Programação Criativa – Tópico 1

***Temática**: Introdução ao Processing e conceitos de programação*
***Duração**: Semana 1*

Actividade 1: Conhecer o Processing como ferramenta de programação criativa e os conceitos básicos por detrás da programação

Competências a desenvolver:

- Instalar o processing e familiarizar-se com o ambiente
- Aprender os conceitos básicos de desenvolvimento de código e programação

1. O que é programar um computador

Apesar de muito falado, nada melhor do que assistir este vídeo curto, que insere uma série de conceitos importantes que iremos utilizar ao longo desta UC. Vá ao YOUTUBE e aceda o seguinte vídeo: https://www.youtube.com/watch?v=iA_8W6saSsc.

2. O que é o Processing?

Processing é uma linguagem de programação de código aberto e ambiente de desenvolvimento integrado (IDE), construído para as artes eletrônicas e comunidades de projetos visuais com o objetivo de ensinar noções básicas de programação de computador em um contexto visual e para servir como base para cadernos eletrônicos. 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 criar alguns novos.

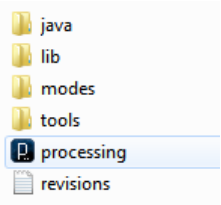
Processing inclui um *sketchbook*, uma alternativa para organizar projetos sem ser o padrão IDE. Cada esboço de processamento é realmente uma subclasse do Java PApplet classe que implementa a maioria das funcionalidades do Processamento de Linguagem.

Ao programar em **Processing**, todas classes adicionais definidas serão tratadas como classes internas quando o código é traduzido para Java puro antes de compilar. Isso significa que o uso de variáveis e métodos estáticos em classes é proibido a menos que você diga que deseja o processamento para o código no modo de Java puro.

3. Instalação e ambiente

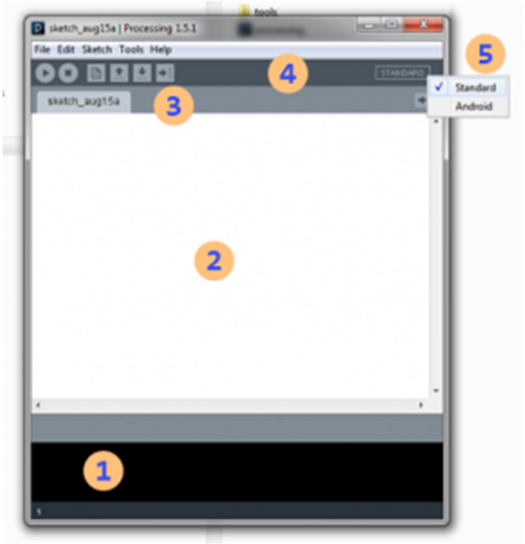
O software pode ser baixado gratuitamente na área de *downloads* do website Processing.org (http://www.processing.org). Existem versões para Linux, Mac OSX e Windows (incluindo JDK) – no caso do Windows, recomenda-se que os iniciantes não escolham a versão *without Java*. O **Processing** não necessita de uma instalação específica, bastando descompactar os ficheiros na pasta de *Programas* ou em qualquer outra pasta do seu computador seu computador.

Ao descompactar a pasta encontramos a estrutura:



O **Processing** pode ser aberto diretamente clicando o ficheiro executável. As outras pastas do pacote incluem bibliotecas e ficheiros importantes para a compilação de seus futuros programas. A pasta *libraries* é o local onde podem ser incluídas bibliotecas e extensões desenvolvidas pela comunidade. O caminho a partir da raiz é: *modes > java > libraries*.

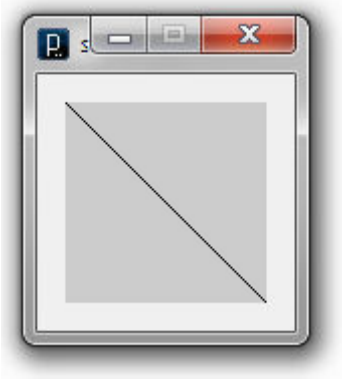
A interface do **Processing** é composta por duas janelas – a principal, contendo um editor onde podemos criar nossa programação e a janela de visualização destinada ao *render* de gráficos, textos, imagens e vídeos.



Zonas de interacção no ambiente Processing

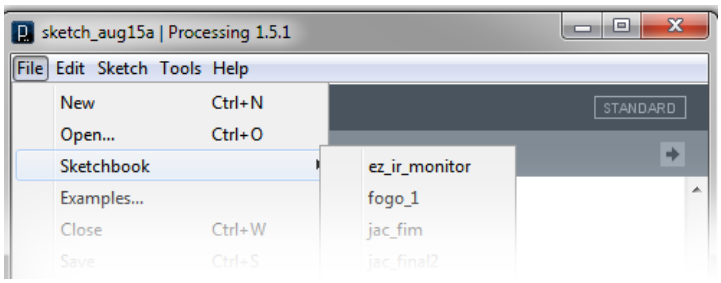
1. Console (área de testes e mensagens de erros)
2. Editor (programação por texto)
3. Barra de objetos (cada etiqueta representa um novo objeto)
4. Barra de ações contendo botões: *Run, Stop, New, Open, Save, Export*
5. Seleciona "modo" de programação – o programa pode ser exportado para Android

A janela de visualização é aberta quando executamos algum programa com o botão "Run".



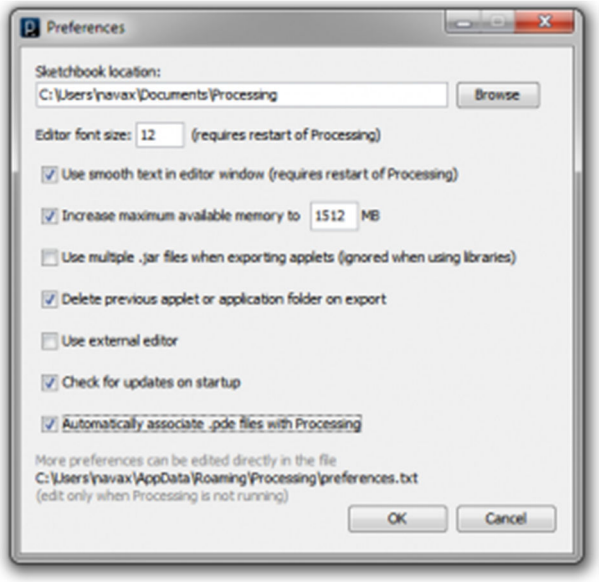
Janela de visualização

Os programas criados no ambiente do **Processing** são salvos numa área padrão chamada "*Sketchbook*" (pasta documentos/processing).



Os ficheiros podem carregados ou salvos no *sketchbook*

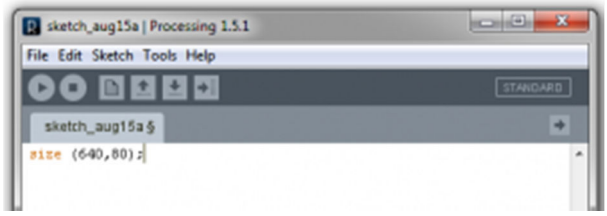
A localização da pasta referente ao *Sketchbook* pode ser configurada no painel de preferências do **Processing** (menu *File>Preferences*). Outra opção de configuração importante é a reserva de memória para a execução do **Processing**. Como algumas aplicações exigem áreas de processo e armazenamento acima de 1Gb, devemos alterar a disposição deste espaço no campo *Increase maximum available memory* – também em *File>Preferences*.



Em preferências podemos alterar configurações sobre o editor, ficheiros e gestão de memória.

Os ficheiros fontes 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 quanto Mac ou Linux (*file > Export Application*).

As dimensões (largura e altura em *pixels*) são configuradas pela função ***size*** (largura, altura) que deve ser incluída na área de edição de programas. Neste exemplo estamos configurando inicialmente uma área de trabalho com 640 x 480 pixels:



As dimensões do *sketch* devem ser configuradas inicialmente na programação. O tamanho padrão é 100 x 100 pixels

Após a instalação, explore o ambiente ao máximo e tente correr alguns dos exemplos que vêem junto. Visite o website oficial onde poderá encontrar muito material interessante para aprender (http://processing.org/reference/).

4. Código

Antes de começarmos a programar, é importante perceber como se estrutura um programa. Um programa é na realidade um conjunto de instruções que você passa para o computador 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 a execução de nada mas é muito útil é o **comentário**. Ele deve ser inserido ao longo do programa, como uma boa prática de explicação do mesmo. Ele serve para o programador explicar o que o seu código faz para outra pessoa que venha o utilizar.

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

```
// Two forward slashes are used to denote a comment.
// All text on the same line is a part of the comment.
// There must be no spaces between the slashes. For example,

// the code "/ /" is not a comment and will cause an error
```

a. Funções

As funções são as peças fundamentais de funcionamento de um programa. Existem **funções pré-definidas que estão prontas para utilizar** e servem para fazer várias coisas: 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 utilizada para declarar funções é sempre com o seguinte formato:

$$<tipo> <nome\ da\ função> (<param_1>, <param_2>, ..., <param_n>)$$

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 chama-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:

```
// 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, 1 para largura e outro para a altura da janela)
size(300, 300);

// 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 os { } é interpretado como instruções a serem
// executadas quando a função é invocada…
float noise (float n, int m){ }
```

b. Expressões, declarações

A maior dificuldade que alguém apresenta para aprender a programar é a de conseguir quebrar as acções nas mais atómicas possíveis e torna-las um conjunto de instruções simples e sequenciais. Para conseguirmos isso, utilizamos expressões e declarações apropriadas.

As expressões em programação são geralmente uma combinação de operadores como +, * e / que são avaliados da esquerda para a direita. Elas podem ser simples e básicas, envolvendo poucos valores e operadores, ou ao contrário, implicarem muitos valores (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 algo textual.

As declarações são na verdade, cada linha de código que identifica uma instrução atómica, terminada por ponto e vírgula (;).

Alguns exemplos:

```
// Definem-se duas variáveis de memória para receberem valores numéricos do tipo
// inteiro de nome a e b (falaremos já a seguir um pouco sobre esse conceito de
// variável de memória)
int a; // declaração que cria a
int b; // declaração que cria b
// Inicializamos a com 1 e depois incrementamos de 2, logo passa a ter valor 3
a = 1; // declaração que atribui 1 a variável a
a = a + 2; // declaração que soma 2 ao valor atual de a
// Testamos se a é maior que 2, se for, b será inicializado com valor igual a 3
// vezes o de a, ou seja passa a ser 6, senão, nada acontece
if (a > 2) b = 3 * a; // declaração que testa o valor de a e altera o valor de b
```

c. Variáveis de memória

As variáveis de memória são outra parcela importante na programação. Toda informação que é passada para o computador através de um programa, deve normalmente ser armazenada na memória deste. A memória 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” variável dinamicamente.

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 momento abrir a “gaveta” desejada e devidamente preparada para receber a informação que queiramos lá pôr.

As variáveis podem ser simples ou ter muitos elementos (arrays) ou ainda dimensões (matrizes) e uma característica importante, é a sua duração ou vida. Elas são criadas, alteradas e destruídas dentro de um programa, conforme a necessidade. É importante se 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. Também temos que ter cuidado **de não ultrapassar o tamanho definido** para um array ou matriz, tentando atribuir/editar/apagar mais elementos do que se definiu na sua criação.

A criação das variáveis de memória obedece sempre este formato:

<tipo> <nome> <dimensões>;

Onde *<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 se define, e *<dimensões>* poderá ou não existir, pois só os *arrays* e matrizes é que apresentam esta parcela. As *<dimensões>* são definidas sempre entre [ ].

Alguns exemplos:

```
int a; // criação da variável simples “a” do tipo numérico inteiro
float b; // criação da variável simples “b” do tipo numérico ponto flutuante
// criação de variável simples do tipo texto, com inicialização
String palavras = “texto….”;
boolean c = true; // criação de variável booleana simples com inicialização
int d[3]; // criação de array de 3 elementos do tipo numérico inteiro
d[1] = 3; // atribui ao 2° elemento do array d o valor 3 (inicia no índice 0)
float e[4][5]; // cria uma matriz de nome e com 4 linhas e 5 colunas
```

5. Blocos de código e parêntesis

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 pelos parêntesis, com o de abertura indicando o início, e o de fecho indicando o fim.

Alguns exemplos:

```
void draw() {// início da função draw()- o que estiver dentro, é “dominado” por ela
        background(255);
        stroke(0);
        fill(175);
        rectMode(CENTER);
        rect(mouseX,mouseY,50,50);

        if (mouseX > 100) // início da estrutura de condição if – o que estiver
dentro é “dominado” pelo if, além de draw()
        {
                Fill(200);
                Rectmode(LEFT);
        } // fim da estrutura if
} // fim da função draw()
```

6. Sintaxe e seus erros

É importante atentar para 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 simplesmente a rejeita, gerando erros. São os chamados erros de sintaxe.

Todas **as instruções são terminadas e separadas por ponto e vírgula** (;), o nome de variáveis ou funções **não podem conter espaços em branco**, ao definirmos variáveis de memória ou funções, temos que seguir o formato de declaração exigido ou ainda, o corpo de uma função está **delimitado por { },** são alguns exemplos das regras que temos que seguir durante a escrita do programa.

A interpretação de **maiúsculas e minúsculas é diferenciada**. Escrever ***size*** é diferente de ***Size***. Temos que evitar a utilização de palavras reservadas na atribuição de nomes de variáveis ou funções. As palavras reservadas são aquelas que são pré-definidas no processing. Por exemplo, ***setup*** está pré-definido (só se utilizarmos ***Setup***…). **Não podemos criar uma função com esse mesmo nome**. 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.

7. Execução e seus erros

Antes da execução o programa **tem que ser traduzido para linguagem de máqu**ina, ou seja, código binário. Essa fase é definida como compilação. Na compilação, **os erros de sintaxe** são detectados e indicados pelo processing. 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 podem haver **erros de lógica**. Nesse caso, o programa "arrebenta" 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 mau geridas, podem causar isso. Outros erros de execução prendem-se ao programa não fazer aquilo que queríamos que fizesse. Ele até pode correr bem, mas o resultado final, não é o esperado.

Para executar o ***debug*** desses erros (que são os mais chatos de encontrar e corrigir), utilizamos muitas vezes a console (área 1) no processing. Com o auxílio de funções como a ***print()*** e ***println()*** somos capazes de imprimir nessa área o conteúdo de variáveis e rastrear a execução de funções, eventos e do programa como um todo.

Exemplos:

```
// To print text to the screen, place the desired output in quotes
println("Processing..."); // Prints "Processing..." to the console
// To print the value of a variable, rather than its name,
// don't put the name of the variable in quotes
int x = 20;
println(x); // Prints "20" to the console
```

8. Utilização de bibliotecas

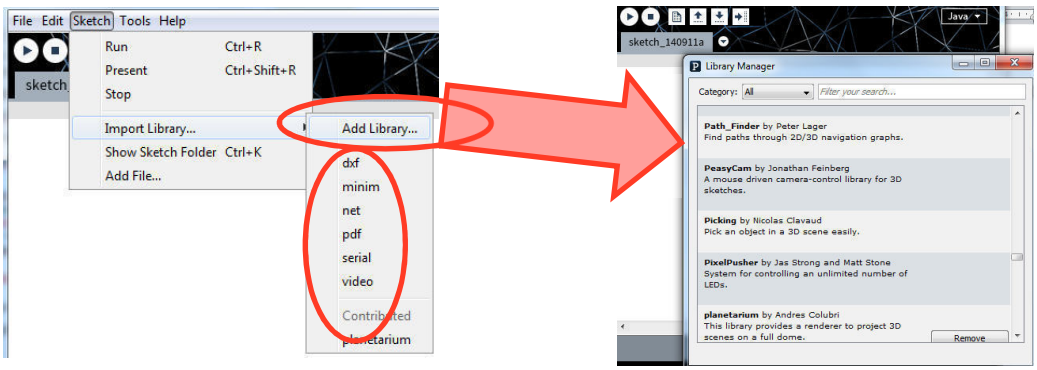
Quando estamos utilizando uma determinada linguagem de programação**, dispomos sempre de um conjunto bastante vasto e variado de funcionalidades prontas** que podemos utilizar sem nos preocuparmos com aspectos de implementação. Essas funções são como "caixas pretas", que se utilizam. Passa-se os parâmetros necessários para o seu bom funcionamento e tira-se partido de seus resultados, sem maiores preocupações. Uma situação semelhante ocorre na calculadora. Quando você precisa de calcular a raiz quadrada de um número, utiliza a função disponível e pronto!

Esse mesmo raciocínio está disponível nas linguagens de programação, só que de uma forma muito mais vasta. Dispomos de **diferentes bibliotecas,** que possuem funcionalidades dedicadas as **mais distintas aplicações**: som, imagem, interacção com outros dispositivos, etc. Por defeito, muitas funcionalidades já estão disponíveis sem que você tenha que incluir uma biblioteca em especial no seu programa, porém, haverá situações que será necessário incluí-las no seu código.

Para isso, basta declarar a biblioteca que possui a funcionalidade desejada no **início do ficheiro de código fonte** que utiliza a respectiva funcionalidade, com a palavra reservada a frente ***import***, e depois, invocar a funcionalidade normalmente. No exemplo a seguir, incluímos as bibliotecas de vídeo e de criação de ficheiros em formato *pdf* (o nome das bibliotecas é **processing.video** e **processing.pdf** respectivamente)

```
import processing.video.*; // Declarações de importação logo nas primeiras linhas…
import processing.pdf.*;
// Corpo do programa
void setup() {
  size(400, 400, PDF, "filename.pdf");
}
// demais código…
```

Outro detalhe na utilização de bibliotecas é que você terá que configura-las no seu ambiente de desenvolvimento, para que elas estejam efetivamente disponíveis e ao seu dispor. No caso de serem bibliotecas já incluídas no processing, bastando apenas ativar a sua utilização, devemos aceder o menu ***Import Library*** e escolher a biblioteca desejada. No caso de ser de contribuição de terceiros, teremos que aceder o menu ***Add Library*** e executar mais alguns passos adicionais:



Existem bibliotecas que têm sido desenvolvidas por contribuição de terceiros. Uma lista completa das disponíveis no processing e por contribuição, bem como os passos necessários para incluí-las você encontra aqui: http://processing.org/reference/libraries/ .

