    float, 40  
    int, 40  
    string, 40  
**exposição de informação, 235**

## F

---

**fill(), função, 141**  
**floco de neve de Koch, 172**  
    regras, 175  
**for, *veja* repetições, estrutura compacta**  
**forma, 95**  
**formatação, 141**  
    estudo de caso, 160  
    gradiente de cores, 160  
**fractais, 171**  
    abstratos, 174  
    aleatórios, 174  
    definição, 172  
    geométricos, 173  
    regras, 175  
**fractal árvore, 176**

**aleatório, 189**  
**assimetria, 185**  
**condição de parada, 181**  
**espessura, 187**  
**folhas, 186**  
**projeto da função, 179**  
**regras, 179**

**frameCount, variável, 121**

**Fry, Benjamin, 8**

**funções, 24**

    de mesmo nome, 57  
    elementos de, 55  
    exemplo de projeto de, 56  
    no Processing, 24

**funções, introdução a, 55**

## G

---

**Galanter, Philip, xix**

**geometria fractal, 171**

    definição, 171

**gerativo, xix**

    na natureza, xx

**gestalt, 235**

**get(), função, 220**

## I

---

**identidade visual, estudo de caso, 231**

    gerativo, 232  
    logotipo, 234  
    paramétrico, 231  
    símbolo, 231

**if/else, *veja* condicionais, estrutura básica**

**imagem digital, 45**

**import, palavra-chave, 223**

**informação, 224**

    construção da, 224

**interrupções**

    introdução a, 58  
    no Processing, 58

## J

janela de saída, 23

limites da, 200

## K

key, variável, 62

Krzynski, Martin, 225

Kwok, Raven, 221

## L

LIA, 8

loadStrings(), função, 228

loop, *veja* repetições

## M

máquina de estado, 142

música gerativa

*Erratum Musica*, 7

*Musikalisch Würfelspiel*, 7

Mandelbrot, Benoit, 171

mapeamento entre domínios, 225

matemática, 115

McCormack, Jon, 8

Minim, biblioteca, 223

modelo tridimensional, 235

mouse

arraste do, 58

posição do, 59

mouse, posição do, 53, 61

movimento de elementos, 197

## N

Nees, Georg, 5

Nervous System, 230

luminária Hyphae, 230

Nodebox, 14

noFill(), função, 141

noise(), função, 100

explicação gráfica, 101

passo, 102

noStroke(), função, 142

## O

Olá Mundo!, 23

OpenFrameworks, 14

operadores, 49

aritméticos, 49

de atribuição, 50

de relação, 51

lógicos, 51

precedência de, 49

ordem e caos, 137

## P

padrões de Truchet, 70

padrões matemáticos

em objetos, 7

na arquitetura, 7

paleta de cores, 146

no Processing, 146

Perlin, Ken, 100

pi, número, 225

interpretações visuais, 225

pixels, 30

plano bidimensional, computação, 30

pontilhismo, 219

println(), função, 41

probabilidade, 109

fórmula, 109

na arte computacional, 109

no Processing, 109

Process, 193

Processing, 15

ambiente, 18

aplicações técnicas, 234

arquivos de áudio no, 20

arquivos de imagem no, 20

arquivos de texto no, 20

bibliotecas no, 19

programa básico, 34  
referência, 24  
vs. outras linguagens, 16

**processo, 193, 203**

**processo do design, 231**

**processo, estudo de caso, 203**  
conexão, 203  
otimização de, 205  
dualidade de cores, 206  
posição inicial, 208  
transparência, 207

