# INTRODUÇÃO À ARTE GENERATIVA E PROGRAMAÇÃO CRIATIVA

## CONCEITOS BÁSICOS

PEDRO ALVES DA VEIGA

2024

# INTRODUÇÃO À ARTE GENERATIVA E PROGRAMAÇÃO CRIATIVA

CONCEITOS BÁSICOS

PEDRO ALVES DA VEIGA
pedro.veiga@uab.pt

DOUTORAMENTO EM MÉDIA-ARTE DIGITAL
UNIVERSIDADE ABERTA

2024

Esta licença permite que os reutilizadores distribuam, remixem, adaptem e construam sobre a presente obra, em qualquer meio ou formato apenas para fins não comerciais, e apenas enquanto a atribuição for dada ao respetivo criador/autor. Se o reutilizador remisturar, adaptar ou desenvolver sobre a presente obra, ele deve licenciar o material assim modificado sob termos idênticos.
CC BY-NC-SA inclui os seguintes elementos:

BY: a atribuição deve ser dada ao criador.

NC: apenas são permitidas utilizações não comerciais da obra.

SA: as adaptações devem ser partilhadas nas mesmas condições.

AGRADECIMENTO

Ao compilar estes materiais de apoio, destinados - numa primeira instância - aos estudantes do Doutoramento em Média-Arte Digital, não posso deixar de manifestar a minha eterna gratidão à Professora Doutora Elizabeth Simão Carvalho, por me ter revelado os horizontes e potencial desta área do conhecimento.

Pedro Alves da Veiga
Outubro de 2024

# ÍNDICE

ÍNDICE ..... 3
ARTE GENERATIVA ..... 5
PANORAMA HISTÓRICO E DEFINIÇÃO ..... 6
  DEFINIÇÃO ..... 9
COMPLEXIDADE ..... 10
  REPETIÇÃO DE FORMAS E TESSELAÇÃO ..... 10
ARTE GENERATIVA NÃO COMPUTACIONAL ..... 12
  OP-ART ..... 12
  LITERATURA ..... 12
  MÚSICA ..... 13
SISTEMAS DE LINDENMAYER ..... 17
  GRÁFICOS DE TARTARUGA ..... 18
  CURVAS DE KOCH ..... 18
SISTEMAS-L ESTOCÁSTICOS ..... 21
  APLICAÇÕES PRÁTICAS ..... 22
ARTE GENERATIVA COMPUTACIONAL ..... 25
  LITERATURA ELETRÓNICA ..... 25
  AUDIOVISUAL ..... 26
CRIANDO ARTE GENERATIVA COMPUTACIONAL ..... 27
ETAPAS DE DESENHO DA OBRA ..... 29
  1 – IDENTIFICAÇÃO DO VOCABULÁRIO ..... 29
  2 – DESENHO DE UM DISPOSITIVO ESTRUTURANTE ..... 29
  3 — AMPLIFICAÇÃO OU MAPEAMENTO ..... 29
  4 — DETEÇÃO E CRIAÇÃO DE EVENTOS ..... 30
TRÊS ABORDAGENS DISTINTAS ÀS OBRAS GENERATIVAS ..... 30
  TEMPO DE EXECUÇÃO ..... 30
  INTERATIVIDADE ..... 30
  CODIFICAÇÃO AO VIVO ..... 30
PROCESSING E P5.JS CÓDIGO CRIATIVO ..... 32
INTRODUÇÃO ..... 33
  O QUE É PROGRAMAR ..... 33
PROCESSING E P5.JS ..... 34
  INSTALAÇÃO E AMBIENTE ..... 34
CÓDIGO ..... 36
  FUNÇÕES ..... 36
  COMANDOS E EXPRESSÕES ..... 37
  VARIÁVEIS ..... 38
  BLOCOS DE CÓDIGO E CHAVETAS ..... 39
  SINTAXE E ERROS ..... 42
  EXECUÇÃO E ERROS ..... 42
DESENHO ..... 44
  LINHA ..... 44
  RETÂNGULO ..... 46
  ELIPSE, CÍRCULO E PONTO ..... 47
TRANSLAÇÃO, ROTAÇÃO E ESCALA ..... 49
  ESTADOS DO SISTEMA DE COORDENADAS ..... 52
COR ..... 55
CICLOS ..... 57
INTERAÇÃO ..... 61
  INTERAÇÃO BÁSICA ..... 61
ANIMAÇÃO ..... 63
TRIGONOMETRIA ..... 72
  UTILIZAÇÕES DE SIN() (SENO) E COS() (COSSENO) ..... 72
CURVAS DE BEZIER ..... 76
SISTEMAS-L EM PROCESSING ..... 79
  VARIAÇÕES ESTOCÁSTICAS ..... 80
RASTER GRAPHICS OU BITMAPS ..... 83
  ALTERANDO AS CORES ..... 84
TEXTO E TIPOS DE LETRA ..... 86
ARRAYS ..... 89
  ARRAYS MULTIDIMENSIONAIS ..... 93
  VISUALIZAÇÃO DE UM ARRAY ..... 95

PRIMEIRA PARTE

# ARTE GENERATIVA

## PANORAMA HISTÓRICO E DEFINIÇÃO

A arte generativa é um subgrupo da família mais alargada da "arte algorítmica", que compreende toda a arte que pode ser gerada de acordo com um conjunto de regras, ou algoritmo.

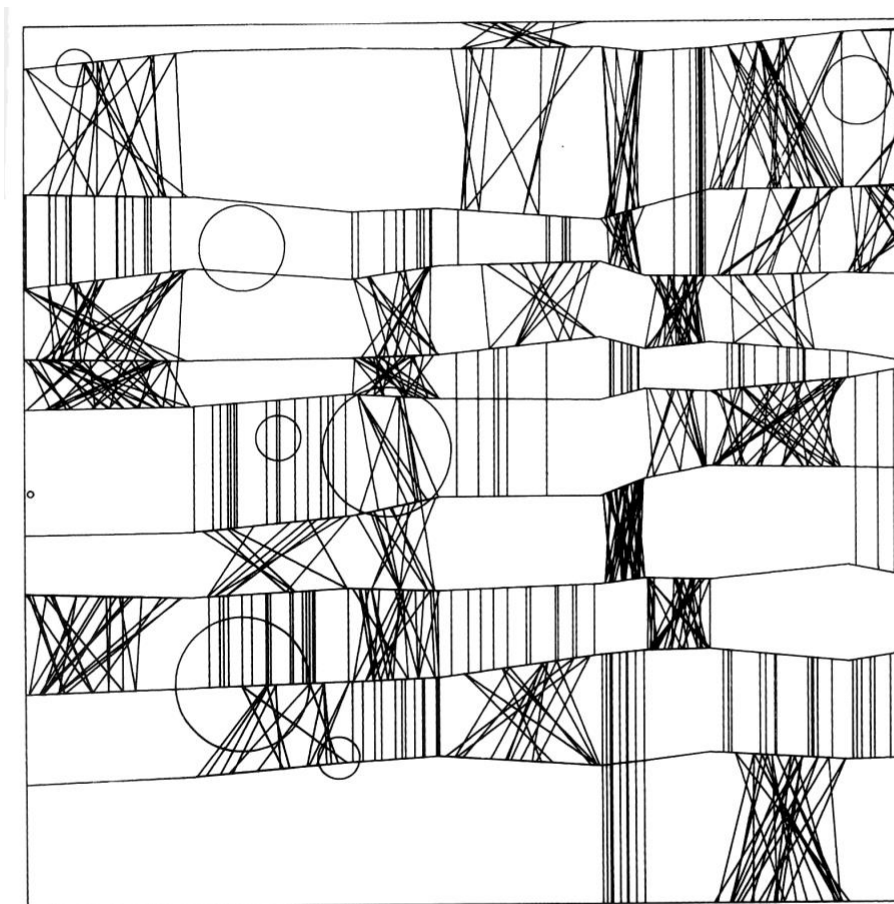


Figura 1 - A obra generativa *Hommage a Paul Klee*, de Frieder Nake, 1965

O que distingue a arte generativa é o facto de existirem várias iterações (repetições) de um determinado processo, mas cada nova iteração é baseada ou influenciada pelo resultado anterior – a "geração" anterior. À medida que as gerações progridem, elas podem permanecer idênticas – como com a azulejaria, por exemplo – ou podem produzir resultados diferentes, afastando-se da geração inicial.

A arte generativa geralmente é produzida através computadores, uma vez que o cálculo de milhares de gerações pode ser um processo longo e monótono, mas não está restrita ao meio digital.

Em termos históricos, pode dizer-se que Max Bense foi o primeiro teórico da área, já que na década de 1950, Bense começou a escrever sobre estética com o objetivo de calcular e quantificar este tema aparentemente intangível (ou, pelo menos, inefável). A sua conceção de arte e estética baseava-se em vários precedentes, nomeadamente a noção de medição estética de Birkhoff, de 1933, que definia matematicamente a estética como uma relação entre ordem e complexidade. Influente também foi a teoria da informação de Claude Shannon, que definia estatisticamente a informação em função da quantidade de ruído que um sinal encontra ao atravessar um canal, bem como a cibernética de Norbert Wiener, que estudava a comunicação em sistemas complexos.

Sintetizando estas fontes, Bense imaginava a arte como uma rede de comunicação em que a informação estética fluía entre o artista, a obra e o público. Ele definia a informação estética como o elemento inesperado dentro deste campo: sinais estéticos transmitem informação na medida em que são imprevisíveis. A sua fórmula expandia a de Birkhoff, calculando a relação entre complexidade e redundância – como, por exemplo, entre ritmos regulares e irregulares, ou entre formas e espaçamentos uniformes e irregulares. A fórmula de Bense

também levava em conta a relação entre o número de variações possíveis dentro de um determinado meio ou máquina (tons na música, espectro de cores, etc.) e aquelas efetivamente realizadas na obra. Para Bense, uma obra de arte bem-sucedida expande os limites da comunicação e da experiência, sem ultrapassar a capacidade do observador de processar a obra. Este equilíbrio entre criatividade e convenção, segundo Bense, é a razão de ser da arte. A sua perspetiva ressoa com as definições de vanguarda como rutura com a tradição, mas, neste caso, essa rutura pode ser automaticamente produzida, medida estatisticamente e definida matematicamente.

Revisitando o conceito de Bense, este aplica 3 níveis de computação a esta estética generativa:

a) métrica, entendida como a aplicação de princípios matemáticos na estética;

b) topológica, para aplicação a sistemas complexos, jogos de ilusão ótica, paradoxos e visões impossíveis;

c) estatística, enquanto forma de afetação de regras com algum grau de imprevisibilidade e variação.

A culminação da teoria de Bense encontra-se num ensaio breve, publicado em 1965, intitulado "Os Projetos da Estética Generativa". Nesse texto, Bense propôs um conjunto de esquemas destinados a avaliar estados estéticos, juntamente com um conjunto de fórmulas destinadas a estimular tais estados através de "uma combinação metódica de planeamento e acaso". Esta abordagem consistia na junção de elementos irregulares ou randomizados, com outros organizados de forma ordenada.

Assim, a produção artística ocupava um lugar de primazia em relação à receção da obra, destacando-se, assim, mais como uma teoria da arte generativa do que propriamente como uma teoria estética tradicional. Este ensaio foi elaborado com o intuito de acompanhar uma exposição de grafismos gerados por computador de Georg Nees, realizada na galeria da Universidade de Estugarda, instituição que o próprio Bense fundara em 1958. Georg Nees, engenheiro de formação, começara, em 1964, a explorar o que designava por "grafismos estatísticos". Estas obras pioneiras no campo da arte generativa, em estreita sintonia com as teorias de Bense, promoviam uma integração inovadora entre a ordem programática e a imprevisibilidade do acaso no processo criativo.

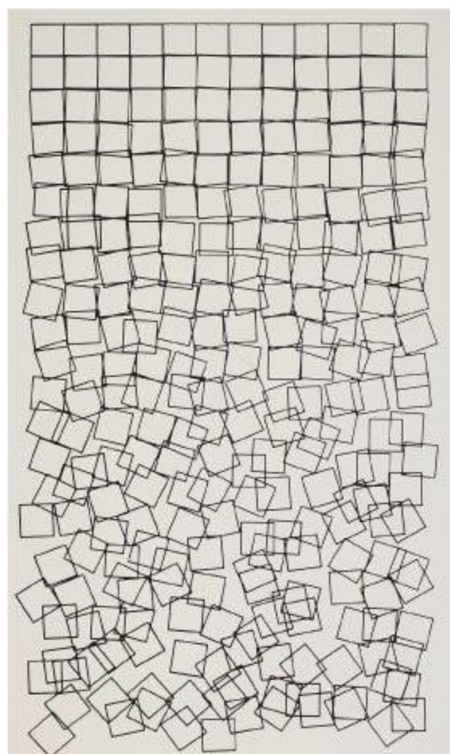


Figura 2 - Schotter, de Georg Nees, 1965

E é assim que a designação "Arte Generativa" aparece em 1965, com a exibição de Georg Nees em Estugarda - *Generative Computergraphik* –, e novamente, no mesmo ano, em conjunto com Frieder Nake. Quatro anos depois, Nees apresenta a primeira tese de doutoramento sobre o tema, e com o mesmo título.

Na obra *Schotter*, de Georg Nees, podemos reconhecer claramente a primeira geração como a primeira linha horizontal de quadrados (no topo), e cada geração seguinte progredindo para baixo, em que os quadrados se apresentam cada vez mais desordenados, rodados, em suma, caóticos.

CELESTINO SODDU

**Generative City Design**

**Aleatority and Urban Species**

Unique, Unrepeatable and Recognizable Identity, like in Nature

DOMUS ARGENIA
ISBN 9788896610411

Figura 3 - Generative City Design, de Celestino Soddu

Em 1989 Celestino Soddu definiu a abordagem de Design Generativo para Arquitetura e Design de Cidades no seu livro *Città Aleatorie* e no mesmo ano Herbert Franke usou o termo 'matemática generativa' como *o estudo de operações matemáticas adequadas para gerar imagens artísticas*.

Podemos, assim, considerar que a arte generativa é um subgrupo da família mais alargada da arte algorítmica, que compreende toda a arte que pode ser gerada de acordo com um conjunto de regras, usando meios computacionais ou não.

Frequentemente confunde-se com a própria arte algorítmica, produzida através de sistemas informáticos, mas que pode também ser obtida através de sistemas químicos, biológicos, mecânicos, matemáticos, por simetria, repetição, etc.

O que distingue a arte generativa é o facto de existirem várias iterações (repetições) de um determinado processo, mas cada nova iteração é baseada ou influenciada pelo resultado anterior – a geração anterior. À medida que as gerações progridem, elas podem permanecer idênticas – como com a azulejaria, por exemplo – ou podem produzir resultados diferentes, afastando-se da geração inicial.

A arte generativa é atualmente produzida maioritariamente através de computadores, uma vez que o cálculo de milhares de gerações pode ser um processo longo e monótono, mas não está restrita ao meio digital.

Deve salientar-se que, dado que a arte generativa não é um estilo ou género artístico, mas sim um processo de obter experiências estéticas pode, por isso, dar origem a artefactos visuais abstratos, figurativos, conceptuais, OpArt, sonoros ou performativos.

Pode parecer estranho que a arte generativa possa ser usada na literatura, mas a verdade é que não precisamos de estar limitados a conceitos visuais. Se em vez de 'azulejos' pensarmos em 'frases', já estamos a meio caminho,

e o mesmo pode ser entendido em qualquer outra forma de expressão criativa, tendo o cuidado de substituir os símbolos e regras pelos do universo artístico que se deseja endereçar.

Apesar de muitas definições irem ao encontro desta visão, Philip Galanter, resume-as de forma eficaz: “Arte generativa refere-se a qualquer prática artística em que o artista usa um sistema, como um conjunto de regras de linguagem natural, um programa de computador, uma máquina ou outra invenção processual, que é acionada com algum grau de autonomia contribuindo ou resultando numa obra de arte completa”.

## DEFINIÇÃO

Podemos então propor a seguinte definição:

> A arte generativa é toda arte que, no todo ou em parte, é criada com o uso de um sistema autónomo, ou seja, um sistema não humano, que pode determinar de forma independente características evolutivas de uma obra de arte que de outra forma exigiriam decisões tomadas diretamente pelo artista. Normalmente o artista assume o papel de designer de estrutura – ou designer de regras – e o sistema evolui livremente dentro dessa estrutura, obedecendo a essas regras.

### FONTES BIBLIOGRÁFICAS

Bense, M. (1965). Projekte generativer ästhetik. *F. von Cube (Flg.), Was ist Kybernetik1 Grundbegriffe, Methoden, Anwendungen, dtv WR, 4079*.

Birkhoff, G. D. (1933). Aesthetic measure. Harvard University Press.

Chomsky, N. (1978). *Topics in the Theory of Generative Grammar*. Walter de Gruyter.

Galanter, P. (2001). Foundations of Generative Art Systems? a Hybrid Survey and Studio Class for Graduate Students. In *Generative Art 2001: Proceedings of the 4th International Conference. Generative Design Lab, Milan Polytechnic, Milan, 2001*.

Nake, F. (2018). The Pioneer of Generative Art: Georg Nees. *Leonardo* , *51* (03), 277-279.

Soddu, C. (1989). *Città aleatorie* (Vol. 1, pp. 1-120). Masson.

## REPETIÇÃO DE FORMAS E TESSELAÇÃO

A repetição de formas é o tipo mais simples de arte generativa nas artes visuais, mas, embora seja simples (porque todas as gerações são semelhantes), pode dar origem a alguns projetos interessantes. Tomando a azulejaria como exemplo, quando todas as peças são geometricamente idênticas, a expressão "azulejos monoédricos" pode ser usada. Contudo a azulejaria não se limita a utilizar uma forma única, e também pode ser diédrica – usando dois azulejos diferentes – como no exemplo abaixo (quadrado e triângulo) ou poliédrica.

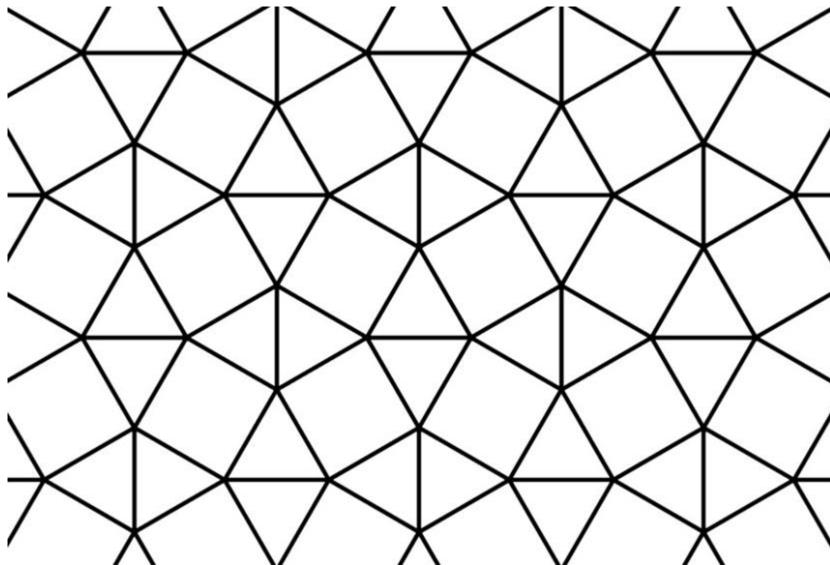


Figura 4 - Tesselação diédrica (Wikimedia Commons)

### ATIVIDADE

> Consegue determinar um possível conjunto de regras (um algoritmo) para gerar a imagem anterior? Sugestão: comece com um quadrado e decida o que acontece com cada lado.

Um padrão mais complexo, é claramente possível, e então entramos no reino da tesselação poliédrica, com alguns bons exemplos datando de há várias centenas de anos, como a seguinte imagem documenta.

Figura 5 - Tesselação triédrica Tawriq em Alhambra, Espanha (Wikimedia Commons)

Outra coisa de que nos apercebemos é que muito rapidamente perdemos o interesse pela progressão do sistema, porque já conhecemos a estrutura: não há surpresas! Apesar dos formatos diferentes, não estamos longe da tradicional azulejaria portuguesa: o resultado final está perfeitamente determinado. Tais sistemas generativos são designados **ordenados**.

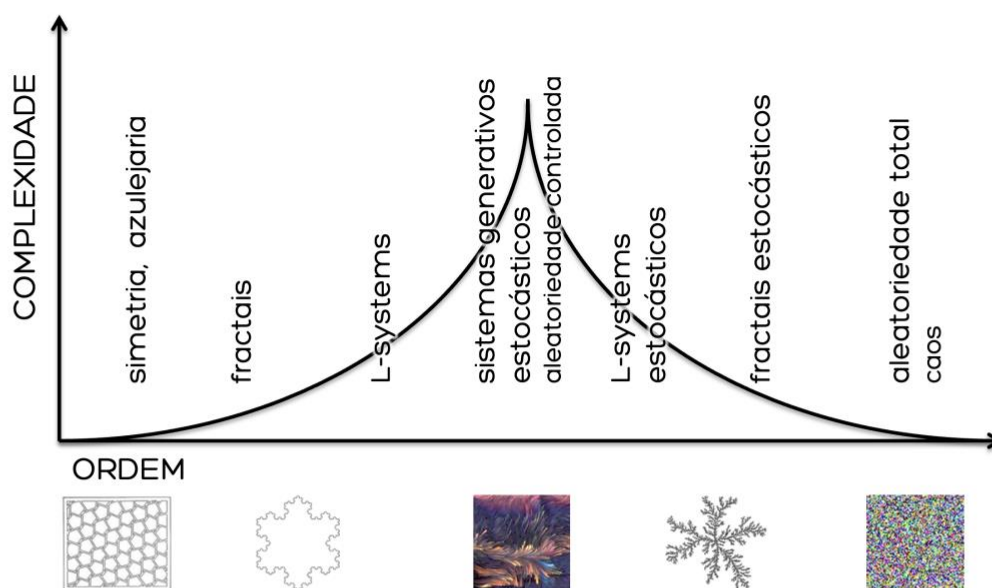



Figura 6 - A relação entre ordem, caos e sistemas complexos (ao centro). Fonte: autor.

E o que é o oposto de “ordenado”? Podemos ter a tentação de pensar em “caótico”, mas a aleatoriedade pura ou o caos puro não são muito interessantes do ponto de vista artístico, pelas mesmas razões que nos levam a desinteressarmo-nos dos sistemas ordenados: tornam-se rapidamente previsíveis, monótonos e desinteressantes. O caos total é tão previsível como a ordem total.

Contudo, no âmbito deste estudo iremos considerar os **sistemas complexos** como o oposto dos sistemas ordenados e caóticos, pois na verdade são a combinação de ambos. Haverá (alguma) estrutura, mas haverá caos (controlado), e os resultados serão imprevisíveis, ainda que sedutores para as nossas capacidades de reconhecimento de padrões.

## OP-ART

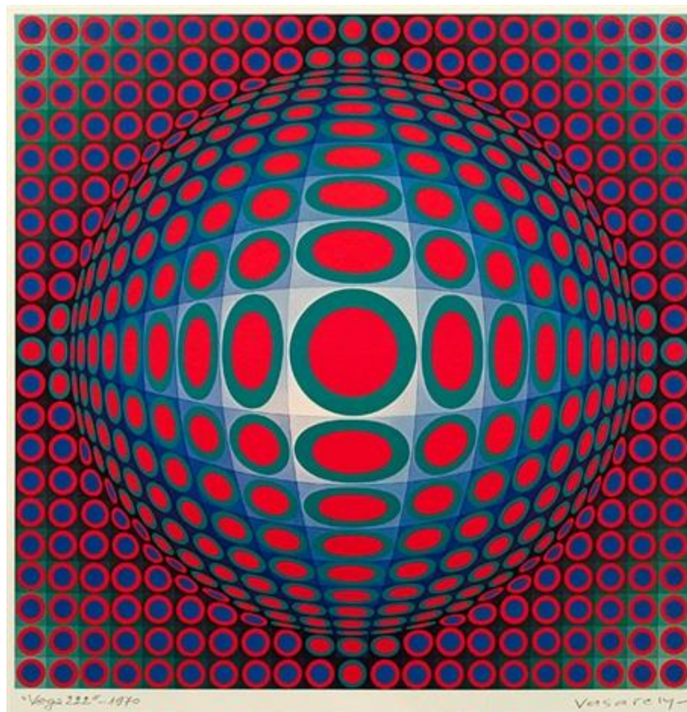


Figura 7 - "Vega 222" de Victor Vasarely (1970). Fonte: WikiArt.

O algoritmo que o nosso sistema autónomo segue pode determinar que os “azulejos” mudam de forma e/ou cor a cada nova geração. Isto é particularmente visível no trabalho de Victor Vasarely, em que o posicionamento no espaço de cada “azulejo” (entendido aqui de forma figurativa, como sendo cada quadrícula que contém um círculo) determina alterações de cor, mas também de forma, deixando de ser quadrados e círculos, e passando a trapézios, losangos e elipses, por deformação iterativa.

Sendo uma forma de arte habitualmente designada por OP-ART (optical-art), na verdade existe um conjunto de regras (um algoritmo) que pode ser seguido sistematicamente para gerar o resultado final. Contudo, os resultados serão sempre idênticos – a menos que algum parâmetro no algoritmo seja alterado durante a execução.

## LITERATURA

Pode parecer estranho que a arte generativa possa ser usada na criação literária, mas a verdade é que não precisamos de estar limitados a conceitos visuais. Se em vez de formas gráficas pensarmos em frases e palavras, torna-se mais imediato perceber como isso pode ser feito. As áreas da literatura eletrónica e da poesia combinatória são, com efeito, muito dinâmicas e autónomas!

*Cem mil biliões de poemas* é um livro de Raymond Queneau, publicado em 1961, e é um exemplo de poesia combinatória. O livro consiste num conjunto de dez sonetos impressos em cartão, com cada linha impressa numa tira separada. Os sonetos não têm apenas o mesmo esquema, mas também a mesma rima, que segue a seguinte regra:

a b a b / a b a b / c c d / e e d

onde a, c, e são rimas femininas: (-ise, -otte, -oques) e b, d são rimas masculinas: ([o] -in).

Assim, quaisquer linhas de um soneto podem ser combinadas com quaisquer outras dos restantes nove, permitindo 100.000.000.000.000 combinações de poemas diferentes.

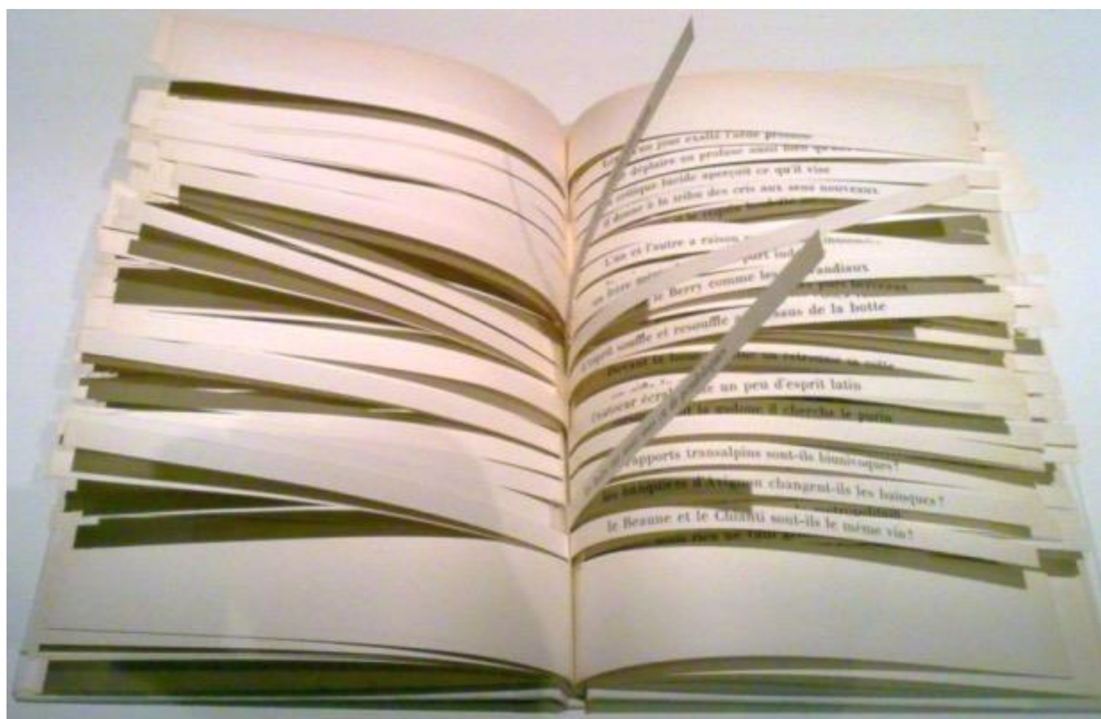



Figura 8 - Cent Mille Milliards de Poèmes (1961) de Raymond Queneau. Fonte : Wikimedia Commons.