Tarefas para este tópico:

- Após instalar o processing, vá ao *website* e assista alguns tutoriais introdutórios disponíveis;
- Baixe algum exemplo e teste…

Programação Criativa – Tópico 2

***Temática**: Introdução ao conceito de visualização e funções em programação*
***Duração**: semanas 2 e 3*

Actividade 2: Aprender a utilidade de funções no processing e como utiliza-las

Competências a desenvolver:

- Conceitos básicos de sistemas de visualização
- Utilizar funções e atributos em programação
- Publicar programas em processing

1. Janela de visualização e sistemas de coordenadas 2D

Quando você desenha no papel, este tem uma determinada dimensão e a localização de cada traço, ponto ou forma, possui uma posição espacial distinta (coordenadas). O mesmo raciocínio deve ser transposto para o computador. A janela de visualização deve ser vista como uma folha de papel para desenho com grelha de referência.

A janela de visualização que é mostrada aquando da execução de um programa possui uma dimensão total definida em *pixels*. A dimensão é definida por um conjunto de valores que definem a quantidade de *pixels* na horizontal (equivale ao eixo X) e na vertical (equivale ao eixo Y). O posicionamento de qualquer elemento gráfico dentro da mesma **tem que obedecer esses limites**. Por exemplo, se quisermos desenhar um ponto numa janela que tenha dimensão de 100 x 100 *pixels*, as coordenadas do ponto tem que ter valores entre 0 a 100 em ambos os sentidos, ou seja, as suas coordenadas x e y devem ter valores entre esses limites, caso contrário, o ponto não fica visível.

No caso do **processing**, a contagem dos *pixels* na janela nas duas direcções é feita a partir do canto superior esquerdo. Portanto, qualquer posicionamento deve ter em conta isso.

A criação da janela com uma determinada dimensão, a sua cor de fundo, bem como o seu posicionamento estão a cargo do programador, e serão normalmente sempre as primeiras instruções a serem dadas para que se crie uma área de visualização na execução do programa.

2. Formas geométricas

São muitas as formas geométricas que o **processing** nos oferece para criar desenhos diferentes. Pontos, círculos, linhas e curvas são facilmente introduzidos na composição visual por a aplicação direta de **funções pré-definidas**. É importante ter atenção a sintaxe de cada uma delas, para fazer o uso correto…

Outro detalhe é o aspecto que a forma terá. Podemos **alterar vários parâmetros relacionados com o seu aspecto** (espessura, cor, transparência, contornos, espacejamento, etc.). Mais importante ainda: **a ordem em que são definidas** as formas dentro do código, bem como as demais instruções. O que for declarado primeiro, é desenhado primeiro. Logo, uma forma que porventura seja definida com um posicionamento e tamanho que calhe de ficar atrás de outra maior ficará automaticamente escondida! Se quisermos que uma forma seja preenchida a vermelho, teremos primeiro que instruir o preenchimento com a cor para depois mandar

desenhar a forma. Logo, é fundamental garantir que as instruções são declaradas na ordem e localização certa do código, para que tudo aconteça como queremos.

3. Cor

A cor é um atributo muito utilizado no desenho. No **processing** a definição da cor é feita utilizando a codificação RGB, ou seja, uma cor é definida pela composição de 3 básicas – vermelho, verde e azul. Cada uma delas pode ter valores entre 0 a 255. A codificação pode ser consultada neste endereço: http://www.tayloredmktg.com/rgb/ .

Portanto, todas as funções que tenham como parâmetro de entrada a cor, adotam o padrão RGB e consequentemente, nós teremos que calibrar cada componente de forma a obter a cor desejada.

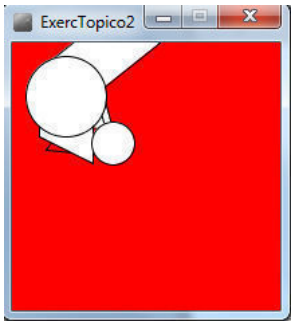
Por fim, podemos dar transparência da cor através de um quarto parâmetro, designado de canal alfa. Daí que quando trabalhamos com funções que recebem parâmetros de cor, normalmente elas podem receber também o parâmetro alfa, causando uma maior ou menor opacidade (veremos um exemplo disso nos exercícios mais abaixo).

4. Alteração de atributos

Muitas das funções que podemos utilizar podem ser afectadas por outras funções que lhe mudam os atributos de funcionamento. Algumas causam o cessar do funcionamento de uma função previamente invocada, outros, causam alteração do aspecto do desenho, como linhas mais finas ou tracejadas. A forma como se conjugam as mesmas pode criar resultados visuais bem diversos.

5. Exercitando

Vamos agora escrever passo-a-passo um código e ver o que vai acontecendo ao longo dele. É importante que codifique e vá dando "run" a medida que se pede. O resultado visual final será este:



1. Abra o **processing**
2. Defina o tamanho da janela de visualização (caso contrário, ela terá um tamanho pequeno de 100 x 100) com a função ***size(largura, altura)***:

```
size(200, 200);
```

3. 

4. Vamos dar uma cor de fundo a janela, neste caso vermelho com a função ***background(r, g, b)***:

```
background(255, 0, 0);
```

Note que os parâmetros *r, g , b* definem a cor com as componentes vermelha, verde e azul.

5. 
6. Vamos agora desenhar alguns pontos e localizações diferentes. Para isso, vamos utilizar a função ***point(x, y).*** Para garantir que todos os pontos estão dentro da janela, vamos definir coordenadas para os pontos que caiam dentro de um máximo de 300 x 300 *pixels*:

```
point(50, 30);
point(55, 30);
point(60, 30);
point(65, 30);
point(70, 30);
```

7. 
8. Repare que desenharam-se pontos na mesma altura e espaçados de forma regular. Se os posicionarmos muito perto (*pixel* a *pixel*), o efeito será o de criar um segmento de reta:

```
point(50, 50);
point(50, 51);
point(50, 52);
point(50, 53);
point(50, 54);
point(50, 55);
point(50, 56);
point(50, 57);
point(50, 58);
point(50, 59);
```

9. 
10. Podemos utilizar outras formas pré-definidas, bastando mais uma vez chamar a função, parametriza-la e executar. Vamos utilizar um triângulo e desenhar mesmo segmentos de reta (sem ser com pontos), com as funções ***triangle(x_1, y_1, x_2, y_2, x_3, y_3)*** e ***line(x_1, y_1, x_2, y_2)***. Note que utilizamos muitas coordenadas, para definir os vértices limites de cada forma (triângulo, 3 e reta, 2):

```
triangle(60, 10, 25, 60, 75, 65);
line(60, 30, 25, 80);
line(25, 80, 75, 85);
line(75, 85, 60, 30);
```

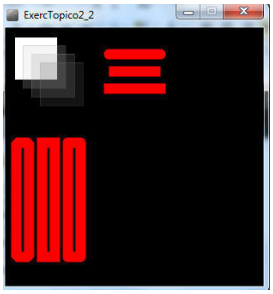
11. 
12. Note que conforme estamos desenhando, há formas que estão ficando escondidas. Isto acontece pois as instruções são executadas de forma sequencial. Para evitar isso, poderíamos trocar a ordem das mesmas ou parametrizar as posições espaciais com valores diferentes…Vamos continuar explorando os métodos para desenhar formas. Testemos agora as funções ***quad(x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4)*** e ***ellipse(x, y, largura,***

altura). A primeira desenha quadrados e suas variantes, com as 4 coordenadas dos 4 cantos como entrada e a segunda, elipses e suas variantes, com x,y identificando a posição central:

```
quad(20, 20, 20, 70, 60, 90, 60, 40);
quad(20, 20, 70, -20, 110, 0, 60, 40);
ellipse(40, 40, 60, 60);
ellipse(75, 75, 32, 32);
```

13. 
14. Para garantir que o seu código não é perdido, salve-o com o nome **ExercTopico2_1**. Vá em *File > Save As*. Notará que é criada uma pasta e o seu programa fonte é gravado com extensão .pde.

Vamos agora explorar um pouco a parte de parametrização desses elementos visuais. Para tal, vamos criar um outro programa de nome **ExercTopico2_2**. Outra pasta será criada e gravado o ficheiro.pde. O resultado final deste exercício será este:



1. Escreva o seguinte trecho de código em seu ficheiro fonte, utilizando a função ***fill (r, g, b)*** que causa o preenchimento dos elementos que sejam declarados a seguir com a cor definida

```
size(300, 300);
rect(10, 10, 50, 50);
fill(204); // Light gray
rect(20, 20, 50, 50);
fill(153); // Middle gray
rect(30, 30, 50, 50);
fill(102); // Dark gray
rect(40, 40, 50, 50);
```

Note que este método pode ser utilizado com cor ou apenas em tom de cinza. Se receber os três parâmetros, será definida uma cor. Se só receber um, será o grau de cinzento. Cada parâmetro deve ter valores entre 0-255.

2. 

3. Vamos substituir a função ***fill*** pela ***stroke(r, g, b)***. Esta serve para definir a cor das bordas dos elementos gráficos e segue a mesma filosofia em termos de parâmetros da ***fill***:

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

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

4. 

5. Poderíamos facilmente combinar os resultados de ambos os efeitos visuais fazendo o seguinte:

```
size(300, 300);
background(0);
rect(10, 10, 50, 50);
fill(204); // Light gray
stroke(102); // Dark gray
rect(20, 20, 50, 50);
fill(153); // Middle gray
stroke(204); // Light gray
rect(30, 30, 50, 50);
fill(102); // Dark gray
stroke(153); // Middle gray
rect(40, 40, 50, 50);
```

6. 

7. Facilmente consegue-se adicionar transparência nas funções ***fill*** e ***stroke***, adicionando um quarto ou segundo parâmetro (designado de alfa), conforme o caso (p. ex. ***fill(r, g, b, alfa), fill(gray, alfa)***). O seu valor varia também entre 0 e 255, e quanto menor, mais transparente é o preenchimento. Para vermos o efeito, adicionemos este parâmetro com valor 50 ao código anterior:

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

Observe que sempre que se ativa uma dada função de controlo de aspecto (cor, grossura e tipo de linha, etc.), todos os elementos gráficos que são desenhados em seguida são afectados pela mesma. Logo, ao definir a cor de preenchimento a vermelho, tudo que de seguida for desenhado, será nessa cor. Para suspendermos alguns efeitos, é possível se utilizarem funções que os desativam. No caso da ***fill*** e ***stroke***, temos as ***noFill()*** e ***noStroke().***

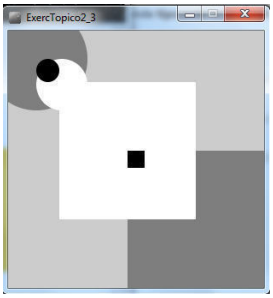
8. Vamos agora alterar um pouco o aspecto do que é desenhado. Para tal incluamos este código onde aparece a função ***smooth()*** que garante suavidade no desenho de linhas, ***strokeWeight(tamanho)*** que define a espessura da linha que é desenhada e ***strokeCAP(tipo)*** que define como as extremidades da linha são desenhadas. O parâmetro tipo pode ser definido por **ROUND**, **SQUARE** ou **PROJECT**:

```
stroke(255,0,0);
smooth();
strokeWeight(12);
strokeCap(ROUND);
line(120, 30, 80, 30); // Top line
strokeCap(SQUARE);
line(120, 50, 80, 50); // Middle line
strokeCap(PROJECT);
line(120, 70, 80, 70); // Bottom line
```

9. 
10. Por fim, alteraremos o aspecto de como os segmentos de reta que são utilizados para desenhar as figuras geométricas são conectados entre si. A função ***strokeJoin(tipo)*** permite fazer isso, sendo que tipo pode ser **BEVEL**, **MITER** ou **ROUND**:

```
smooth();
strokeWeight(12);
strokeJoin(BEVEL);
rect(12, 133, 15, 133); // Left shape
strokeJoin(MITER);
rect(42, 133, 15, 133); // Middle shape
strokeJoin(ROUND);
rect(72, 133, 15, 133); // Right shape
```

Para terminarmos os exercícios, vamos agora explorar um pouco a alteração do comportamento de algumas funções face a invocação desta com determinados parâmetros pré-definidos. Salve o programa e crie um terceiro ficheiro de código fonte com o nome **ExercTopico2_3.pde**. O resultado final será este:



1. Vamos alterar o modo de desenho da elipse com a função ***ellipseMode(tipo)***, onde **tipo** pode ser **RADIUS**, **CORNER** ou **CORNERS**:

```
size(300, 300);
smooth();
noStroke();
ellipseMode(RADIUS);
fill(126);
ellipse(33, 33, 60, 60); // Gray ellipse
fill(255);
ellipseMode(CORNER);
```

```
ellipse(33, 33, 60, 60); // White ellipse
fill(0);
ellipseMode(CORNERS);
ellipse(33, 33, 60, 60); // Black ellipse
```

A orientação da elipse fica alterada, pois dependendo da opção do tipo, os parâmetros são utilizados em outra ordem para alterar a orientação da forma.

2. Vamos alterar o modo de desenho do rectângulo da mesma forma com a função ***rectMode(tipo)***, onde **tipo** pode ser **CENTER**, **CORNER** ou **CORNERS**:

```
noStroke();
rectMode(CORNER);
fill(126);
rect(140, 140, 160, 160); // Gray ellipse
rectMode(CENTER);
fill(255);
rect(140, 140, 160, 160); // White ellipse
rectMode(CORNERS);
fill(0);
rect(140, 140, 160, 160); // Black ellipse
```

Explore outras formas e outras combinações dos exercícios que foram feitos. Note como a ordem afeta o resultado final, bem como a parametrização que faça. Faça a actividade formativa proposta para esta actividade e participe do fórum para colocar as suas questões.
Por fim, poderá encontrar alguns exemplos no YOUTUBE de programação criativa com formas básicas geométricas:
https://www.youtube.com/watch?v=JE-tLmp6UMs
https://www.youtube.com/watch?v=Mm2nwYdhrfg
https://www.youtube.com/watch?v=bdRSuKr9UXY

Até agora utilizamos várias funções prontas e pré-definidas do processing (disponíveis nas bibliotecas por carregadas por defeito) para facilmente desenhar elementos diferentes na nossa janela de visualização. Para isto, basta invocar o nome da mesma, passando-lhe os parâmetros necessários (e na ordem certa), caso esta necessite de algum (ns).Caso desejássemos utilizar funções mais específicas e "exóticas" teríamos que adicionar ao ambiente a biblioteca que a contivesse.

Nos próximos tópicos, iremos ver como se faz isso, bem como, começaremos a criar nossas próprias funções. ☺

6. Publicação do programa

Depois de terminar a criação de seu programa, poderá publica-lo como uma aplicação *stand alone* ou por exemplo, uma *applet*. O processing permite configurar o ambiente de produção para outros modos além do JAVA. Dependendo do modo que esteja ativado, ao acionar FILE → EXPORT, poderá gerar uma publicação em modo diferente do seu programa. Note que os demais modos (JavaScript, Android, etc.) deverão ser ativados no seu ambiente antes que possa efetivamente utiliza-los.

Veja maiores detalhes aqui: http://wiki.processing.org/w/Export_Info_and_Tips

Não esqueça que apesar de esta disciplina lhe ensinar a programar em **processing**, o resultado deve ser sempre com um objectivo criativo.

Tarefas

- Com base nos exemplos facultados neste tópico, tente altera-los, alterando os parâmetros de entrada de determinados elementos, seu estilo ou ainda cor. Coloque-os em ordem diferente e veja o que acontece.
- Poderá consultar a tabela de cor RGB por exemplo aqui: http://pt.wikipedia.org/wiki/Anexo:Tabela_de_cores. Utilize-a para alterar o valor da cor nos exemplos.

Programação Criativa – Tópico 3

***Temática**: Interação e estruturas de controlo e repetição em programação*
***Duração**: semanas 3, 4, 5 e 6*

Actividade 3: Aprender a desenvolver interacção nos programas

Competências a desenvolver:

- O significado de setup() e draw()
- Interação com o rato
- Variáveis de memória
- Estruturas de controlo e repetição

1. *Setup() e Draw()*

Existem dois "esqueletos" de funções que você pode facilmente utilizar no processing: a *setup()* e a *draw()*. Elas estão pré-definidas e prontas para receber no seu interior blocos de código que você especifique. Elas são tratadas de forma especial pelo processador, uma vez que o programa comece a ser executado:

- *setup()* tem seu código executado **apenas 1 vez**, logo no arranque do programa;
- *draw()* tem o seu código **executado a taxa de refrescamento dos quadros** (*frame rate*, que pode ser configurado, permitindo controlar melhor as vezes em que esta função é repetida). Logo, todo o seu código é vezes sem fim executado durante o programa. Daí resulta uma sobrecarga de processamento, e devemos ter cuidado com o que especificamos dentro desta!).