**programação orientada a objetos, 81**

classe, *veja classe*  
objeto, 81

**R**

**radians(), função, 118**

**random(), função, 97**

**reação ao áudio, 221**

no Processing, 223

**Reas, Casey, 8**

**rect(), função, 24, 31**

**recursividade, 174**

condição de parada, 174

função, estrutura básica, 175

**recursividade infinita, 181, 244**

**reflexão, física, 201**

**repetições**

aninhamento de, 72

condição de parada, 67

estrutura básica, 66

estrutura compacta, 69

introdução a, 66

**resolução, 30, 34**

**reta centrada em círculo, 157**

**return, palavra-chave, 56**

**Ross, Bob, 123**

**ruído Perlin, 100**

visualizado através de

cores, 104

curva, 101  
espiral, 137  
retas, 157

**S**

**seletor de cores, 46**

**setup(), função, 34**

**simetria**

de escala, 152  
radial, 127, 152  
reflexiva, 152

**sintaxe por ponto, 83**

**stroke(), função, 142**

**strokeWeight(), função, 142**

**StructureSynth, 14**

**switch, *veja* condicionais, estrutura alternativa**

**T**

**teclado**

pressionar do, 62

**tratamento de imagens, 218**

no Processing, 219

**Tresset, Patrick, 8**

**trigonometria, 116**

**Truchet, Sébastien, 75**

**V**

**variáveis**

declaração de, 39

do sistema, 41

introdução a, 38

nomeando, 39

**variável**

inicialização, 39

**vetores de variáveis**

índices em, 44

atribuição por elemento, 43

declaração de, 42

introdução a, 42

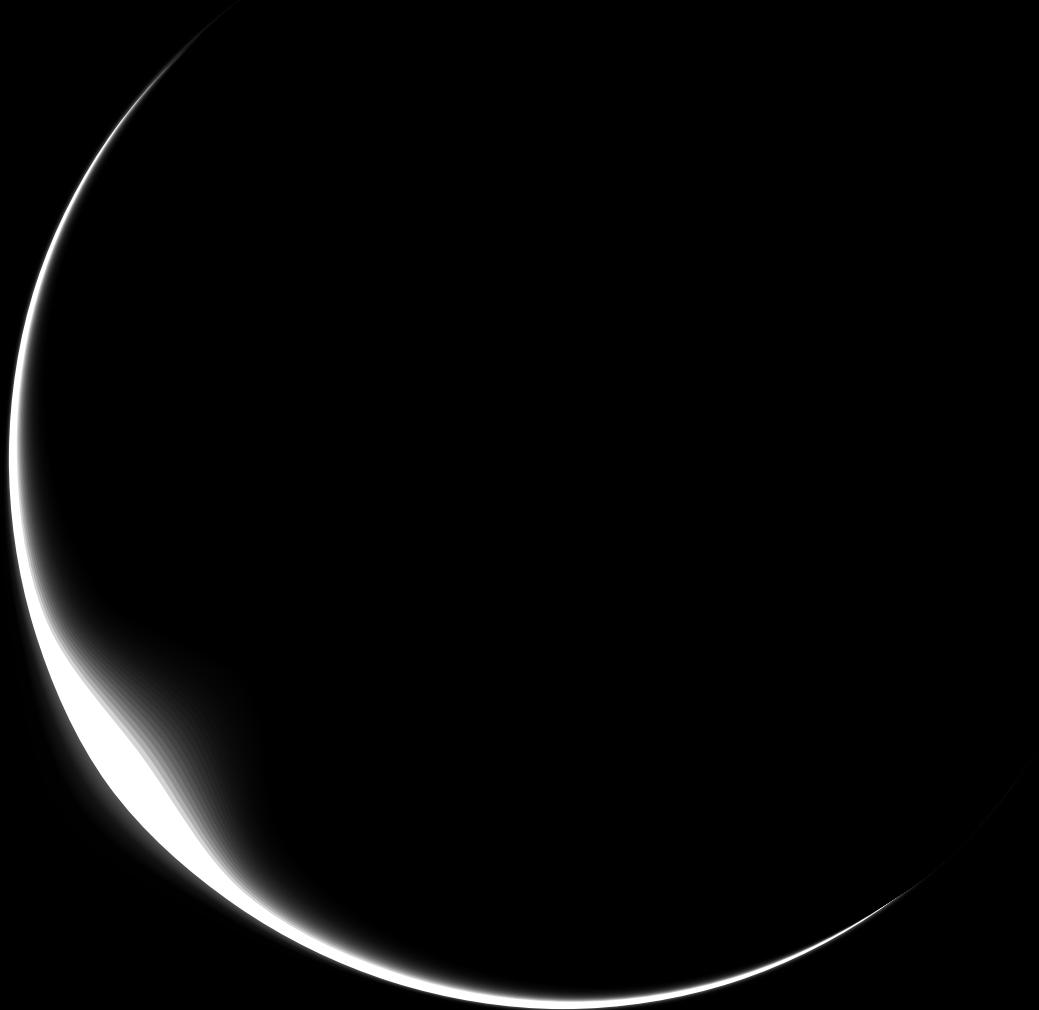
**visualização de dados, 224**  
**void, palavra-chave, 56**  
**VVVV, 14**

## W

---

**while, *veja* repetições, estrutura básica**





ISBN: 978-65-902096-0-3