E porque é isto arte generativa? Considere-se cada geração como um soneto completo. O leitor determina as regras sobre como gerar o próximo soneto: alterando várias linhas ao mesmo tempo, mudando apenas 3 de cada vez, abrindo um soneto aleatório de cada vez, as possibilidades são ... não infinitas, mas 100.000.000.000.000, mesmo se as regras forem caóticas.

Esta é a primeira indicação de que o caos não implica possibilidades infinitas, como podemos ser tentados a acreditar, mas apenas uma falta de estrutura.

## MÚSICA

A música generativa pode ser obtida de forma semelhante à literatura generativa, substituindo o vocabulário visual/gráfico por sons, frases musicais, notas, timbre, duração, etc.

Um exemplo é *Discreet Music* (1975), o quarto álbum de Brian Eno. A sua intenção era explorar várias maneiras de criar música ambiente com planeamento ou intervenção limitada, usando *delay* induzido por *feedback* através de fita magnética e sintetizadores.

As anotações do álbum contêm um diagrama mostrando como a peça foi criada, começando com duas frases melódicas de diferentes comprimentos executadas a partir do sistema de memória digital de um sintetizador, que são depois reproduzidas através de um equalizador gráfico para ocasionalmente se lhes alterar o timbre.

O resultado é então processado através de uma unidade de eco antes de ser gravado em bobines de fita magnética. A fita é passada para um segundo leitor/gravador e a saída dessa máquina é retroalimentada – proporcionando assim a base de uma nova geração – no primeiro gravador de bobines, que regista os sinais sobrepostos.

**"DISCREET MUSIC"** (30·35)
Recorded at Brian Eno's studio 9·5·75
**THREE VARIATIONS ON THE CANON IN D MAJOR BY JOHANN PACHELBEL**
**(i) "FULLNESS OF WIND"** (9·57)
**(ii) "FRENCH CATALOGUES"** (5·18)
**(iii) "BRUTAL ARDOUR"** (8·17)
Performed by The Cockpit Ensemble conducted by Gavin Bryars (who also helped arrange the pieces).
Recorded at Trident Studios 12·9·75
Engineered by Peter Kelsey.
Produced by Brian Eno
℗ 1975 E.G. Records Ltd.
© 1975 E.G. Records Ltd.

Since I have always preferred making plans to executing them, I have gravitated towards situations and systems that, once set into operation, could create music with little or no intervention on my part.

That is to say, I tend towards the roles of planner and programmer, and then become an audience to the results.

Two ways of satisfying this interest are exemplified on this album. "Discreet Music" is a technological approach to the problem. If there is any score for the piece, it must be the operational diagram of the particular apparatus I used for its production. The key configuration here is the long delay echo system with which I have experimented since I became aware of the musical possibilities of tape recorders in 1964. Having set up this apparatus, my degree of participation in what it subsequently did was limited to (a) providing an input (in this case, two simple and mutually compatible melodic lines of different duration stored on a digital recall system) and (b) occasionally altering the timbre of the synthesizer's output by means of a graphic equalizer.

It is a point of discipline to accept this passive role, and, for once, to ignore the tendency to play the artist by dabbling and interfering. In this case, I was aided by the idea that what I was making was simply a background for my friend Robert Fripp to play over in a series of concerts we had planned. This notion of its future utility, coupled with my own pleasure in "gradual processes" prevented me from attempting to create surprises and less than predictable changes in the piece. I was trying to make a piece that could be listened to and yet could be ignored… perhaps in the spirit of Satie who wanted to make music that could "mingle with the sound of the knives and forks at dinner"

In January this year I had an accident. I was not seriously hurt, but I was confined to bed in a stiff and static position. My friend Judy Nylon visited me and brought me a record of 18th century harp music. After she had gone, and with some considerable difficulty, I put on the record. Having laid down, I realized that the amplifier was set at an extremely low level, and that one channel of the stereo had failed completely. Since I hadn't the energy to get up and improve matters, the record played on almost inaudibly. This presented what was for me a new way of hearing music–as part of the ambience of the environment just as the colour of the light and the sound of the rain were parts of that ambience. It is for this reason that I suggest listening to the piece at comparatively low levels, even to the extent that it frequently falls below the threshold of audibility.

Another way of satisfying the interest in self-regulating and self-generating systems is exemplified in the 3 variations on the Pachelbel Canon. These take their titles from the charmingly inaccurate translation of the French cover notes for the "Erato" recording of the piece made by the orchestra of Jean François Paillard. That particular recording inspired these pieces by its unashamedly romantic rendition of a very systematic Renaissance canon.

In this case the "system" is a group of performers with a set of instructions–and the "input" is the fragment of Pachelbel. Each variation takes a small section of the score (two or four bars) as its starting point, and permutates the players' parts such that they overlay each other in ways not suggested by the original score. In "Fullness of Wind" each player's tempo is decreased, the rate of decrease governed by the pitch of his instrument (bass = slow). "French Catalogues" groups together sets of notes and melodies with time directions gathered from other parts of the score. In "Brutal Ardour" each player has a sequence of notes related to those of the other players, but the sequences are of different lengths so that the original relationships quickly break down.

London, September 1975

The cover photograph is from a video by Brian Eno.

**Operational diagram for "Discreet Music"**
The black line indicates the signal path

Synthesizer with digital recall system
graphic equalizer
echo unit
delay return
output stored on master tape
delay line
record
playback
tape recorders
combined monitor output

Printed in Canada

Figura 9 - Discreet Music (1975) de Brian Eno, as anotações do álbum.

Fonte: https://knowuh.github.io/IDV2/frankenbits/ps-02/D/d.html

Se visualmente podemos facilmente reconhecer padrões, azulejos e tesselações, musicalmente, isso também é possível, à medida que as frases musicais recorrentes ecoam em segundo plano. Ouça o trabalho de Brian Eno e veja (escute) esses padrões.

Brian Eno - Discreet Music https://www.youtube.com/watch?v=hhGb2O1DIF0

**BIBLIOGRAFIA:**

Galanter, P. (2016). An introduction to complexism. *Technoetic Arts: A Journal of Speculative Research*, *14*(1-2), 9-31.

Montfort, N. (2013). *World clock*. Bad Quarto.

Queneau, R. (1961). *Cent mille milliards de poèmes.* Paris, Gallimard.

Steele, S. (2021). Brian Eno: Oblique Music: edited by Sean Albiez and David Pattie, London, Bloomsbury Academic, 2016, 297 pp., $30.95 (paperback), ISBN 978-1441155344.

Vasarely, V., & Spies, W. (1971). *Victor Vasarely*. Thames and Hudson.

**QUESTÕES:**

**A Arte Generativa é...**

1) um género ou estilo, porque o seu tipo pode ser facilmente reconhecido e identificado.

2) um processo de fazer arte, seguindo um conjunto de regras onde cada nova iteração é construída com base na iteração anterior.

3) somente possível usando computadores.

**Os sistemas caóticos generativos têm o mesmo nível de complexidade que...**

1) os sistemas generativos complexos.

2) sistemas ordenados.

3) sistemas de música.

Procure responder e depois confirme as respostas corretas na página seguinte.

**RESPOSTAS:**

**A Arte Generativa é...**

1) um género ou estilo, porque o seu tipo pode ser facilmente reconhecido e identificado.

Na verdade, não. Há muitos géneros e estilos dentro da ampla qualificação de Arte Generativa, que efetivamente se refere a um processo, uma forma de alcançar um resultado, em vez de um estilo ou género artístico.

2) um processo de fazer arte, seguindo um conjunto de regras onde cada nova iteração é construída com base na iteração anterior.

Correto!

3) somente possível usando computadores.

Não exatamente, porque os seres humanos também podem seguir regras e procedimentos. Podem demorar apenas (muito) mais tempo do que um computador para alcançar uma geração / iteração avançada.

**Os sistemas caóticos generativos têm o mesmo nível de complexidade que...**

1) os sistemas generativos complexos.

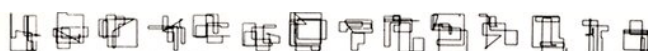
Não propriamente, já que os sistemas puramente caóticos não são complexos, são simplesmente... aleatórios.

2) sistemas ordenados.

Exatamente! Ambos são sistemas simples. Imagine colocar azulejos de cozinha exatamente com o mesmo padrão e também colocar azulejos de cozinha com padrões e formas totalmente diferentes, de qualquer maneira. Ambas as tarefas têm o mesmo nível (baixo) de complexidade.

3) sistemas de música.

Na verdade, os sistemas de música generativos podem ser qualquer coisa entre muito simples ou muito complexos, ordenados ou caóticos, e por isso esta realmente não é a resposta correta.



# SISTEMAS DE LINDENMAYER

Aristid Lindenmayer foi um biólogo húngaro que, em 1968, desenvolveu uma sistematização para a descrição de linguagens formais, e que hoje são designadas por sistemas-L (L-systems) ou sistemas de Lindenmayer. Usando esses sistemas, Lindenmayer modelou o comportamento de células de plantas, mas atualmente os sistemas-L também são usados para modelar plantas inteiras e outras entidades figurativas e abstratas.

Lindenmayer trabalhou com leveduras e fungos de filamentos e estudou os padrões de crescimento de vários tipos de algas, como a bactéria azul/verde *anabaena catenula*. Originalmente, os sistemas-L foram concebidos para fornecer uma descrição formal do desenvolvimento de tais organismos multicelulares simples, e para ilustrar as relações de vizinhança entre células de plantas. Posteriormente, este sistema foi ampliado para descrever plantas superiores e estruturas de ramificação complexas.

Os sistemas-L reescrevem repetidamente uma cadeia de caracteres de acordo com regras simples: o ponto de partida é uma cadeia de caracteres (uma ou mais letras ou símbolos) chamada axioma, que funcionará como uma semente. É o ponto de partida a partir do qual o sistema evolui. Regulando esse crescimento existe um conjunto de regras, também chamadas de produções. Ao aplicar as produções ao axioma, aparecem novas cadeias de caracteres que representam uma ***iteração*** ou ***geração*** de crescimento. Todas as cadeias que se conseguirem produzir dentro desse sistema, usando essas regras, diz-se que fazem parte da sua linguagem.

Modelo original de Lindenmayer para modelar o crescimento das algas:

variáveis: A B

axioma: A

regras (também designadas por *produções*): A → AB, B → A

gerações resultantes:

n = 0 : A
n = 1 : AB
n = 2 : ABA
n = 3 : ABAAB
n = 4 : ABAABABA
n = 5 : ABAABABAABAAB
n = 6 : ABAABABAABAABABAABABA
n = 7 : ABAABABAABAABABAABABAABAABABAABAAB

```
n=0:         A
            /  \
n=1:       A    B
          /|      \
n=2:    A  B       A
       / |   \      | \
n=3:  A B    A     A  B
     / |  \   | \   | \ \
n=4: A B  A A  B  A  B  A
```

Podemos observar no exemplo acima que, se invertermos verticalmente as gerações, com a geração 0 na base, obtemos algo idêntico à estrutura da alga, mesmo sem necessidade de um desenho mais complexo. Contudo, os símbolos (A e B ou outros que constem da gramática usada) podem ser substituídos por indicações, como "desenhar um ramo", ou "produzir um som", entre tantas outras possibilidades.

**BIBLIOGRAFIA:**

Lindenmayer, A. (1986, October). Models for multicellular development: Characterization, inference and complexity of L-systems. In *International Meeting of Young Computer Scientists* (pp. 138-168). Berlin, Heidelberg: Springer Berlin Heidelberg.

## GRÁFICOS DE TARTARUGA

A interpretação mais comum dos sistemas-L é gráfica, utilizando os chamados gráficos de tartaruga, um conceito criado por Seymour Papert, no qual uma tartaruga imaginária se move no ecrã do computador, criando um desenho sob o controle das instruções dadas por um programa de computador. Este tipo de programação e os resultados que produz, são fáceis de entender porque são imediatamente visualizados, e em que os movimentos da tartaruga são representados por símbolos como F (desenhe para frente uma unidade de comprimento), + (vire à esquerda num ângulo predefinido), - (vire à direita num ângulo predefinido).

Figura 10 - Exemplo de gráficos de tartaruga em ação na figura abaixo, em que o ângulo de viragem é de 60º

Pode ver-se alguns exemplos aqui (http://www.kevs3d.co.uk/dev/lsystems/), e também pode testar-se o exemplo do sistema-L original, alterando o número de gerações:

variáveis: A B
constantes: nenhuma
axioma: A
regras: (A → AB), (B → A)

**BIBLIOGRAFIA:**

Solomon, C. J., & Papert, S. (1976, June). A case study of a young child doing Turtle Graphics in LOGO. In *Proceedings of the June 7-10, 1976, national computer conference and exposition* (pp. 1049-1056).

## CURVAS DE KOCH

Niels Fabian Helge von Koch (25 janeiro 1870 – 11 março 1924) foi um matemático sueco. Ele descreveu a curva, que atualmente tem o seu nome, num artigo de 1904 intitulado "Numa curva contínua sem tangentes obtidas por construção geométrica elementar" (título francês original: “*Sur une courbe continue sans tangente obtenue par une construction géométrique élémentaire*”).

Esta é uma das primeiras curvas fratais a serem descritas, com base no seguinte algoritmo:

```
1 - comece com um segmento de reta (AB), designado por iniciador
2 - divida-o em três partes iguais (AC, CE e EB)
3 - substitua a parte do meio (CE) por dois segmentos (CD e DE), ambos do
mesmo comprimento que os três primeiros, criando um triângulo equilátero com a
linha inferior
4 - apague depois o segmento CE. A forma agora consiste em quatro linhas retas
com o mesmo comprimento.
5 - Para cada um desses segmentos (AC, CD, DE e EB), repita o processo acima e
depois continue a transformação, dando origem a uma nova geração a cada
iteração.

Se se aplicar o mesmo processo a um triângulo equilátero, o gerador, em que
cada lado é tratado como o segmento AB original, obtém-se algo semelhante à
figura 9, e daí o nome "floco de neve".
```

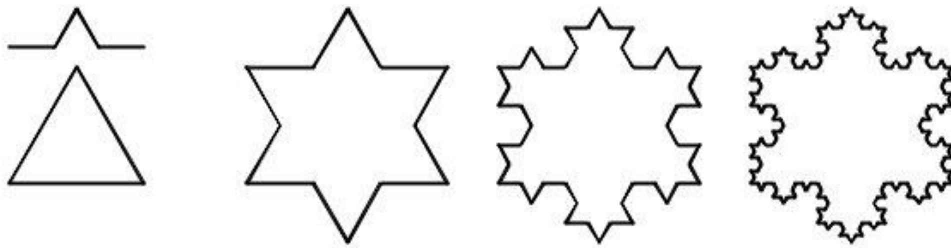


Figura 11 - Gerador e iniciador; primeira, segunda e terceira iterações da curva de Koch.

As curvas de Koch podem ser descritas (e geradas) usando sistemas-L, e o "floco de neve" é apenas uma das muitas curvas de Koch (chamadas curvas embora, na prática, elas sejam compostas por segmentos de reta). No entanto, se o número de gerações tender para o infinito, os segmentos tornam-se progressivamente menores, ficando individualmente indetetáveis, e a figura geral parece ser curva.

## ATIVIDADE

Calcular a segunda geração da curva quadrática de Koch:

variável: F
constantes: +, -
axioma: F+F+F+F
regra: F → F+F-F-FF+F+F-F

Desenhe a geração obtida usando gráficos de tartaruga e ângulos de 90º.

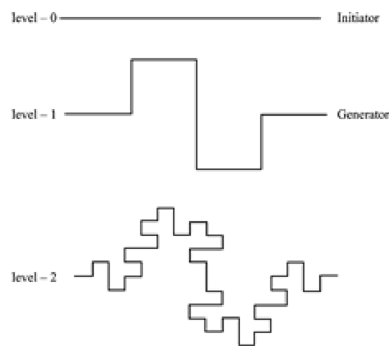



Figura 12

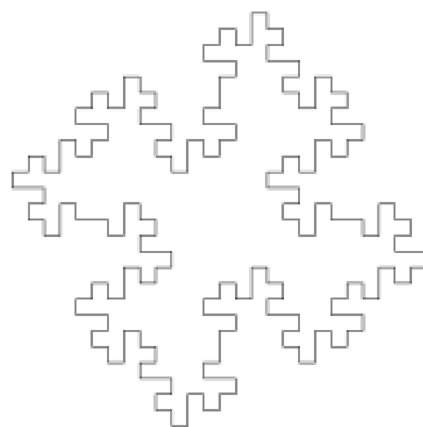


Figura 13 - A curva quadrática de Koch.

Pode explorar mais exemplos de curvas de Koch, como os seguintes:

Axioma: F-F-F-F  Regra: F → FF-F-F-F-F-F+F

Axioma: F-F-F-F  Regra: F → FF-F-F-F

Axioma: F-F-F-F  Regra: F → F-F+F-F-F

e utilizar um sistema online para as animar (garantindo que se utiliza um número baixo de iterações, como 3 ou 4): http://www.kevs3d.co.uk/dev/lsystems/

**ATIVIDADE**

Crie seu próprio sistema-L e use o site anterior para testá-lo. Lembre-se de que pode alterar o ângulo, bem como o axioma inicial.

**BIBLIOGRAFIA:**

Ungar, Š. (2007). The Koch curve: A geometric proof. *The American Mathematical Monthly*, *114*(1), 61-66.

## SISTEMAS-L ESTOCÁSTICOS

Os sistemas-L estocásticos são praticamente idênticos aos sistemas-L discutidos anteriormente. A principal diferença consiste no facto de cada regra poder definir múltiplos resultados, sendo cada um deles definido por uma probabilidade pré-definida. Por exemplo, em vez de "A → AB", podemos ter "A 50% → AB" em conjunto com "A 50% → BB".

Considere o seguinte exemplo com gráficos de tartaruga onde "F" significa avançar e desenhar, "+" significa "virar à esquerda a 30 graus", "-" significa "virar à direita a 30 graus", "[" significa "memorizar esta posição ","] "Significa "quando terminar, regresse para a última posição memorizada e esqueça-a":

Axioma: F
Regra: F → F[+F][-F]F

As iterações 2, 3 e 4 são mostradas abaixo:

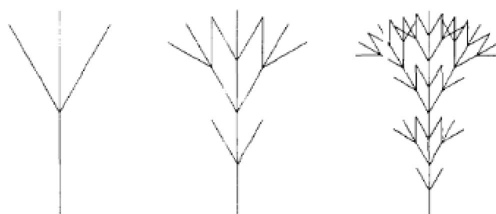


Figura 14 - 2 iterações, 3 iterações e 4 iterações

É fácil identificar a extrema regularidade deste sistema-L e, portanto, o resultado final é demasiado regular para parecer natural, a partir de uma perspetiva de modelação do crescimento da planta. Agora considere sua adaptação estocástica:

Axioma: F

Regras:
F 50% → F[+F][-F]F
F 30% → F[-F]F
F 20% → F[+F]F

Seis iterações executadas por três vezes resultariam agora em plantas distintas, consoante o gerador aleatório decidisse:

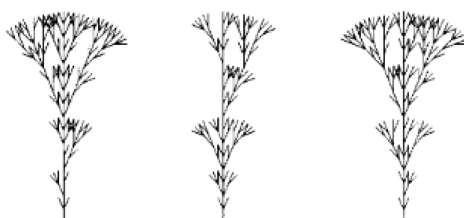


Figura 15 - Três resultados do sistema-L estocástico, todos distintos, e com um ar bastante mais natural.

Isto implica que, para cada nova geração, precisamos de um gerador aleatório (lançar uma moeda, dados, etc.), o que também é algo que os computadores podem replicar / produzir com bastante facilidade. Refinar este modelo e expandi-lo para lidar com três dimensões pode produzir alguns resultados interessantes e de aparência natural. E se juntarmos a regra adicional, que determina que em cada ramo de terminação criamos uma folha, e se aplicamos uma variação estocástica a esta última regra, a partir da qual também podemos obter folhas de tamanho diferente e flores diversas, então o(s) resultado(s) seria(m) algo como exemplificado na figura 12:

Figura 16 - Vários tipos de plantas criadas a partir de Sistemas-L estocásticos. Fonte: Wikimedia Commons.

Veja de seguida um vídeo com vários exemplos de geração de plantas através de sistemas-L: https://www.youtube.com/watch?v=feNVBEPXAcE

## APLICAÇÕES PRÁTICAS

Os sistemas-L podem, no entanto, ser usados para além do domínio de modelação geométrica de plantas, e podem ver-se alguns exemplos no site MorphoCode (https://morphocode.com/branching-structures-l-systems-study-with-rabbit/), em que os sistemas-L foram aplicados para criar uma variedade de conceitos em arquitetura e design.

Figura 17 - Alguns exemplos do site MorphoCode

**BIBLIOGRAFIA:**

Prusinkiewicz, P., & Hanan, J. (2013). *Lindenmayer systems, fractals, and plants* (Vol. 79). Springer Science & Business Media. http://algorithmicbotany.org/papers/lsfp.pdf

Prusinkiewicz, P., & Lindenmayer, A. (2012). *The algorithmic beauty of plants*. Springer Science & Business Media. http://algorithmicbotany.org/papers/abop/abop.pdf

Pearson, M. (2011). *Generative art: a practical guide using processing*. Simon and Schuster. https://www.mat.ucsb.edu/~g.legrady/academic/courses/20f594/txt/generativeArt2.pdf

## QUESTÕES

**Um sistema Lindenmayer é**

1) um estilo de arte generativa.

2) um processo de descrever a reprodução ou crescimento de organismos multicelulares.

3) um processo para descrever desenhos simples.

**Os sistemas-L estocásticos diferem dos sistemas-L regulares**

1) porque não se baseiam em regras.

2) porque podem ser aplicados ao modelo de crescimento celular.

3) porque eles podem ter várias regras baseadas em probabilidades para o mesmo símbolo.

**Nos gráficos de tartaruga os símbolos "F", "+", "-", "[" e "]" geralmente significam**

1) avance e desenhe, vire à esquerda, vire à direita, memorize a posição atual, uma vez que termine regresse à última posição memorizada e esqueça-a.

2) avance e não desenhe, memorize a posição atual, uma vez que termine regresse à última posição memorizada e esqueça-a, vire à esquerda, vire à direita.

3) avance e não desenhe, vire à esquerda, vire à direita, memorize a posição atual, uma vez que termine regresse à última posição memorizada e esqueça-a.

**RESPOSTAS**

**Um sistema Lindenmayer é**

1) um estilo de arte generativa.

> Não propriamente, os sistemas-L não estão restritos à arte generativa, apesar de poderem ser usados para produzir obras de arte generativa.

2) um processo de descrever a reprodução ou crescimento de organismos multicelulares.

> Correto!

3) um processo para descrever desenhos simples.

> Não é bem isso, os sistemas-L são abstrações e não implicam necessariamente uma ligação a sistemas de desenho ou gráficos.

**Os sistemas-L estocásticos diferem dos sistemas-L regulares**

1) porque não se baseiam em regras.

> Não, os sistemas-L estocásticos baseiam-se tanto em regras como os sistemas-L regulares.

2) porque podem ser aplicados ao modelo de crescimento celular.

> Não exatamente, todos os sistemas-L, estocásticos ou regulares, podem potencialmente ser aplicados à modelação de sistemas de crescimento celular.

3) porque eles podem ter várias regras baseadas em probabilidades para o mesmo símbolo.

> Correto!

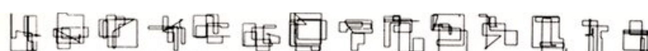
**Nos gráficos de tartaruga os símbolos "F", "+", "-", "[" e "]" geralmente significam**

1) avance e desenhe, vire à esquerda, vire à direita, memorize a posição atual, uma vez que termine regresse à última posição memorizada e esqueça-a.

> Correto.

2) ~~avance e não desenhe~~, ~~memorize a posição atual~~, ~~uma vez que termine regresse à última posição memorizada e esqueça-a~~, ~~vire à esquerda~~, ~~vire à direita~~.

3) ~~avance e não desenhe~~, vire à esquerda, vire à direita, memorize a posição atual, uma vez que termine regresse à última posição memorizada e esqueça-a.





É, pois, quando chegamos ao meio digital que a arte generativa parece explodir em termos de possibilidades e complexidade. A arte generativa digital não só engloba todos os exemplos anteriores, pois é capaz de reproduzi-los, mas aproveita o poder de cálculo do computador para produzir resultados mais complexos e sofisticados.

## LITERATURA ELETRÓNICA

Na literatura, Nick Montfort fornece um exemplo interessante (entre muitos) do que pode ser alcançado, com o romance "World Clock", que é na realidade o resultado de 165 linhas de programação em Python (https://nickm.com/code/world_clock.py). Analise o seguinte fragmento de abertura:

> *It is now exactly 05:00 in Samarkand. In some ramshackle dwelling a person who is called Gang, who is on the small side, reads an entirely made-up word on a box of breakfast cereal. He turns entirely around.*
>
> *It is now right about 18:01 in Matamoros. In some dim yet decent structure a man named Tao, who is no larger or smaller than one would expect, reads a tiny numeric code from a recipe clipping. He smiles a tiny smile.*
>
> *It is now as it happens 19:02 in Grand Turk. In some sturdy yet undistinguished habitat a youth named Peng, who is quite sizable and imposing, reads a stained card. He sits up straight.*
>
> *It is now only a moment before 02:03 in Windhoek. In some suitable structure someone named Ezra, who is significantly smaller than others of the same age, reads a canary-colored manuscript. He hums quietly.*
>
> *(...)*

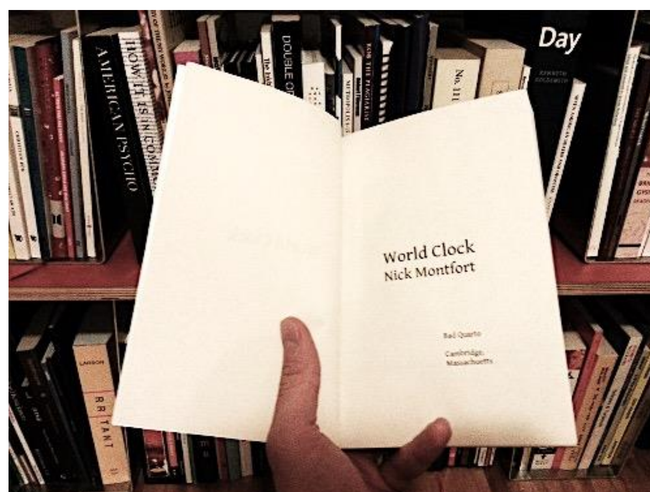



Figura 18 - World Clock de Nick Monfort, a edição impressa.

Fonte: https://nickm.com/post/wp-content/stuff/world_clock_title_page.jpg

A estrutura torna-se bastante evidente. Cada parágrafo (cada geração) acontece um minuto depois do anterior e leva em consideração o fuso horário. A primeira frase localiza a ação em tempo e lugar. A frase seguinte começa sempre com "In some" e define uma localização e uma pessoa, com uma característica particular, que está a fazer uma determinada ação. A última frase define uma ação complementar.