Note que a função *draw()* é uma estrutura de repetição implícita. Em programação, existem várias estruturas que controlam a repetição, como veremos mais adiante. No caso do *draw()* é **desaconselhável utilizar qualquer uma**, pois devemos olhar o código que está dentro dela, como já estando dentro de uma estrutura desse tipo…

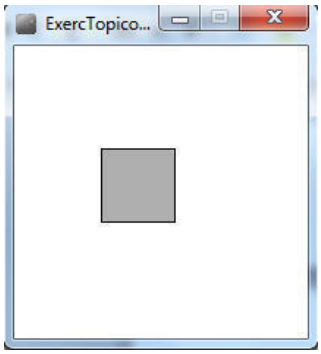
2. *Posição do rato*

A interacção com o utilizador final de seu programa deve sempre utilizar algum dispositivo com este intuito: rato, teclado, ecrã táctil, etc. Num programa podemos aceder esses dispositivos de forma privilegiada, obtendo desde a posição actual do mesmo, até verificar o seu estado (clicado, arrastado, etc.). No caso do processing, ele actualiza a informação sobre o rato de muitas formas, permitindo uma fácil utilização da mesma. A informação sobre o rato fica disponível em variáveis de sistema ou ainda estão disponíveis funções (como o caso de *setup()* e *draw(*) dedicadas para gerir os eventos do rato, dentro das quais você pode escrever o código que quiser:

- mouseButton
- mouseClicked()
- mouseDragged()
- mouseMoved()
- mousePressed()
- mousePressed

- mouseReleased()
- mouseWheel()
- mouseX
- mouseY
- pmouseX
- pmouseY

Vamos agora escrever passo-a-passo um código e ver o que vai acontecendo ao longo dele. É importante que codifique e vá dando "run" a medida que se pede. O resultado visual final será este, na primeira parte do código:



1. Abra o **processing**
2. Defina o tamanho da janela de visualização (caso contrário, ela terá um tamanho pequeno de 200 x 200) com a função ***size(largura, altura)***. Desta vez, vamos utilizar a função pré-definida ***setup(),*** garantindo que esta linha de código é executada apenas 1x no arranque do programa:

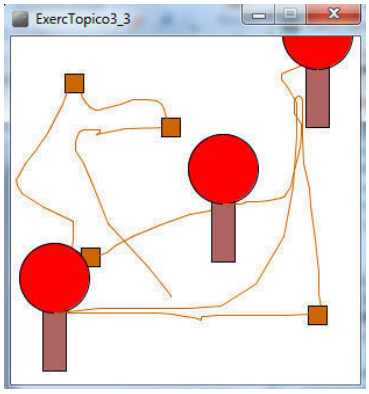
```
void setup() {
  size(300,300);
}
```

3. Vamos agora completar o que queremos que seja executado a cada refrescamento da janela, nomeadamente, pintar o fundo de branco (***background***), alterar a grossura do traço (***stroke***) e desenha um rectângulo, com parametrização central (***rectMode*** e ***rect***):

```
void draw() {
  background(255);
  stroke(0);
  fill(175);
  rectMode(CENTER);
  rect(mouseX,mouseY,50,50);
}
```

4. 
5. Se você mexer o rato, o rectângulo se moverá de acordo com a posição deste. ***mouseX*** e ***mouseY*** são **variáveis do sistema** que são constantemente actualizadas com as posições X e Y do rato. Se você suspender a limpeza do fundo, o quadrado vai deixar um rasto (comente a linha: `// background(255)`). Experimente! Não esqueça de salvar com o nome de ExercTopico3_1.pde e ExercTopico3_2.pde.

Vamos analisar um segundo exemplo. O resultado visual é este:



1. Crie um segundo *sketch* com o nome de ExercTopico3_3.pde
2. Defina o tamanho da janela de visualização para 300 x 300. Coloquemos o fundo a branco e regule a taxa de refrescamento dos quadros para 30 por segundo com ***frameRate(30).*** Esta taxa não é a melhor, mas permite uma boa performance a nível de processamento. Para evitar o desagradável efeito escada no desenho das retas[1], vamos invocar a função ***smooth().*** Esse código deve ser executado apenas 1x no arranque do programa:

```
void setup() {
  size(300,300);
  background(255);
  frameRate(30) ;
  smooth();
}
```

3. Vamos fazer com que ao mexer o rato, vá sendo desenhada uma linha continuamente, com base na posição do rato (***line(pmouseX ,pmouseY ,mouseX ,mouseY)***). Essa linha tem uma cor acastanhada definida com ***stroke***:

```
void draw() {
  stroke(204, 102, 0);
  line(pmouseX ,pmouseY ,mouseX ,mouseY);
}
```

4.  e mova o seu rato…
5. De seguida, vamos fazer que ao clicar o rato, vá aparecendo pequenos rectângulos na posição clicada (***rect(mouseX,mouseY,16,16)***). Como existe a função pré-definida ***mousePressed(),*** vamos tirar partido dela, e todo o código que vai fazer isso ficará dentro dela. Ela é invocada automaticamente pelo sistema sempre que o rato é clicado:

```
void mousePressed() {
  stroke(0);
  fill(204, 102, 0);
  rectMode(CENTER);
  rect(mouseX,mouseY,16,16);
```

[1] O efeito escada ocorre por causa do ecrã ser uma matriz de pontos rectangular. Qualquer forma ao ser desenhada tem que ser adaptada nessa matriz de pontos, daí, resultando em efeitos visuais desagradáveis. A técnica anti-aliasing é utilizada para minorar o efeito visual causado por esse motivo.

```
}
```

6.  e mova o rato, clicando de vez em quando…
7. Agora vamos fazer algo mais interessante. Toda vez que o botão central do rato rodar, vai aparecer uma figura naquela posição. Vamos utilizar a função pré-definida ***mouseWheel()*** para conter o código que fará isso. O desenho da figura será também condicionada pela posição do rato com ***rect(mouseX,mouseY,20,100)*** e ***ellipse(mouseX ,mouseY -30,60,60)*** :

```
void mouseWheel() {
  stroke(0);
  fill(175, 100, 100);
  rect(mouseX,mouseY,20,100);
  stroke(0);
  fill(255, 0, 0);
  ellipse(mouseX ,mouseY -30,60,60);
}
```

8.  e mova o rato, clicando de vez em quando e rodando o botão central…
9. Por fim, vamos garantir que a todo momento conseguimos limpar a janela, fazendo uma espécie de "reset". Vamos utilizar a função pré-definida ***keyPressed()*** que é invocada automaticamente toda vez que alguma tecla é premida (qualquer uma):

```
void keyPressed() {
  background(255);
}
```

10. 

Podemos inclusive tornar o código mais elaborado, fazendo testes com as funções que indicam se o rato foi ou não premido, por exemplo, e caso isso ocorresse, executar um determinado bloco de código. Antes de fazermos isso, vamos ver em mais detalhe as estruturas que permitem que você controle o fluxo de seu programa e as variáveis de memória.

3. Variáveis de memória

Para a execução de um programa ser flexível, é necessário que ele possa ser configurado de formas diferentes durante a sua execução. As estruturas de controlo ajudam muito nesse sentido, mas para que elas sejam realmente flexíveis, é necessário que aquilo que elas controlam possa variar ao longo da execução. Nesse ponto, é importante esclarecer melhor o que são variáveis de memória.

O seu computador disponibiliza a memória RAM para você a utilizar como um caderno de rascunho através de um mecanismo designado de variáveis de memória. As variáveis permitem que você armazene e altere informação na memória de seu computador durante a execução de um programa. Na declaração temos que lhes dar um nome (que tem que obedecer certas regras) e definir a sua dimensão (pode ser um *array*, uma matriz, ou ainda possuir apenas 1 dimensão) e tipo (pode conter números decimais, inteiros, texto, etc.). Uma vez a variável declarada, podemos-lhe atribuir valores, fazendo com que estes fiquem

associados a variável de memória. Quando o programa é executado, a execução de cada declaração faz com que seja reservado um endereço de memória específico.

Os nomes das variáveis não podem conter espaços em branco, serem apenas compostos de números, ou ainda serem nomes reservados pelo sistema operativo e linguagem de programação (p. ex. *mouseX* ou *else*). Devemos começar sempre por letras minúsculas e criar nomes que sejam significativos para a utilização da variável. Se o nome for composto por mais de um, devemos colocar os demais iniciando com maiúscula: p. ex. contadorCiclos, sinalPositivoMaior, etc.

Os tipos podem ser os seguintes:

- ***boolean***: true or false
- ***char***: um caracter, ‘ a ’ ,‘b ’ ,‘c ’ , etc.
- ***byte***: números de –128 a 127
- ***short***: números de –32768 a 32767
- ***int***: números inteiros de –2147483648 a 2147483647
- ***long***: números muito grandes
- ***float***: números decimais 3.14159
- ***double***: números decimais com muitas casas decimais

Alguns exemplos:

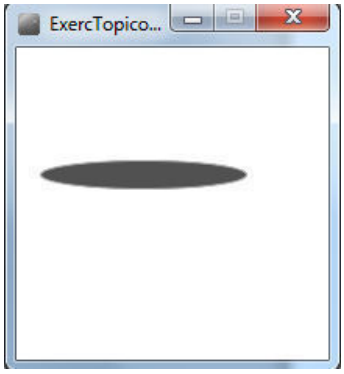
```
int count = 0; // nome count, atribuído valor 0
char letter = ' a'; // Declara um caracter com valor 'a'
double d = 132.32; // nome d do tipo double valor atribuído 132.32
boolean happy = false; // Variável de nome feliz com valor false e do tipo booleana
float x = 4.0; // Do tipo float, nome x, inicializada com valor 4.0
float y; // Declarada com nome y do tipo y, sem valor atribuído (está vazia)
y = x + 5.2; // Variável y conterá o valor da variável x + 5.2, ou seja, 9.2
float z = x * y + 15.0; // Declarada uma variável de nome z do tipo float que conterá 4.0 x 9.2 + 15.0
```

Por fim, a **localização da variável num programa pode torna-la visível a todo este** (todas as funções existentes no código) **ou apenas localmente a uma determinada função**. Quando ela é declarada no início do código, antes de qualquer função, ela é dita **global**, e portanto, visível em qualquer ponto do código. Quando ela é declarada dentro de uma determinada função, ela é dita **local**, e só existe dentro da função. Caso a queira aceder fora desta, é dito que a variável saiu de âmbito, e como resultado, ela não está disponível, pois ao sair de âmbito, é apagada da memória RAM.

Uma pergunta que surge é a de quando devemos utilizar variáveis de memória. Uma das grandes razões já foi referida atrás: **flexibilidade**. Podemos criar uma (ou mais) variável de memória e inseri-la em vários pontos de nosso código, atribuindo-lhe algum valor ao declara-la, ou fazendo com que o seu valor seja calculado a cada ciclo de repetição em função de alguma expressão, por exemplo (para se obter animação, tem que se utilizar variáveis de memória, p. ex.). Outra grande razão é sermos práticos. Imagine que o seu código vai utilizar várias vezes o valor 10. Você terá que escrever esse valor em todos os lugares que o utilizar. Se criar uma variável de memória para conter esse valor, evita possíveis erros de escrita e reduz o

processamento (hoje em dia, isso já não pesa muito, mas no passado, evitar sobrecargas desnecessárias era fundamental!).

Vamos executar um exemplo com variáveis. O resultado final será dinâmico, pois como temos variáveis dentro do programa, conseguimos alterar os valores contidos pelas mesmas durante a execução. O resultado em dado momento será este:



1. Crie um outro *sketch* com o nome de ExercTopico3_4.pde
2. Vamos declarar as variáveis que iremos utilizar. Como queremos que elas sejam globais (ou seja, visíveis a todas as funções do programa), as colocamos logo no início do código. São todas do tipo *float* e são inicializadas com valores em simultâneo a sua declaração:

```
float circleX = 0;
float circleY = 0;
float circleW = 50;
float circleH = 100;
float circleStroke = 255;
float circleFill = 0;
float backgroundColor = 255;
float change = 0.5
```

   Na execução deste código, o seu computador irá reservar na memória RAM 8 "caixinhas" cujo tamanho é apropriado para conter um *float* e o endereço, fica associado ao nome que foi-lhe dada.

3. Vamos inicializar a janela com tamanho 300 x 300 *pixels* e eliminar o efeito escada:

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

4.  nada de especial (visível) ainda acontece…
5. Agora, na função ***draw()***, vamos fazer com que um círculo seja desenhado, mas este será continuamente alterado em termos de posição (***circleX*** e ***circleY***) e de dimensões (***circleH*** e ***circleW***), com o auxílio das variáveis de memória. Ao desenho de cada quadro, ***draw***() é executada, e as variáveis vão tendo seus valores aumentados ou diminuídos do valor contido em ***change***, ou seja, 0.5:

```
void draw() {
```

```
// Desenha o fundo e uma elipse
  background(backgroundColor);
  stroke(circleStroke);
  fill(circleFill);
  ellipse(circleX,circleY,circleW,circleH);
// Altera o valor inicial das variáveis
  circleX = circleX + change;
  circleY = circleY + change;
  circleW = circleW + change;
  circleH = circleH - change;
  circleStroke = circleStroke - change;
  circleFill = circleFill + change;
}
```

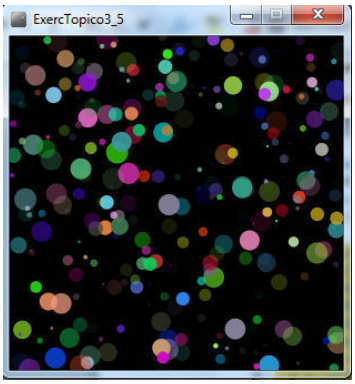
6.  (vai aparecer uma elipse a se mover e "parecendo" que gira sobre si própria…)
7. Salve

Repare que tivemos que mandar continuamente limpar o fundo, de forma a parecer que o círculo se movia (efeito animado). Se você suspender a função ***background***, verá que o que está acontecendo é o desenho sucessivo de elipses em posições e com dimensões gradualmente alteradas a cada repetição de ***draw()***.

Como já referido antes, existem variáveis pré-definidas e prontas para você utilizar. Elas são designadas de variáveis do sistema. Sua informação é automaticamente actualizada e mantida pelo próprio sistema podendo ser muito útil em muitas ocasiões. Vejamos algumas das disponíveis (existem muitas mais…):

- ***width*** —largura em pixels da janela de visualização.
- ***height*** — altura em pixels da janela de visualização.
- ***frameCount*** —Número de quadros processados.
- ***frameRate*** —Taxa de processamento de quadros por segundo.
- ***screen.width*** —largura em pixels do ecrã.
- ***screen.height*** — altura em pixels do ecrã.
- ***key*** —a tecla que foi premida por ultimo no teclado do computador.
- ***keyCode*** —o código numérico da tecla premida.
- ***keyPressed*** —Retorna verdadeiro se alguma tecla foi premida, caso contrário, falso.
- ***mousePressed*** — Retorna verdadeiro se o botão do rato foi premido, caso contrário, falso
- ***mouseButton*** —Indica qual botão foi premido no rato: esquerdo, direito ou central.

Um bom exemplo de necessidade da utilização de variáveis é com a função ***random()***. Esta função gera valores inteiros ou com casa decimal de forma aleatória. Os limites dos valores são definidos consoante o parâmetro que se passa. Se não utilizarmos variáveis de memória para conter o valor retornado a cada chamada de ***random()*** perderíamos a sua funcionalidade. Vejamos o exemplo seguinte:



1. Crie um outro *sketch* com o nome de ExercTopico3_5.pde.
2. Vamos declarar as variáveis que iremos utilizar. Faremos com que sejam também globais. São todas do tipo *float* e não são inicializadas, pois teremos o seu conteúdo carregado durante a execução, de forma dinâmica:

```
float r;
float g;
float b;
float a;
float diam;
float x;
float y;
```

3. Vamos inicializar a janela com tamanho 300 x 300 *pixels*, colocar o fundo a preto e eliminar o efeito escada:

```
void setup() {
  size(300,300);
  background(0);
  smooth();
}
```

4. Por fim, em ***draw()*** vamos utilizar a função ***random()***, fazendo com que o valor da cor (carregado em ***r***, ***g*** e ***b***), da transparência (***a***), da posição (***x*** e ***y***) e o diâmetro da elipse (***diam***), sejam todos aleatórios, mudando a cada execução de ***draw()***. No caso da cor, colocamos 255 para garantir que os valores serão entre 0-255 para cada componente. O diâmetro será entre 0-20 pixels e a posição sempre em função da largura e altura da janela de visualização em pixels (utilização de duas variáveis do sistema, ***width*** e ***height***).

```
void draw() {
  r = random(255);
  g = random(255);
  b = random(255);
  a = random(255);
  diam = random(20);
  x = random(width);
  y = random(height);
  noStroke();
  fill(r,g,b,a);
  ellipse(x,y,diam,diam);
}
```

5.  e salve.

4. *If-then-else e switch*

Qualquer programa, para poder ser versátil, permite que o seu fluxo de execução siga caminhos diferentes, consoante as condições que se defina. Por exemplo, podemos querer que o traço a ser desenhado pelo rato passe a ter cor amarela quando o botão for clicado, ou ainda que tudo na janela seja apagado quando a posição do rato for superior a uma determinada posição. Podemos criar as condições mais variadas possíveis para alternar o fluxo de execução do código e com isso, mais uma vez, ganhar **flexibilidade**.

Para que isto seja possível, temos que definir "caminhos" diferentes dentro do nosso código. Esses caminhos são controlados por estruturas *if-then-else* e condições lógicas (booleanas) que são lhe associadas. Quando definimos as condições, temos que garantir que o resultado final é algo que pode ser avaliado como verdadeiro ou falso. As expressões que podem ser utilizadas são muitas. Podemos utilizar os seguintes operadores relacionais nas mesmas:

- Maior que
- < Menor que
- >= Maior ou igual
- <= Menor ou igual
- == igual
- != diferente

Podemos também combinar as expressões lógicas que vão ser testadas, utilizando operadores lógicos como o "E" (&&), "OU" (||) ou "NÃO" (!).

Podemos utilizar variáveis de memória ou o valor retornado por funções nessas expressões, por exemplo, pois isso permite dar flexibilidade ao teste, dependendo do resultado que a variável ou função comportem. Exemplificando:

- x < 10 → resulta em verdadeiro se a variável x contiver um valor menor que 10
- y != 20 → resulta em verdadeiro se a variável y contiver um valor diferente de 20
- mousePressed → resulta em verdadeiro se o botão do rato for clicado
- (x<10) && (y!=20) → resulta em verdadeiro se ambas as expressões forem verdadeiras
- (x<10) && (y!=20) → resulta em verdadeiro se pelo menos uma das expressões for verdadeira
- !(x<10) → resulta em verdadeiro sempre que a expressão resulte em falso

Note ainda que você pode construir testes aninhados, ou seja, *if-then-elses* dentro *de if-then elses*. Vamos ver alguns exemplos disso em código. Começamos com mais um exemplo simples da utilização desta estrutura de controlo.

Outra estrutura de controlo possível de utilizar é a ***switch-case***. Ela é muito mais rígida que o ***if-then-else***, pois não aceita a utilização de expressões relacionais e/ou lógicas. Na ***switch*** os

valores têm que ser exactos e iguais as situações case que ela oferece. Sua sintaxe é a seguinte:

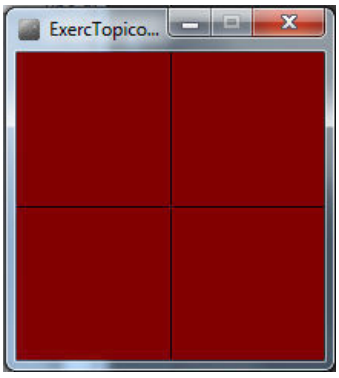
```
switch (<variável>){
      case <valor1>:
            instrução1;
            instrução2;
            instruçãon;
            break;
      case <valor2>:
            instrução1;
            instrução2;
            instruçãon;
            break;
      case <valor3>:
            break;

      default:
            instrução1;
            instrução2;
            instruçãon;
            break;
}
```

Um exemplo:

```
switch (n){
      case 0:
            ++i;
            break;
      case 1:
            --i;
            break;
      case 2:
            i*=2;
            break;
      case 3:
            i/=2;
            break;
      default:
            i = 0;
            break;
}
```

Dependendo do valor contido na variável ***n***, uma das 4 situações será executada. Caso seja **igual** a 0, a variável ***i*** será incrementada de 1, se for **igual** a 1, decrementada de 1, se for **igual** a 2, ***i*** é dobrado, se for **igual** a 3, é dividido por 2, e se **nenhum desses casos** ocorrer, é atribuído zero a ***i***. Note a utilização do ***break*** sempre no final de cada ***case***. Isso faz com que os testes terminem e ele saia fora do ***switch***.

Uma estrutura ***switch*** pode ser substituída por um conjunto de ***if-then-elses***, não sendo a recíproca verdadeira. No próximo exemplo, iremos utilizar a ***if-then-else***.



1. Crie um novo *sketch* e nome de ExercTopico3_6.pde
2. Vamos definir como usual um tamanho de janela 300 x 300 *pixels*, e criar 3 variáveis globais para controlo da cor (***r***, ***g*** e ***b***), inicializando-as com valor igual a zero:

```
float r = 0;
float b = 0;
float g = 0;
void setup() {
      size(200,200);
}
```

3. Desenharemos duas linhas, cujas dimensões são balizadas pelas variáveis de sistema que informam a largura e altura da janela de visualização. A cor que será utilizada para pintar o fundo, dependerá da posição do rato. Dentro da estrutura de controlo, fazemos testes para verificar se a posição horizontal do rato é maior do que a metade da largura da janela de visualização. Se isso ocorrer a componente vermelha da cor do fundo é incrementada, caso contrário, decrementada. Utilizamos a função ***constrain (<variável>, <limite mínimo>, <limite máximo>)*** para garantir que os valores não ultrapassam os limites aceitáveis para cada componente de cor:

```
void draw() {
      background(r,g,b);
      stroke(0);
      line(width/2,0,width/2,height);
      line(0,height/2,width,height/2);
      if(mouseX > width/2) {
            r = r + 1;
      } else {
            r = r - 1;
      }

      r = constrain(r,0,255);
      g = constrain(g,0,255);
      b = constrain(b,0,255);

}
```

6. E mova o rato…

7. Vamos agora fazer com que a componente azul também varie se a posição vertical do rato ultrapassar um valor igual a metade da altura da janela de visualização. Acrescente este trecho de código em ***draw()*** antes da linha de código ***r = constrain(…)***:

```
if (mouseY > height/2) {
      b = b + 1;
} else {
      b = b - 1;
}
```

8.  E mova o rato…

9. Por fim, vamos inserir o controlo para o caso de algum botão ser premido no rato. Se for, a componente verde é incrementada, caso contrário, decrementada:

```
if (mousePressed) {
      g = g + 1;
} else {
      g = g - 1;
}
```

10.  E mova o rato premindo o botão do mesmo…

Agora você começa a ver que o céu é o limite para a sua criação ☺

Com pouco código, podemos facilmente criar resultados diferentes no nosso programa utilizando a estrutura de controlo *if-then-else*, pois conseguimos criar "caminhos" diferentes de execução no nosso programa de acordo com as condições que definamos. Além disso, com a utilização das variáveis de memória, conseguimos variar o valor guardado na memória de nosso computador, conseguindo flexibilizar os resultados que se obtém.

Vamos elaborar um pouco mais. Iremos agora percorrer um exemplo que utiliza *if-then-elses* aninhados e condições lógicas mais elaboradas.

1. Crie um novo sketch e nome de ExercTopico3_7.pde
2. Vamos definir um tamanho de janela 200 x 200 *pixels*:

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

3. Colocaremos o fundo a branco e desenharemos duas linhas para dividir a janela em 4 áreas.

```
void draw() {
  background(255);
  stroke(0);
  line(100,0,100,200);
  line(0,100,200,100);
  noStroke();
```

```
}
```

4. 

5. Agora vem a parte mais "complexa" do código. Queremos que toda vez que o rato esteja posicionado num determinado quadrante, o mesmo fique a preto. A solução exige que monitoremos a posição do rato constantemente, e de acordo com a sua posição, seja desenhado naquele quadrante um rectângulo negro, com dimensões e localização condizente com a do próprio quadrante. Como já deve ter percebido, o teste lógico a ser feito terá que testar em simultâneo 2 coisas: a posição horizontal e a posição vertical. Ativaremos de início a cor negra, para que tudo que seja desenhado depois, fique nessa cor. Inclua este código antes do parêntesis de fecho de ***draw():***

```
fill(0);
if (mouseX < 100 && mouseY < 100) {
    rect(0,0,100,100);
}
```

5.  Notará que toda vez que move o rato numa posição que seja inferior a 100 pixels na horizontal e vertical, aparece o rectângulo negro no 1º quadrante…Como faremos para fazer com que os demais quadrantes também fiquem a preto, caso o rato se posicione dentro de seus limites? O raciocínio é o mesmo, apenas mudando os limites contra os quais testamos. Inclua este código abaixo, em seguida ao anterior:

```
if (mouseX > 100 && mouseY < 100) {
     rect(100,0,100,100);
}
 if (mouseX < 100 && mouseY > 100) {
          rect(0,100,100,100);
 }
 if (mouseX > 100 && mouseY > 100) {
          rect(100,100,100,100);
 }
```

6. 

7. Verá que agora ao mexer por cada quadrante aparecerá o fundo preto, que é o rectângulo que está sendo desenhado. Surge entretanto um pormenor no código que deve ser melhorado. Note que toda vez que uma das condições *if* é satisfeita, não há necessidade das demais serem testadas. Da forma como o código está, forçosamente **todos** os *ifs* serão testados, independente de apenas 1 satisfazer a condição a cada instante. O que fazemos nesses casos é utilizar *if-then-elses* aninhados. Dessa forma, quando um dos *ifs* for aprovado, os demais são descartados. É uma boa maneira de reduzir tempo de processamento e tornar o programa mais eficiente. Reescrevamos o código com esse objectivo:

```
if (mouseX < 100 && mouseY < 100) {
    rect(0,0,100,100);
} else if (mouseX > 100 && mouseY < 100) {
          rect(100,0,100,100);
       } else if (mouseX > 100 && mouseY < 100) {
                  rect(100,0,100,100);
             } else if (mouseX < 100 && mouseY > 100) {
                        rect(0,100,100,100);
```

```
} else if (mouseX > 100 && mouseY > 100)
        {
                rect(100,100,100,100);
        }
```

8.  E salve o programa.

9. E se quiséssemos que o quadrante ficasse a preto apenas quando clicássemos o rato? Uma solução possível seria adicionar o teste dessa condição nas expressões lógicas dos *ifs*, ou seja, ficaria algo como***: (mouseX < 100 && mouseY < 100 && mousePressed)***. Outra solução, melhor, seria inserir todos os *ifs* dentro de um *if* que testasse se o rato foi premido ou não:

```
if (mousePressed) {
    if (mouseX < 100 && mouseY < 100) {
      rect(0,0,100,100);
    } else  if (mouseX > 100 && mouseY < 100) {
…
}
```

10. E se para além disso, quiséssemos alternar a cor de azul para vermelho, consoante o clicar do rato, ou seja, clica fica a cor a vermelho, clica novamente, a cor fica a preto? Teríamos que criar uma variável para funcionar como semáforo. Essa variável poderia ser do tipo booleana (verdadeiro ou falso) e ter o seu valor trocado cada vez que o botão fosse clicado. Consoante o seu estado, a cor ativa era a azul ou vermelha. Para isso, bastaria utilizar o operador lógico NÃO para esse efeito. Teríamos que utilizar a função pré-definida ***mousePressed()***, que é invocada automaticamente toda vez que um evento "clicar rato" acontece. Dentro dela, faremos a mudança do valor da variável semáforo. Por fim, utilizaríamos a variável pré-definida de sistema ***mousePressed***, que contém sempre verdadeiro na ocorrência desse evento. Vejamos as alterações no código necessárias (a vermelho para evidenciar as alterações):

```
boolean semaforo;
void setup() {
  size(200,200);
  semaforo = true;
}
void draw() {
  background(255);
  …

  noStroke();

  if (semaforo) fill(0, 0, 255);
  else fill(255, 0, 0);

   if (mousePressed) {
      if (mouseX < 100 && mouseY < 100) {
        rect(0,0,100,100);
      } else  …


    }
}
void mousePressed() {
```

```
        semaforo = !semaforo;
    }
```

Para terminar a exploração da utilização de estruturas de controlo em programação, vamos implementar um último exemplo, que terá como objectivo, criar uma janela em que se divide a mesma em duas cores. Estas vão se alternando gradualmente na cor de forma automática e cíclica. Além disso, toda vez que mexemos o rato sobre a mesma, um pequeno rectângulo branco percorre as bordas da janela. A velocidade do rectângulo é controlada pelas teclas UP e DOWN do teclado (aumenta e diminui). Como evidente, teremos que criar várias variáveis de memória para controlar a situação, além de utilizar funções pré-definidas que são invocadas nos eventos "clica rato" ou "premir de tecla".

Para dividirmos a janela, iremos desenhar 2 retângulos, cujas dimensões e localizações são definidas de acordo com a dimensão em pixels da janela. Em termos da cor, sabemos que ela pode variar de 0-255, logo a alternância da cor **deverá obedecer estes limites**. O que faremos é incrementar e decrementar as variáveis de memória que definem a cor do lado direito e esquerdo da janela, ou seja, do rectângulo à direita e à esquerda. Quando uma aumenta, a outra diminui, e vice-versa. Utilizaremos um artifício muito comum em matemática e consequentemente, em programação: **utilização de uma variável de incremento**, cujo sinal, alterna. Isto é, fazemos com que ela ora seja positiva, ora seja negativa, e com isso, ela permite aumentar ou diminuir o valor da variável que define a cor do rectângulo. Testando os limites da cor, saberemos se os mesmos foram ultrapassados, e de seguida, faremos a correcção do sinal da variável de incremento, multiplicando-a por -1 (**o que causa a troca de polaridade da variável de incremento**…).

Por fim, para desenharmos o rectângulo que anda a roda na janela, **teremos que controlar em que lado da janela ele está**. Se estiver nas bordas superiores ou inferiores, apenas a componente X de sua posição deve se alterar. Como queremos que ande no sentido horário, teremos que definir valores de incremento convenientes para não cair fora dos limites da janela (consoante o tamanho da mesma) e que X seja incrementado quando andar na borda superior e decrementado, quando na borda inferior. No caso de estar nas bordas direita e esquerda da janela, só a componente Y deverá ser alterada, incrementando-a quando estiver na borda esquerda e decrementando-a em caso contrário. Note que teremos que controlar se os limites não são ultrapassados, e o comportamento do cálculo deverá ser ligeiramente diferente, consoante a borda. O controlo poderia ser baseado no teste das coordenadas x e y do rectângulo, mas podemos colocar uma solução mais elegante, e incluir uma variável, designada de ***estado***, que tem seu valor alterado, consoante a borda que o rectângulo está. A utilização de estruturas *if-then-else* aninhadas, aumenta a eficiência do código.

Por fim, para mais legibilidade e flexibilidade no código, iremos criar duas funções para encapsular os códigos que tratam essas duas partes. Criaremos as funções ***oscilaCor()*** e ***moveRetângulo()*** para segmentar bem o que se faz e quando. O resultado final é esse. Vamos ver o código:

1. Crie um novo sketch e nome de ExercTopico3_8.pde
2. Vamos declarar várias variáveis globais para controlarmos tudo que precisamos, inicializando-as com valores convenientes:

```
float c1 = 0; // cor do rectângulo à esquerda
float c2 = 255; // cor do rectângulo à direita
float c1dir = 0.1;// incrementos da cor
float c2dir = -0.1;
int x = 0; // coordenadas do pequeno retângulo
int y = 0;
int vel = 5; // velocidade base do retângulo
int estado = 0; // indicador da borda
boolean semaforo = false; // indicador do status do rato
int inc = 0; // incremento para a velocidade
```

3. Vamos definir um tamanho de janela 300 x 300 *pixels*:

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

4. Definiremos a função oscilaCor():

```
void oscilaCor(){
  noStroke();
  fill(c1,0,c2); // c1 é a componente r e c2 a b da cor

// desenha rectângulo à esquerda com dimensão consoante
// tamanho da janela
  rect(0,0,width/2, height);
  fill(c2,0,c1); // c1 é a componente b e c2 a r da cor

// desenha rectângulo à direita com dimensão consoante
// tamanho da janela
  rect(width/2,0,width/2,height);

// actualiza os valores de c1 e c2
  c1 = c1 + c1dir;
  c2 = c2 + c2dir;

// se c1 ou c2 estiverem for a dos limites da componente da
// cor, é invertido o sinal da variável de incremento
  if (c1 < 0 || c1 > 255) c1dir *= -1;
  if (c2 < 0 || c2 > 255) c2dir *= -1;
}
```

Note que utilizamos uma forma reduzida para a operação de multiplicação. Em programação, podemos definir uma operação de soma, multiplicação, divisão ou subtracção simples com uma variável das seguintes formas:

```
a = a + 1; ou a += 1;
a = a - 1; ou a -=1;
a = a * 2; ou a *= 2;
a = a / 3; ou a /= 3;
```

5. Para que oscilaCor() seja executada, teremos que invoca-la em setup() ou draw(). Neste caso, como queremos que ela seja sempre executada, será colocada em draw():

```
void draw() {
 oscilaCor();
}
```

6. 

Verá que a cor oscila de forma gradual de vermelho a azul, alternando os lados. Se quiser que seja mais rápido, faça o incremento maior…

7. Vamos agora definir outra função importante:

```
void moveRetangulo()
{
  fill(255);
// A velocidade base é incrementada ou decrementada
// dependendo do valor de inc, que é alternado com as
// teclas UP e DOWN
  vel = vel + inc;

// Recolocado a zero, pois se não fosse, como é um ciclo,
// iria acumular…
  inc = 0;
// Garantia de que o valor não cai acima de 10 e abaixo de
// 0
  constrain(vel, 10, 0);

// o rectângulo é continuamente desenhado, deixaria rastro
// se oscilaCor() não estivesse sendo executada…
  rect(x,y,10,10);

  if (estado == 0) {
    x = x + vel;
// significa que está no limite da borda superior
    if (x > width-10) {
      x = width-10;
      estado = 1; // altera o estado
    }
  } else if (estado == 1) {
            y = y + vel;
// significa que está no limite da borda à direita
            if (y > height-10) {
                y = height-10;
                estado = 2; // altera o estado
            }
          } else if (estado == 2) {
                    x = x - vel;
// significa que está no limite da borda inferior
                    if (x < 0) {
                      x = 0;
                      estado = 3; // altera o estado
                    }
                  } else if (estado == 3) {
                            y = y - vel;
// significa que está no limite da borda à esquerda
                            if (y < 0) {
                            y = 0;
```

```
                    estado = 0;
                }
            }
        }
```

8. Para concluirmos, teremos que incluir as funções pré-definidas para sinalização de eventos:

```
void mouseMoved()
{
  semaforo = true; // se o rato se mover, fica verdadeira
}

void keyPressed()
{
// Coloca inc com valor positivo ou não consoante o premir
// das teclas UP e DOWN
  if (key == CODED) {
    if (keyCode == UP) {
      inc = 1;
    } else if (keyCode == DOWN) {
      inc = -1;
    }
  }
}
```

9.  E mova o rato, premindo as teclas UP e DOWN…

5. Ciclos de repetiçã: for e while

Um outro controlo importante em programação é a repetição de troços de código. A função ***draw()*** já executa automaticamente uma repetição de todo o código que contenha (depende do número de quadros por segundo…). Porém, existem situações em que desejamos mesmo, que um determinado e específico código seja repetido mesmo dentro de ***draw()***. Por exemplo, como faria se quisesse desenhar vários círculos ou quadrados concêntricos na janela? A única diferença da instrução a dar seriam os diâmetros da circunferência, ou a dimensão do quadrado. Se não utilizarmos uma estrutura de repetição, seremos obrigados a escrever várias vezes a mesma instrução, variando o incremento. Se utilizarmos uma, a única coisa a variar será o incremento, que poderá ser definido por uma variável de memória!

Toda vez que o seu código apresentar conjuntos de instruções muito semelhantes e repetidas, é uma boa indicação que deverá converter esse código para um formato mais flexível (utilizar variáveis de memória) e inseri-las dentro de alguma estrutura de repetição.

Outro detalhe é a localização da estrutura de repetição. Se você a colocar dentro da função ***setup()***, ela será repetida apenas as vezes que você definiu para controlo do ciclo. Se for colocada em ***draw()*** a situação é diferente, pois o ciclo se repetirá a cada quadro. Isso torna o processamento mais pesado, mas pode ser necessário, dependendo dos resultados visuais desejados. Se colocar em outra função, dependerá se esta é invocada por ***setup()*** ou ***draw()***.

O processing oferece duas estruturas para criar ciclos de repetição, ou melhor, iterações no seu código. A ***for*** e a ***while***. Ambas são controladas por expressões lógicas, cujo formato

obedecem o mesmo que se utiliza nas estruturas ***if-then-else***. É muito usual utilizarmos variáveis de memória também no controlo destas estruturas. Podemos criar expressões bastante elaboradas, combinando operadores relacionais e lógicos. Dentro delas, podemos colocar qualquer instrução que desejemos, e incluir outras estruturas de repetição (***while***, ***for***) ou/e de controlo (***if-then-else***). Por fim, também podemos ter estruturas aninhadas de repetição, que devem ser utilizadas com muita cautela!