Essencialmente, o autor definiu grupos de frases, ou partes de frases, que podem ser usadas de forma intercambiável, e o programa combina-as aleatoriamente, definindo a hora como sendo um minuto após a combinação anterior. O mesmo resultado poderia ser alcançado se tivéssemos sete caixas com recortes de papel, contendo todas as partes variáveis das frases, e tirássemos um pedaço de papel de forma aleatória de cada caixa para completar cada novo parágrafo:

> *It is now exactly* [número da geração+fuso horário do local da caixa 1] *in* ["lugar" - caixa 1]. *In some* ["localização" - caixa 2] ["pessoa que se chama" - caixa 3] ["nome" - caixa 4] *who is* ["característica física" – caixa 5], *reads* ["coisas para ler" - caixa 6]. *He* ["faz algo" - caixa 7].

**ATIVIDADE**

Experimente este tipo de sistema e produza a sua própria literatura generativa, com base na abordagem das "caixas". Lembre-se de que cada caixa (chamemos-lhe uma variável) tem um tipo específico de conteúdo (chamemos-lhe um tipo de dados): em alguns casos, será um nome, noutros casos, um país, e noutros ainda, um verbo ou substantivo que descreva uma ação, e todos os pedaços de papel (chamemos-lhes dados) que vamos usar devem produzir significado no parágrafo estrutural que precisamos de criar.

Por exemplo, vamos considerar o seguinte exemplo com 3 variáveis:

[CAIXA1 – NOME DE PESSOA] por hábito sempre [CAIXA2 – AÇÃO] antes de [CAIXA3 – AÇÃO].

É evidente que CAIXA2 e CAIXA3 devem ter dados diferentes: CAIXA2 pode ter expressões verbais como "saía" ou "ficava na cama e pensava na vida", enquanto a CAIXA3 deve ter expressões verbais como "ir para o trabalho" ou "ouvir música".

## AUDIOVISUAL

De entre os múltiplos exemplos de arte generativa audiovisual, existem vários disponíveis como "Experiments with Google" (https://experiments.withgoogle.com/search?q=Generative) de entre os quais destaco:

Generative Tubes: https://labs.fluuu.id/tubes/

Generative Geometries: https://labs.fluuu.id/geo/dist/

A maioria dos projetos resulta em grafismos abstratos, embora existam algumas exceções, como as seguintes:

Ghost Maps: https://ojack.xyz/ghost-map/

Let me dream again: https://artsexperiments.withgoogle.com/let-me-dream-again/

**ATIVIDADE**

Divirta-se explorando o WeaveSilk (http://weavesilk.com/), ilustrado acima, que é um sistema complexo generativo. Cada nova geração (de linhas) irá seguir (aproximadamente) o cursor do rato, mas ao mesmo tempo parecerá ter uma vida própria, e a imagem final é decididamente mais atraente do que se todas as linhas fossem desenhadas aleatoriamente (sem nenhuma estrutura), ou se estivessem limitadas a seguir o cursor do rato (sem qualquer variação).

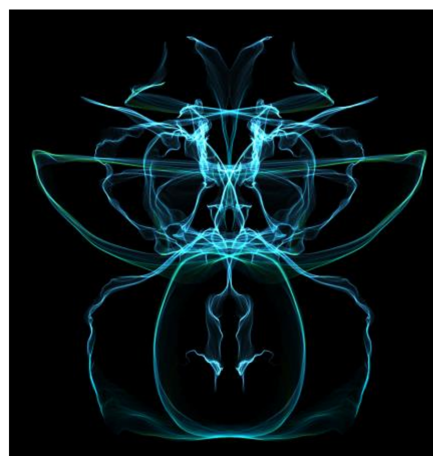
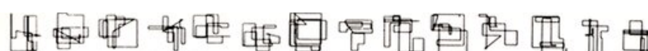


Figura 19 - Testando WeaveSilk



## CRIANDO ARTE GENERATIVA COMPUTACIONAL

A arte generativa é criada através dum sistema autónomo, o que habitualmente se traduz pela utilização de uma máquina (elemento não humano) que segue determinado conjunto de comandos, tomando autonomamente decisões que definem as características da obra de arte resultante que, de outra forma, exigiriam decisões feitas diretamente pelo artista.

O autor/programador pode afirmar que o sistema generativo representa o seu conceito artístico – ou pelo menos um conjunto de fronteiras estéticas dentro das quais o sistema evolui, ainda que o sistema assuma o papel de executante da obra final.

O termo ***arte generativa*** é frequentemente usado para se referir à arte algorítmica (arte gerada por computador, que é determinada algoritmicamente), e muitas vezes é assumido como sendo arte visual, tanto estática como animada.

Mas a arte generativa também pode ser feita usando sistemas químicos, biológicos, mecânicos e robóticos, com materiais inteligentes, aleatorização manual, matemática, mapeamento de dados, simetria, repetição de padrões, e pode dar origem a obras de arte audiovisual, incluindo literatura e poesia, arquitetura e planeamento urbano.

Celestino Soddu propôs a seguinte definição:

> A arte generativa é a ideia realizada como código genético de eventos artificiais, como a construção de sistemas complexos dinâmicos capazes de gerar infinitas variações. Cada projeto generativo é um conceito de software que funciona produzindo eventos únicos e não repetíveis, como música ou objetos 3D, possíveis e múltiplas expressões da ideia geradora, altamente reconhecíveis como uma visão pertencente a um artista / designer / músico / arquiteto / matemático.

E Philip Galanter também sugeriu uma abordagem semelhante:

> A arte generativa refere-se a qualquer prática de arte onde o artista cria um processo, como um conjunto de regras de linguagem natural, um programa de computador, uma máquina ou outra invenção processual, que é então posta em movimento com algum grau de autonomia, contribuindo ou resultando numa obra de arte completa.

A arte visual generativa é suficientemente rica para fornecer uma grande variedade de experiências estéticas, através de uma diversidade de géneros, desde a glitch-art, audiovisuais abstratos ou até mesmo retratos.

Figura 20 - Glitch-Art: Rosa Menkmann, To Smell and Taste Black Matter 1 & 2, 2009

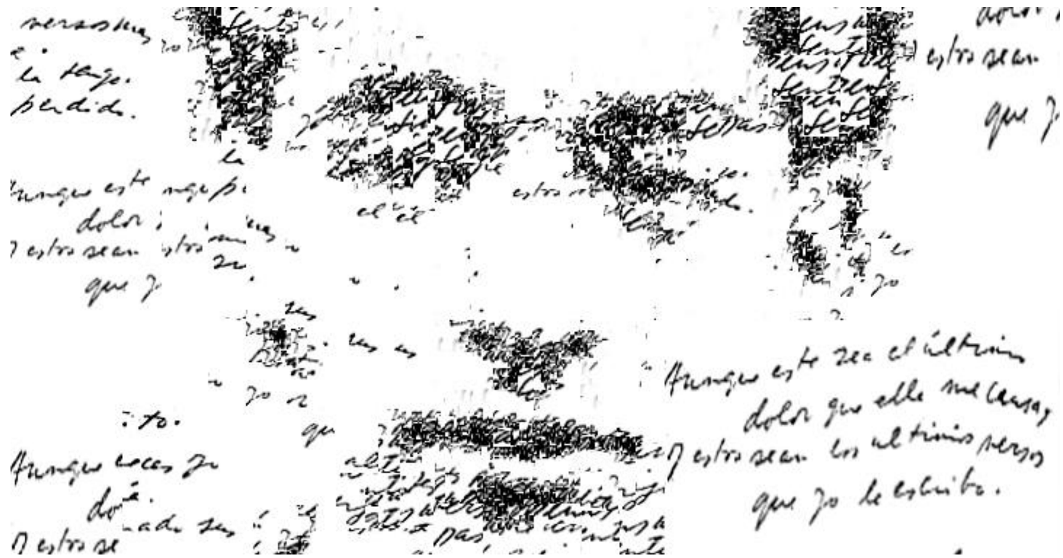



Figura 21 - Retratos generativos: Sergio Albiac, Manuscript self portrait of Pablo Neruda (1904 – 1973), 2012

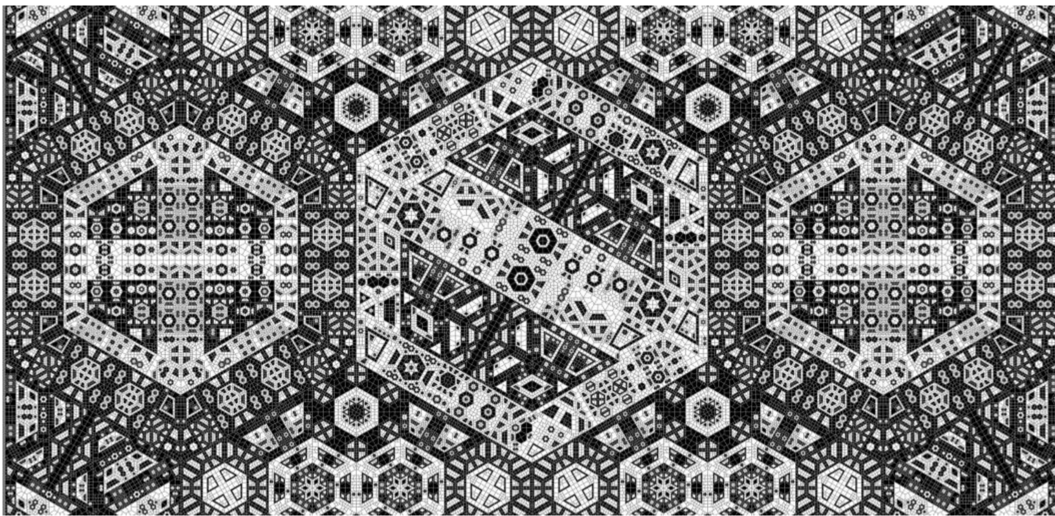


Figura 22 - Tesselação generativa: Fleen / John Greene, 2015 03 15, 2015

Figura 23 - Videoarte Generativa: Pedro Alves da Veiga, Post-Digital City, 2022

Com os sistemas L, fomos expostos à interpretação de "F" como "avançar e desenhar". Mas e se a interpretação de "F" fosse algo como "desenhar um círculo vermelho centrado em x, y com raio z; mova o centro do próximo círculo para x + z, y + z", ou "utilize uma palavra dum conjunto de substantivos, outra palavra dum conjunto de verbos e crie uma frase do tipo – Este SUBSTANTIVO não pode VERBO", ou mesmo "junte os pés, erga os braços acima da cabeça, mova-se um passo para trás, olhe para a esquerda, baixe os braços". Torna-se evidente que todas as formas de arte, incluindo a dança e a performance, podem ser abordadas por esta abordagem metodológica.

Consideremos então a abordagem da arte generativa como um sistema estruturante e autónomo, cuja execução resulta em criação artística usando um alfabeto próprio e variável, consoante o autor e a instância: linhas, cores, sons, formas, mas também conceitos de interpretação, expressão corporal – se houver um nível de envolvimento físico com a obra de arte, incluindo movimento, gesto, diálogo empático, ouvindo com todos os sentidos e influenciando a "próxima geração". Se se considerar a abordagem da arte generativa para todos os aspetos do *sensorium* humano, as obras de arte resultantes irão fornecer experiências mais ricas e mais significativas, pois, assim que os símbolos são reunidos, eles transformam-se em algo mais complexo: performances, experiências estéticas e cognitivas.

# ETAPAS DE DESENHO DA OBRA

Em toda a criação de arte generativa podemos distinguir um conjunto de etapas comuns no seu desenho e desenvolvimento.

## 1 – IDENTIFICAÇÃO DO VOCABULÁRIO

A primeira fase corresponde à identificação dos média que irão ser processados: linhas, pontos e cores, áudio e vídeo, dispositivos eletrónicos, entre outros. Nesta etapa efetuam-se os primeiros testes (manuais) de combinação para ver se o resultado obtido está de acordo com as expectativas do artista.

## 2 – DESENHO DE UM DISPOSITIVO ESTRUTURANTE

Segue-se a conceção do dispositivo estruturante, através do qual o artista/criador define a evolução estética da obra de arte, bem como os seus limites. Este é essencialmente um conjunto de regras e procedimentos, um algoritmo, um conjunto de regras de aquisição. Define ainda como o vocabulário, anteriormente selecionado, será usado no sistema e um conjunto de mecanismos de potencialização ou modulação, através dos quais o vocabulário será manipulado, alterado ou combinado.

Existem vários estudos sobre as correspondências de modalidades cruzadas na perceção humana. Por exemplo, os sons agudos são geralmente relacionados a luzes pequenas e brilhantes, e a um posicionamento espacial mais elevado, enquanto o movimento lento está associado a ambientes mais escuros, sons prolongados e graves. Mais abrangentemente, o volume sonoro é geralmente associado ao brilho e ao tamanho; o ritmo sonoro relaciona-se facilmente com a frequência espacial, ou luminosa. Essas relações sugerem mapeamentos entre o som e o posicionamento espacial, o movimento, a colocação e a forma, o que, obviamente, pode ser ignorado, e mesmo contrariado, mas também expandido, e aqui reside uma parte significativa da próxima etapa de design: mapeamento para vários parâmetros.

## 3 — AMPLIFICAÇÃO OU MAPEAMENTO

A etapa seguinte é a da amplificação, onde são adicionadas as extensões cognitivas ao sistema, onde as correlações são feitas entre diferentes tipos de media e podem ocorrer práticas colaborativas. A arte generativa é muitas vezes recursiva, e os mecanismos de *feedback* podem ser desencadeados pela informação obtida a partir da própria interação, e serem usados para influenciar a direção e a evolução da obra de arte generativa. Desta forma, som, imagem, luz, movimento, emoção, podem ser interpretados e manipulados numa performance interativa dinâmica, ou como um sistema evolutivo independente.

## 4 — DETEÇÃO E CRIAÇÃO DE EVENTOS

Finalmente, na etapa final, o artista já fez os ajustes no sistema, tanto em termos de mecanismo estruturante como nos mecanismos de amplificação, e agora está preocupado em identificar as ocorrências mais interessantes, quais as parametrizações que conduzem a resultados esteticamente mais ricos, à medida que o sistema é executado. O artista pode detetar esses eventos através de tentativa e erro, e depois identificar conjuntos de gerações únicos como expressões artísticas completas do conceito e estética iniciais, ou então assumi-los como uma pauta, um guia. Mas também pode optar por uma performance gerada em tempo real, pelo artista, performers e audiência, com tantos graus de imprevisibilidade quanto o artista decidiu usar a aleatoriedade e a interatividade no sistema.

# TRÊS ABORDAGENS DISTINTAS ÀS OBRAS GENERATIVAS

Projetar uma obra de arte generativa leva em consideração três abordagens performativas possíveis, embora todas elas contemplem as etapas anteriormente mencionadas: (1) tempo de execução (*runtime*), (2) interatividade e (3) codificação ao vivo (*live-coding*).

## TEMPO DE EXECUÇÃO

O código é executado e a obra de arte corresponde à experiência visual, auditiva, cinemática ou sensorial que dele resulta. Tanto o programador como o público são passivos durante a execução do código, à medida que o programa evolui de acordo com as suas próprias regras e parâmetros.

## INTERATIVIDADE

O código é executado, mas o resultado final é influenciado pela participação ativa do público ou de dados externos que, ao atuar sobre parâmetros/variáveis do código, alteram o resultado em tempo real.

Figura 24 - Beau Jackson, utilizando WeaveSilk, sem título. Fonte: WeaveSilk.

## CODIFICAÇÃO AO VIVO

O código é escrito e executado ao vivo pelo programador, diante dum público que é presenteado com a capacidade técnica e criativa do programador e com a performance/experiência resultante, numa atuação ao vivo. A interação do público pode ou não ocorrer.

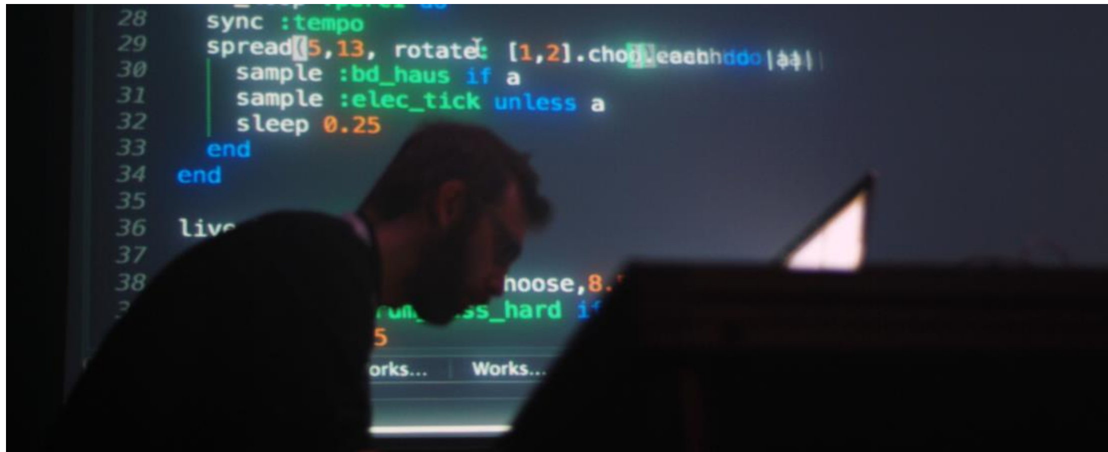



Figura 25 - Martin Zeilinger codificando ao vivo com Sonic Pi. Fonte: https://vectorfestival.org/

## BIBLIOGRAFIA:

Spence C. Crossmodal correspondences: a tutorial review. Atten Percept Psychophys. 2011 May;73(4):971-95. doi: 10.3758/s13414-010-0073-7. PMID: 21264748.

Veiga, P. A. (2023, September). Generative Ominous Dataset: Testing the Current Public Perception of Generative Art. In *Proceedings of the 20th International Conference on Culture and Computer Science: Code and Materiality* (pp. 1-10).

Veiga, P. A. (2022). Patient Zer0: Creating Online Generative Art During the COVID-19 Pandemic. International Journal of Art, Culture, Design, and Technology (IJACDT), 11(3), 1-19. https://doi.org/10.4018/IJACDT.314953

## RECURSOS ADICIONAIS:

Arte Generativa Aumentada: https://vimeo.com/437973449

Some Thoughts on Generative Art: https://inconvergent.net/thoughts-on-generative-art/

SEGUNDA PARTE

# PROCESSING E P5.JS
# CÓDIGO CRIATIVO

# INTRODUÇÃO

É importante ver a programação como aquilo que é: um meio que permite vários tipos de criação, incluindo a arte e o design. Tendo uma boa compreensão do que é a programação e a codificação, pode-se entender melhor as suas possibilidades e refinar-se a sensibilidade de acordo com a sua utilização. O código é uma língua, com a qual se explica como desenhar, escrever poemas ou assinar contratos. Prática, conhecimento e cultura definirão o que cada um de nós cria com o código.

Existem várias linguagens de programação que são usadas na prática de arte generativa:

PURE DATA

https://puredata.info/

SUPER COLLIDER

http://supercollider.github.io/

CONTEXT FREE

https://www.contextfreeart.org/

EXTEMPORE

https://extemporelang.github.io/

NODEBOX

https://www.nodebox.net/

PROCESSING

https://processing.org/

A nossa atenção ficará focada no Processing e/ou no P5.JS. Ambos são ambientes de programação baseados em Java, criados por artistas, otimizados para interatividade e artes visuais, embora exista a possibilidade de manipulação de áudio, bem como de audiovisuais. O Processing é sobretudo usado offline e o P5.JS online, já que a edição e execução ocorrem dentro de um browser.

## O QUE É PROGRAMAR

Antes de embarcarmos na programação de computadores, com vista a criar obras de média-arte digital, online e offline, convém esclarecer dois conceitos importantes: o de **algoritmo** e o de **programa**.

### ALGORITMO

É uma sequência finita de ações executáveis, que visam descrever uma solução para um determinado tipo de problema. Este conceito é frequentemente ilustrado pelos exemplos das receitas de culinária, que descrevem passos e quantidades até chegar ao resultado final. Esses passos podem ser repetidos (iterações) ou precisar de decisões (tais como comparações ou análise lógica) até que a tarefa seja completada. Um algoritmo corretamente executado não irá resolver um problema se estiver implementado incorretamente ou se não for apropriado ao problema.

### PROGRAMA

É uma sequência de instruções seguidas por um computador para produzir um determinado resultado, ou seja, é a tradução de um algoritmo para uma linguagem que o computador "compreende" para produzir esse mesmo resultado.

# PROCESSING E P5.JS

O Processing é uma linguagem de programação de código aberto e ambiente de desenvolvimento integrado (IDE), construído para as artes eletrónicas e comunidades de projetos visuais com o objetivo de ensinar noções básicas de programação de computador em contexto visual.

O projeto foi iniciado em 2001 por Casey Reas e Ben Fry, ambos ex-membros do Grupo de Computação do MIT Media Lab. Um dos objetivos do Processing é atuar como uma ferramenta para não-programadores iniciados com a programação, através da satisfação imediata com um retorno visual. A linguagem tem por base as capacidades gráficas da linguagem de programação Java, simplificando características e criando alguns novos.

O P5.js é desenvolvido com JavaScript utilizada para programação criativa, focada em tornar a programação acessível e inclusiva para artistas, designers, educadores, iniciantes e qualquer outra pessoa. O P5.js é gratuito e de código aberto. Tal como o Processing, que lhe está na origem, utiliza a metáfora de um esboço/sketch, e possui um conjunto completo de funcionalidades de desenho. No entanto, não estamos limitados ao espaço de desenho. Podemos pensar em toda a página do navegador como o nosso esboço/sketch, incluindo objetos HTML5 para texto, rato, vídeo, webcam e som. O P5.js é uma interpretação do Processing para a web de hoje, criado com o apoio da Processing Foundation.

Existem muitas diferenças entre o Processing e o P5.js, mas podemos resumi-las desta forma:

O Processing permite criar diretamente aplicações artísticas offline e o P5.js permite criá-las diretamente online. Formalmente, as duas linguagens de programação partilham muitos conceitos, mas apresentam algumas pequenas diferenças.

**RECURSOS:**

www.processing.org

www.p5js.org

## INSTALAÇÃO E AMBIENTE

O software para o Processing pode ser baixado gratuitamente na área de downloads do website Processing.org (http://www.processing.org). Existem versões para Linux, Mac OSX e Windows.

O Processing não necessita de uma instalação específica, bastando descompactar os ficheiros na pasta de Programas ou em qualquer outra pasta do seu computador seu computador. O Processing pode ser aberto diretamente clicando o ficheiro executável.

A interface do Processing é composta por duas janelas – a principal, contendo um editor onde podemos criar a nossa programação e a janela de visualização onde é apresentado o resultado gráfico dos nossos programas, com linhas, texto, imagens e vídeos.

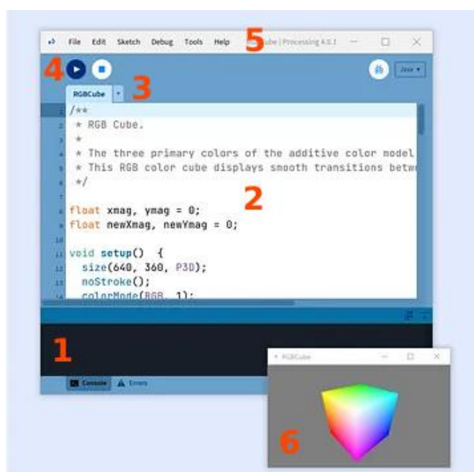



Figura 26 - Zonas de interação no ambiente Processing

1. Consola (área de testes e mensagens de erros)
2. Editor (programação por texto)
3. Barra de objetos (cada etiqueta representa um novo objeto)
4. Barra de ações contendo botões: Run, Stop
5. A janela de visualização é aberta quando executamos algum programa com o botão "Run".

Os ficheiros fonte do Processing possuem a extensão .pde e podem ser alterados em qualquer editor de texto. Já os ficheiros e bibliotecas exportados para web (applets) possuem as extensões .java e .jar. O Processing também permite a exportação de aplicações para desktop, tanto para Windows, como para Mac, como para sistemas Android.

Cada programa criado no ambiente do Processing é guardado numa pasta padrão com o mesmo nome que o ficheiro fonte, em que é guardado o código. Um sketch de Processing deve ser partilhado comprimindo a pasta e todos os ficheiros nela contidos – não basta partilhar o ficheiro com a extensão .pde.

A janela de visualização tem as suas dimensões (largura e altura em pixels) configuradas pela função size(largura, altura) que deve ser definida inicialmente na programação. O tamanho padrão é 100 x 100 pixels.

O espaço de visualização nessa janela é endereçado através de coordenadas, sendo que o ponto (0,0) corresponde por defeito ao canto superior esquerdo, e o ponto com a máxima largura e altura que definirmos ao canto inferior direito.

Tanto o Processing como o P5.js permitem-nos usar duas variáveis pré-definidas, chamadas **width** e **height**, que contêm sempre o valor correto da largura e altura do nosso espaço de visualização.

**ATIVIDADE**

> Após a instalação, explore o ambiente ao máximo e tente correr alguns dos exemplos incluídos. Visite o website oficial https://processing.org/ onde poderá encontrar muito material interessante para se familiarizar.

Já o P5.js não precisa de qualquer instalação, bastando aceder com o seu navegador a https://editor.p5js.org/ e, caso ainda não tenha uma conta para poder guardar os seus trabalhos, poderá fazê-lo aqui https://editor.p5js.org/signup.

O ambiente de trabalho do P5.js considera também a existência de uma zona de edição de código, do lado esquerdo, uma zona de execução/visualização, do lado direito, e a consola em baixo.

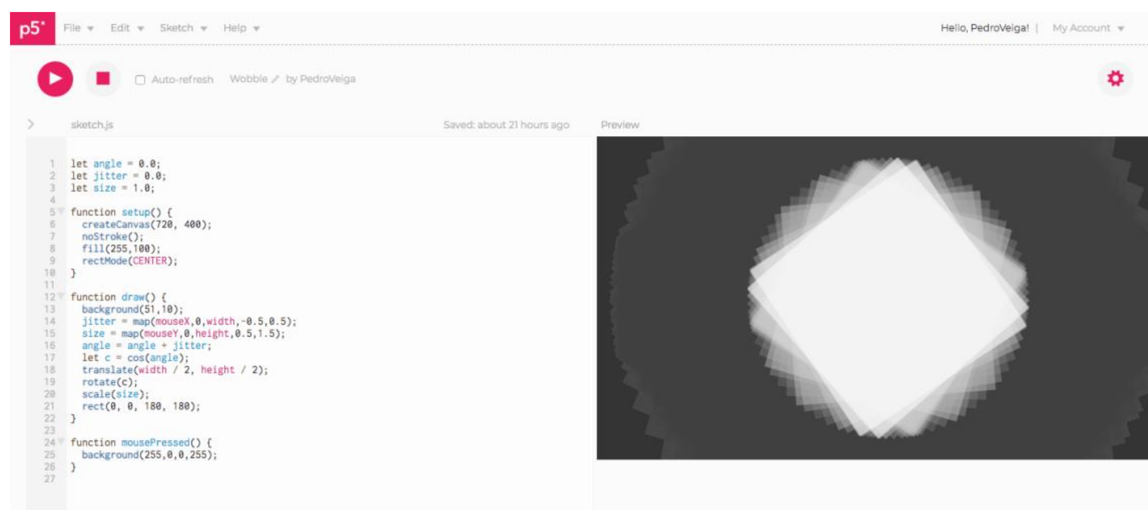



Figura 27 - Aspeto do editor do P5.js em https://editor.p5js.org/

# CÓDIGO

Antes de começarmos a programar, é importante perceber como se estrutura um programa. Um programa é na realidade um conjunto de instruções que o programador fornece ao computador para executar. Cada linha de código é interpretada como uma instrução, passível de execução ativa, ou não. As instruções são executadas de forma sequencial.

Por exemplo, um elemento que não causa qualquer execução, mas é muito útil é o comentário. Ele deve ser inserido ao longo do programa, como uma boa prática de explicação do código. Ele serve para o programador explicar o que o seu código faz para outra pessoa que o venha a (re)utilizar ou analisar.

Em processing, as linhas de comentário são criadas com a inserção de duplas barras no início da linha:

```
// isto é um comentário
```

## FUNÇÕES

As funções são as peças fundamentais do funcionamento de um programa em Processing e P5.js.

Existem funções pré-definidas que podem ser utilizadas de imediato, e servem para fazer várias tarefas: definir o tamanho de uma janela, alterar a cor, desenhar polígonos, etc.

Existem funções que são criadas pelo próprio programador, e neste caso, elas podem conter um conjunto variado de instruções que executam as mais diversas operações ou cálculos e utilizar inclusivamente outras funções (pré-definidas ou não).

As funções podem receber um número variado de parâmetros de entrada, ou simplesmente não receber nenhum.

A sintaxe em Processing utilizada para declarar funções é sempre com o seguinte formato:

<tipo> <nome da função> (<param1>, <param2>, …., <paramn>)

Onde <tipo> identifica o tipo de dado retornado pela função no ponto de chamada (números inteiros ou ponto flutuante, texto, booleano, etc.), <nome da função> é o nome atribuído a mesma e utilizado para chamá-la (deve ser em minúscula, pela boa prática de programação) e <param>, define os parâmetros a serem passados para função quando ela é invocada (número variável, separado por vírgulas).

**Alguns exemplos:**

Uma função pré-definida que permite definir o tamanho da janela de desenho a ser lançada. Neste caso, seria 300 x 300 pixels (recebe 2 parâmetros do tipo inteiro, o primeiro para definir a largura e o segundo para a altura da janela)

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

Uma função criada pelo programador que recebe 2 parâmetros de entrada, o primeiro numérico decimal e o segundo, numérico inteiro, e retorna um valor numérico decimal no seu ponto de chamada. O nome da função é **noise**. Tudo o que for definido entre { } é interpretado como instruções a serem executadas quando a função é invocada…

```
float noise (float n, int m){ }
```

No P5.js as funções são ligeiramente diferentes, e são sempre declaradas com a palavra **function**. A função usada em P5.js para dimensionar a janela de visualização chama-se **createCanvas**, e recebe os mesmos parâmetros que **size**, em Processing:

```
function setup() {
  createCanvas(720, 400);
  noStroke();
  fill(255,100);
  rectMode(CENTER);
}
```

### SETUP E DRAW

Existem duas funções com particular importância em Processing e P5.js, que são as funções setup() e draw().

**Setup** inclui todos os comandos que façam parte da inicialização do nosso sistema. Com a analogia das receitas de culinária, **setup** é a fase da preparação em que separamos os ingredientes, definimos as suas quantidades iniciais e preparamo-los para o trabalho que se segue. Esta função é executada apenas uma vez, no arranque do programa.

**Draw** é uma função que, como o nome sugere, se encarrega do desenho, isto é, pode manter o nosso programa em execução, a *desenhar*, por tempo indeterminado. Esta função tem uma característica muito própria: ela é executada em *loop*, ou seja, cada vez que chega ao fim o conjunto de instruções que contém, ela começa uma nova iteração, desde o início. Este aspecto faz com que o Processing (e o P5.js) sejam particularmente adequados para projetos de arte generativa, pois estão constantemente a calcular/desenhar novas *gerações*.

Em Processing, um programa que cria uma janela de 200 x 200 pixels e desenha um quadrado vermelho sobre fundo preto, tem este aspeto:

```
void setup() {
  size(200,200);
  background(0);
  fill(255,0,0);
}

void draw() {
  rect(10,10,180,180);
}
```

O mesmo programa em P5.js tem este aspeto:

```
function setup() {
  createCanvas(200,200);
  background(0);
  fill(255,0,0);
}

function draw() {
  rect(10,10,180,180);
}
```

## COMANDOS E EXPRESSÕES

A maior dificuldade que alguém apresenta para aprender a programar é a de conseguir reduzir as ações complexas às suas componentes atómicas e transformá-las num conjunto coerente de instruções ou comandos simples e sequenciais.

As expressões em programação são geralmente uma combinação de operadores como +, - , * e / que são avaliados da esquerda para a direita, embora com prioridades que, tal como na matemática, podem ser quebradas através da utilização de parênteses curvos.

Elas podem ser simples e básicas, envolvendo poucos valores e operadores, ou ao contrário, implicarem muitos valores (ou variáveis de memória) e operadores.

Qualquer expressão deve resultar num valor, que por sua vez, é utilizado no programa para alguma coisa. Pode ser um valor numérico, lógico (verdadeiro ou falso) ou até em uma sequência de caracteres.

Os comandos são, na verdade, cada uma das linhas de código que identificam uma instrução atómica, terminada por ponto e vírgula (;).

Por exemplo, se pretendermos desenhar um retângulo com o dobro da altura em relação à sua largura, no centro do ecrã:

```
rect(width/2, height/2, 100, 200);
```

A linha acima é um comando, e usamos duas expressões como parâmetros da função rect(), nomeadamente width/2 e height/2 (dividimos cada um dos valores por 2 para obter o ponto médio). Como tinha sido referido acima, **width** contém sempre o valor da largura da nossa janela de visualização, e **height** o da altura, e assim estamos a indicar que o retângulo vai ter um canto no centro (metade da largura e metade da altura) do nosso espaço de visualização.

## VARIÁVEIS

Neste momento já sabemos que tudo o que inserirmos dentro da função draw() vai ser executado vezes sem conta, mas se os comandos que lá inserirmos não contemplam alterações, então não nos apercebemos de qualquer diferença no que está a ser desenhado, dado que é sempre igual.

Para contrariar esse efeito vamos introduzir o conceito de variável. As variáveis são uma forma de armazenarmos determinados valores que depois podemos reutilizar e alterar.

A memória de um computador funciona como um armário com muitas gavetas, onde colocamos e retiramos informação quando necessário. Nesse *armário*, as *gavetas* podem ser maiores ou menores e estarem preparadas para armazenar tipos diferentes de informação (número, texto, etc.). O *armário* é flexível, e o número de *gavetas* é dinamicamente variável. Cada "gaveta" corresponde a uma variável de memória, que se rotula com um determinado nome e se define como sendo de um determinado tipo. Isso é feito para que possamos a todo o momento abrir essa *gaveta*, cujo conteúdo é expectável, pois ela foi devidamente preparada para receber a informação que queremos lá colocar ou consultar.

As variáveis são criadas, alteradas e destruídas dentro de um programa, conforme a necessidade. É importante gerir bem as variáveis de memória, pois elas ocupam recursos computacionais valiosos. É importante também só tentar colocar dados dentro de uma variável que sejam compatíveis com o tipo da mesma, logo uma variável numérica só pode receber valores numéricos, enquanto uma textual, apenas texto. A criação das variáveis de memória obedece sempre este formato em Processing:

<tipo> <nome>;

e em P5.js

var <nome>;

No Processing <tipo> identifica a natureza dos dados que a variável conseguirá conter (numéricos inteiros, ponto flutuante, texto, etc.), o <nome> é o rótulo de identificação, que convém espelhar a utilização que lhe vamos dar. Alguns exemplos em Processing:

```
int lado;
```

criação da variável simples "lado" do tipo numérico inteiro

```
float angulo;
```

Os mesmos exemplos em P5.js:

```
var lado;
```

criação da variável simples "lado" do tipo numérico inteiro

```
var angulo;
```

Por vezes criamos a nossa variável e atribuímos de imediato um valor inicial:

(Processing)

```
float angulo = 1.1;
```

(P5.js)

```
var angulo = 1.1;
```

Consideremos agora o nosso programa original:

```
void setup() {
  size(200,200);
  background(0);
  fill(255,0,0);
}

void draw() {
  rect(10,10,180,180);
}
```

E vamos então utilizar variáveis para o tornar mais interessante. Vamos criar uma variável chamada “lado”, que vai conter a medida do lado da nossa forma geométrica, e a cada iteração da função draw() vamos aumentar aquele valor.

```
int lado;

void setup() {
  size(200,200);
  background(0);
  fill(255,0,0);
  lado=10;
}

void draw() {
  lado=lado+1;
  rect(10,10,lado,lado);
}
```

Uma vez que usamos a variável lado quer na função setup(), quer na função draw(), temos que fazer a sua declaração fora das duas, caso contrário ela apenas seria visível dentro do bloco de chavetas de uma das duas funções. Em setup() atribuimos o valor 10, e em draw() aumentamos a sua medida em uma unidade, através da expressão lado=lado+1; que se pode ler assim:

o novo valor da variável lado é igual ao seu valor anterior mais um.

Quando executamos o programa vemos o nosso quadrado a crescer gradualmente até ultrapassar as medidas da janela de visualização, e a partir desse momento parece que o programa não está a fazer nada, pois deixamos de ver o crescimento continuado.

## BLOCOS DE CÓDIGO E CHAVETAS

O código está sempre organizado em blocos. Esses blocos estão sempre debaixo do domínio de alguma estrutura de controlo, que pode ser uma função, ou até mesmo estruturas de repetição ou de condição (veremos mais a frente). Os blocos de código são sempre delimitados por chavetas {}, com o de abertura { indicando o início, e o de fecho } indicando o fim.

Exemplo:

```
void setup() {
  size(200,200);
  background(0);
  fill(255,0,0);
  lado=10;
}
```

A função setup() consiste nos vários comandos compreendidos entre as duas chavetas.

Por outro lado, os parêntesis () podem ser usado para controlar a avaliação das expressões.

Por exemplo: lado = lado + 1 * 2;

não é o mesmo que: lado = (lado + 1) * 2;

No primeiro caso primeiro multiplica-se 1 * 2 (é um operador prioritário) e só depois é que se soma com lado. No segundo caso primeiro soma-se um a lado e depois o valor resultante é multiplicado por dois.

Há ainda situações em que iremos utilizar estruturas no nosso código que nos permitem tomar decisões em função de determinadas condições.

Recordemos o nosso programa em Processing:

```
int lado;

void setup() {
  size(200,200);
  background(0);
  fill(255,0,0);
  lado=10;
}

void draw() {
  lado=lado+1;
  rect(10,10,lado,lado);
}
```

Como vimos, a dada altura os quadrados desenhados ultrapassam o limite da área de visualização e deixamos de perceber diferenças. Vamos introduzir duas variações neste programa.

Para começar, vamos então impedir que o nosso quadrado passe dos limites da zona de visualização, testando o valor da variável lado:

```
if (lado > 180) lado=0;
```

Isto pode ser lido assim:

se lado for maior que 180, então lado passa a ser igual a zero.

A segunda é um comando de background() no ciclo draw(). Vamos fazer variar o tom de fundo de acordo com o valor da variável “lado”.

```
int lado=10;

void setup() {
  size(200,200);
  background(0);
  fill(255,0,0);
}

void draw() {
  background(lado);
  lado=lado+1;
  rect(10,10,lado,lado);
  if (lado > 180) lado=0;
}
```

Se apenas estipularmos um parâmetro na função background() – e tudo isto é também válido nas restantes funções que lidam com cor, como fill() – preenchimento e stroke() – contorno – ele é entendido como cor. Se escrevermos três parâmetros, eles são entendidos como os valores dos primários vermelho, verde e azul que permitem criar vários milhares de cores. Se introduzirmos dois ou quatro parâmetros, são situações idênticas às anteriores, mas em que os últimos valores são traduzidos como transparência. Os limites destes parâmetros são 0 e 255, sendo 0 o valor mínimo (totalmente transparente ou ausência de cor) e 255 o máximo (cor saturada ou sem transparência).

Vamos então adaptar o código anterior para o seguinte:

```
int lado=10;
int cor=0;

void setup() {
  size(200,200);
  background(0);
  noFill();
  rectMode(CENTER);
}
```

```
void draw() {
  lado=lado+1;
  if (lado > width) {
    lado=0;
    if (cor<155)cor=cor+15;
    else cor=0;
  }
  stroke(100+cor,cor*2,255-cor*2,50);
  translate(width/2,height/2);
  rotate(lado/200.0*PI);
  rect(0,0,lado,lado);
}
```

Introduzimos uma nova variável – cor – e em vez de utilizarmos o preenchimento do quadrado, agora vamos utilizar apenas o seu contorno. Para isso usamos a função noFill() em setup(), dizendo assim que nada deve ser preenchido com cor.

Também definimos que vamos desenhar os retângulos a partir do seu centro através de rectMode(CENTER), e assim os dois primeiros parâmetros de rect() passam a referir-se ao centro do retângulo, e não ao seu canto.

Surge depois o nosso teste do valor da variável lado, mas desta vez queremos fazer não apenas uma ação, mas várias:

```
if (lado > width) {
  lado=0;
  if (cor<155)cor=cor+15;
  else cor=0;
}
```

Para isso utilizamos as chavetas para delimitar tudo que que iremos fazer se a nossa condição for verdadeira. Neste caso, isso inclui uma outra verificação, desta vez para a variável cor.

Depois definimos a cor da linha de contorno através da função stroke():

```
stroke(100+cor,cor*2,255-cor*2,50);
```

Estamos a usar quatro parâmetros, o que significa que estamos a fazer variar os valores de vermelho, verde e azul através de expressões matemáticas, e o último parâmetro é a transparência, que está fixa em 50.

Finalmente usamos um comando que altera o centro do nosso espaço visual. Habitualmente o ponto central (0,0) é o canto superior esquerdo, mas por uma questão de conveniência podemos querer deslocá-lo para outro lado. Frequentemente esse outro lado é o centro do ecrã, e isso faz-se através do comando:

```
translate(width/2,height/2);
```

Neste caso, o centro é obtido calculando a metade da largura e da altura.

O comando seguinte é a razão de fazermos esta operação: vamos rodar o nosso espaço!

```
rotate(lado/200.0*PI);
```

Essa rotação é sempre feita em torno do ponto (0,0), daí termos feito a sua translação para o centro do ecrã, porque pretendemos dar o efeito do quadrado a rodar sobre o seu centro. O valor do ângulo de rotação é dado pela expressão:

```
lado/200.0*PI
```

Estamos a dividir o valor de lado (que é um número inteiro) por 200. Em programação, frequentemente as operações matemáticas respeitam o tipo dos seus argumentos, e se escrever lado/200, o resultado é o valor da divisão inteira, dado que tanto lado é um inteiro (foi assim definido) como 200 também o é. Mas na realidade nós queremos valores decimais, e por isso em vez de escrever 200, escrevemos 200.0, e assim o Processing já assume uma divisão com casas decimais. Seguidamente multiplicamos esse valor pela constante PI (já pré-definida em Processing) para obter o valor final do ângulo de rotação.

Ao fim de alguns ciclos/iterações de execução, vamos obter algo semelhante à figura seguinte:

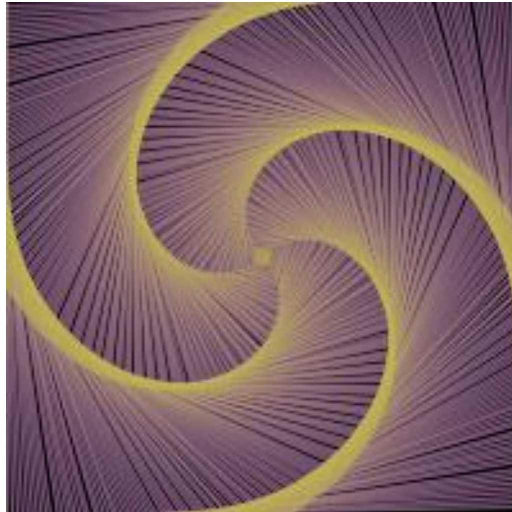


Figura 28 - Resultado da execução do código

## SINTAXE E ERROS

É importante estar atento a todos os detalhes de sintaxe que a programação exige. Um programa é uma espécie de carta que se escreve para o computador interpretar e executar. Se essa "carta" contiver erros gramaticais, o computador rejeita-a, gerando erros. São os chamados erros de sintaxe.

Esses erros são reportados no espaço da consola, tanto pelo Processing como pelo P5.js.

Algumas das regras básicas são:

- Todas as instruções são terminadas e separadas por ponto e vírgula (;).
- Os nomes de variáveis ou funções não podem conter espaços em branco.
- Ao definirmos variáveis ou funções, temos que seguir o formato de declaração exigido.
- O corpo de uma função está delimitado por { }.
- A interpretação de maiúsculas e minúsculas é diferenciada. Escrever size é diferente de Size e de sIzE.
- Temos que evitar a utilização de palavras reservadas (pré-definidas) na atribuição de nomes de variáveis ou funções. As palavras reservadas são aquelas que são pré-definidas no Processing e P5.js. Por exemplo, **setup** está pré-definido (só se utilizarmos Setup ou SetUp, mas mesmo assim não é recomendável porque prejudica a leitura por humanos e pode dar origem a más interpretações). A mesma coisa se aplica a estruturas que sejam oferecidas pela linguagem, como por exemplo**: if, else, while, for, true, false,** etc. São palavras que não podemos utilizar para definir algo no nosso programa, pois elas já estão reservadas para a própria linguagem.

## EXECUÇÃO E ERROS

Antes da execução o programa tem que ser traduzido para linguagem máquina, ou seja, código binário. Essa fase é definida como compilação. Na compilação, os erros de sintaxe são detetados e indicados pelo Processing ou P5.js. O código binário, ou seja, o executável, não será gerado até que todos esses erros tenham sido corrigidos pelo programador.

Uma vez o executável gerado, o programa pode ser executado, porém ainda pode haver erros de lógica, por exemplo, alterando o valor de uma variável de tal forma que ela irá provocar uma divisão por zero. Nesse caso, o programa "rebenta" ou sofre um "crash" durante a sua execução, não iniciando e terminando de forma correta e estável, ou produz resultados inesperados. Por exemplo, criar repetições em trechos de código, com o auxílio de estruturas de controlo mal geridas, podem causar *loops* infinitos em que nada parece acontecer.

Outros erros de execução prendem-se com o programa não fazer aquilo que desejamos que faça. Ele até pode executar sem erros, mas o resultado final não é o esperado, e isso apenas se deve a um erro humano de codificação do algoritmo.

Para executar o *debug* desses erros (que são os mais complicados de encontrar e corrigir), utilizamos muitas vezes a consola (área 1). Com o auxílio de funções como a print() e printl() somos capazes de imprimir nessa área o conteúdo de variáveis e rastrear a execução de funções, eventos e do programa como um todo.

Vamos inserir o comando println(lado, " ", cor) na função draw() do nosso programa. Estamos assim a dar a indicação de escrever o valor da variável lado, seguindo-se-lhe um espaço em branco e depois o valor da variável cor. Ao executar, atente-se no que acontece na zona da consola:

```
void draw() {
  lado=lado+1;
  if (lado > width) {
    lado=0;
    if (cor<155)cor=cor+15;
    else cor=0;
  }
  println(lado, " ", cor);
  stroke(100+cor,cor*2,255-cor*2,50);
  translate(width/2,height/2);
  rotate(lado/200.0*PI);
  rect(0,0,lado,lado);
}
```

**ATIVIDADE**

Modifique o código do exemplo em Processing de forma a poder executar com o mesmo efeito em P5.js. Depois partilhe a ligação no fórum.

Atente sobretudo às diferenças das funções e variáveis nas duas linguagens.

# DESENHO

Vamos ver como usar o Processing para desenhos básicos. Começamos a partir de linhas e retângulos. Na verdade, o Processing pode ser usado como uma ferramenta avançada de desenho digital, mas também de pós-processamento de desenho digital. Além disso, pode-se obter resultados interessantes ao combinar o desenho vetorial com lógica algorítmica. Para isso pode criar um desenho numa ferramenta como o Adobe Illustrator e exportá-lo no formato SVG, e posteriormente importá-lo diretamente no Processing.

## LINHA

Vamos desenhar com uma linha, criando um novo sketch e depois copiando o código do Exemplo 1. Depois disso, execute-o.

### EXEMPLO 1

```
1    void  setup() {
2           size(300, 300);
3           background(0);
4           smooth();
5      }
6
7
8    void draw() {
9           strokeWeight(30);
10          stroke(100);
11          line(100, 100, 200, 200) ;
12   }
```

Depois de executar o código acima, vai obter uma janela com fundo escuro e com uma linha diagonal, conforme demonstrado na Figura 26.

Para a definição de um segmento de reta num plano precisamos de conhecer as coordenadas das suas extremidades. Na verdade, no Processing, temos apenas linhas que são assumidas como segmentos de reta. Para definir as coordenadas das extremidades, podemos combinar coordenadas horizontais (X) e verticais (Y) para cada ponto (X,Y). O plano de desenho tem a coordenada 0,0 no canto superior esquerdo, e o valor de ambas as coordenadas aumenta à medida que nos afastamos para a direita e para baixo.

A linha 11 da Listagem 1 corresponde à definição dos pontos de extremidade: os valores (100, 100) definem as coordenadas X e Y, respetivamente. O valor (200, 200) define X e Y como coordenadas do segundo ponto do segmento de reta.

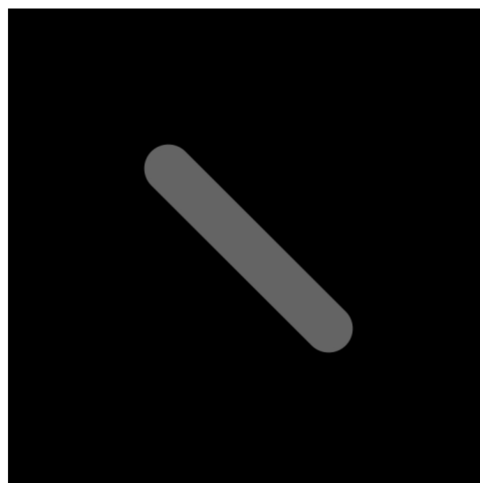


Figura 29 – Resultado da execução do código

**ATIVIDADE**

Modifique o código do Exemplo 1 de modo a desenhar um segmento de reta entre o ponto (50, 50) e o ponto (250, 250). Nota: basta alterar a linha 11. Experimente com outros valores.

**ATIVIDADE**

Modifique o código do Exemplo 1 de modo a obter um desenho como a da figura seguinte. Nota: basta acrescentar uma linha.

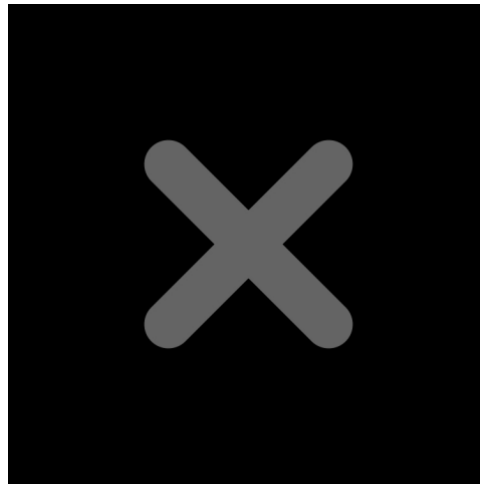


Figura 30

Vamos descrever agora em detalhe o exemplo 1.

Existem duas funções principais em Processing: setup () e draw ().

Setup () contém todos os comandos que definem o estado inicial do nosso sistema, e draw () contém os comandos que irão ser executados ciclicamente. Assim, apesar de parecer um desenho estático, a nossa linha (ou linhas) estão constantemente a ser redesenhadas.

```
2   size(300, 300);
```

Este comando define a dimensão do nosso plano de desenho, que vai desde o ponto (0,0) (canto superior esquerdo) até (300,300) (canto inferior direito).

```
3   background(0);
```

Este comando define a cor de fundo, que neste caso é 0 (ou seja, negro).

```
4   smooth();
```

Este comando determina que as linhas são desenhadas de forma suave, com *anti-aliasing*.

```
9   strokeWeight(30);
```

Este comando define a espessura da linha, sendo o valor 30 correspondente a uma linha grossa. Um valor mais pequeno corresponde a uma linha mais fina.

```
10  stroke(100);
```

Este comando define a cor da linha, numa escala de cinzentos (entre o negro, com o valor 0, e o branco, com o valor 255). O valor 100 corresponde a um cinzento médio.

Note ainda como as funções setup () e draw () são definidas, contendo o código respetivo entre chavetas.

A função setup () é executada primeiro e apenas uma vez pelo Processing. Após a execução do setup (), a função draw () foi executada automaticamente pelo Processing, e permanece em execução em *loop* contínuo, embora existam forma de limitar esse comportamento.

A designação de **função** é utilizada, por semelhança com a matemática, sempre que escrevemos algo como **nome ()** ou ainda **nome (valor1, valor 2)**. Assim, torna-se imediato perceber que **size (300, 300)** é também uma função, que tem um comportamento pré-definido (neste caso o de criar o plano de desenho, tal como **line (100, 100, 200, 200)**.

Os valores que aparecem dentro dos parênteses e separados por vírgulas designam-se por "parâmetros" da função. Assim, podemos ter funções com zero argumentos (setup(), draw(), entre outras), com um argumento (background(0), stroke(100), entre outras), com dois ou mais argumentos (line utiliza 4, por exemplo).

#### ATIVIDADE

Experimente alterar os valores dos parâmetros das várias funções utilizadas no Exemplo 1. Note que se utilizar um valor para o fundo (background) que seja exatamente igual ao de stroke, não vai poder observar nada, porque estará a desenhar com a mesma cor sobre o fundo.

## RETÂNGULO

Para definirmos um retângulo basta-nos considerar um ponto de origem (x,y) e uma medida de largura e comprimento a partir desse ponto. Assim, o Processing tem uma função para desenhar retângulos com base naqueles quatro valores: a função rect (x, y, l, c) utiliza os parâmetros origem (x e y) seguindo-se largura (l, medido na horizontal) e comprimento (c, na vertical).

#### EXEMPLO 2

```
1    void setup(){
2           size(400, 400);
3           smooth();
4           noLoop();
5           background(10);
6           strokeWeight(10);
7           stroke(150);
8    }
9
10   void draw() {
11          fill(250);
12          rect(100, 100, 100, 100);
13          fill(50);
14          rect(200, 200, 50, 100);
15   }
```

O resultado da execução do código do exemplo 2 pode ser visto na figura seguinte, e consiste em dois retângulos que se tocam num dos vértices e estão preenchidos com cores distintas.

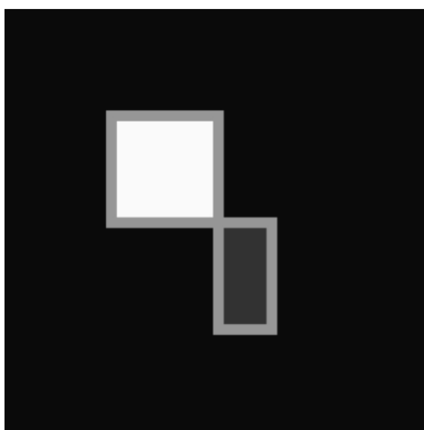


Figura 31

Atente que na função setup () usamos agora outra função nova: noLoop(). Esta função assegura que a função draw () apenas é executada uma vez, e que não fica a ser executada em loop continuamente.

Nas linhas 11 e 13, a função fill(250) e fill(50) é a responsável por aquela diferença. Enquanto a função stroke atua sobre as linhas, a função fill atua sobre o preenchimento do interior das formas. Tudo o que for desenhado após a última instrução (de stroke() ou fill()) vai ter as cores que nelas forem determinadas e, assim, se queremos obter diferenciação de cores, temos de usar aquelas funções tantas vezes quantas as cores diferentes que desejamos, colocando-as imediatamente antes das instruções de desenho (neste caso do Exemplo 2, o desenho dos retângulos).