A sintaxe da while é a seguinte:

```
While (<expressão de controlo>) {
      Instrução1;
      Instrução2;
      …
      Instruçãon;
}
```

Onde a expressão de controlo pode ser elaborado com a combinação de operadores lógicos e relacionais, incluindo mais de uma variável de memória se necessário. A estrutura ***while*** executa o seu conteúdo enquanto a expressão de controlo for verdadeira.

A sintaxe da ***for*** é diferente, pois ela funciona como um contador. Ela utiliza **sempre uma variável de memória para a contagem**. Quando uma determinada condição é alcançada, ela pára de repetir o seu conteúdo. A condição normalmente envolve o teste do valor actual da variável utilizada para a contagem. Essa variável é utilizada normalmente em algumas das instruções contidas no ciclo ***for***.

```
For (<variável> = <valor inicial>; <expressão de controlo>;
<actualização da variável>){
      Instrução1;
      Instrução2;
      …
      Instruçãon;
}
```

Vamos percorrer um exemplo para entender melhor como elas funcionam. O exemplo seguinte desenha 4 figuras geométricas.

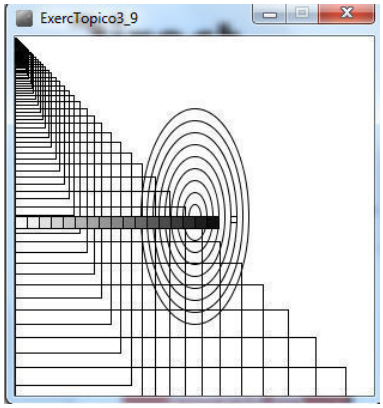
A primeira será um rectângulo cuja posição em ***x*** varia de 20 a 180, mantendo o valor de ***y*** constante em 150 (*height*/2) e uma largura e altura de 5 *pixels*. A posição é variada dentro do ciclo ***for*** em função da variável contadora ***i***. Ela varia de 0 até 9 e é incrementada de 1 unidade a cada ciclo (***++i***).

A segunda figura será uma elipse, cuja posição é sempre a mesma, mudando o seu diâmetro, que aumenta a cada ciclo. Novamente a variável ***i*** é utilizada para alterar esse valor a cada ciclo de repetição. Ela irá variar de 0 a 9. O ***while*** testa o valor de ***i*** a cada ciclo, sendo que dentro dele, a variável é incrementada de 1 unidade a cada repetição. Repare que no caso do for, dentro da própria estrutura isso já ocorre, não sendo necessário incluir uma instrução para esse efeito.

A terceira figura é também um rectângulo, mas cuja posição e dimensão é variada dentro do ciclo com a variável ***j***. Essa variável terá valores de 1 até 290.99 no máximo. A cada ciclo ela é actualizada com o seu valor anterior multiplicado por 1.1.

Por fim, a terceira figura também é um rectângulo, cuja cor é alterada através do ciclo ***for***, com a variável c variando de 255 até 1, no máximo. A cada ciclo, ela é actualizada com o seu valor anterior subtraído de 15. A sua posição em ***x*** também é alterada dentro do ciclo, sendo incrementada de 10 a cada passagem.

Vamos ao passo-a-passo do código:



1. Crie um novo sketch e nome de ExercTopico3_9.pde
2. Vamos utilizar apenas a função ***setup()***, pois como temos ciclos e não queremos que sejam repetidos várias vezes dentro de ***draw()***. O efeito final seria outro (poderá testar por si mesmo!). Colocaremos o fundo a negro, sem preenchimento e contornos a branco. Vamos utilizar variáveis globais, isto é, variáveis que só existem dentro do âmbito da função em que elas são declaradas, neste caso, ***setup():***

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

  background(255);
  stroke(0);
  noFill();

// Variáveis locais a setup(), todas do tipo inteiro
  int i, x= 0, c;
}
```

3. Vamos definir o primeiro ciclo para desenho da primeira figura, ou seja o rectângulo. Insira o código logo abaixo da declaração das variáveis:

```
  for (i = 0; i < 10; i++) {
    rect(i*20,height/2, 5, 5);
  }
```

4. 

5. Para desenho da segunda figura, definimos um novo ciclo de repetição, porém, como vamos utilizar a variável ***i***, ela tem que ser limpa, pois ao final do primeiro ciclo, ela está com valor 10! Insira este código após o anterior.

```
i = 0;
while (i < 10) {
  ellipse(width/2,height/2, i*10, i*20);
  i++;
}
```

6. 

7. A terceira figura, volta a utilizar um ciclo ***for***, porém com precisão decimal. Note que podemos declarar a variável contadora dentro da própria estrutura de repetição ***(float j***). Neste caso, a variável é também local, **porém ao ciclo**…isto significa que ao terminar o ciclo, ela sai de âmbito e é apagada da memória! O mesmo não acontece com ***c, i*** ou ***x***, pois elas são locais a função ***setup(),*** só sendo apagadas quando a função termina a sua execução ☺:

```
// j é local a este for…
for (float j = 1.0; j < width; j *= 1.1) {
   rect(0,j,j,j*2);
 }
```

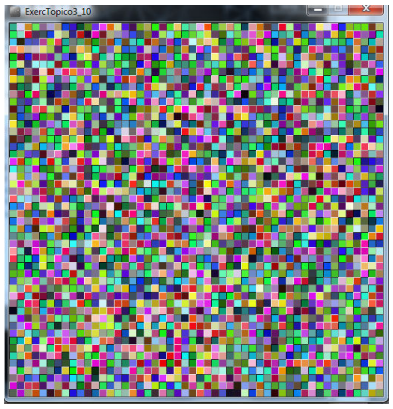
8. 

9. Por fim, vamos definir a última figura, que é também um rectângulo. Ele muda de core de posição no sentido horizontal. Execute o código todo e o salve. Crie a função ***draw()*** e mova o código para dentro dela e veja o resultado visual. Só conseguirá obter o mesmo, se mover para ***draw()*** também a função background:

```
for (c = 255; c > 0; c -= 15) {
  fill(c);
  rect(x, height/2, 10, 10);
  x = x + 10;
}
```

Para concluirmos este tópico faremos um último exemplo. Imagine que deseja desenhar uma janela estroboscópica com vários rectângulos, que mudam de cor sistematicamente. Como podemos fazer?

Uma possível saída será desenhar vários rectângulos ao longo da janela (linha a linha), que são repetidamente actualizados com uma cor de preenchimento escolhida de forma aleatória. Para desenhar os rectângulos, podemos utilizar duas estruturas de repetição aninhadas. Uma controla a posição horizontal, que deve variar de 0 até a largura total da janela enquanto a outra estrutura, controla a variação da posição vertical do rectângulo, fazendo-a variar de 0 até a altura da janela de visualização. Como queremos que o efeito seja estroboscópico, isto é, sempre se actualizando, devemos coloca-las dentro de ***draw()***. Para o efeito aleatório, podemos utilizar a função ***random***, já nossa conhecida. Vamos ver o exemplo:



1. Crie um novo sketch e nome de ExercTopico3_10.pde
2. Vamos utilizar apenas a função ***setup()*** para definir a dimensão da janela:

```
void setup() {
  size(500,500);
}
```

3. Dentro de ***draw(),*** declaramos os dois ciclos for aninhados. Assim, conseguimos ter o seguinte efeito: ***x*** = 0, ***y*** varia de 0 até ***height***; x =1, ***y*** varia de 0 até ***height***, ***x*** = 2, idem…:

```
  void draw() {
// Variáveis locais para conter a cor
  float r, g, b;

// x vai dar saltos de 10 em 10
  for (int x = 0; x < width; x+=10) {

// y vai dar saltos de 10 em 10, sendo que para cada
// ciclo de x, o ciclo de y realiza-se por completo!
    for (int y = 0; y < height; y+=10) {

// Utilizamos random para aleatoriedade
      r = random(255);
      g = random(255);
      b = random(255);
      stroke(r);
      fill(r, g, b);
      rect(x, y, 10, 10);
    }
  }
```

4. 

Tarefas

- Considerando o último exemplo dado, se quisermos outros efeitos, poderíamos alterar a forma para círculos, por exemplo. A dimensão da figura, também causaria um efeito visual final diferente. Poderíamos mexer também na transparência, fazendo com que esta variasse de forma a alguns rectângulos ficarem invisíveis ou parcialmente visíveis.

Portanto, as combinações são várias, dependendo da imaginação ☺ Faça isso e veja o resultado obtido!

Programação Criativa – Tópico 4

***Temática**: Funções, objectos, transformações 2D e arrays*
***Duração**: semanas 7, 8, 9 e 10*

Actividade 4: Aprender a utilizar funções de uma forma abrangente, criar objectos e arrays.

Competências a desenvolver:

- Funções com passagem de parâmetros e retorno de valor
- Criação e utilização de objetos
- Utilização de *arrays* para gerir os dados do programa
- Transformações espaciais 2D

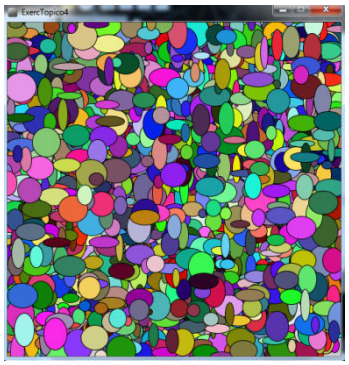
1. Funções

Já falamos sobre funções no início desta formação, e estivemos utilizando-as nos exemplos e exercícios até agora apresentados e propostos. Entretanto, falta vermos mais exemplos, e mais complexos e completos de sua utilização.

As funções são muito úteis para conseguirmos obter **modularidade** e **reutilização** de código. Imagine que possui muitas linhas idênticas e semelhantes dentro de ***setup()*** ou ***draw()***. Uma solução pode ser remete-las para dentro de estruturas de repetição, como já vimos. Mas e se o troço de código puder ser reutilizado em outras funções? Ou se ele tiver um objectivo e conceito completo dentro de si? (desenhar um determinado objecto ou componente específico de sua visualização). Coloca-lo dentro de uma função pode ser a melhor solução.

As funções podem receber parâmetros de entrada e retornar valores (resultados) no seu ponto de chamada. Isso pode ser muito útil também. Imagine que quer desenhar um conjunto variável de bolas e no final quer saber quantas bolas foram com a componente vermelha diferente de zero. Como fazer isso?

Uma solução inteligente seria criar uma **função específica no seu código para desenhar bolas**. Ela receberia como parâmetro de entrada o número total de bolas que quisesse desenhar a cada vez. No desenho das bolas, dentro da função, poderia utilizar a função ***random()*** para atribuir a cor de forma aleatória e testar de seguida para saber se a cor tem componente vermelha diferente de 0. Se fosse, contaria, para uma variável contadora. No final da execução, retornaria esse valor no ponto de chamada, e poderia desenhar um quadrado ao centro da janela cujo tamanho era calibrado pelo número de vezes que a cor vermelha tivesse sido utilizada…Vamos implementar?



1. Abra o **processing** e crie um programa de nome **ExercTopico4_1.pde**
2. Vamos começar por criar a ***setup()*** e definir o ***stroke***:

```
void setup() {
  size(500,500);
  stroke(0);
}
```

3. Vamos fazer com que tudo seja aleatório. Criamos uma variável local ***n*** para conter o número de bolas que será definido de forma aleatória por ***random***. Repare na utilização de ***(int)*** antes de ***random***. Isso ocorre, pois ***random*** retorna sempre um valor do tipo ***float***. **Se não forçarmos a conversão do valor retornado para inteiro**, dará erro (espaço de memória diferentes…), pois estaremos querendo colocar algo do tipo ***float***, dentro de algo que é do tipo ***int***!! Essa conversão é designada de ***cast*** em programação, e é muito útil em situações desse tipo, ou seja, quando temos tipos que são parcialmente compatíveis (numéricos, no caso), mas por razões de dimensão do espaço de memória reservado, estão incompatíveis (***float*** exige espaço maior que ***int***…). Invocaremos a função para desenho das bolas (***desenhaBolas(n)***)só quando o botão do rato é premido. Esse controlo é feito pela estrutura ***if*** e a variável de sistema ***mousePressed***. No caso de o botão ser premido, calculamos a posição e tamanho do rectângulo a partir do centro, em vermelho. A posição e tamanho variam consoante o número de bolas desenhadas (na expressão, utilizamos ***n*** para calcular estes valores). Os valores são colocados nas variáveis locais ***x1, y1, x2*** e ***y2***:

```
void draw() {
    int n = (int)random(4, 50);// gera valores entre 4 e 50
    if (mousePressed) {
      println(desenhaBolas(n)); // invoca desenhaBolas
      fill(255, 0, 0); // ativa a cor vermelha
      float x1 = 0.5 * (width - n);
      float y1 = 0.5 * (height - n);
      float x2 = 0.5 * n;
      float y2 = 0.5 * n;
      rect(x1, y1, x2, y2);// desenha o retângulo
  }
}
```

4. Vamos definir ***desenhaBolas***. A função vai receber como entrada um parâmetro do tipo inteiro, que indica o número de bolas a desenhar. Para podermos ter a cor e as dimensões das bolas totalmente aleatórias, vamos ter que definir variáveis locais, de

forma a conter os valores nelas. Também precisaremos de uma variável contadora para controlo do número de bolas criadas com a componente ***r*** da cor diferente de 0. A utilização de um ciclo ***for***, garantirá que se repetirá o desenho as ***n*** vezes necessárias, passadas na entrada. Por fim, vamos retornar (***return <valor>***) ao ponto de chamada o total contado para r diferente de 0:

```
int desenhaBolas(int n) {
    float r, g, b, x, y, w, h;
    int cont = 0;// sempre inicializado a 0 o contador

// Repetirá n vezes o desenho de bolas
  for (int i = 0; i < n; ++ i) {
// Atribuição aleatória as dimensões, posições e cor
    r = random(255);
    g = random(255);
    b = random(255);
    x = random(0, width);
    y = random(0, height);
    w = random(10, 50);
    h = random(10, 50);

    fill(r, g, b);
    if (r!=0) ++ cont; // Contabiliza r diferente de 0
    ellipse(x, y, w, h);
  }
  return cont; // retorna valor
}
```

5.  clique o botão do rato várias vezes…

Há alguns detalhes importantes a referir nas funções com parâmetros e com valores retornados:

1. Garanta que o **tipo declarado para a função é igual ao do valor retornado**: se ela é ***int***, deve retornar algo do tipo inteiro, se for ***boolean***, deverá ser do tipo lógico, etc.
2. Garanta que os **parâmetros passados são condizentes com o número e tipo** de parâmetros expectados e a **ordem**, correta!
3. O valor retornado pode ser utilizado para se carregado numa variável de memória, ou outra operação qualquer, pois o "nome" da função, equivale ao "valor" retornado pela mesma no ponto de chamada ☺
4. Os valores passados como parâmetros, se em variáveis, **são passados como cópias**…isto significa que eles **não são afectados por qualquer operação** que ocorra com eles dentro da função que os recebeu, permanecendo intactos na sua origem…

Vamos fazer um outro exemplo no seguimento deste, que além de fazer tudo o que este já faz, irá fazer mais uma coisa: desenhar curvas de Bézier aleatoriamente. Para isso, utilizaremos a função pré-definida ***bezier(x1, y1, cx1, cy1, cx2, cy2, x2, y2)***. O detalhe do código está na utilização do valor retornado por ***desenhaBolas*** como parâmetro de entrada para a função ***desenhaCurvas***. Em lugar do desenho das formas ocorrer quando se clicar o botão do rato, irá ocorrer quando se move-lo sobre a janela. Vamos ver o que este exemplo alterou do código do anterior:

1. Crie um programa de nome **ExercTopico4_2.pde**
2. Vamos inserir a nova função, ***desenhaCurvas***. Na mesma, o número de curvas a desenhar vai depender do valor passado como entrada. Este é na verdade o total de vezes que a componente ***r*** em ***desenhaBolas*** foi diferente de 0:

```
void desenhaCurvas(int n) {
   float r, g, b, x1, y1, x2, y2;
   float cx1, cx2, cy1, cy2;

   for (int i = 0; i < n; ++ i) {
      r = random(255);
      g = random(255);
      b = random(255);
      x1 = random(0, width);
      y1 = random(0, height);
      x2 = random(0, width);
      y2 = random(0, height);
      cx1 = random(0, width);
      cy1 = random(0, height);
      cx2 = random(0, width);
      cy2 = random(0, height);
      fill(r, g, b);
      bezier(x1, y1, cx1, cy1, cx2, cy2, x2, y2);
  }
}
```

3. Em ***draw()*** vamos fazer as alterações necessárias. Como vê, o valor retornado por ***desenhaBolas*** é repassado como entrada para ***desenhaCurvas***. A variável global moveu é do tipo lógico e controla o mover do rato:

```
void draw() {
  int n = (int)random(4, 50);

   if (moveu) {
    desenhaCurvas(desenhaBolas(n));
    . . .
  }
}
```

4. A variável ***moveu*** tem que ser declarada e actualizada. Precisamos de utilizar a função pré-definida ***mouseMoved*** para capturar o evento. Vamos proceder as últimas correcções

```
boolean moveu;

void setup() {
  size(500, 500);
  stroke(0);
  moveu = false;
}

void mouseMoved(){
  moveu = true;
}
```

5.  mova o rato várias vezes…salve!

2. *Variáveis Arrays*

Um outro conceito muito importante e útil em programação é o de *array*. Até agora, vimos variáveis unidimensionais, isto é, declaradas de forma a poder conter apenas um determinado valor por vez. Mas o que acontece se tivermos que armazenar e processar conjuntos de dados, todos do mesmo tipo? Imagine que necessita memorizar as três últimas posições no eixo do X do rato. Poderia criar três variáveis para essas posições, ou apenas uma do tipo *array*, com 3 elementos para armazenar as posições em X. Um *array* pode ser composto por vários elementos, armazenando os dados como um vector, e pode ser bidimensional, isto é, armazenar os dados como se fosse uma matriz. Isto significa que para você aceder um determinado elemento terá sempre que indicar em que índice ele se encontra. No caso de ser bidimensional, terá que utilizar dois índices: um a indicar a linha e outro a coluna da matriz. Pode ainda haver *arrays* tridimensionais, mas não iremos aborda-los nesta formação.

Vamos ilustrar como os *arrays* são declarados no processing, detalhando a sintaxe e o que cada declaração causa:

```
// Declara e inicializa um vetor de 3 elementos do tipo inteiro
int [ ] listaPosicoesX = new int[3];

// Declara e inicializa um vetor de 10 elementos do tipo float
float [ ] listaCoresR = new float[10];

// Declara e inicializa uma matriz com número de linhas é igual a
// largura da janela e colunas, a altura
Float [ ][ ] valorTransp = new float[width] [height];

// Carrega o valor 2 para o 1º elemento de listaPosicoesX
listaPosicoesX[0] = 2;
// Carrega o valor 255 para o 3º elemento de listaCoresR
listaCoresR[2] = 255;
// Carrega o valor 50 para o elemento na 3º linha e 5º coluna de
// valorTransp
valorTransp[2][4] = 50;
```

Repare que temos **sempre que inicializar os *arrays***, com a utilização da declaração **new <tipo> [nº elementos]** no caso de vectores e **new <tipo> [linhas][colunas]** antes de fazer uso dos mesmos. Isso serve para que o espaço de memória seja limpo e preparado para receber os dados dali para frente. Se não o fizer, poderá ter erro na execução de seu programa.

Um outro pormenor com *arrays*, é que se utilizará muitas vezes estruturas de repetição para percorre-los. Não irá acontecer sempre, mas grande parte das situações em programação, obriga que se percorra os *arrays* como um todo, visitando elemento a elemento para se avaliar ou calcular algo, por exemplo. Nada melhor do que se utilizar estruturas do tipo ***for*** (e

aninhado, no caso de matrizes), para com isso conseguirmos facilmente gerar os índices e percorrer cada elemento do *array* ☺.

Os *arrays* podem ser de tipos padrão, como já vimos até agora: *boolean, int, float,* etc. Podem também ser *arrays* de objectos. Os objectos são um conceito poderoso em programação que iremos abordar de seguida.

Vamos ver um exemplo com *arrays*. Nesse exemplo, o objectivo será criar uma janela que é preenchida com vários pontos com cores diferente. Para isso, iremos criar 3 matrizes para conter as componentes r, g e b da cor. As matrizes terão um tamanho igual ao da janela de visualização, ou seja, o total de linhas é igual a altura da janela, e o de colunas, é igual a largura da mesma. No início do programa, as matrizes são carregadas com valores aleatórios *r*, *g*, *b* para cada posição. Posteriormente, os elementos das matrizes são acedidos também de forma aleatória, e o ponto a ser desenhado em cada execução de ***draw(),*** tem a cor atribuída em função do resultado da combinação de elementos aleatórios da três matrizes. Como ***draw()*** já é um ciclo por si só, a posição em x e y do ponto é calculada de forma incremental a cada execução. Duas estruturas de controlo ***if-then-else***, permitem calibrar os valores da posição do ponto a ser desenhado. Repare que se ***draw()*** não fosse um ciclo por si só, teríamos que ter utilizado estruturas de repetição (p. ex. ***for***) para obter o mesmo efeito! O preenchimento é feito linha a linha, e recomeçará quando a última linha e coluna da janela for atingida. Ao recomeçar, novas cores serão utilizadas para pintar cada ponto ☺

Vamos ao código:

1. Crie um programa de nome **ExercTopico4_3.pde**
2. Vamos começar por declarar as variáveis globais:

```
// Declaração das variáveis globais
        int cols , linhas;
// Declaração de 3 matrizes, cada uma para uma das componentes da cor
        float[][] r;
        float[][] g;
        float[][] b;
// Variáveis para controlar o acesso as linhas e colunas
        int col = 0, lin = 0;
```

3. Vamos inicializar as variáveis e preencher as matrizes de cor em ***setup():***

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

          cols = width;
```

```
            linhas = height;
// Inicialização dos arrays
            r = new float[cols][linhas];
            g = new float[cols][linhas];
            b = new float[cols][linhas];

// Preenchimento aleatório de cor para cada elemento dos arrays
            for (int i = 0; i < cols; i++) {
              for (int j = 0; j < linhas; j++) {
                r[i][j] = random(0, 255);
                g[i][j] = random(0, 255);
                b[i][j] = random(0, 255);
              }
            }
          }
```

4. Por fim, em ***draw()*** vamos fazer o resto do trabalho:

```
            void draw()
            {
              int x = (int) random(0, cols-1);
              int y = (int) random(0, linhas-1);
// Cor do ponto escolhido em posição aleatória em cada array
              stroke(r[x][y], g[x][y], b[x][y]);
              point(col,lin);

// Controlo da posição do ponto
              if (col < cols) ++col;
              else {
                  col = 0;
                  ++lin;
              }
              if (lin > linhas) lin = 0;

            }
```

5.  Não esqueça de salvar e esperar a execução de ***draw()*** várias vezes, até preencher a janela toda e recomeçar…Tente fazer o controlo do desenho do ponto por dois ciclos ***for*** (aninhados), de forma a janela ser totalmente desenhada a cada execução de ***draw()*** e veja como se altera o resultado visual!

```
            void draw(){
             for (col = 0; col < cols; ++ col)
               for (lin = 0; lin < linhas; ++ lin)
               {
                   int x = (int) random(0, cols-1);
                   int y = (int) random(0, linhas-1);
                   stroke(r[x][y], g[x][y], b[x][y]);
                   point(col,lin);
               }
            }
```

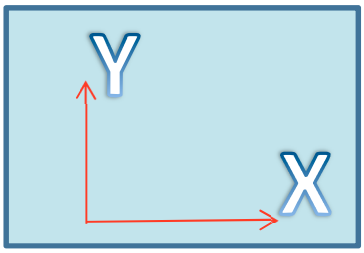
Note que com a utilização das matrizes de cor, definimos apenas uma vez as cores para cada posição da janela, e depois, apenas vamos buscando o que já lá está definido. Isso é muito prático, pois conseguimos definir um "molde de cor" para todos os pontos que desenhamos. Poderíamos criar "moldes" para outras coisas: o tipo de traço, o tamanho, a transparência, etc.

3. Transformações espaciais 2D

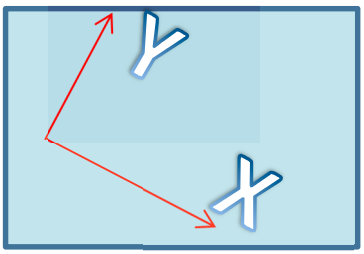
Muitas vezes desejamos que determinados objectos presentes na visualização girem, alterem o seu tamanho ou simplesmente se desloquem. Em última análise, o que ocorre é sempre a alteração da posição do objecto no seu desenho na janela. A translação faz com que ele fique mais acima e/ou abaixo da sua posição original, a alteração de escala, causa um deslocamento mais para frente ou para trás, enquanto a rotação, faz com que a posição que está a frente fique mais acima, abaixo, à direita, ou/e à esquerda, dependendo do ângulo utilizado. As transformações espaciais são uma parte importante e muito útil na construção de cenas gráficas por computador.

Na geometria encontramos todas as **formulações matemáticas** que calculam a **translação**, **rotação** e **alteração de escala** dos objectos. Elas reflectem uma sequência de operações matemáticas que são necessárias para conseguirmos executar a transformação espacial de um objecto. Podemos lançar mão dessas formulações para facilmente calcular as transformações espaciais em 2D e até em 3D, porém isso seria extremamente trabalhoso e torna-se desnecessário, pois as **linguagens de programação oferecem funcionalidades** que permitem executar as **transformações espaciais** de forma cómoda (esse trabalho já foi implementado…). Dessa forma, temos apenas que invocar as funcionalidades apropriadas, passar-lhe os parâmetros necessários e incluí-las nos pontos convenientes de nosso código.

É importante perceber o mecanismo de funcionamento dessas funcionalidades, para fazer bom uso das mesmas. O primeiro conceito relevante é o de **matriz de transformação**. Todas as equações que representam as transformações são representadas numa forma matricial. Essa matriz é **mantida pelo sistema**. Cada vez que aplicamos uma funcionalidade para transformação espacial (translação, alteração de escala ou rotação) de algo em nossa cena, a **matriz é actualizada**, de forma a reflectir a situação actual. As transformações vão sendo aplicadas de **forma acumulativa**. Isto significa que o nosso **sistema referencial vai sendo também afectado ao longo das transformações**, alterando a sua orientação e posição, ou seja:



Limites da janela de visualização



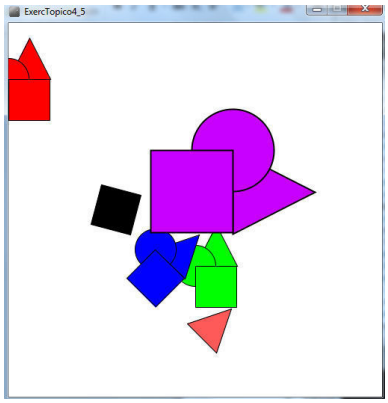
Após uma rotação de 45 graus

Entretanto, para flexibilizar o processo e permitir melhor controlo do que se transforma e como, a matriz transformação **é mantida numa pilha**, ou seja, conforme uma nova versão surge (pois alguma transformação foi solicitada), a versão mais actual fica no topo e a anterior, desce um nível. Isso significa que podemos **desfazer as últimas transformações** solicitadas desempilhando a pilha de matrizes de transformação. Esse mecanismo permite que executemos as transformações de **forma isolada em certos objectos** de uma cena, como veremos mais a frente. A pilha no processing não suporta **mais do que 32 matrizes**, ou seja, só se consegue fazer 32 *pushs* seguidos, sem fazer *pops*, antes que a mesma estoure…

Outro pormenor importante é que as transformações não são comutativas, isto é, a ordem em que são executadas gera um resultado final diferente. Uma translação seguida de uma rotação resulta em uma localização espacial diferente se a rotação ocorrer primeiro que a translação, por exemplo. A **única forma de garantir que a rotação ou alteração de escala não movem o objecto também**, é **posicionar o objecto na origem**, ou seja, a localização (0, 0), executar a transformação, e depois devolve-lo a posição em que estava…

Em termos de funcionalidades, as que vamos utilizar no caso 2D são a ***rotate()***, ***translate()***, ***scale()***, ***pushmatrix()***, ***popmatrix()***, ***shearX()*** e ***shearY()***. A ***pushmatrix*** e ***popmatrix*** permitem que façamos o empilhar e desempilhar da pilha de matrizes de transformação enquanto a ***shear*** executa o cisalhamento do objecto na direcção do eixo X ou do eixo Y. Note que existem outras funcionalidades orientadas para o caso em 3D.

Para compreender melhor a transformação espacial 2D, nada melhor que um exemplo. Vamos desenhar 3 objectos: um rectângulo, um círculo e um triângulo. Vamos criar 3 funções para desenhar cada objecto e de forma a explorar bastante as transformações e conjugação de matrizes de transformação, iremos criar situações isoladas de transformação e também combinadas em todos os objectos da cena.



1. Crie um programa de nome **ExercTopico4_4.pde**
2. Vamos começar por criar as funções para desenhar os objetos:

```
void desenhaTriangulo(float r, float g, float b){
  fill(r, g, b);
  triangle(0, 75, 28, 20, 56, 75);
}

void desenhaCirculo(float r, float g, float b){
  fill(r, g, b);
  ellipse(0, 75, 55, 55);
}

void desenhaRect(float r, float g, float b){
  fill(r, g, b);
  rect(0, 0, 55, 55);
}
```

3. Vamos agora especificar as transformações em ***setup(),*** de forma a ilustrar etapa a etapa as transformações. Começamos por apenas desenhar os objetos sem fazer nada de especial:

```
void setup() {
  size(500, 500);
  background(255);

// Desenha as figuras sem sofrer transformações a vermelho
// note que a posição inicial é a mesma para todos,
// ou seja, o canto superior esquerdo da janela
  desenhaTriangulo(255, 0, 0);
  desenhaCirculo(255, 0, 0);
  desenhaRect(255, 0, 0);
}
```

4. 
5. Insira o código a seguir abaixo da última instrução em ***setup():***

```
// Executa-se uma translação de 250 em X e 250 em Y
// ficando as figuras agora no centro da janela
  translate(width/2, height/2);
// São desenhadas a verde
  desenhaTriangulo(0, 255, 0);
  desenhaCirculo(0, 255, 0);
  desenhaRect(0, 255, 0);
```

6. 
7. E em seguida, este:

```
// Gira 45 graus e desenha todas as figuras novamente
// Elas vão ser deslocadas, pois não estão na origem...
  rotate(PI/4.0);
  desenhaTriangulo(0, 0, 255); // desenha a azul
  desenhaCirculo(0, 0, 255);
  desenhaRect(0, 0, 255);
```

8. 
9. Mais este, que ilustra a utilização do bloco *push-pop*. É utilizada indentação para realçar como as instruções internas estão isoladas das demais transformações:

```
// Isola a transformação no triângulo, empilhando a nova
// matriz no topo da pilha
  pushMatrix();
// Move-o 100 pixels em X e 40 em Y
// Note que o "X" e "Y" já não está referido a uma referência
// paralela aos limites da janela, pois fizemos antes
// uma rotação!!!
    translate(100, 40);
    desenhaTriangulo(255, 90, 90); // desenha a rosa
 // e a desempilhando no final, ou seja, volta a ser
 // a matriz que estava antes desta translação...
  popMatrix();
```

10. 
11. Este também:

```
// Repetimos o processo, agora numa rotação isolada
// do retângulo
```

```
  pushMatrix();
    rotate(PI/3.0);
    desenhaRect(0, 0, 0); // desenha a preto
  popMatrix();
```

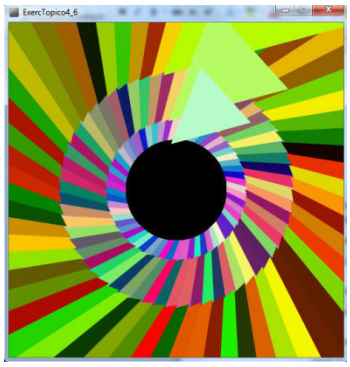
12. 

13. E por fim este:

```
// Fazemos agora 3 transformações seguidas, tendo como base na
// matriz que incluiu a translate(width/2, height/2) e
// rotate(PI/4)
  pushMatrix();
    scale(2.0); // dobramos o tamanho
    rotate(PI/4); // giramos + 45 graus
// Note que a translação tem posição negativa, pois a
// referência já não está alinhada com os limites da janela!!!
    translate(-40, -100); // desenha em roxo
    desenhaTriangulo(200, 0, 255);
    desenhaCirculo(200, 0, 255);
    desenhaRect(200, 0, 255);
  popMatrix();
```

14.  salve…

Para exemplificar melhor a situação de as transformações não mudarem a localização do objecto se este estiver na origem, vamos escrever mais um programa. Neste, faremos um efeito "túnel do tempo", com vários triângulos, com diferentes factores de escala, sendo girados e desenhados continuamente com cores diferentes. No centro, uma esfera negra dá a noção de "fundo". O rectângulo está posicionado na origem e ativamos o modo de desenho considerando a origem no meio deste. Notará que as rotações não o tiram da posição, ao contrário do que ocorre com o triângulo. Se quiséssemos que ele também não saísse do lugar, teríamos que garantir que o centro de gravidade estava também em (0, 0).Vamos reaproveitar o programa anterior, e só fazer algumas alterações em ***setup()*** e definir ***draw().***



1. Crie um programa de nome **ExercTopico4_5.pde**
2. Vamos começar por criar a declarar as variáveis globais e definir a inicialização do programa:

```
float angulo = 0;
float[] b = {0, 100, 200};
```

```
float[] s = {5.0, 3.0, 2.0};

void setup() {
  size(500, 500);
  background(255);
  rectMode(CENTER); // Garante desenho com centro em (0, 0)
  noStroke();
}
```

3. Vamos agora definir ***draw().*** Veja como é feito o encadeamento do *push* e *pop* e o resultado que se obtém. Como o triângulo não tem o seu baricentro na origem, ele se desloca também com a rotação. O mesmo não ocorre com o quadrado:

```
void draw() {
  float r = random(0, 255);
  float g = random(0, 255);

// Posicionado no centro da janela
  translate(width/2, height/2);
  desenhaCirculo(0, 0, 0);
  for (int i = 0; i < 3; ++ i) {
    pushMatrix();
      rotate(angulo); // Rectângulo gira no centro
      desenhaRect(r, g, b[i]);
      scale(s[i]);    // Triângulo, não…
      desenhaTriangulo(r, g, b[i]);
    popMatrix();
  }

  if (angulo > TWO_PI) angulo = 0;
  else angulo += TWO_PI/60; // Gira de 6 em 6 graus
}
```

4. Por fim, as alterações em ***desenhaRect()*** e ***desenhaCirculo():***

```
void desenhaCirculo(float r, float g, float b) {
  fill(r, g, b);
  ellipse(0, 0, 150, 150);
}

void desenhaRect(float r, float g, float b) {
  fill(r, g, b);
  rect(0, 0, 55, 55);
}
```

5.  salve…

4. Objetos

A programação orientada a objectos (POO) foi uma alteração significativa a nível de paradigma que ocorreu nos últimos anos. Antes, utilizávamos apenas a designada programação procedimental, que é a que você tem visto até agora. Todo o código é estruturado dentro de funções/procedimentos, que podem ou não conter parâmetros de entrada e retornar resultados no ponto do chamado. Para "montar" o seu programa, vai definindo as funções, que são organizadas segundo objectivos diferentes, de forma ao "todo" resultar naquilo que deseja. Organizar as funções, determinando o que cada uma resolve e como é a base de uma boa programação.

Na POO é criado o conceito de objecto, com seus atributos e métodos. Cada objecto é definido de forma a organizar o código melhor. Ele pode ser reutilizado em funções ou ainda, por outros objectos. Um objecto é definido por uma classe, que quando instanciada, passa a ser um objecto.

Os objectos possuem métodos (ou funções) especiais, designados de **construtor**, que são **automaticamente invocados** quando o objecto é instanciado (ou criado). Esses métodos não possuem um "tipo" e podem receber ou não parâmetros de entrada. O nome do método **construtor é sempre igual ao do objecto** ao qual ele está associado. Para podermos utilizar objectos, temos que cria-los (instancia-los) utilizando o operador ***new*** (como no caso dos *arrays*).