**ATIVIDADE**

Mude o código do exemplo de forma que o retângulo claro fique debaixo do retângulo escuro. Mude ainda a cor da linha exterior de um dos retângulos.

**ATIVIDADE**

Consulte a referência online do Processing (em https://processing.org/reference/ ) e estude a função rectMode() e os diferentes modos de desenho de retângulos que permite. Altere o código do Exemplo 2 por forma a desenhar dois quadrados lado a lado, dando como parâmetros **o centro** de cada quadrado (à sua escolha), com os lados iguais a 100.

## ELIPSE, CÍRCULO E PONTO

Considere agora o Exemplo 3. Entre as linhas1 e 9 definimos a função setup(), como anteriormente. Utilizamos a função fill(), desta vez com dois parâmetros, em que o segundo parâmetro se refere à transparência da cor escolhida.

**EXEMPLO 3**

```
1   void  setup() {
2    size(500 , 500);
3    smooth();
4    background(255);
5    noLoop();
6    fill(50, 80);
7    stroke(100);
8    strokeWeight(3);
9   }
10
11  void  draw () {
12   ellipse (250, 250-50, 100, 100);
13   ellipse (250-50, 250, 100, 100);
14   ellipse (250+50, 250, 100, 100);
```

```
15   ellipse (250, 250+50, 100, 100);
16  }
```

Definimos a função draw() entre as linhas 11 e 16, utilizando para o efeito uma novidade: a função ellipse. Tal como com o retângulo, esta função aceita quatro parâmetros, sendo os dois primeiros o centro da elipse e os dois últimos o diâmetro horizontal e vertical da elipse. Torna-se evidente que, se os dois valores forem idênticos, em vez de elipses desenharemos círculos, como no caso presente. Tal como no caso do retângulo, existe ainda uma função (ellipseMode) que permite alterar o comportamento dos dois últimos parâmetros, e sugerimos a sua consulta e experimentação.

Na figura seguinte pode ver o resultado da execução do código. Observe como entre as linhas 12 e 15 posicionámos os centros dos círculos, somando e subtraindo valores diretamente nos parâmetros de posicionamento x e y, por forma a tornar-se mais evidente (50 pontos para a esquerda equivale a subtrair 50, 50 pontos para a direita equivale a somar 50, 50 pontos para cima equivale a subtrair 50 e 50 pontos para baixo equivale a somar 50).

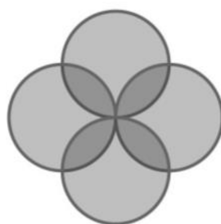


Figura 32

Procure na documentação online de referência saber mais sobre outros operadores (como, por exemplo: - * / % -- ou ++).

Vamos agora introduzir o conceito de variável, como forma de melhorar a legibilidade de um programa.

Considere as seguintes alterações ao Exemplo 3, que resultam no Exemplo 4:

**EXEMPLO 4**

```
1   int  largura = 500;
2   int  altura = 500;
3   int  diametro = 100;
4
5   void setup() {
6    size(largura, altura);
7    smooth();
8    background(255);
9    fill(50, 80);
10   stroke(100);
11   strokeWeight(3);
12   noLoop();
13  }
14
15
16  void draw() {
17   ellipse(largura/2, altura/2-diametro/2, diametro, diametro);
18   ellipse(largura/2-diametro/2, altura/2, diametro, diametro);
19   ellipse(largura/2+diametro/2, altura/2, diametro, diametro);
20   ellipse(largura/2, altura/2+diametro/2, diametro, diametro);
21  }
```

A primeira diferença é a existência das linhas antes da função setup(). Nessas linhas estamos a declarar que iremos utilizar as variáveis **largura**, **altura** e **diametro**, todas elas do tipo **int** (números inteiros), e estamos desde logo a atribuir-lhes valores iniciais. Os nomes que escolhemos ajudam à compreensão do seu papel no nosso programa, e assim convém que sejam ilustrativos da forma como irão ser utilizadas ou do que representam os seus valores.

As variáveis são uma espécie de contentores, cujo conteúdo pode ser alterado no decurso do próprio programa (ou não). Nem todos os nomes são permitidos para as variáveis (por exemplo, as palavras que constituem as próprias definições do Processing, nomes de funções, etc.).

Cada variável tem um tipo. No nosso exemplo todas são do tipo inteiro (**int**), mas existem outros tipos numéricos, como **float**, por exemplo, para o caso de termos de usar valores com casas decimais (ex: 0.5, 3.14159, etc.).

Existem ainda tipos distintos, como lógico/booleano (**boolean**) cujos valores podem apenas ser verdadeiro ou falso (**true**, **false**) e **char**, para armazenar caracteres tipográficos, como “a”, “n” ou “-”. Para descobrir todos os tipos de dados possíveis em Processing, por favor consulte a referência online do Processing, na secção Data.

Pelo facto de termos definido as variáveis **largura**, **altura** e **diametro** fora das funções setup() e draw() logo no início do programa, nós podemos usá-las dentro das duas funções. Mas se tivéssemos colocado a sua definição dentro de uma das funções (de setup(), por exemplo), já não poderíamos tentar usá-las na função draw(), dado que elas não existiriam foram da função ou âmbito em que são definidas. Assim, todas as variáveis que definirmos dentro de setup(), apenas poderão ser usadas dentro dessa mesma função, e o mesmo é válido para draw(), ou qualquer outra função que venhamos a definir no futuro.

**ATIVIDADE**

Altere o valor da variável **diametro** na linha 3 para 200, execute o código e veja como todos os círculos são afetados pela alteração.

# TRANSLAÇÃO, ROTAÇÃO E ESCALA

Há situações em que manter a origem do nosso espaço de desenho no canto superior esquerdo não é conveniente, sobretudo quando pretendemos explorar uma forma geométrica regular, centrada no espaço.

Para isso podemos utilizar a função **translate(x,y)** em que os parâmetros x e y indicam a deslocação que queremos impor ao ponto (0,0). Assim, sabendo que o Processing disponibiliza duas variáveis de sistema chamadas **width** e **height**, que em qualquer momento devolvem o valor da largura e altura do nosso espaço de desenho, podemos assumir que **translate(with/2, height/2)** desloca o ponto (0,0) rigorosamente para o centro da nossa janela.

É isso que iremos fazer no exemplo que se segue.

Adicionalmente vamos mostrar que alterações sucessivas de translações e rotações são acumuladas, isto é, somam-se às anteriores.

Vamos também usar a função **rotate(ang)**, em que **ang** é o valor do ângulo pelo qual queremos rodar o nosso espaço de desenho, em radianos. Também aqui o Processing disponibiliza a variável PI, exatamente com o valor matemática de PI (3.14159…). Assim, uma rotação de 2 x PI radianos corresponde a uma volta completa.

**EXEMPLO 5**

```
1   int cor = 255;
2
3   void setup()  {
4     size(600, 600) ;
5     noLoop();
6     smooth();
7     background(100);
```

```
8      strokeWeight(50);
9    }
10
11
12   void draw() {
13
14    translate(width/2, height/2);
15    stroke(cor);
16    line(0, 0, 250, 0);
17
18    cor=cor-30;
19    rotate(PI/4);
20    stroke(cor);
21    line(0, 0, 250, 0);
22
23    cor=cor-30;
24    rotate(PI/4);
25    stroke(cor);
26    line(0, 0, 250, 0);
27
28    cor=cor-30;
29    rotate(PI/4);
30    stroke(cor);
31    line(0, 0, 250, 0);
32
33    cor=cor-30;
34    rotate(PI/4);
35    stroke(cor);
36    line(0, 0, 250, 0);
37
38    cor=cor-30;
39    rotate(PI/4);
40    stroke(cor);
41    line(0, 0, 250, 0);
42
43    cor=cor-30;
44    rotate(PI/4);
45    stroke(cor);
46    line(0, 0, 250, 0);
47
48    cor=cor-30;
49    rotate(PI/4);
50    stroke(cor);
51    line(0, 0, 250, 0);
52   }
```

Atente ainda na seguinte expressão, que encontra nas linhas 18, 23, 28 etc.:

```
18    cor=cor-30;
```

Isto significa: veja qual o valor anterior da variável cor, subtraia 30 e guarde o novo valor atualizado na mesma variável cor. Na prática poderíamos dizer “vamos subtrair 30 a cor”, e existe uma forma simplificada de escrever exatamente isso:

```
cor-=30;
```

Esta versão parece um pouco estranha, mas significa exatamente o mesmo que a que utilizámos no código do exemplo. O Processing tem ainda alguns atalhos que podem ser úteis, como por exemplo:

```
cor++;
```

Aqui estamos a aumentar o valor da variável cor em uma unidade. Conforme pode imaginar, se desejássemos diminui-lo em uma unidade, iríamos escrever:

```
cor--;
```

Olhe agora para a figura seguinte e verifique que é possível determinar, devido à sobreposição das linhas mais recentes sobre as iniciais, qual a ordem pela qual as linhas são desenhadas, da mais clara para a mais escura.

Figura 33

## ATIVIDADE

Altere o código por forma a modificar o sentido de rotação das linhas, isto é, no sentido anti-horário.

Veja agora a seguinte alteração ao código, no Exemplo 6.

## EXEMPLO 6

```
1   int cor = 255;
2   float escala = 1;
3
4   void setup()  {
5     size(600, 600) ;
6     noLoop();
7     background(100);
8     smooth();
9     strokeWeight(50);
10  }
11
12  void draw() {
13
14    translate(width/2, height/2);
15    stroke(cor);
16    line(0, 0, 100, 0);
17
18    cor=cor-30;
19    escala=escala+0.02;
20    rotate(PI/4);
21    scale(escala);
22    stroke(cor);
23    line(0, 0, 100, 0);
24
25    cor=cor-30;
26    escala=escala+0.02;
27    rotate(PI/4);
28    scale(escala);
29    stroke(cor);
30    line(0, 0, 100, 0);
31
32    cor=cor-30;
33    escala=escala+0.02;
34    rotate(PI/4);
35    scale(escala);
36    stroke(cor);
37    line(0, 0, 100, 0);
38
39    cor=cor-30;
```

```
40      escala=escala+0.02;
41      rotate(PI/4);
42      scale(escala);
43      stroke(cor);
44      line(0, 0, 100, 0);
45
46      cor=cor-30;
47      escala=escala+0.02;
48      rotate(PI/4);
49      scale(escala);
50      stroke(cor);
51      line(0, 0, 100, 0);
52
53      cor=cor-30;
54      escala=escala+0.02;
55      rotate(PI/4);
56      scale(escala);
57      stroke(cor);
58      line(0, 0, 100, 0);
59
60      cor=cor-30;
61      escala=escala+0.02;
62      rotate(PI/4);
63      scale(escala);
64      stroke(cor);
65      line(0, 0, 100, 0);
66    }
```

Introduzimos uma nova variável – escala – e alteramos o seu valor, relacionando-a com a função scale(s). O valor s, se for igual a 1, mantém toda a proporção do nosso desenho, mas se o valor s descer abaixo de 1, reduz essa mesma proporção, e de igual forma, se subir acima de 1, aumenta-a.

Veja o resultado da execução deste código na figura seguinte, e verifique que não só o tamanho das linhas foi afetado, como também a sua espessura.

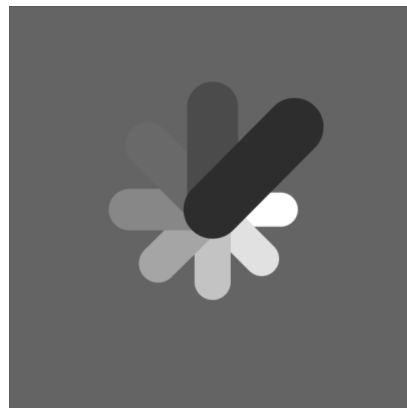


Figura 34

**ATIVIDADE**

Altere o código por forma a conseguir um efeito de escala inverso, isto é, de diminuição da escala.

## ESTADOS DO SISTEMA DE COORDENADAS

Já falámos das funções setup() e draw(), e agora vamos ver como podemos criar as nossas próprias funções. Conforme ficou explícito no exemplo anterior, por vezes repetimos blocos de código, e isso pode ser melhorado com o recurso a funções, que não são mais do que uma forma de encapsularmos esses mesmos blocos de código, dotando-os de parâmetros, caso sejam necessários.

No exemplo que se segue criamos uma função cujo objetivo é desenhar uma cruz (duas linhas cruzadas). Essa função recebe como parâmetro a cor.

Existem agora também duas novas funções do Processing: **pushMatrix()** e **popMatrix()**. Conforme tínhamos referido antes, todas as transformações vão-se acumulando, e por vezes torna-se complicado voltar ao início, ou desfazer alguma rotação ou mudança de escala. Podemos contornar essa situação chamando a função pushMatrix() antes de fazermos qualquer translação, rotação ou mudança de escala. Depois fazemos todas as alterações que queremos (rotações, translações, etc.) e no final chamamos a função popMatrix() e o sistema volta ao estado exato em que estava antes de termos chamado pushMatrix(), ou seja, desfaz automaticamente todas as transformações contidas entre pushMatrix() e popMatrix().

Para cada chamada a pushMatrix() deve obrigatoriamente existir uma chamada de popMatrix(), e não pode existir uma chamada a popMatrix() sem se ter feito previamente um pushMatrix().

**EXEMPLO 7**

```
1
2   void  setup() {
3     size (600, 600);
4     noLoop();
5     background(20);
6     smooth();
7     noStroke();
8   }
9
10  void  cruz(float cor){
11    fill(cor);
12    rect(0, 50, 150, 50);
13    rect(50, 0,  50, 150);
14  }
15
16  void draw() {
17
18    pushMatrix();
19    translate(100, 0);
20    rotate( PI /4) ;
21    cruz(180);
22    popMatrix();
23
24    pushMatrix();
25    translate(220, 110);
26    rotate(PI/6);
27    scale(2);
28    cruz(220) ;
29    popMatrix();
30
31    pushMatrix();
32    translate(520, 350);
33    rotate(PI/3);
34    scale(1.4);
35    cruz(80);
36    popMatrix();
37
38  }
```

Figura 35

A figura anterior mostra o resultado de se chamar a função cruz() com as várias transformações.

**ATIVIDADE**

Altere o código acima por forma a contemplar a função cruz() definida da seguinte forma:

```
void cruz(float cor, int x, int y, float angulo, float escala){
  pushMatrix();
  translate(x,y);
  rotate(angulo);
  scale(escala);
  fill( cor);
  rect(0, 50, 150, 50);
  rect(50, 0, 50, 150);
  popMatrix()
}
```

E a função draw() da seguinte forma:

```
void draw() {
  cruz(180, 100, 0, PI/4, 1);
  cruz(220, 220, 110, PI/6, 2) ;
  cruz(80, 520, 350, PI/3, 1.4);
}
```

# COR

Vamos agora revisitar as funções do Processing que permitem trabalhar com cor. Até agora usávamos apenas um parâmetro e o resultado era um tom de cinzento, variando desde o preto (0) até ao branco (255).

Mas podemos usar as componentes vermelha, verde e azul para especificar qualquer cor. Nesse caso, basta-nos usar 3 parâmetros e, automaticamente, o Processing assume que estamos a especificar as componentes pela ordem indicada. Veja o seguinte exemplo:

## EXEMPLO 1

```
1  void setup()  {
2    size(500, 500);
3    smooth();
4    noLoop();
5    background(100);
6  }
7
8  void  draw ()  {
9    stroke(150, 50, 250);
10   strokeWeight(110);
11   line(100, 150, 400, 150);
12
13   stroke(200, 100, 255);
14   strokeWeight(60);
15   line(100, 250, 400, 250);
16
17   stroke(255, 20, 20);
18   strokeWeight(110);
19   line(100, 350, 400, 350);
20 }
```

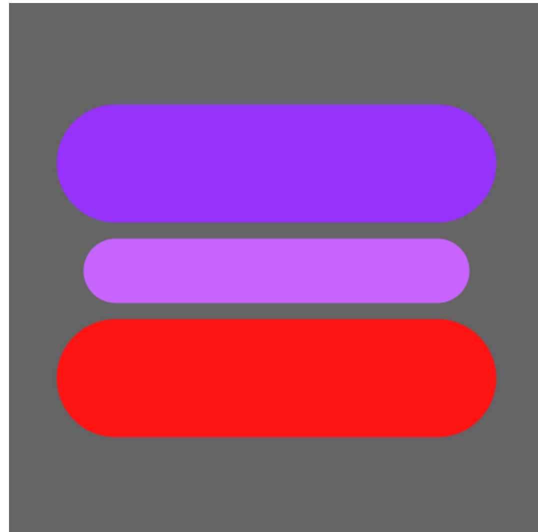


Figura 36

## ATIVIDADE

Tente agora variar os valores das componentes vermelha, verde e azul no exemplo dado por forma a obter um resultado semelhante ao da figura seguinte.

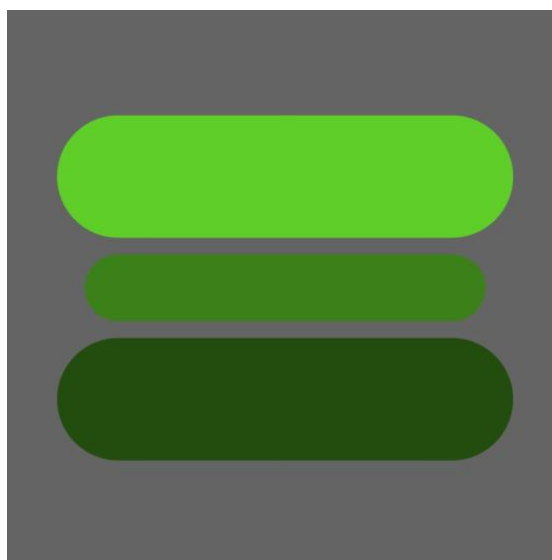


Figura 37

Conforme vimos anteriormente, se usarmos apenas dois parâmetros em **stroke()** ou **fill()**, o segundo valor é interpretado como uma transparência (255 é sinónimo de opaco e 0 é totalmente transparente, portanto invisível). O mesmo sucede com as cores, e neste caso iremos ter mais um parâmetro, num total de quatro, sendo a sua ordem vermelho, verde, azul e transparência).

**EXEMPLO 2**

```
1  void setup()  {
2    size(500, 500);
3    smooth();
4    noLoop();
5    background(0);
6  }
7
8  void  draw ()  {
9    stroke(95, 200, 40, 100);
10   strokeWeight(200);
11   line(150, 150, 350, 150);
12
13   stroke(60, 130, 25, 100);
14   line(150, 250, 350, 250);
15
16   stroke(35, 80, 15, 100);
17   line(150, 350, 350, 350);
18 }
```

Neste exemplo temos três linhas que se sobrepõem, e nas zonas de sobreposição é possível observar o efeito de transparência – ver figura seguinte.

Figura 38

**ATIVIDADE**

Altere o código fornecido e experimente com valores diferentes de transparência, cor, e com funções de desenho diferentes (retângulos, círculos, elipses).

# CICLOS

Se quisermos desenhar vários objetos idênticos, até agora a única maneira de o fazer era repetir as várias instruções, alterando os seus parâmetros. Isto torna-se viável para um reduzido número de repetições, mas muito cansativo se forem várias centenas.

Um dos mecanismos que o Processing faculta são os ciclos "for". Estes ciclos definem-se com recurso a uma variável, que frequentemente é criada dentro do próprio ciclo (dado não ter interesse fora dele) e é do tipo inteiro – um contador. De seguida estabelecemos um valor inicial para essa variável, um valor final, e um incremento (ou decremento) por forma a podermos passar do valor inicial ao valor final num determinado número finito de passos.

Por exemplo:

for ( int i = 0; i < 3; i++ ) { *xxxxxxxx* }

significa: vamos fazer 3 vezes o comando que vier a seguir a esta instrução (todos os comandos que estiverem entre duas chavetas, tal como nas funções, e que aqui exemplificamos com *xxxxx*).

Como verificamos que é mesmo três vezes? Começamos com a variável i com o valor 0 (uma vez), que é menor do que 3, e por isso vai ser aumentado em uma unidade e executamos o que quer que venha depois (*xxxxx*).

A seguir i já tem o valor 1 (segunda execução). Ainda é menor do que 3, por isso aumentamos o seu valor em uma unidade e fazemos a mesma instrução.

Agora i já vale 2, ainda é menor do que 3, por isso aumentamos o seu valor em uma unidade e fazemos a mesma instrução pela terceira vez.

Agora i é igual 3, e 3 não é menor do que 3, por isso o nosso ciclo interrompe-se aqui.

Vamos usar agora um código parecido com o do exemplo 9, mas onde substituímos as instruções sequenciais para cada uma das linhas por um ciclo for.

**EXEMPLO 3**

```
1    void setup() {
2     size(500, 500);
3     smooth();
4     noLoop();
5     background(0);
6     strokeWeight(200);
7   }
8
9   void draw() {
10    for (int i=0; i<3; i++) {
11      stroke(95-i*30, 200-i*30, 40-i*15, 100);
12      line(150, 150+i*100, 350, 150+i*100);
13    }
14  }
```

A função draw() ficou muito mais curta, e podemos observar que entre chavetas temos duas instruções:

```
11      stroke(95-i*30, 200-i*30, 40-i*15, 100);
12      line(150, 150+i*100, 350, 150+i*100);
```

Já vimos que a variável i vai assumir sequencialmente os valores 0, 1 e 2, e, portanto, o que estamos a fazer é, no caso da linha 12, por exemplo, 150+i*100 significa que a cada novo valor de i vamos ter um valor de 150 (com i=0), depois de 250 (com i=1) e finalmente de 350 (com i=2).

Veja agora este exemplo e o seu resultado na figura seguinte:

**EXEMPLO 4**

```
1  void setup() {
2    size(500, 500) ;
3    smooth();
4    background(255);
5    strokeWeight(30);
6    noLoop();
7    stroke(20) ;
8  }
9
10 void draw() {
11   for (int  i=1; i<8; i++ ){
12     line(i*50, 200, 150+(i-1)*50, 300);
13     line(i*50+100, 200, 50+(i-1)*50, 300);
14   }
15 }
```

Figura 39

**ATIVIDADE**

Modifique o código por forma a desenhar com transparência, conforme mostrado na figura seguinte.

Figura 40

**ATIVIDADE**

Modifique o código do ciclo **for** de maneira a aumentar a variável **I,** não em uma unidade a cada passo, mas em duas unidades, para obter um resultado semelhante ao da figura seguinte.

Figura 41

Veja agora o resultado de se inserir mais uma linha, com uma instrução stroke(), baseada na variável i, e o resultado na figura seguinte.

```
10 void draw() {
11   for ( int  i=1; i<8; i++ ){
12      stroke(20+20*i);
13      line (i*50, 200, 150+(i-1)*50, 300);
14      line (i*50+100, 200, 50+(i-1)*50, 300);
15    }
16  }
```

Figura 42

Conforme foi indicado, dentro das chavetas que seguem a um ciclo for podemos colocar um conjunto grande de instruções. E porque não outro ciclo for? Já percebemos que ao variar um determinado incremento podemos deslocar figuras horizontalmente, e de igual forma podemos fazê-lo verticalmente, cobrindo assim ambas as dimensões.

Considere o exemplo seguinte:

**EXEMPLO 5**

```
1  void setup() {
2    size(500, 500) ;
3    smooth();
4    background(255);
5    noStroke();
6    noLoop();
7  }
8
9  void draw() {
10   for (int i=0; i<10; i++ ) {
11     for (int j=0; j<10; j++) {
12       fill(i*20+j*10);
13       rect(i*40+50, j*40+50, 35, 35);
14     }
15   }
16 }
```

Pode ver o resultado na figura abaixo. O primeiro ciclo, com a variável i, é usado para incrementar a posição horizontal de desenho dos quadrados, e o segundo, com a variável j, para incrementar a posição vertical. Adicionalmente combinamos as variáveis i e j para fazer variar a cor ao longo do plano.

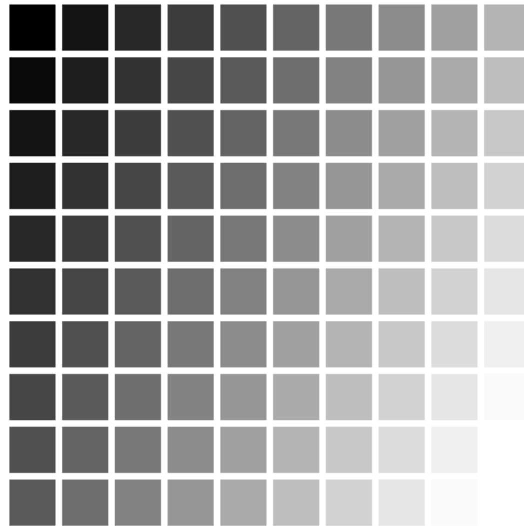


Figura 43

**ATIVIDADE**

Modifique o código do exemplo anterior por forma a obter um resultado semelhante ao da próxima figura.

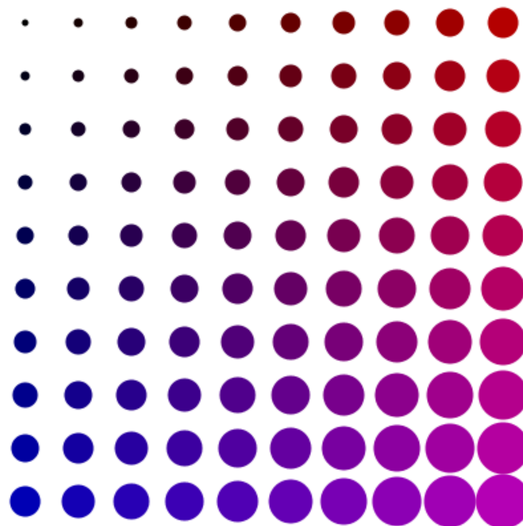


Figura 44

# INTERAÇÃO

Iremos começar com exemplos simples de interação com rato e teclado. Mais tarde poderão ser utilizados outros dispositivos, como uma webcam ou uma Kinect, por exemplo. O Processing dispõe de várias bibliotecas (*libraries*) adequadas a vários dispositivos.

## INTERAÇÃO BÁSICA

O Processing faculta-nos a utilização de duas variáveis pré-definidas que permitem saber em qualquer momento a posição do ponteiro do rato, em termos de coordenadas espaciais (x,y), e são elas **mouseX** e **mouseY**.

No exemplo abaixo veja como as coordenadas do rato são usadas para centrar o desenho, e como a cor é obtida através de uma variável que aumenta a cada ciclo da função draw(), mas à qual é aplicado o operador % (módulo), que permite obter o resto inteiro da divisão pelo valor que se segue, ou seja***:*** **cor%256** irá dar-nos sempre um valor inteiro entre 0 e 255.

Execute o programa e mova o rato para obter resultados semelhantes ao da figura abaixo.

### EXEMPLO 1

```
int cor=0;
float ang=0;

void setup() {
  size (500, 500);
  smooth();
  strokeWeight(1);
  background(0);
}

void draw() {
  cor++;
  ang+=0.01;
  stroke(cor%256, 20);
  pushMatrix();
  translate(mouseX, mouseY);
  rotate(ang);
  line(-50, -50, 50, 50);
  line(50, -50, -50, 50);
  popMatrix();
}
```

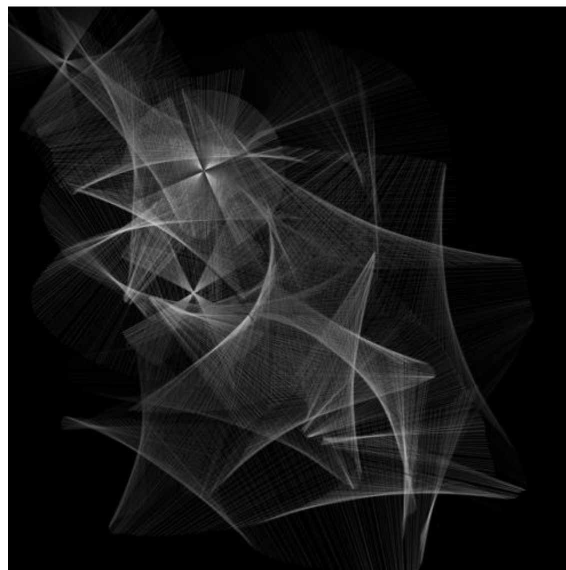


Figura 45

### ATIVIDADE

Altere o código e faça com que as linhas sejam desenhadas a vermelho.

No segundo exemplo usamos três variáveis para controlar a cor, e limitamos os valores que elas podem assumir através da função módulo (o resto da divisão inteira), identificada pelo símbolo % (ex: se a variável **red** tiver o valor 35, o resultado da operação **red % 256** é 35. Se o valor da variável for 260, o resultado da mesma operação é 4).

**EXEMPLO 2**

```
int red=0, green=0, blue=0;
float ang=0;

void setup() {
  size (500, 500);
  smooth();
  strokeWeight(1);
  background(0);
}

void draw() {
  red+=5;
  green+=2;
  blue+=3;
  ang+=0.01;
  stroke(red%256, green%256,
blue%256, 100);
  pushMatrix();
  translate(mouseX, mouseY);
  rotate(ang);
  line(-50, -50, 50, 50);
  line(50, -50, -50, 50);
  popMatrix();
}
```

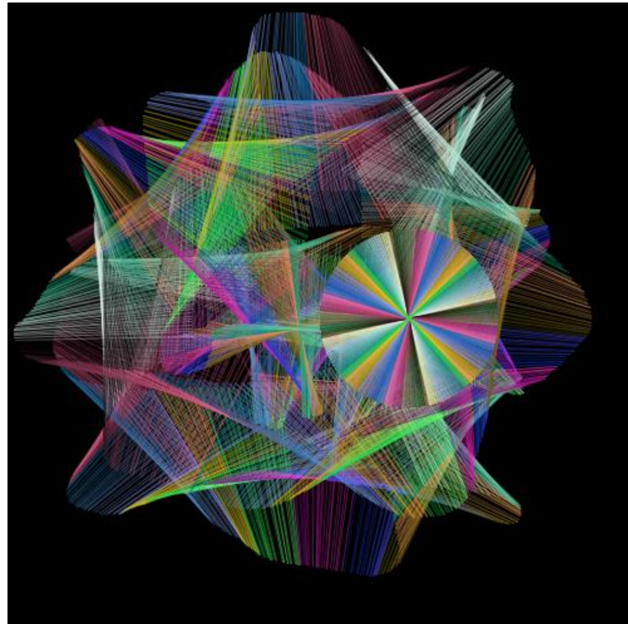


Figura 46

**ATIVIDADE**

Altere o código e faça com que as linhas sejam desenhadas a tons de cinzento.

# ANIMAÇÃO

Até agora a maior parte dos exemplos desenham de forma a acrescentar linhas a um ecrã, de forma cumulativa. Mas podemos limpar o nosso ecrã a cada iteração da função draw(), utilizando a instrução background() com a cor apropriada, como no exemplo abaixo, em que essa é a única diferença do anterior.

**EXEMPLO 3**

```
int red=0, green=0, blue=0;
float ang=0;

void setup() {
  size (500, 500);
  smooth();
  strokeWeight(10);
  background(0);
}

void draw() {
  background(0);
  red+=5;
  green+=2;
  blue+=3;
  ang+=0.01;
  stroke(red%255, green%255, blue%255, 100);
  pushMatrix();
  translate(mouseX, mouseY);
  rotate(ang);
  line(-50, -50, 50, 50);
  line(50, -50, -50, 50);
  popMatrix();
}
```

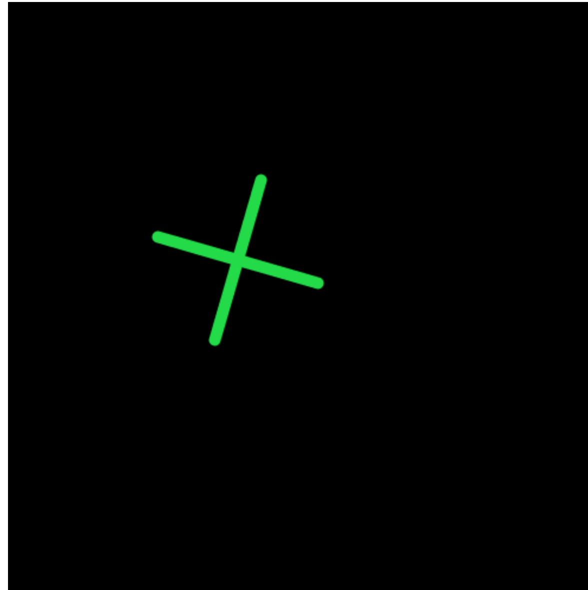


Figura 47

No exemplo que se segue utilizamos o código do exemplo anterior, mas introduzimos condições. As condições em Processing expressam o seguinte raciocínio:

se algo é verdade, então faz-se alguma coisa. Senão, faz-se outra coisa (ou nada).

Isto codifica-se assim:

**if ( red > 250 ) incR=-3;** (se a variável **red** tiver um valor superior a 250, então a variável **incR** passa a ter o valor -3).

A estrutura **if** admite apenas um comando a seguir, mas se precisarmos de colocar vários comandos dependentes da nossa condição, basta envolvê-los em chavetas, como nos ciclos **for** ou nas funções.

Neste exemplo, se a condição for verdadeira, então as duas instruções entre chavetas são executadas.

```
if (red > 250 ) {
  incR=-3;
  red = 0;
}
```

Neste exemplo seguinte, uma vez que não há chavetas, apenas a primeira instrução é condicionada. A segunda instrução (**red=0**) é executada sempre.

```
if (red > 250 )
  incR=-3;
  red = 0;
```

No exemplo seguinte, a instrução que aparece depois da palavra **else** é executada apenas se a condição do **if** for falsa.

```
if (red > 250 )
  incR=-3;
else
  red = 0;
```

**EXEMPLO 4**

```
int red=10, green=0, blue=100;
int incR=1, incG=1, incB=1;
float ang=0;

void setup() {
  size (500, 500);
  smooth();
  strokeWeight(10);
  background(0);

  incR=1;
  incG=1;
  incB=1;
}

void draw() {
  background(0);

  if(red>250) incR=-3;
  if(red<10)incR=1;
  red+=incR;

  if(green>250) incG=-1;
  if(green<10)incG=2;
  green+=incG;

  if(blue>250) incB=-2;
  if(blue<10)incB=3;
  blue+=incB;

  ang+=0.01;

  stroke(red, green, blue, 100);
  pushMatrix();
  translate(mouseX, mouseY);
  rotate(ang);
  line(-50, -50, 50, 50);
  line(50, -50, -50, 50);
  popMatrix();
}
```

**ATIVIDADE**

Altere o código experimentando com valores distintos para as variáveis incrementais incR, incG e incB.

No exemplo 5, acrescentamos uma nova função, chamada **keyPressed()**, que é automaticamente executada sempre que uma tecla for pressionada. A variável **key** contém o valor da tecla que tiver sido pressionada, e podemos testar esse valor e tomar ações específicas conforme a tecla.

Usamos ainda a função **saveFrame("nome.extensão")**, que grava o conteúdo gráfico que estiver a ser mostrado naquele preciso momento, num ficheiro cujo tipo é determinado pela extensão que escolhermos (por exemplo, jpg, png) e com o nome que dermos. No caso do exemplo, vai gravar uma imagem chamada oMeuProjeto.png. O nome completo deve vir entre aspas.

**EXEMPLO 5**

```
int red=10, green=0, blue=100;
int incR=1, incG=1, incB=1;
float ang=0;

void setup() {
  size (500, 500);
  smooth();
  strokeWeight(2);
  background(0);

  incR=1;
  incG=1;
  incB=1;
}

void draw() {

  if(red>250) incR=-3;
  if(red<10)incR=1;
  red+=incR;

  if(green>250) incG=-1;
  if(green<10)incG=2;
  green+=incG;

  if(blue>250) incB=-2;
  if(blue<10)incB=3;
  blue+=incB;

  ang+=0.01;

  stroke(red, green, blue, 100);
  pushMatrix();
  translate(mouseX, mouseY);
  rotate(ang);
  float linha=random(-100,100);
  line(-50-linha, -50-linha, 50+linha, 50+linha);
  line(50+linha, -50-linha, -50-linha, 50+linha);
  popMatrix();
}

void keyPressed() {
 if(key=='g') saveFrame("oMeuProjeto.png");
}
```

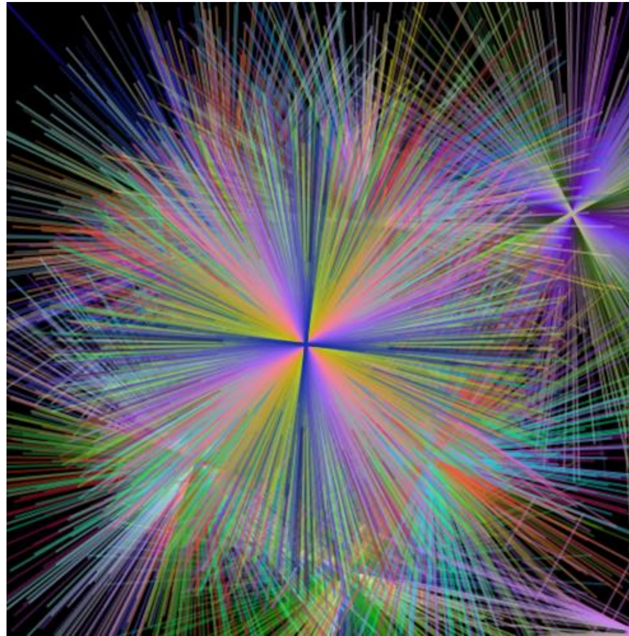


Figura 48

No exemplo seguinte mostra-se como encadear várias estruturas condicionais (vários **if**), e introduz-se também o tipo **boolean** (lógico), cujos únicos valores podem ser **true** ou **false**. Quando temos uma variável do tipo booleano e queremos obter o valor contrário ao que ela armazena (se for **true**, queremos **false**, e vice-versa) usamos o operador ! (ponto de exclamação, lê-se NOT). Assim, este comando **a = !b;** significa: na variável de tipo **boolean** chamada a vamos guardar o oposto (a negação lógica) do conteúdo da variável de tipo **boolean** chamada **b**. Se quisermos comparar duas variáveis para saber se o seu valor é idêntico usamos o operador **==**. Se quisermos compará-las para saber se o seu valor ´r diferente, usamos o operador **!= .**

**ATENÇÃO: if (a==b) a=10; else a=b;** significa: se o valor de **a** for igual a **b**, então **a** passa a ter o valor **10**, senão, **a** passa a ter o valor de **b**. Não confundir == (serve para comparar) com = (serve para atribuir um valor).

**EXEMPLO 6**

```
int red=10, green=0, blue=100;
int incR=1, incG=1, incB=1;
float ang=0;
boolean desenha=false;

void setup() {
  size (500, 500);
  smooth();
  strokeWeight(2);
  background(0);

  incR=1;
  incG=1;
  incB=1;
}

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

  if(red>250) incR=-3;
  else if(red<10)incR=1;
  red+=incR;

  if(green>250) incG=-1;
  else if(green<10)incG=2;
  green+=incG;
```

```
    if(blue>250) incB=-2;
    else if(blue<10)incB=3;
    blue+=incB;
    ang+=0.01;

    stroke(red, green, blue, 100);
    if(desenha) {
      pushMatrix();
      translate(mouseX, mouseY);
      rotate(ang);
      float linha=random(-100,100);
      line(-50-linha, -50-linha, 50+linha, 50+linha);
      line(50+linha, -50-linha, -50-linha, 50+linha);
      popMatrix();
    }
}

void keyPressed() {
 if(key=='g') saveFrame("oMeuProjeto.png");
 if(key=='d') desenha=!desenha;
}
```

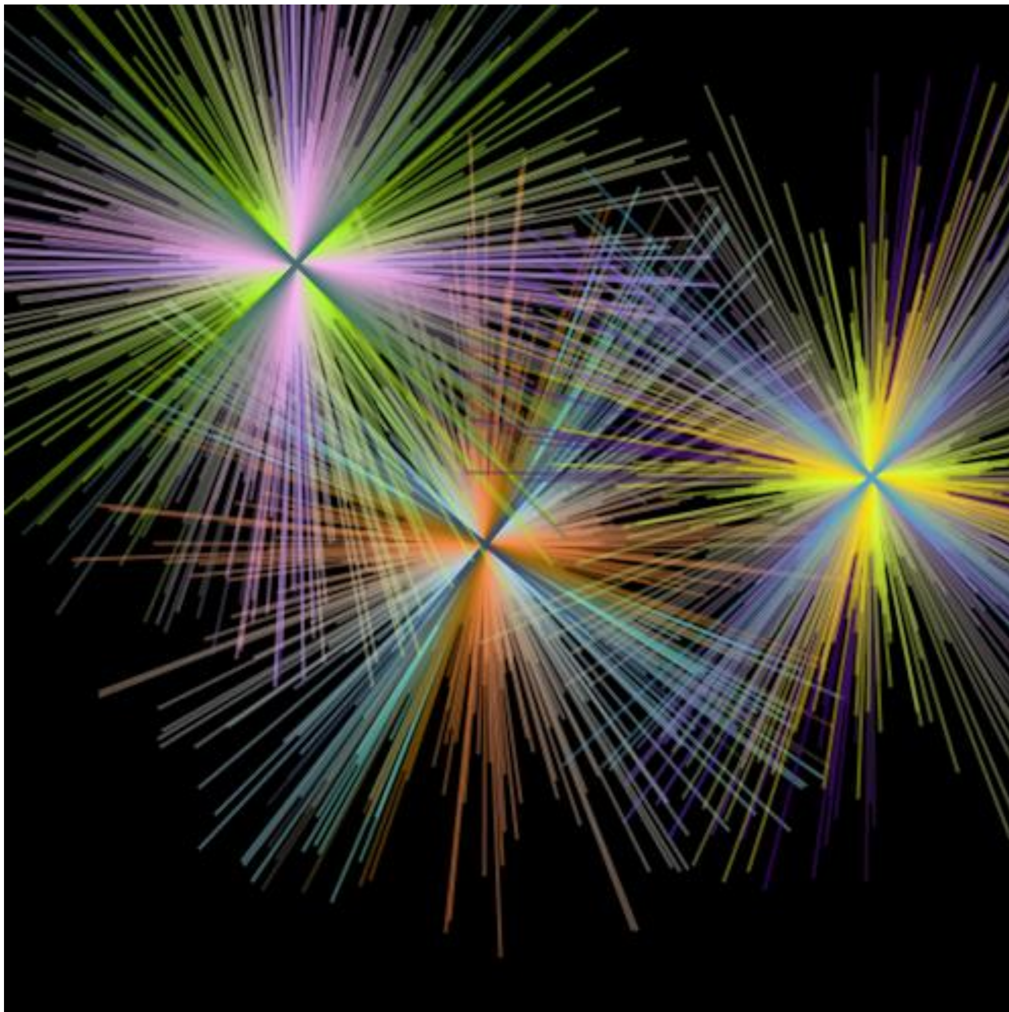


Figura 49

**ATIVIDADE**

Altere o código por forma a obter resultados semelhantes ao da imagem abaixo.

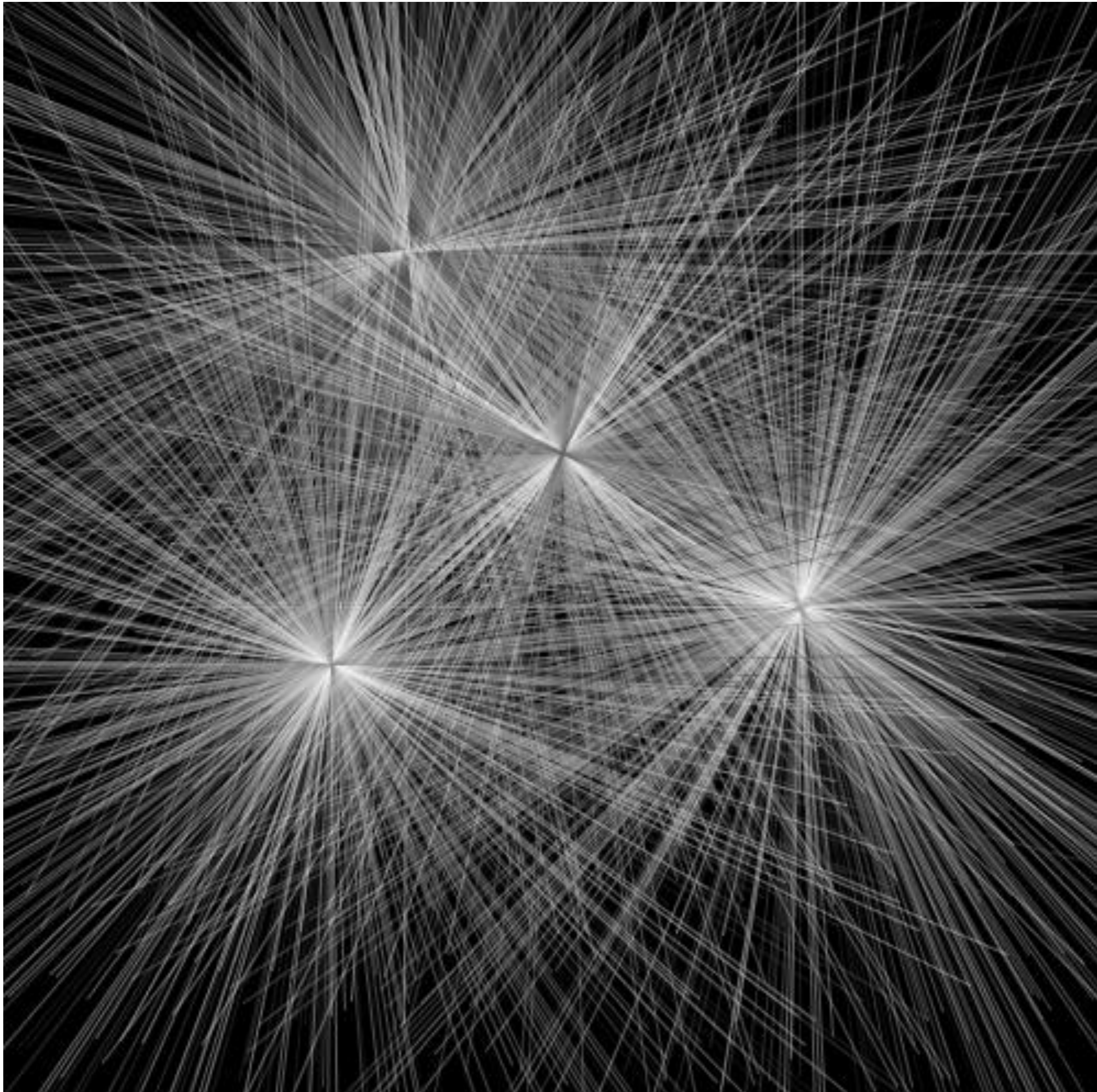


Figura 50

No exemplo 7 vemos mais uma variação de animação. Já usámos sketches em que desenhamos tudo no ecrã, nuns casos permitindo apenas uma execução única da função draw() (usando a função **noLoop()** dentro de **setup()**), noutros casos sem que nada seja removido a cada iteração da função **draw()**, e noutros ainda removendo tudo a cada iteração da função **draw()**. Agora vamos mostrar como podemos apagar subtilmente os desenhos anteriores, deixando algum rasto. Para isso desenhamos um retângulo com a dimensão do ecrã, usamos a mesma cor do fundo (de **background()**) mas aplicamos uma transparência nesse mesmo retângulo, para que ele não oculte tudo.

No exemplo abaixo, usamos a função **rectMode(CENTER);** e assim todos os nossos retângulos são desenhados a partir do centro. Por isso **rect(width/2,height/2,width+2,height+2);** usa as coordenadas do centro do ecrã (metade da largura e metade da altura) como ponto de origem, e ultrapassa a dimensão do ecrã em 1 pixel para cada lado**.**

**fill(0, 5);** assegura que usamos a cor preta de fundo, com apenas 5 de transparência (portanto, muito transparente, sendo que 255 será totalmente opaco).

## EXEMPLO 7

```
int col=100;
int incC=1;
boolean horiz=true;

void setup() {
  size (500, 500);
  smooth();
  noStroke();
  background(0);
  rectMode(CENTER);
  incC=1;
}

void draw() {
  fill(0, 5);
  rect(width/2,height/2,width+2,height+2);

  if(col>250) incC=-1;
  else if(col<100)incC=1;
  col+=incC;

  float linha=random(-width,width);
  fill(col, 30);

  if(horiz) {
    pushMatrix();
    translate(mouseX, mouseY);
    rect(0, 0, linha, 20);
    popMatrix();
  }
  else {
    pushMatrix();
    translate(mouseX, mouseY);
    rect(0, 0, 20, linha);
    popMatrix();
  }
}

void keyPressed() {
 if(key=='g') saveFrame("oMeuProjeto.png");
 if(key=='d') horiz=!horiz;
}
```

Figura 51

**ATIVIDADE**

Altere o código acima experimentando usar elipses, linhas e cores, bem como variando a transparência do retângulo que usamos para “apagar” os desenhos antigos.

Até agora usámos a posição do rato como forma de influenciar espacialmente os nossos desenhos, Mas podemos usar o rato também para controlar a cor desses mesmos desenhos. Basta-nos assumir que os extremos horizontais do ecrã correspondem a extremos de valores permitidos para as cores.

O Processing tem a função **map(variável, valorMin, valorMax, novoMin, novoMax)** que recebe como parâmetros uma variável (no nosso caso iremos usar **mouseX** e **mouseY**), os valores mínimo e máximo que essa variável pode ter (no nosso caso serão zero e with, ou zero e height), e os limites mínimo e máximo que queremos obter (no nosso caso, zero e 255, os limites dos valores permitidos para as cores). Assim, ao mexermos o rato, iremos influenciar as componentes vermelha e azul da cor.

**EXEMPLO 8**

```
float ang=0;

void setup() {
  size (500, 500);
  smooth();
  noStroke();
  background(0);
  ellipseMode(CENTER);
}

void draw() {

  fill(0,10);
  rect(0,0,width,height);

  translate(width/2, height/2);
  fill(map(mouseX,0,width,0,255),50,map(mouseY,0,height,0,255), 100);

  ang+=0.01;
  rotate(ang);

  pushMatrix();
  translate(random(width/2), 0);
  float linha=random(0,100);
  ellipse(0, 0, linha, linha);
  popMatrix();
}

void keyPressed() {
 if(key=='g') saveFrame("oMeuProjeto.png");
}
```

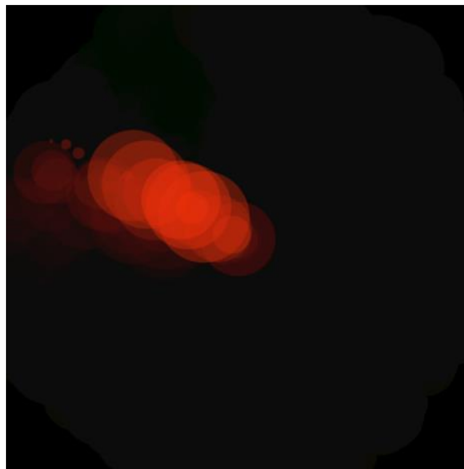


Figura 52

**ATIVIDADE**

Altere o código acima experimentando variar a transparência quer do retângulo que apaga o conteúdo do ecrã, quer do próprio desenho, para obter desenhos semelhantes aos abaixo.

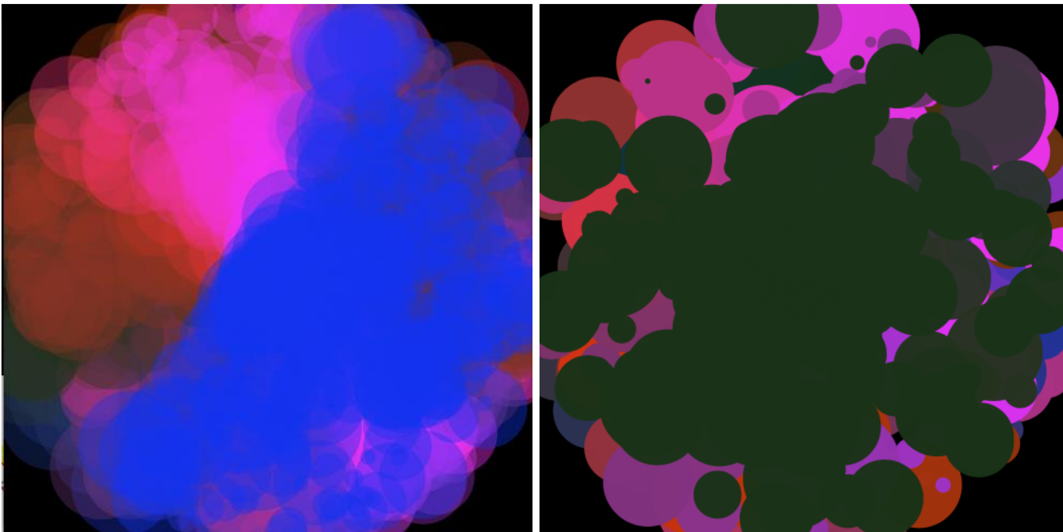


Figura 53

## UTILIZAÇÕES DE SIN() (SENO) E COS() (COSSENO)

As funções seno **sin()** e cosseno **cos()** são extraordinariamente úteis quando pretendemos criar movimentos circulares ou pulsantes. Veja no exemplo abaixo como conseguimos criar vários pontos (x,y) para desenhar vários círculos, ao longo de um traçado também ele circular.

### EXEMPLO 1

```
float cx=10, cy=10;
float raio=100;
float ang=0;

void setup() {
  size (500,500);
  smooth();
  background(50);
  noStroke();
}

void  draw ()  {
  fill(0,50);
  rect(0,0,width,height);

  translate(width/2, height/2);
  ang+=0.01;
  if(ang>2*PI) ang=0.01;
  rotate(ang);

  fill(250,50);
  for (float i=0; i<2*PI; i+=ang) {
    float x=cos(i)*raio;
    float y=sin(i)*raio;
    ellipse(x,y,raio,raio);
  }
}
```

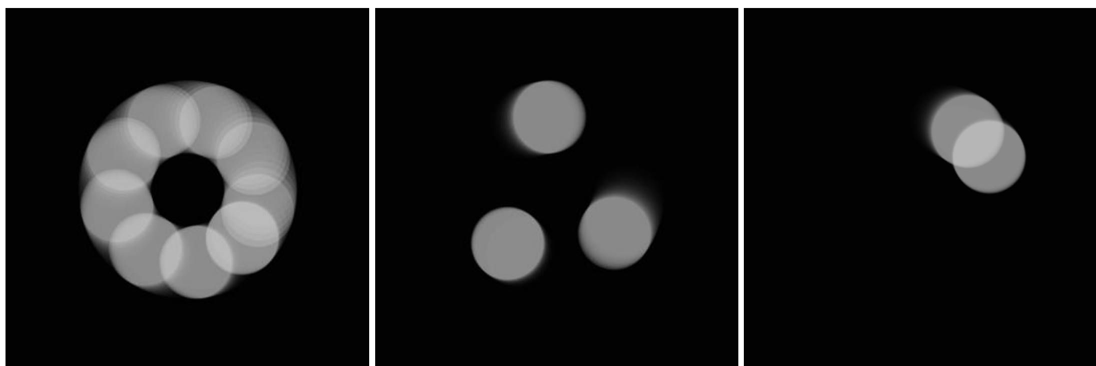


Figura 54

Podemos construir formas muito complexas a partir das funções trigonométricas, e o exemplo 10 ilustra essa possibilidade.

## EXEMPLO 2

```
float ang1, ang2;
float cx=250, cy=250;
float raio=200;

void  setup() {
  size(500,500) ;
  smooth();
  background(255);
  strokeWeight(1);
 }

void draw() {
  stroke(0,30);

  float nx=sin(ang2)*raio+cx;
  float ny=cos(ang2)*raio+cy;

  float x1=nx-sin(ang1)*200;
  float y1=ny-cos(ang1)*200;
  float x2=nx+sin(ang1)*200;
  float y2=ny+cos(ang1)*200;

  line(x1,y1,x2,y2);

  ang1+=PI/30;
  if(ang1>2*PI) ang1=0;

  ang2+=0.01;
  if(ang2>2*PI) ang2=0;
}
```

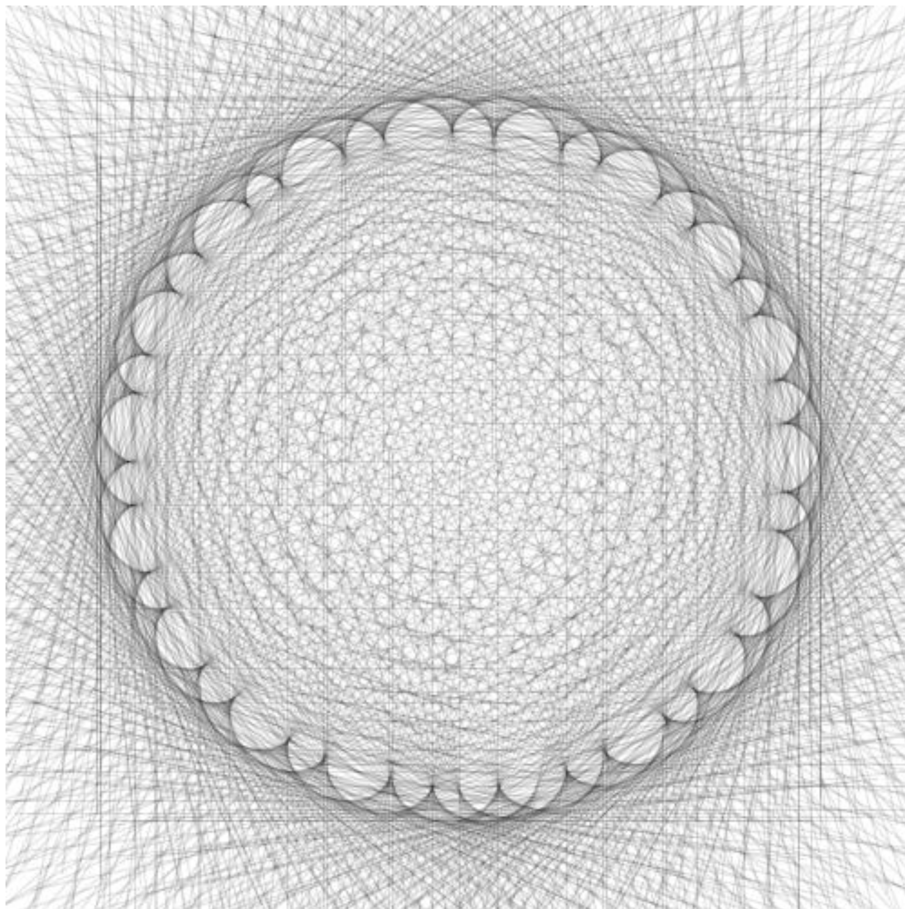


Figura 55

Mas a utilização das funções trigonométricas não está restrita ao cálculo de posições espaciais em torno de trajetórias circulares, elas também podem ser usadas para controlar cores, por exemplo.

**EXEMPLO 3**

```
float ang1, ang2;
float raio=300;
float x1, y1, x2, y2;

void setup() {
  size(1000,1000);
  smooth();
  background(100);
  strokeWeight(1);
}

void draw() {
  translate(width/2, height/2);
  float nx=sin(ang2)*raio/2;
  float ny=cos(ang2)*raio/2;

  stroke(abs(sin(ang2)+cos(ang2))*255,50);

  x1=nx-sin(ang1)*raio;
  y1=ny-cos(ang1)*raio;
  x2=nx+sin(ang1)*raio;
  y2=ny+cos(ang1)*raio;

  line(x1,y1,x2,y2);

  ang1+=0.01;
  if(ang1>2*PI) ang1=0;
  ang2+=0.05;
  if(ang2>2*PI) ang2=0;
}
```

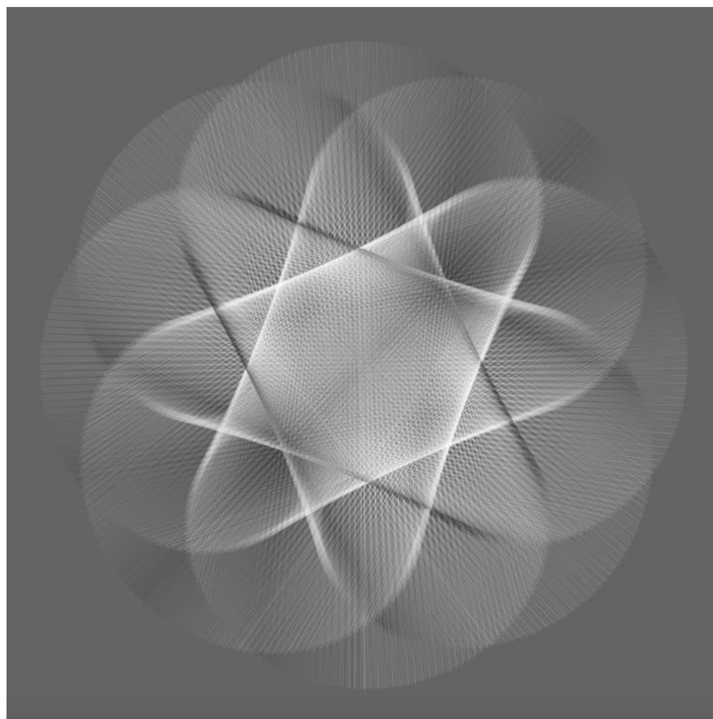


Figura 56

Podemos ainda utilizar uma lógica mais complexa, alterando cores e desenho em função de valores retornados por funções trigonométricas, para obter formas mais estimulantes, menos regulares, mais desafiadoras.

## EXEMPLO 4

```
float ang1, ang2;
float raio=300;
float x1, y1, x2, y2;

void setup() {
  size(1000,1000);
  smooth();
  background(100);
  strokeWeight(1);
}

void draw() {
  translate(width/2, height/2);

  float nx=sin(ang2)*raio/2;
  float ny=cos(ang2)*raio/2;

  stroke(abs(sin(ang2)+cos(ang2))*255,50);

  if(sin(ang1)+cos(ang2)<0) {
    x1=nx-sin(ang1)*raio;
    y1=ny-cos(ang1)*raio;
    x2=nx+sin(ang1)*raio;
    y2=ny+cos(ang1)*raio;
  } else {
    x1=nx-cos(ang1)*raio;
    y1=ny-sin(ang1)*raio;
    x2=nx+cos(ang1)*raio;
    y2=ny+sin(ang1)*raio;
  }

  line(x1,y1,x2,y2);

  ang1+=0.01;
  if(ang1>2*PI) ang1=0;

  ang2+=0.05;
  if(ang2>2*PI) ang2=0;
}
```

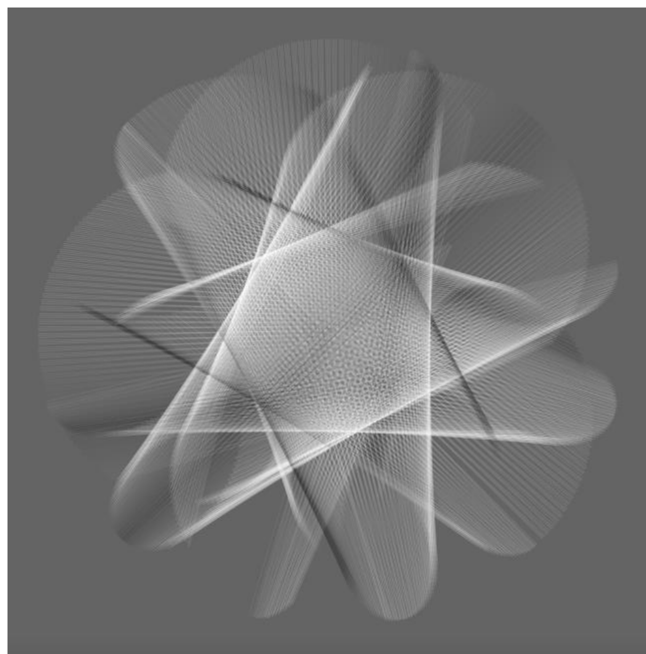


Figura 57

# CURVAS DE BEZIER

A curva de Bézier é uma curva expressa como a interpolação entre alguns pontos representativos, chamados de pontos de controle. É uma curva utilizada em diversas aplicações gráficas como o Illustrator, Freehand, Fireworks, GIMP, Photoshop, Processing, Inkscape e CorelDRAW, e formatos de imagem vetorial como o SVG. Esse tipo de curva também pode originar Superfícies de Bézier, bastante utilizadas em modelagem tridimensional, animações, design de produtos, engenharia, arquitetura entre outras aplicações.

O exemplo abaixo mostra a relação entre os pontos de controle e a curva obtida a partir dos mesmos através da função **bezier()** em Processing.

**EXEMPLO 1**

```
void setup() {
  size(660, 400);
  smooth();
  rectMode(CENTER);
  ellipseMode(CENTER);
}

void draw() {
  background(255);

  float anc1X = mouseX;
  float anc1Y = mouseY;
  float cont1X = width/2;
  float cont1Y = height/2-100;
  float cont2X = width/2;
  float cont2Y = height/2+100;
  float anc2X = width-mouseX;
  float anc2Y = height-mouseY;

  noFill();
  stroke(100);
  strokeWeight(5);
  bezier(anc1X, anc1Y, cont1X, cont1Y, cont2X, cont2Y, anc2X, anc2Y);

  stroke (50);
  strokeWeight(1);
  line (anc1X, anc1Y, cont1X, cont1Y);
  line (anc2X, anc2Y, cont2X, cont2Y);

  noStroke();

  fill(150);
  rect(cont1X, cont1Y, 6, 6);
  rect(cont2X, cont2Y, 6, 6);

  fill(170, 0, 0);
  ellipse(anc1X, anc1Y,10, 10);
  ellipse(anc2X, anc2Y,10, 10);
}
```

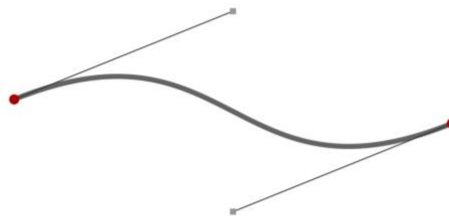
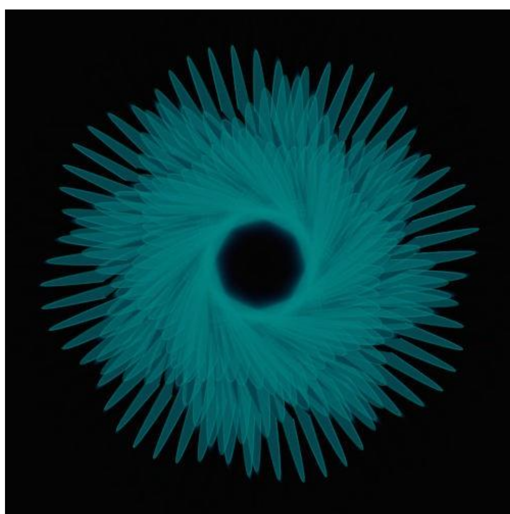


Figura 58

As utilizações gráficas da função bezier() são muitas e interessantes, e aqui ficam alguns exemplos obtidos a partir do site openprocessing.com, onde se encontram muitos mais exemplos.

**EXEMPLO 2** (https://www.openprocessing.org/sketch/159891)

```
float num=0, pathR, pathG, pathB;

void setup() {
  size (500, 500);
  background(0);
}

void  draw() {
  //limpar o ecrã mas com transparência
  fill(0, 10);
  rect(-1, -1, width+1, height+1);

  float maxX=map(mouseX, 0, width, -150, 150);
  float maxY=map(mouseY, 0, height, -150, 150);

  // posicionar as corrdenadas no centro
  translate (width/2, height/2);

  for(int i=0; i<360; i+=2) {
   float angle=sin(i+num);
   float x=sin(radians(i))*(maxX+angle*30);
   float y=cos(radians(i))*(maxX+angle*30);

   float x2=sin(radians(i+num*50))*(maxY+angle*60);
   float y2=cos(radians(i+num*50))*(maxY+angle*60);

   pathR=50+angle+125*sin(PI+num*3);
   pathG=50+angle+125*sin(TWO_PI+num*3);
   pathB=50+angle+125*sin(HALF_PI+num*3);

   stroke(pathR, pathG, pathB, 50);
   fill(pathR, pathG, pathB, 30);

   bezier(x, y, x+x, y+y, x+x2 , y+y2 , x+x2, y+y2);
  }
  num+=0.01;
}
```



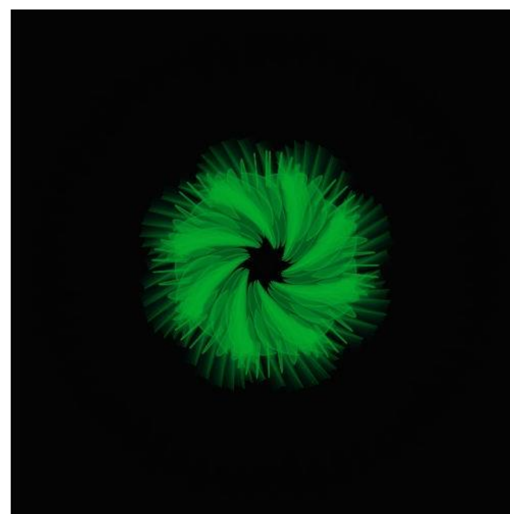


Figura 59

**EXEMPLO 3** (http://www.openprocessing.org/sketch/160305)

```
float amount = 20, num;

void setup() {
  size(640, 640);
  stroke(0, 150, 255, 100);
}

void draw() {
  fill(0, 40);
  rect(-1, -1, width+1, height+1);

  float maxX = map(mouseX, 0, width, 1, 250);

  translate(width/2, height/2);
  for (int i = 0; i < 360; i+=amount) {
    float x = sin(radians(i+num)) * maxX;
    float y = cos(radians(i+num)) * maxX;

    float x2 = sin(radians(i+amount-num)) * maxX;
    float y2 = cos(radians(i+amount-num)) * maxX;
    noFill();
    bezier(x, y, x-x2, y-y2, x2-x, y2-y, x2, y2);
    bezier(x, y, x+x2, y+y2, x2+x, y2+y, x2, y2);
    fill(0, 150, 255);
    ellipse(x, y, 5, 5);
    ellipse(x2, y2, 5, 5);
  }

  num += 0.5;
}
```

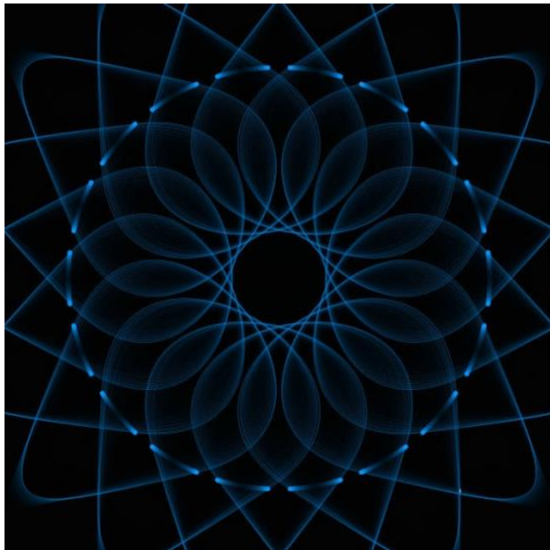
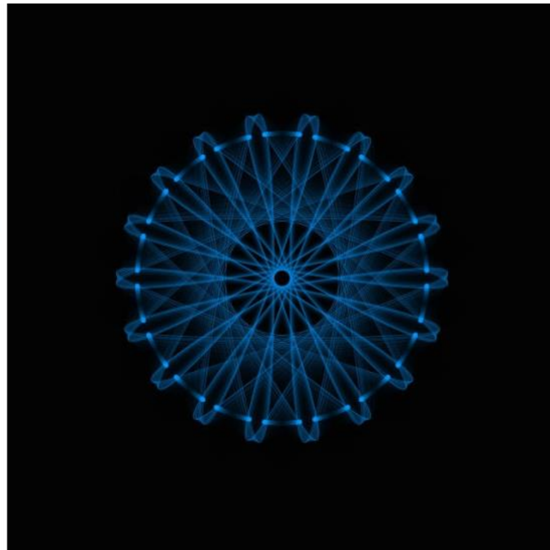



Figura 60

## SISTEMAS-L EM PROCESSING

Considere o seguinte exemplo de um sketch de Processing:

**EXEMPLO 1**

```
int length=100;

void setup() {
  size(1000, 1000);
  stroke(255);
  background(0);
}

void draw() {
  angle = PI/2 * mouseX / width;
  translate(width/2,height*3/4);
  line(0,0,0,-length);
  translate(0,-length);
  if (mousePressed) {
    background(0);
    branches(length);
  }
}

void branches(float height) {
  height *= 0.8;
  if (height > 5) {
    pushMatrix();
    rotate(angle);
    line(0, 0, 0, -height);
    translate(0, -height);
    branches(height);
    popMatrix();
    pushMatrix();
    rotate(-angle);
    line(0, 0, 0, -height);
    translate(0, -height);
    branches(height);
    popMatrix();
  }
}
```

Deslocando o ponteiro do rato e clicando em diferentes zonas do ecrã, afetamos o ângulo (ver linha 11), e assim produzimos os diferentes resultados, ilustrados pelas imagens seguintes.

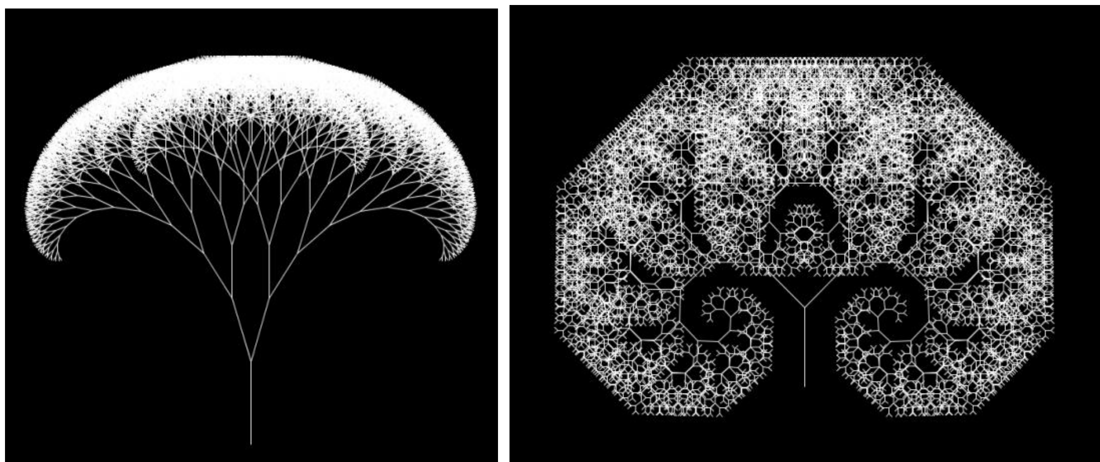


Figura 61 - Duas capturas resultantes do sketch acima, obtidas com dois ângulos diferentes, com o mesmo algoritmo

## VARIAÇÕES ESTOCÁSTICAS

#1: ao introduzir um fator aleatório junto ao ângulo, possibilitando que qualquer ângulo entre 0 e o valor armazenado na variável **angle** seja usado, o nosso modelo exibe diferenças significativas.

### EXEMPLO 2

```
(...)
void branches(float height) {
  height *= 0.8;
  if (height > 5) {
    pushMatrix();
    rotate(angle*random(1));
    line(0, 0, 0, -height);
    translate(0, -height);
    branches(height);
    popMatrix();
    pushMatrix();
    rotate(-angle*random(1));
    line(0, 0, 0, -height);
    translate(0, -height);
    branches(height);
    popMatrix();
  }
}
```

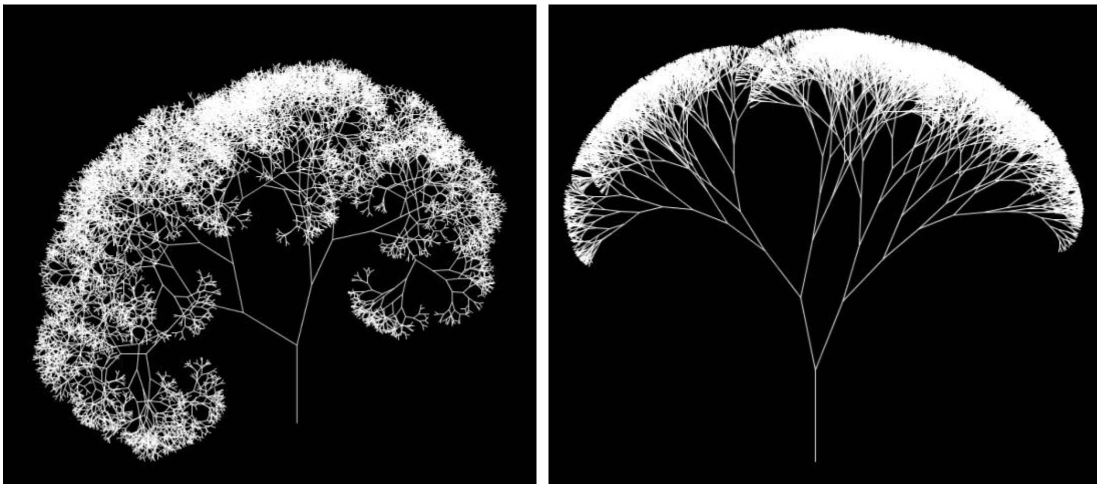


Figura 62 - Duas capturas da variação acima, obtidas com dois ângulos diferentes, com o mesmo algoritmo

#2: Se se aprofundar o uso da aleatoriedade e se a usarmos para decidir se ambos os ramos serão gerados, então os resultados tornam-se também mais interessantes:

### EXEMPLO 3

```
(...)
void branches(float height) {
  height *= 0.8;
  if (height > 5) {
    if(random(1)>.2) {
      pushMatrix();
      rotate(angle*random(1));
      line(0, 0, 0, -height);
      translate(0, -height);
      branches(height);
      popMatrix();
    }
```

```
    if(random(1)>.2) {
      pushMatrix();
      rotate(-angle*random(1));
      line(0, 0, 0, -height);
      translate(0, -height);
      branches(height);
      popMatrix();
    }
  }
}
```

Figura 63 - Duas capturas da variação acima, obtidas com dois ângulos diferentes, com o mesmo algoritmo

Neste exemplo, identificamos o dispositivo estruturante como o algoritmo abstrato, a amplificação como o mapeamento num fundo escuro e linhas brancas, e a deteção de eventos como a nossa escolha consciente de capturar um resultado em detrimento doutros resultados menos "interessantes". O(s) exemplo(s) anterior(es) representa(m) outro aspeto da conceção de uma obra de arte generativa: todos os três estágios – dispositivo estruturante, amplificação e deteção de eventos – podem ser (e normalmente serão) revistos e alterados, testados, aperfeiçoados.

Para mostrar como o esboço anterior pode ser transformado, simplesmente, para produzir resultados muito diferentes, consideremos estas poucas mudanças, principalmente nas funções *setup()* e *draw()*, essencialmente removendo a dependência do rato – agora o sistema evolui sempre livremente – e apagando gradualmente todas as iterações anteriores por meio da transparência:

**EXEMPLO 4**

```
void setup() {
  size(1000, 1000);
  background(0);
  colorMode(HSB,100,100,100);
  angle = 2*PI;
  rectMode(CENTER);
}

void draw() {
  fill(0,20);
  translate(width/2, height/2);
  rect(0,0,width,height);
  stroke((millis()/100)%100,100,100);
  branches(length);
}

(...)
```

E o resultado é significativamente diferente dos exemplos anteriores, embora as alterações ao código fossem reduzidas.

Figura 64 - Uma captura da variação acima, ilustrando as enormes diferenças no resultado visual através de pequenas alterações ao código

**ATIVIDADE**

Tente mudar as cores dos ramos durante o processo geracional da sua criação (desenho), em vez de ter toda a "árvore" na mesma cor. Posteriormente altere o “vocabulário” da obra, e substitua as linhas por círculos ou quadrados.

Se completou com êxito a atividade acima, agora consegue perceber quão crítico é a fase da amplificação no que toca ao resultado final, e também pode apreciar como é fácil substituir uma linha ou um quadrado por qualquer outro conceito, desde um som pré-gravado, uma fotografia ou vídeo, ou até mesmo uma emoção, uma sequência de movimentos ou palavras e frases.

**BIBLIOGRAFIA:**

Pearson, M. (2011). *Generative Art: A Practical Guide Using Processing*. Manning Publications Co. disponível em https://livebook.manning.com/book/generative-art/table-of-contents/

Shiffman, D. (2012). *The Nature of Code: Simulating Natural Systems with Processing*. ISBN-13: 978-0985930806

Veiga, P. A. (2017). Generative theatre of totality. *Journal of Science and Technology of the Arts, 9*(3), 33-43. DOI 10.7559/citarj.v9i3.422

# RASTER GRAPHICS OU BITMAPS

*Raster graphics* ou *bitmaps* são termos usados para designar as imagens que habitualmente partilhamos, resultantes de fotografia ou ilustração não vetorial. O Processing admite vários tipos de bitmaps (jpg, png, tga), mas os mais comuns são de facto o jpg, pela sua elevada compressão, e também o png, porque nos permite trabalhar com transparências já definidas na imagem.

Figura 65 - Esta figura exemplifica a possibilidade de, através do formato png, podermos usar uma laranja perfeitamente recortada do fundo, para a poder animar num sketch em Processing.

Vamos ver agora como conseguimos reproduzir uma imagem em Processing. Para isso deverá guardar o seu sketch e colocar na mesma pasta uma imagem, cujo nome deve substituir no código abaixo, onde está **000.png**. Observe os três passos importantes no código abaixo: a criação da variável do tipo **PImage**, o carregamento da imagem durante o setup() através do comando **loadImage()**, e o desenho da imagem em draw(), através de **image(variável do tipo PImage, coordenada x, coordenada y)**.

Procure obter uma imagem em formato png, com transparência.

**EXEMPLO 1**

```
PImage img;

void setup (){
  background(100);
  smooth();
  size(800, 800);
  img = loadImage("000.png");
}

void draw() {
  background(100);
  image(img, mouseX, mouseY);
}
```

Figura 66 - O resultado da execução do código acima é uma laranja que segue o rato, sobre um fundo cinzento.

## ALTERANDO AS CORES

Vamos agora alterar as cores da nossa imagem, recorrendo à função tint().

### EXEMPLO 2

```
PImage img;

void setup (){
  background(100);
  smooth();
  size(800, 800);
  imageMode(CENTER);
  img = loadImage("000.png");
}

void draw() {
  background(100);
  float red=map(mouseX,0,width,100,255);
  float green=20;
  float blue=map(mouseY,0,height,100,255);
  tint(red,green,blue);
  image(img, mouseX, mouseY);
}
```

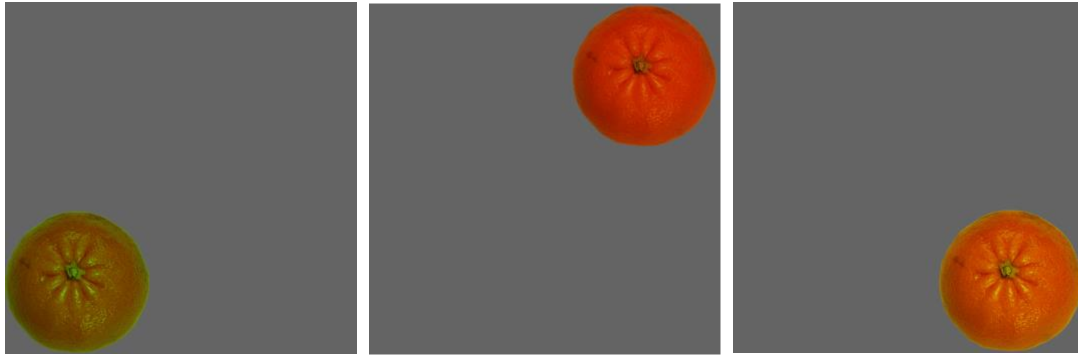


Figura 67 - Ao fazermos variar as componentes vermelha e verde com as posições horizontal e vertical do rato, a nossa imagem vai mudar de cor.

Vamos agora ver como usar transparência com as imagens, recorrendo também à função tint().

Usamos para isso duas imagens, e vamos fazer com que uma delas seja completamente transparente quando o rato está do lado esquerdo, e a outra seja completamente transparente do lado direito. Ao centro, ambas estarão semi-opacas, a 50%.

**EXEMPLO 3**

```
PImage img1, img2;

void setup (){
  size(800, 800);
  background(100);
  smooth();
  imageMode(CENTER);
  img1 = loadImage("000.png");
  img2 = loadImage("001.png");
}

void draw() {
  background(100);

  float transp1=map(mouseX,0,width,0,255);
  tint(255, transp1);
  image(img1, mouseX, mouseY);

  float transp2=map(mouseX,0,width,255,0);
  tint(255, transp2);
  image(img2, mouseX, mouseY);
}
```

Figura 68

O primeiro passo é criar o ficheiro com o tipo de letra que iremos usar. Para isso vamos ao menu Tools (ferramentas) do Processing, e à opção Create Font.

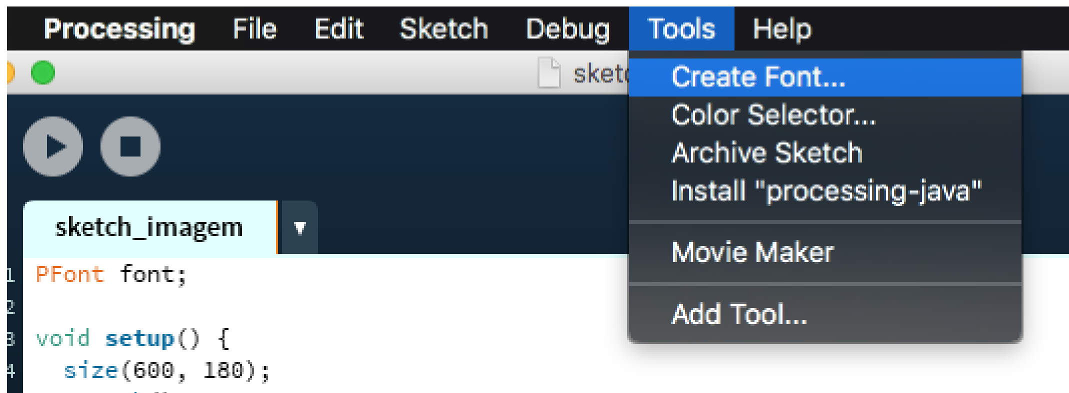



Figura 69 – Criando um Tipo de Letra (Font)

De seguida escolhemos o tipo de letra e o tamanho que queremos usar, e o sistema gera um ficheiro que ficará na pasta Data, que por sua vez fica automaticamente dentro da pasta do nosso sketch. Precisamos depois de usar o nome desse ficheiro com o comando **loadFont("Bauhaus93 -48.vlw")**, como no exemplo 19.

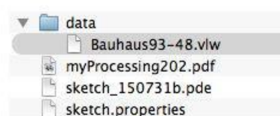



Figura 70 – Vários Tipos de Letra

**EXEMPLO 1**

```
PFont font;

void setup() {
  size(600, 180);
  smooth();
  background(0);
  noLoop();
  font = loadFont("Bauhaus93 -48.vlw");
  textFont(font, 48);
  fill(178, 7, 157);
}

void draw() {
  text("Este é um dia histórico.", 120, 100);
}
```

Vamos agora exercitar algumas variações com o texto, utilizando as transformações que já conhecemos.

**EXEMPLO 2**

```
PFont font ;
float rotateCounter = 0;

void setup() {
  size(600, 600);
  smooth();
  background(0) ;
  font = loadFont("Serif-48.vlw");
  textFont(font, 48);
}

void draw() {
  translate(width/2, height/2);

  pushMatrix();
  rotate(rotateCounter);
  fill(255) ;
  text("BLACK", mouseX-width/2, mouseY-height/2);
  popMatrix();

  pushMatrix();
  rotate(-rotateCounter*1.5);
  fill(0);
  text("WHITE", width/2-mouseX, height/2-mouseY);
  popMatrix();

  rotateCounter+=0.05;
}
```

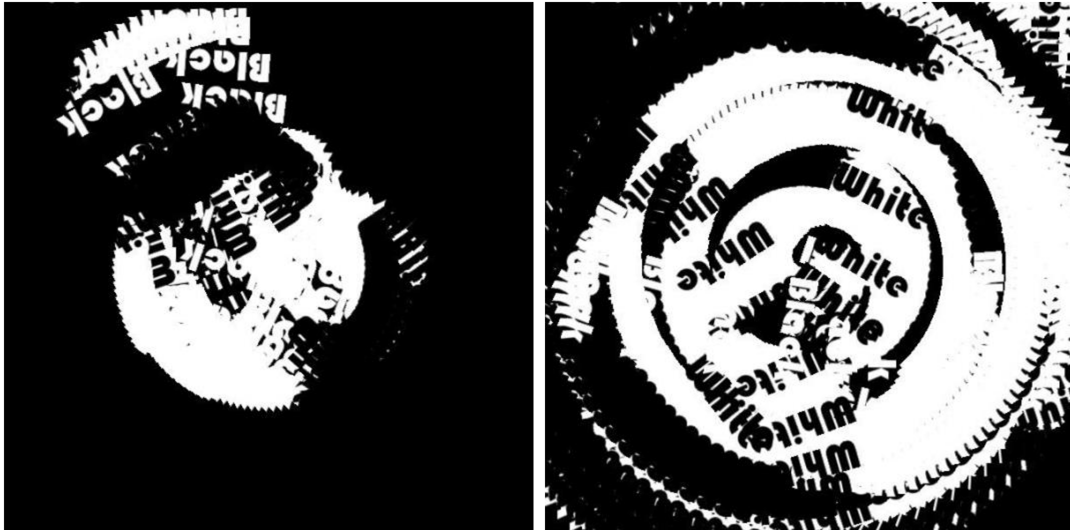



Figura 71

**ATIVIDADE**

Crie um programa onde utilize texto, transformações e imagens.

## EXEMPLO 1: DESENHAR UM ARRAY DE ELIPSES

```
int [] xCoordinate = new int[10];

void setup() {
  size(500, 500);
  smooth();
  noStroke();
  for (int i=0; i<xCoordinate.length; i++) {
    xCoordinate[i] = 35*i+90;
  }
}

void draw() {
  background(50) ;
  for (int coordinate: xCoordinate) {
    fill(200);
    ellipse(coordinate, 250, 30, 30);
    fill(0);
    ellipse(coordinate, 250, 3, 3);
  }
}

void keyPressed() {
  if (key=='s') saveFrame("myProcessing.png");
}
```

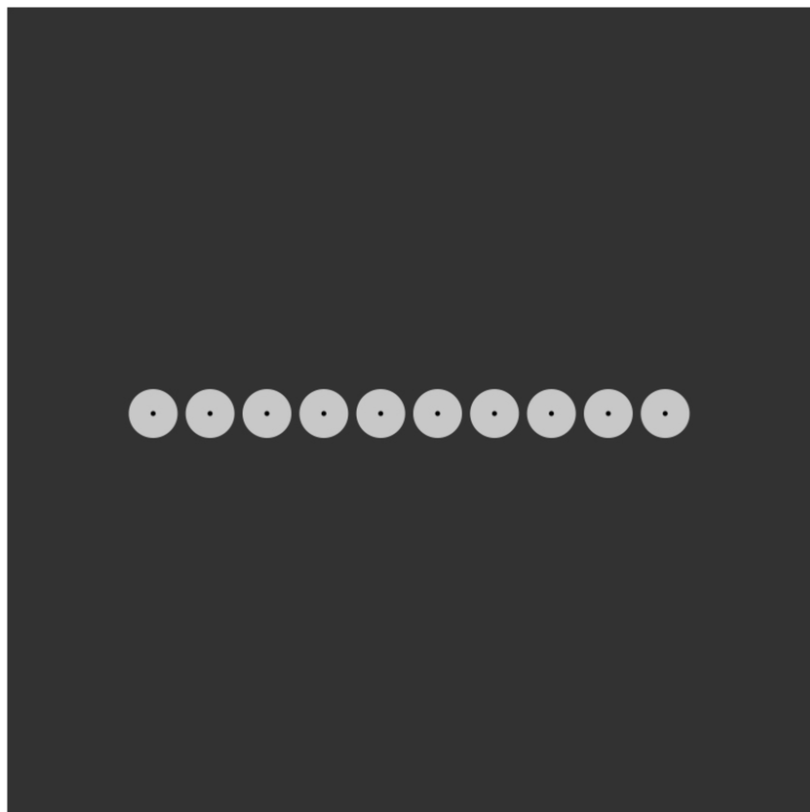


Figura 72 - Resultado do Exemplo 1.

## EXEMPLO 2: EXCERTO DE CÓDIGO PARA ARRAY DE DIMENSÃO ALEATÓRIA

```
   int  k  =  (int) random(5 ,15);
   int []  xCoordinate = new int[k];
```

## EXEMPLO 3: ARRAY DE ELIPSES DE DIMENSÃO CRESCENTE

```
int [] xCoordinate = new int[10];

void setup() {
  size(500, 500);
  smooth();
  noStroke();
  for (int i=0; i<xCoordinate.length; i++) {
    xCoordinate[i] = 35*i+90;
  }
}

void draw() {
  background(50) ;
  for (int coordinate: xCoordinate) {
    fill(200, 40);
    ellipse(coordinate, 250, (coordinate-90)/4, (coordinate-90)/4);
    fill(0);
    ellipse(coordinate, 250, 3, 3);
  }
}

void keyPressed() {
  if (key=='s') saveFrame("myProcessing.png");
}
```

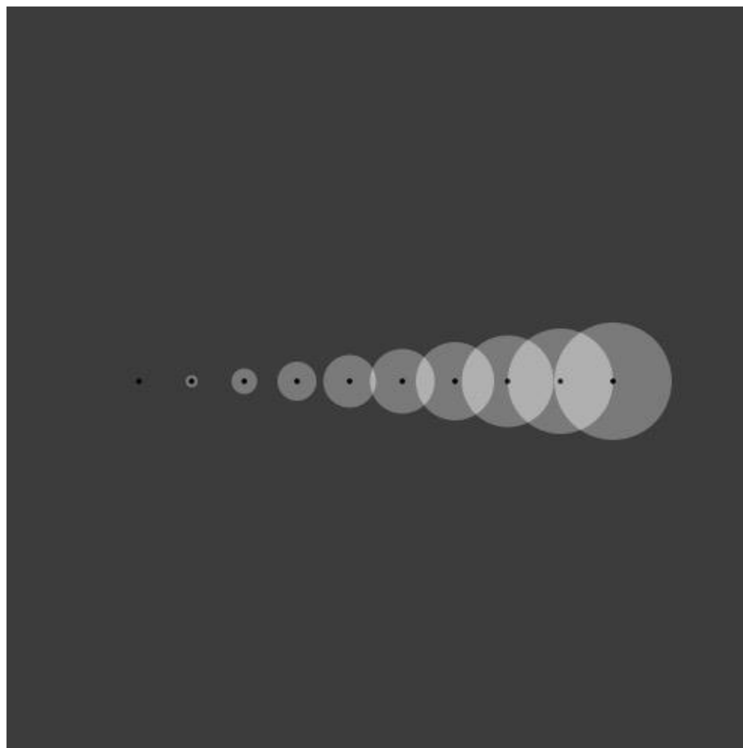


Figura 73 - Exemplo 3

**ATIVIDADE**

Altere o código, acrescentando mais um array, de forma a obter o resultado da figura abaixo.

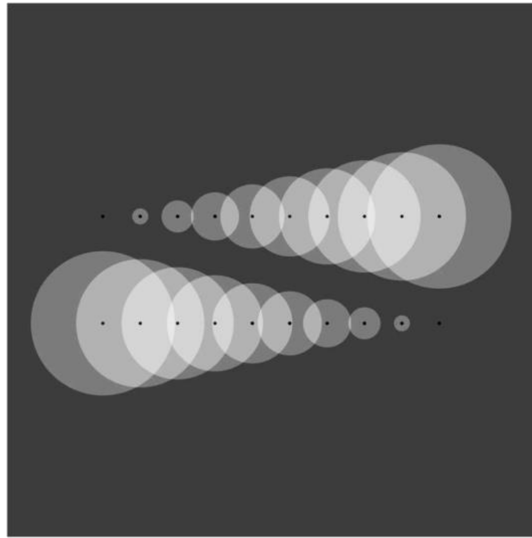


Figura 74 - Resultado da Atividade acima

**EXEMPLO 4: UMA NUVEM DE ELIPSES, RENOVADA ATRAVÉS DA POSIÇÃO HORIZONTAL DO RATO**

```
int [] xCoordinate = new int[30];

void setup() {
  size(500, 500);
  smooth();
  noStroke();
  init();
}

void draw() {
  background(50) ;
  for (int i=0; i<xCoordinate.length; i++) {
    fill(200, 40);
    ellipse(xCoordinate[i], 250, 10*i, 10*i);
    fill(20);
    ellipse(xCoordinate[i], 250, 5, 5);
  }
  if(mouseX > 250) init();
}

void init() {
  for (int i=0; i<xCoordinate.length; i++) {
    xCoordinate[i] = 250+(int)random(-100, 100);
  }
}

void keyPressed() {
  if (key=='s') saveFrame("myProcessing.png");
}
```

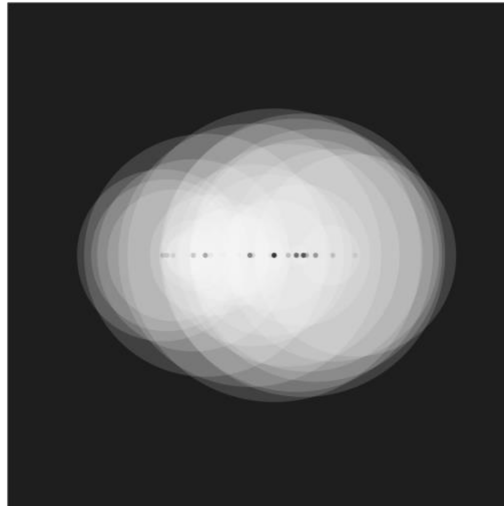


Figura 75 - Exemplo 4

Ao deslocar o rato no ecrã, alteramos a posição em que são desenhados os círculos. Estas posições são registadas num array e executadas a cada ciclo da função draw(). A cada ciclo é adicionado o valor da posição corrente (mais recente) e eliminado o valor mais antigo. Usamos a função módulo (%) para variar mais rapidamente os índices à medida que os ciclos se evoluem (e são contabilizados em **which**).

### EXEMPLO 5: ANIMAÇÃO ATRAVÉS DO REGISTO DAS ÚLTIMAS POSIÇÕES DO PONTEIRO DO RATO

```
int  num = 60;
float mx[] = new float[num];
float my[] = new float[num];

void setup() {
  size(640, 360);
  noStroke();
  fill(255, 153);
}

void draw() {
  background(51) ;

  int which = frameCount%num;
  mx[which] = mouseX;
  my[which] = mouseY;

  for (int i=0; i<num; i++) {
    int index = (which+1+i)%num;
    ellipse(mx[index], my[index], i, i);
  }
}
```

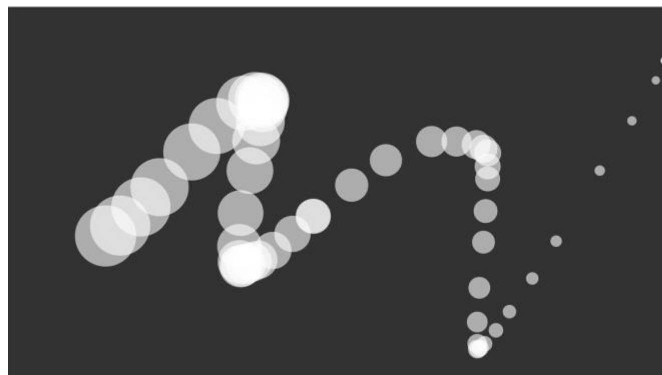


Figura 76 - Exemplo 5

## ARRAYS MULTIDIMENSIONAIS

### EXEMPLO 6: DESENHAR ELIPSES A PARTIR DE UM ARRAY MULTIDIMENSIONAL

```
int numberOfEllipse = 10;
int step = 50;
float counter = 0;
int [][] a = {
  {50, 133, 40},
  {132, 87, 90},
  {12, 287, 10},
  {300, 407, 30},
  {232, 187, 190},
  {448, 300, 79},
  {460, 450, 32}
};

void setup() {
  size(500, 500);
  smooth();
  noStroke();
}

void draw() {
  background(127);
  for (int i=0; i<a.length; i++) {
    float angle = counter/(float)(1+i);
    float myC = map(sin(angle), -1, 1, 0, 255);
    fill(myC);
    ellipse(a[i][0], a[i][1], a[i][2], a[i][2]) ;
    counter+=0.01;
  }
}
```

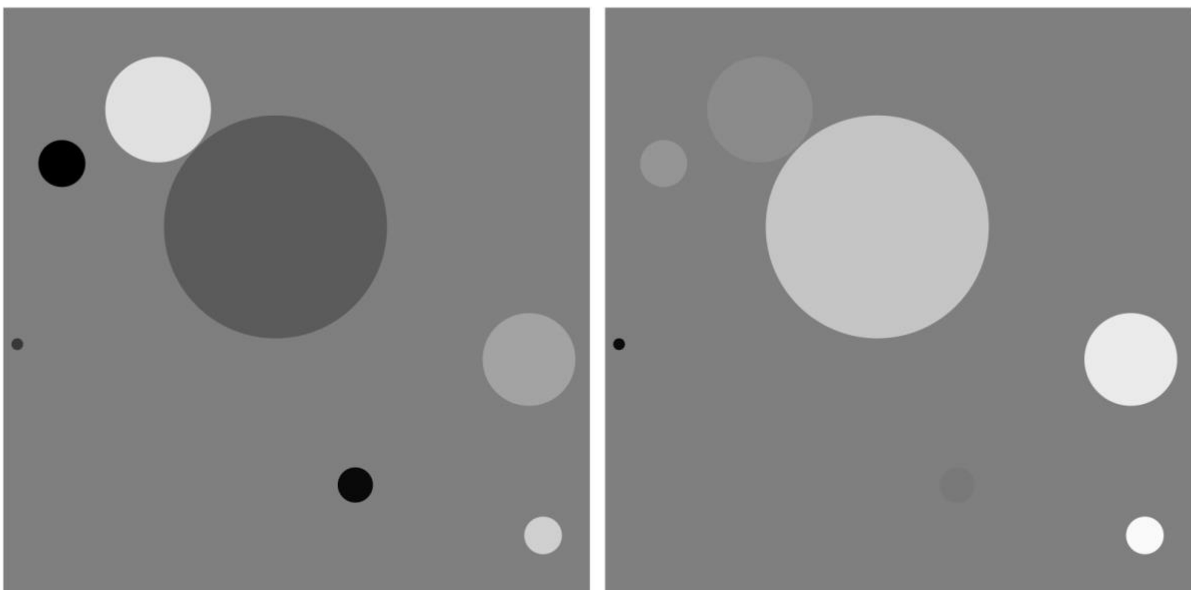


Figura 77 - Exemplo 6

## EXEMPLO 7: DESENHAR ELIPSES E CONTROLAR O SEU TAMANHO ATRAVÉS DA POSIÇÃO DO RATO

```
float [][] a = new float[500][2];

void setup() {
  size(500, 500);
  for(int i=0; i<a.length; i++) {
    for(int j=0; j<a[i].length; j++) {
      a[i][j] = random(10, 490);
    }
  }
}

void draw() {
  smooth();
  noStroke();
  background(0);

  for(int i=0; i<a.length; i++) {
    float eDist = dist(mouseX, mouseY, a[i][0], a[i][1]) ;
    float eSize = map(eDist, 0, 200, 5, 100) ;
    float eColor = map(eDist, 0, 200, 50, 255) ;
    fill(eColor, 200);

    float cx = noise(mouseX)*10+a[i][0];
    float cy = noise(mouseY)*10+a[i][1];

    ellipse(cx, cy, eSize, eSize);
  }
}
```

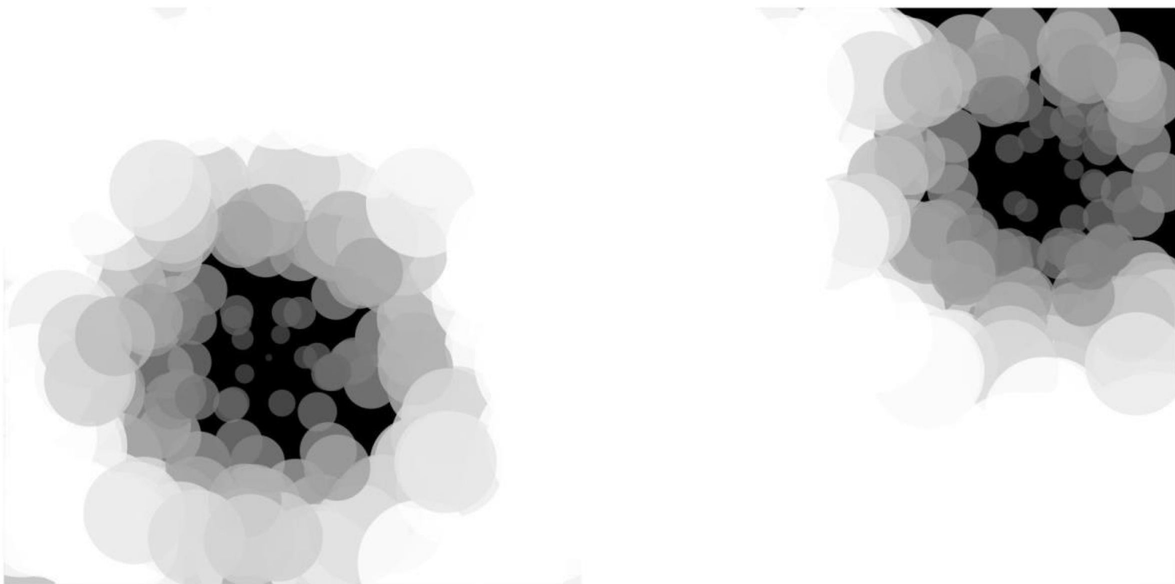


Figura 78 - Exemplo 7

## ATIVIDADE

Altere o código para desenhar linhas ou retângulos em vez de elipses.

### EXEMPLO 8: EXEMPLO DE INSTANCIAÇÃO DE VALORES A CÉLULAS DE UM ARRAY

```
int  a [][] = new int [3][5];
a[0] = new int[7];
a[1] = new int[0];
a[2] = null;
```

## VISUALIZAÇÃO DE UM ARRAY

### EXEMPLO 9: VISUALIZAÇÃO DE UM ARRAY

```
private int [][] a = new int[10][];
private int step = 30;
void setup() {
  size(500, 500);
  smooth();
  noStroke();
  myInit();
}

void  myInit() {
  for(int i=0; i<a.length; i++) {
    a[i] = new int [(int) random(0, 10)];
    for( int j=0; j<a[i].length; j++) {
      a[i][j] = (int) random(0, 30);
    }
  }
}

void draw() {
  fill(180, 50);
  background(10);
  for( int i=0; i<a.length; i++) {
    for( int j=0; j<a[i].length; j++) {
      stroke(100);
      strokeWeight(1) ;
      fill(50) ;
      rect( i*step+100, j*step+100, step, step);
      noStroke();
      fill(250, 90);
      ellipse(i*step+115, j*step+115, a[i][j], a[i][j]);
    }
  }
}

void  mouseClicked() {
  myInit ();
}
```

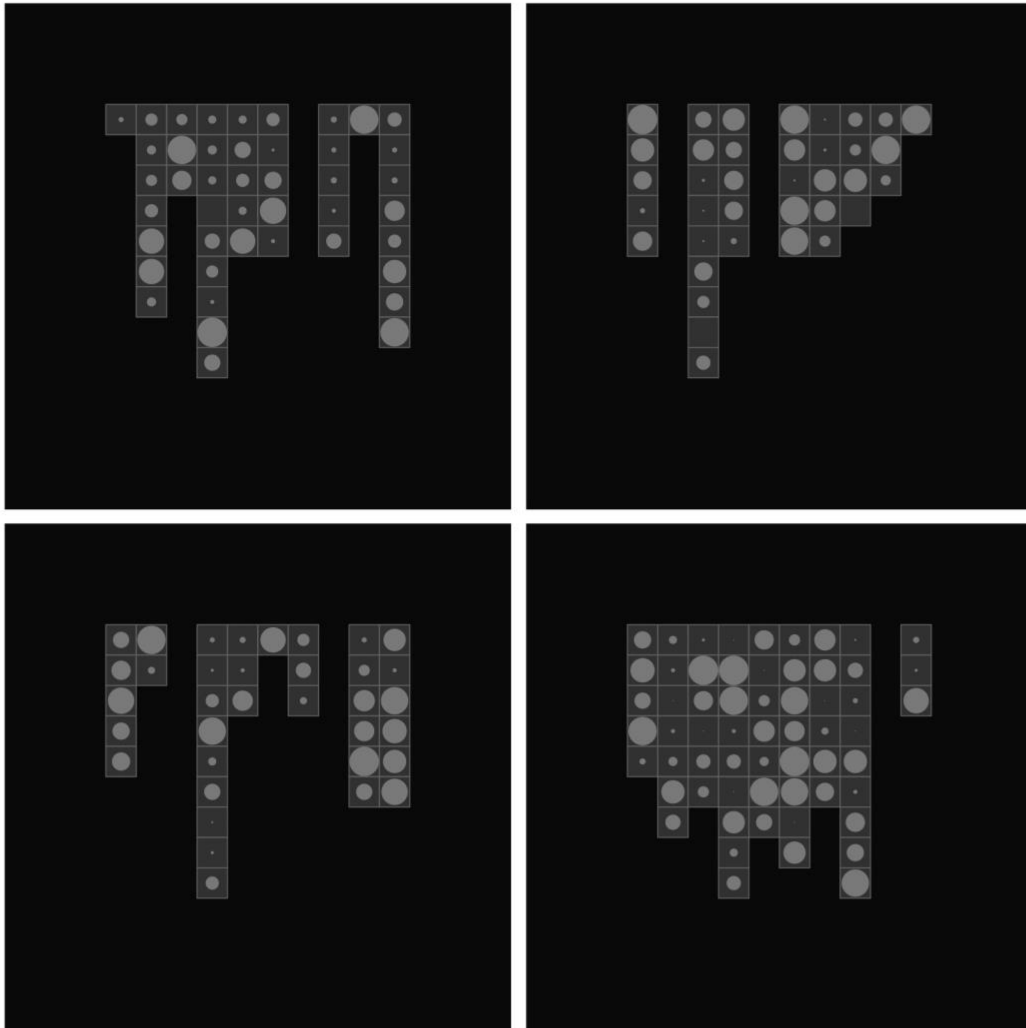


Figura 79