Criando uma correspondência entre a POO e a procedimental, temos que os métodos dos objectos equivalem a funções enquanto seus atributos, as variáveis globais, visíveis dentro da classe que define o objecto. O construtor desempenha normalmente um papel muito semelhante a função ***setup()*** do processing, ou seja, parametrização inicial dos atributos e outras funcionalidades do objecto.

A POO tem muitas vantagens sobre a tradicional. Permite obter-se uma melhor gestão da informação, encapsula-la, pois podemos conseguir controlar o acesso aos atributos e métodos com a utilização da palavra ***private*** na declaração do atributo ou função. Podemos também obter a **sobrecarga de métodos**, ou seja, criar métodos com nomes idênticos, variando apenas a quantidade e/ou tipo de parâmetros de entrada e tipo do método em si. Isto também é válido para o método construtor.

A classe de um objecto é sempre definida num ficheiro a parte. No ambiente processing deverá adicionar um novo separador (*new tab*) e dar o nome do ficheiro igual ao nome da classe que vai declarar nele, que por fim, é o nome do próprio objecto. A sintaxe para declarar segue o seguinte formato:

```
Class <nome do objecto> {
 atributo1;
 atributo2;
 atributon;
 <método construtor>() {
       atributo1;
       atributo2;
       atributon;
 }
 <tipo> <método1>(<lista de parâmetros>){
       atributo1;
       atributo2;
       atributon;
 }
 <tipo> <métodon>(<lista de parâmetros>){
       atributo1;
       atributo2;
       atributon;
}
```

Por exemplo:

```
Class Carros {
      // Atributos de Carros
      int matricula;
      int serie;
      String marca;
      float [] cor;
      // Métodos constructores, 2 versões
      Carros(){
            cor = new float[3];
            cor[0] = 255;
            cor[1] = 0;
            cor[2] = 0;
            marca = "porsche";
            serie = 345672;
            matricula = 678944;
      }
      Carros(int m, int s, string marc, float[] c) {
            Cor = new float[3];
            matricula = m;
            serie = s;
            cor = c;
            marca = marc;
      }
      // Demais métodos
      int getSerie() {return serie;}
      int getMatricula() {return matricula;}
      void mudaCor(float [] c){
            cor[0] = c[0];
            cor[1] = c[1];
            cor[2] = c[2];
      }
}
```

Ao definirmos um objecto, devemos analisar sempre quais **são as propriedades** do objecto que precisamos de controlar/utilizar e que **acções** queremos que ele execute ou esteja envolvido. Esse raciocínio é importante, pois as **propriedades são os atributos** da classe e as **acções correspondem aos métodos** a implementar. O objecto deve conter em si um determinado âmbito. Por exemplo, considere que temos que gerir a informação sobre alunos em termos escolares. Os atributos típicos de um aluno são o *nome*, a *morada*, o *telefone*, a *matrícula*, o seu *histórico escolar*, e as *propinas* que pagou. As acções que seriam necessárias poderia ser a de *registar o aluno*, *cancelar a matrícula*, *consultar histórico* ou ainda *consultar propinas pagas*. Numa POO o objecto **Aluno** seria definido com esses atributos (variáveis de memória) e métodos (funções) seriam implementados para executar cada uma das acções mencionadas.

Para utilizarmos um objecto, basta declararmos uma variável cujo tipo é a do objecto e utilizar a palavra *new* para instancia-lo. Vejamos o formato da sintaxe:

```
<nome da classe> <variável> = new <método construtor>(<lista de
                        parâmetros);
```

Por exemplo:

```
Carros porsche = new Carros();
Carros fusca = new Caros(45632, 768543, "Fusca", cor);
```

Para invocar os métodos públicos da classe, basta utilizarmos a seguinte sintaxe:

```
<variável>.<nome do método>;
```

Exemplificando com a classe Carros, acima exemplificada:

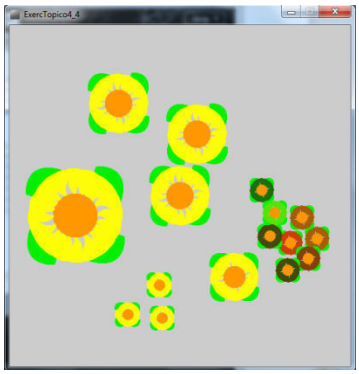
```
porsche.mudaCor(0, 255, 255);
int m = fusca.getMatricula();
```

Por fim, os programas podem **utilizar um número variado** de objectos e um **objecto pode ser composto por outros objectos**. Isso cria uma flexibilidade muito grande no código, além de reaproveitamento. Por exemplo, se considerássemos um objecto para representar uma empresa, ela seria composta por vários objectos do tipo funcionário e do tipo cliente.

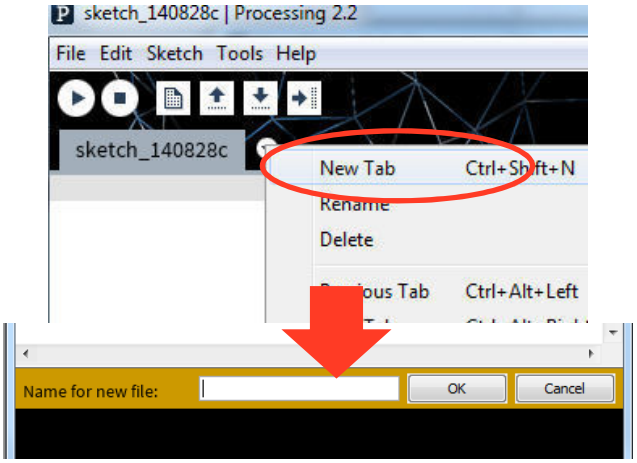
Vamos fazer um exemplo mais complexo para ilustrar passo-a-passo a definição de um objecto. O objectivo será criar um programa que seja capaz de desenhar uma flor a cada clique do rato na janela. Será possível alterar a cor e tamanho da flor utilizando a tecla "C" e o botão giratório do rato respectivamente.

Uma flor é um objecto para nós. No âmbito do objectivo pretendido, precisamos de controlar as propriedades **cor** e **tamanho** da flor. A nível de acções, queremos **desenhar a flor**, que para simplificar, pode ser decomposta em partes, além de **modificar a localização** em que é desenhada (quando o rato é clicado na janela, é onde ela deverá aparecer), **alterar o seu tamanho** (quando movemos o botão giratório do rato para trás ou para frente), ou ainda a sua **cor** (quando premimos a tecla "C"). Ao criarmos o objecto, embora ele possa ter o tamanho e cor modificados, **terão que ser atribuídos valores iniciais para estes**. Isto é uma tarefa para ser executada dentro do **construtor da classe**.

A gestão dos eventos será feita no programa principal, ou seja, o rato e o premir de teclas estará sendo controlado no código principal, e quando algo ocorrer, invoca-se o método apropriado do objecto. O resultado visual do programa é o seguinte:



1. Crie um programa de nome **ExercTopico4_6.pde**. Crie uma nova TAB e dê-lhe o nome de **Flor**:



2. Vá para a nova janela, com o nome de **Flor**. Iremos codificar a classe **Flor** para o efeito desejado. Vamos declarar os atributos para conter o tamanho da flor, a cor de suas pétalas e a cor do seu centro. Utilizaremos *arrays* do tipo ***float*** com 3 elementos para conter as componentes *r*, *g* e *b* respectivamente e uma variável unidimensional também ***float*** para o tamanho:

```
class Flor
{
  // Atributos da classe cor
  float []corPetala = new float[3]; // array com 3 elementos
  float []corCentro = new float[3];
  float tamanho;
```

3. Vamos definir o construtor da classe. O objectivo é garantir que tudo é inicializado com algum valor. O construtor definido vai receber 3 parâmetros de entrada, as cores das pétalas e centro e o tamanho da flor a criar:

```
// Construtor
  Flor (float []corP, float corC[], float t) {
      corPetala[0] = corP[0];
      corPetala[1] = corP[1];
      corPetala[2] = corP[2];
      corCentro[0] = corC[0];
      corCentro[1] = corC[1];
      corCentro[2] = corC[2];
      tamanho = t;
  }
```

4. Para desenhar a flor, vamos utilizar 3 métodos. Um principal (**desenhaFlor**) irá coordenar a montagem da flor com pétalas (**desenhaPetalas**), folhas (**desenhaFolhas**) e centro. Uma flor é composta por várias pétalas, folhas e um centro. As pétalas e folhas são desenhadas em círculo em torno do centro da flor. No caso das pétalas, definimos que elas são desenhadas a cada 6 graus (**TWO_PI/60**), ou seja, dividimos 360 graus (variável do sistema **TWO_PI**) por 60, desenhando 60 pétalas ao todo. No caso das folhas, desenhamos apenas 4 em intervalos de 90 graus (**TWO_PI/4**). Utilizamos ciclos ***for*** para controlar a variação dos graus de rotação. A flor e a folha são desenhadas da mesma forma, variando-se a localização dos vértices. Utilizamos funções de transformação espacial 2D (***scale***, ***rotate***) para facilitar o processo de

alteração de escala e rotação dos elementos gráficos. Adicione este método a seguir do construtor:

```
void desenhaFlor(){
      scale(tamanho); // Altera o tamanho
      for (float ang = 0; ang <= TWO_PI; ang +=TWO_PI/4) {
        rotate(ang); // Gira de 0 a 360°, com incremento de 90°
        desenhaFolha();
      }

      for (float ang = 0; ang <= TWO_PI; ang +=TWO_PI/60) {
        rotate(ang);// Gira de 0 a 360°, com incremento de 6°
        desenhaPetala();
      }

      fill(corCentro[0], corCentro[1], corCentro[2]);
      ellipse(0, 0, 80, 80);
  }
```

5. Falta definir os métodos **desenhaFolha** e **desenhaPetala**. Para desenhar a forma que vai representar a folha e a pétala, utilizamos o bloco ***beginShape()-endShape()*** e as funções ***vertex*** e ***bezierVertex***. Eles nos auxiliam no desenho de formas não regulares. Os parâmetros passados correspondem a posições coordenadas nos eixos X e Y:

```
// Método responsável pelo desenho da folha
  void desenhaFolha() {
    fill(0, 240, 0);
    beginShape(); // Utilização de desenho complexo de formas
      vertex(90, 39); // com beginShape() - endShape()
      bezierVertex(90, 39, 54, 17, 26, 83);
      bezierVertex(26, 83, 90, 107, 90, 39);
    endShape();
  }

// Método responsável pelo desenho das pétalas
  void desenhaPetala(){
     fill(corPetala[0], corPetala[1], corPetala[2]);
     beginShape();
      vertex(30, 20);
      bezierVertex(80, 0, 80, 75, 30, 75);
      bezierVertex(50, 80, 60, 25, 30, 20);
    endShape();
  }
```

6. Por fim, vamos codificar os demais métodos que vão permitir alterar o valor contido em ***tamanho*** e nos *arrays* de cor. No caso de move-la, utilizamos a função ***translate*** para executar a transformação espacial 2D:

```
// Método para alterar o fator de escala
  void alteraTamanho(float t){
    tamanho = t;
  }

  // Método para alterar a posição da flor
  void move(float dx, float dy) {
    translate(dx, dy);
  }
```

```
  // Método para alterar a cor da flor
  void mudaCor(float []corP, float []corC){
     corPetala[0] = corP[0];
     corPetala[1] = corP[1];
     corCentro[0] = corC[0];
     corCentro[1] = corC[1];
  }
} // Fecho da declaração da classe Flor
```

7. Vamos agora completar o programa principal. Salve **Flor** e vá para a *tab* do programa principal. Vamos precisar de 3 variáveis globais. Uma para o objecto flor (**flor**), duas do tipo *array* para a cor das pétalas (**corP**) e centro (**corC**) e outra para o fator (**fator**) de escala:

```
// Variáveis globais
Flor flor;
float [] corP= new float[3];
float [] corC= new float[3];
float fator;
```

8. Em ***setup()*** definimos uma janela de 500 x 500, suavizando o efeito escada e sem contornos nas formas. Também vamos inicializar as variáveis globais, instanciando o objecto **flor**:

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

// Inicializa as variáveis globais de cor e escala
  corP[0] = 253; corP[1] = 253 ; corP[2] = 7;
  corC[0] = 255; corC[1] = 152 ; corC[2] = 0;
  fator = 0.5; // 50% da dimensão normal

// Cria o objeto Flor e posiciona-o no meio da janela
  flor = new Flor(corP, corC, 0.5); // instancia o objeto
  flor.move(width/2, height/2); // posiciona no meio da janela
  flor.desenhaFlor(); // solicita o seu desenho
}
```

9. 

10. Em ***draw()*** vamos garantir que só quando o botão do rato é premido, algo acontece, neste caso, e **flor** a ser desenhada é posicionada na posição actual do rato, actualizado o fator de escala e a cor do objecto **flor**, e finalmente, solicitado o seu desenho:

```
void draw() {
  // Quando o rato é premido, a flor é desenhada na sua posição
   if (mousePressed) {
      flor.move(mouseX, mouseY);
      flor.alteraTamanho(fator);
      flor.mudaCor(corP, corC);
      flor.desenhaFlor();
  }
}
```

11. Para concluir a tarefa, temos que monitorar os eventos de rato e teclado. Vamos utilizar as funções pré-definidas para esse propósito, bem como as variáveis de sistema apropriadas. Acontecendo algo, temos que actualizar a cor ou aumentar ou diminuir o fator de escala. Isso garante que quando clicamos o rato, os valores já estão todos espelhando o *status* dos eventos que eventualmente ocorreram ☺
    No caso do fator de escala, para evitar problemas de cair em valores impróprios, salvaguardamos com a utilização da função ***constrain***:

```
// Quando a tecla C é premida, a cor da flor é alterada
// de forma aleatória
void keyPressed() {
  if (keyCode == 67){
    corP[0] = random(0, 255);
    corP[1] = random(0, 255);
    corC[0] = random(0, 255);
    corC[1] = random(0, 255);
  }
}

// Quando o botão central do rato é girado para frente
// é incrementado o fator de escala de 0.1, sendo decrementado
// em situação contrária.
void mouseWheel(MouseEvent event) {
  float e = event.getCount();
  if (e > 0) fator -= 0.1;
  else fator += 0.1;

// Garante que o fator nunca ultrapassa estes valores
  constrain(fator, 1, 5);

}
```

12.  Guarde e clique o rato em vários locais, rodando o botão giratório e premindo a tecla "C" de vez em quando.

Tarefas

- Crie um programa que crie um objecto de nome **Linhas**. Os atributos do objecto são a sua localização (x, y), a cor, a espessura e a utilização ou não de contornos. O objecto deve conter métodos apropriados que permitam alterar os atributos do objecto quando se deseja, e o desenho de linhas. A posição e a alternância de atributos é feita com o auxílio do teclado e/ou rato (Dica: utilize a posição do rato para definir as posições limites da linha e as funções de gestão de eventos de teclado/rato para controle)

Programação Criativa – Tópico 5

***Temática**: Utilização de imagens, tipografia, transformações em 3D e funções trigonométricas na composição visual*
***Duração**: semanas 12, 13, 14 e 15*

Actividade 5: Aprender a utilizar imagens e texto, além de executar transformações espaciais em 3D e aplicar devidamente algumas funções trigonométricas para obter efeitos visuais originais.

Competências a desenvolver:

- Utilização de imagens
- Transformações espaciais 3D
- Utilização de funções trigonométricas
- Tipografia

1. Utilização de imagens

É muito normal utilizarmos imagens na montagem de determinadas visualizações. A inclusão de uma fotografia pode ser facilmente executada com o auxílio de certas funcionalidades que o processing faculta. Os formatos admissíveis são: GIF, JPG, TGA e PNG. As funções ***loadImage, image*** ou ***createImage*** são alguns exemplos de funções que permitem que carreguemos imagens e manuseemo-las no nosso programa.

Existem também várias funcionalidades que permitem que se **altere o matiz**. A função ***tint*** pode receber como entrada de 1 até 4 parâmetros. Dependendo da quantidade, podemos afetar apenas o brilho da imagem (1 parâmetro), a tonalidade do brilho em termos de cores RGB (3 parâmetros), a tonalidade e transparência (4 parâmetros), ou o brilho e a transparência (2 parâmetros).

Por fim, uma imagem nada mais é do que um conjunto de *pixels* disposto em forma matricial. Se conseguirmos aceder cada *pixel* que a compõe, podemos facilmente alterar o valor deste e com isso, criar filtros ☺ Existem também várias funcionalidades que permitem que operemos a nível dos *pixels* que compõem uma imagem: ***loadPixels***, ***updatePixels***, etc.

Quando trabalhamos com uma imagem, ela deve **estar sempre disponível dentro da pasta** do projeto, pois é a partir daí que o programa procura-a para carrega-la. Outro detalhe importante é que toda imagem **é tratada como um *array* unidimensional** cujo número total de elementos é o número total de *pixels* que a compõe (ou seja, o produto dos valores de sua resolução – p. ex. 1024 x 798). Quando queremos aceder a um determinado *pixel*, temos que ter isso em mente. Temos que ter atenção nas **conversões necessárias para poder correlacionar devidamente uma determinada posição linear nesse *array* com a posição bi-dimensional** (matricial) que a imagem possui quando representada na janela de visualização ☝

Considere que possui uma janela de visualização com uma dimensão de 500 x 500 e uma imagem cuja resolução é 1024 x 798, que é mapeada nessa janela. Com base no que foi referido antes, o *array* de *pixels* da imagem terá 1024 x 798 elementos (817152 no total). Isso significa que a posição 0 no *array* corresponderá a posição x = 1 e y = 1 na janela de visualização, x = 0 e y = 0 na imagem e a posição 817251, a posição x = 500 e y = 500 na janela e x = 1023 e y = 797 na imagem. Como podemos saber qual será a posição no *array* linear da

imagem do *pixel* situado na linha 300 e coluna 200 (x = 200 e y = 300) da janela de visualização? Basta calcularmos da seguinte forma:

*Posição X na imagem = (Posição X na janela * largura da imagem)/ largura da janela*

*Posição Y na imagem = (Posição Y na janela * altura da imagem)/ altura da janela*

*Posição linear no array = Posição X na imagem + Posição Y na imagem * largura da imagem*

Com esse raciocínio em conta, teríamos:

- *Posição X na imagem = (200 * 798)/500 = 319*
- *Posição Y na imagem = (300 * 1024)/500 = 614*
- *Posição linear no array = 319 + 614 * 1024 = 629055* ☺

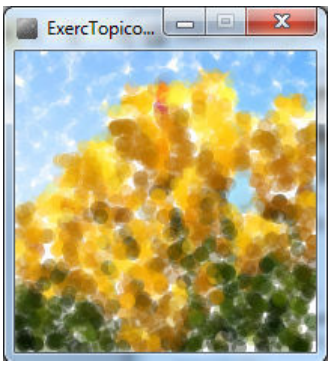
Temos também que ter em conta que as posições X e Y, tanto na imagem como na janela, variam de zero até (largura – 1) ou (altura – 1). Por exemplo, consideremos a última posição na janela, ou seja x = 499 e y = 499, isso daria, arredondando:

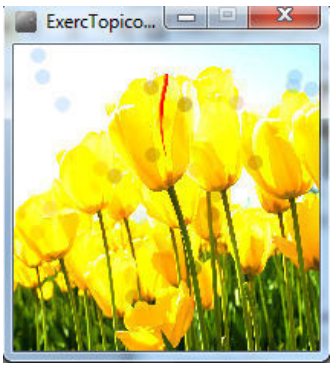
- *Posição X na imagem = (499 * 798)/500 = 797*
- *Posição Y na imagem = (499 * 1024)/500 = 1023*
- *Posição linear no array = 1023 + 797 * 1024 = 817151*

ou seja, o último elemento…

Para incluir a imagem, você pode anexa-la a pasta do projeto ou utilizar a interface do processing, por *Sketch/File Add*.

Vamos percorrer um exemplo, tendo como base filtros disponíveis no próprio *website* do processing. Nesse exemplo, vamos construir um objecto de nome ***ImagemFiltro***. Dentro desse objecto, iremos incluir dois métodos para aplicar dois filtros distintos na imagem passada no construtor do objecto. O primeiro filtro desenha círculos centrados na posição de cada *pixel* da imagem. Os *pixels* são escolhidos de forma aleatória e pintado na cor do *pixel*. O segundo filtro, altera o brilho de cada *pixel* em função da posição do rato na janela quando clicado. Teremos que em ambos os casos executar conversões das coordenadas da janela para as da imagem e vice-versa, pois não vamos utilizar uma janela cuja dimensão seja igual a da imagem. Se assim fosse, não seria necessário calcular essas conversões.





1. Abra o **processing** e crie um programa de nome **ExercTopico5_1.pde**
2. Vamos começar por criar a o objecto ***ImagemFiltro.*** Vamos adicionar mais um ficheiro para a classe, dando-lhe o nome igual ao desta. Dentro dele, vamos adicionar os atributos e métodos necessários:

```
class ImagemFiltro {
  PImage img;
  int pointillize;
// Construtor que recebe como entrada o nome do ficheiro imagem
  ImagemFiltro(String imagem){
    img = loadImage(imagem);// Lê a imagem
  }

// O primeiro filtro desenha círculos com a cor do pixel
  void filtroA(int diametro) {
      // Verifica se o diâmetro enviado é aceitável
      if ((diametro > 16)||(diametro <= 0)) pointillize = 16;
      else pointillize = diametro;

      // Seleção aleatória de um ponto na imagem
      int x = int(random(img.width));
      int y = int(random(img.height));

      // Localização no array linear de pixels da imagem
      int loc = x + y * img.width;

      // Verifica a cor atribuída ao pixel na imagem
      loadPixels();
      float r = red(img.pixels[loc]);
      float g = green(img.pixels[loc]);
      float b = blue(img.pixels[loc]);
      noStroke();

      // Desenha uma elipse na localização, com essa cor
      fill(r,g,b,100);

    // Converte coordenadas do ponto na imagem para as da janela
      int xj = (width * x)/img.width;
      int yj = (height * y)/img.height;

   // Desenha a elipse com dimensão variável, conforme definido
      ellipse(xj, yj, pointillize, pointillize);
    }
// Altera o brilho de acordo com a relação da posição do rato com
// a largura da janela de visualização
// É invocada apenas quando o rato é clicado
  boolean filtroB() {
      loadPixels(); // Força a leitura dos pixels
      for (int x = 0; x < img.width; x++) {
        for (int y = 0; y < img.height; y++ ) {
        // Calcula a posição no array linear de pixels da imagem
          int loc = x + y * img.width;
          // Captura a cor do pixel na imagem
          float r = red   (img.pixels[loc]);
          float g = green (img.pixels[loc]);
          float b = blue  (img.pixels[loc]);
        // Altera o seu brilho de acordo com a posição do rato
          float adjustBrightness = ((float) mouseX / width) *
8.0;
          r *= adjustBrightness;
```

```
                g *= adjustBrightness;
                b *= adjustBrightness;
              // Garante que o RGB cai entre 0-255
                r = constrain(r,0,255);
                g = constrain(g,0,255);
                b = constrain(b,0,255);
    // Altera a cor do pixel correspondente na janela de visualização
                color c = color(r,g,b);

       // Converte coordenadas do ponto na imagem para as da janela
                int xj = (width * x)/img.width;
                int yj = (height * y)/img.height;

                // Calcula posição no array de pixels da janela
                int locj = xj + yj * width;

                pixels[locj] = c;
              }
            }
            // Força atualização dos pixels
            updatePixels();
            return false; // Valor retornado no ponto de chamada
          }
    }
```

3. Por fim, vamos ao programa principal e definir o ***setup()*** e ***draw().*** Teremos também que utilizar a ***mousePressed(),*** pois queremos que o clicar do rato faça o controle da execução dos métodos de filtragem. Utilizaremos duas variáveis globais: uma para ajudar no controlo do evento de clique do rato e outra para o objecto ***ImagemFiltro***:

```
boolean clicou = false;
ImagemFiltro img;

void setup() {
  size(200,200);
  img = new ImagemFiltro("Tulips.jpg");
  background(255);
  smooth();
}

void draw() {
  img.filtroA(10);
  if (clicou) clicou = img.filtroB();
}

void mousePressed() {
    clicou = true;
}
```

4.  Execute e espere um bocado de forma ao primeiro filtro conseguir preencher toda a janela com os círculos. Clique o botão do rato. Salve o projeto.

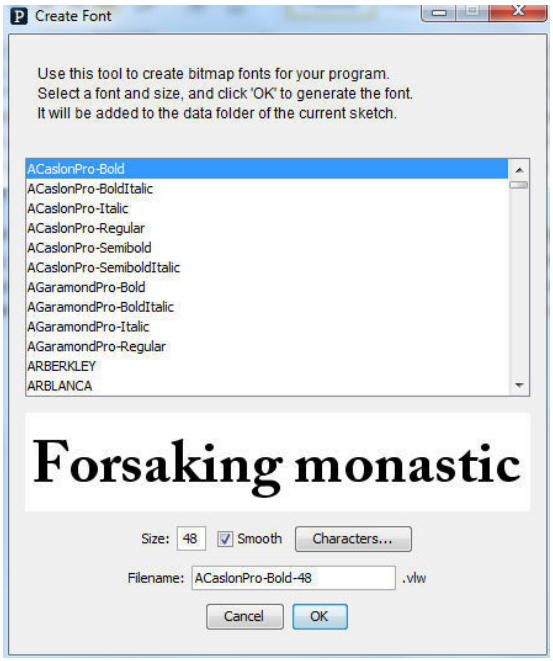
2. Tipografia

Antes de fazermos o próximo exemplo, de forma a abordarmos outras funcionalidades com imagens, vamos também ver o que podemos fazer com texto em termos de tipografia. Em

processing, existe a possibilidade de carregarmos fontes diferentes de texto para utiliza-las na janela de visualização.

Os tipos ***char*** e ***String*** são o que utilizamos quando lidamos com texto, como já visto no início desta formação. No caso das variáveis do tipo ***String***, podemos contar com várias funcionalidades que permitem saber desde o seu tamanho, verificar se duas variáveis ***String*** são iguais, extrair parte da mesma ou ainda verificar qual a letra com que iniciam, entre outras facilidades. Variáveis desse tipo é o que devemos utilizar no caso de explorar a tipografia disponível no processing.

As funcionalidades disponíveis para tipografia são: ***PFont, loadFont(), textFont(), text(), textSize(), textLeading(), textAlign()*** e ***textWidth().*** Porém, antes de utiliza-las, teremos que carregar para o nosso projeto a fonte que desejamos utilizar. Portanto, tendo um projeto aberto, devemos aceder o menu ***Tools/Create Font*** para carregarmos a fonte que desejamos utilizar:



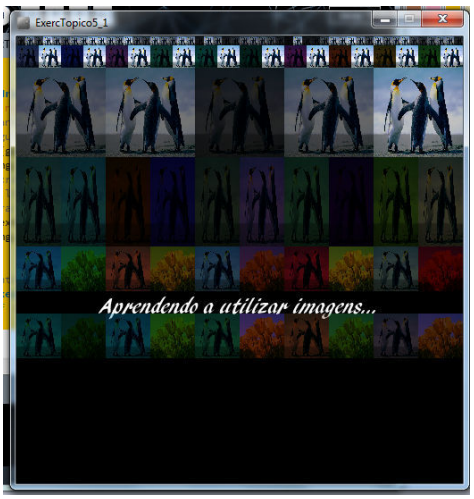
Uma vez a fonte adicionada ao seu projeto, poderá utilizar as funcionalidades para carregar a fonte em memória, posicionar o texto, alinha-lo, etc.

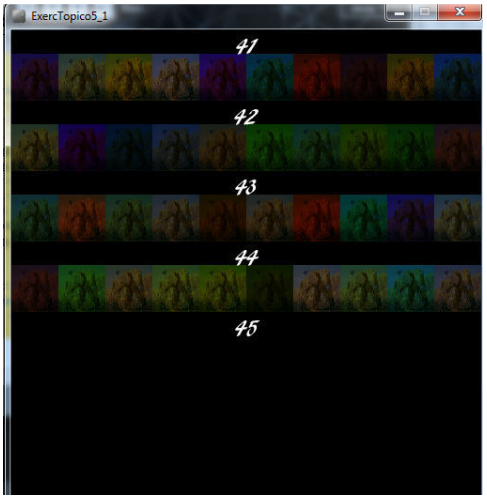
Vamos examinar mais um exemplo. Neste, iremos explorar a utilização de **matizes** em imagens, conjuntamente, com a criação de **fontes (tipografia)** e funcionalidades de texto. O nosso objectivo será o de criar um objecto com o nome ***Imagem*** que irá conter atributos que permitam que ele contenha duas imagens, uma fonte de texto e controle sua localização na janela e alternância de efeitos de matiz.

Serão criadas 5 versões (5 métodos) para apresentação das imagens, sendo que todos se basearam sempre no posicionamento das imagens de cima para baixo e da direita para a esquerda da janela. Todos desenham linhas da janela com imagens afectadas pela funcionalidade ***tint***. É também adicionado um método para escrita de texto e ainda um outro, para ajudar a calcular o posicionamento da próxima linha a ser desenhada, já que o espaço

ocupado varia de linha para linha (dimensão da imagem é variada). Caso a janela seja totalmente preenchida até a sua última linha, a mesma é limpa, e recomeça o desenho por linhas, de cima para baixo.

Por fim, note que o código poderia ser ainda muito mais flexível se utilizássemos *arrays*. Poderíamos em lugar de 2 imagens, permitir que o objecto ***Imagem*** contivesse um número flexível de imagens ou fontes, por exemplo. Poderíamos criar uma quantidade variável de objectos ***Imagem*** em ***setup()***. No exemplo, está fixa essa quantidade, mas sempre é possível tornarmos os nossos programas mais flexíveis e adaptáveis! ☺





1. Abra o **processing** e crie um programa de nome **ExercTopico5_2.pde**
2. Vamos começar por criar a o objecto ***Imagem.*** Vamos adicionar mais um ficheiro para a classe, dando-lhe o nome igual ao desta. Dentro dele, vamos adicionar os atributos e métodos necessários. Note que deverá ter criado anteriormente a fonte **ARBLANCA-25** seguindo os passos anteriormente referidos. Vai ser criada uma pasta com o nome de data e depositado o ficheiro fonte. Vamos a definição da classe ***Imagem***:

```
// Atributos da classe
  PImage imgA, imgB;
  float x, y;
  int inc;
  boolean flag;
  PFont font;
// Construtor com parâmetros
  Imagem (String nomeA, String nomeB){
    x = 0;
    y = 0;
    flag = true;
    imgA = loadImage(nomeA); // Carrega a 1ª imagem
    imgB = loadImage(nomeB); // Carrega a 1ª imagem
    font = loadFont("ARBLANCA-25.vlw");
    textFont(font);
    fill(255);
  }
  PImage getImagemA() { return imgA; }
  PImage getImagemB() { return imgB; }

  // Desenha a imagem A sempre com efeito matiz, apenas
  // afetando o brilho
```

```
  // n indica quantas vezes na posição horizontal atual
  // na janela
  void imagemVersao1(int n) {
    inc = 500/n; // calcula espacejamento
    posiciona(n);
    for (int i = 0; i < inc; ++i) {
  // Brilho calculado aleatoriamente
     tint(random(0, 255));
  // Posiciona a imagem na janela
     image(imgA, x, y, n, n);
     x += n; // atualiza posição vertical
    }
  // Atualiza a coordenada y, posição horizontal
  // para a próxima solicitação de desenho
    y += n;
  }
// Idem, porém alterna entre a utilização e não do efeito
// matiz
  void imagemVersao2(int n) {
    inc = 500/n;
    posiciona(n);
    for (int i = 0; i < inc; ++i) {
       flag = !flag; // controlo de alternância de efeito
       tint(random(0,  255),  random(0,  255),  random(0,  255),
80);
       if (flag) noTint(); // idem
       image(imgA, x, y, n, n);
       x += n;
    }
     y += n;
  }

 // Idem, porém a imagem é escalada para a metade de sua
 // largura, fazendo um efeito "espremido"
  void imagemVersao3(int n) {
 // espacejamento calculado levando em conta isso...
    inc = 500*2/n;
    posiciona(n);
    for (int i = 0; i < inc; ++i) {
       tint(random(0,  255),  random(0,  255),  random(0,  255),
60);
// posicionamento escalonado em x
       image(imgA, x, y, n * 0.5, n);
       x += n * 0.5; // idem
    }
     y += n;
  }

 // Idem, fazendo alternância entre duas imagens diferentes
  void imagemVersao4(int n) {
    inc = 500/n;
    posiciona(n);

    for (int i = 0; i < inc; ++i) {
       tint(random(0,  255),  random(0,  255),  random(0,  255),
90);
       if (flag) image(imgA, x, y, n, n);
       else image(imgB, x, y, n, n);
// Flag é utilizada para controlo da alternância
       flag=!flag;
       x += n;
```

```
    }
     y += n;
  }

  // Idem, fazendo sobreposição das imagens diferentes
  void imagemVersao5(int n) {
    inc = 500/n;
    posiciona(n);

    for (int i = 0; i < inc; ++i) {
       tint(random(0,  255),  random(0,  255),  random(0,  255),
50);
       image(imgA, x, y, n, n);
       image(imgB, x, y, n, n);
       x += n;
    }
     y += n;
  }
 // Método para escrever texto na janela abaixo da última //
posição horizontal utilizada
  int escreve(String msg) {
    posiciona(25);
    textAlign(CENTER); // alinhado ao centro
    text(msg, 250, y + 25);// posiciona no meio da janela
    y += 25; // tamanho da fonte é 25 pixels

    return 0;
  }
// Método que re-inicializa as coordenadas x e y
  void posiciona(int n) {
    x = 0;
// Se ultrapassar a altura da janela, é limpado o fundo
// e recomeça o desenho de cima para baixo
    if ((y + n) > 500) {
      background(0, 0, 0);
      y = 0;
    }
  }
}
```

3. Vamos agora definir o código no programa principal desse exemplo. Utilizaremos três variáveis globais para nos auxiliar no controle e o na visibilidade do objecto Imagem instanciado em ***setup()*** em todo o programa principal. Utilizaremos a função ***keyPressed()*** para auxiliar a gestão de eventos com o teclado, e com isso, criar alguma interacção:

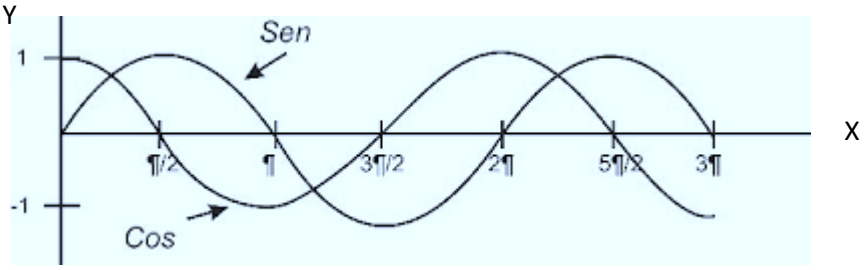
```
boolean go; // controlo de interação com tecla
Imagem img; // objeto Imagem
int contador; // controlo de ciclo

void setup() {
  size(500, 500);
  background(0, 0, 0);
  go = false;
  contador = 0;
// Cria o objeto Imagem que é composto por dois objetos PImage
  img = new Imagem("Penguins.jpg", "Tulips.jpg");
// Invoca vários métodos do objeto
  img.imagemVersao1(10);
  img.imagemVersao2(25);
```

```
  img.imagemVersao1(100);
  img.imagemVersao3(100);
  img.imagemVersao4(50);
  int    tamanho    =    img.escreve("Aprendendo    a    utilizar
imagens...");
  img.imagemVersao4(50);
}

void draw() {
  String texto;
// Quando premir a tecla ENTER limpa tudo e executa
// seguidamente estes 2 métodos
  if (go) {
    img.imagemVersao5(50);
// Escreve na janela o valor do contador
// utiliza a função str() para converter de númerico
// para string...
    texto = str(++contador);
    img.escreve(texto);
  }
}
// Controlo de evento premir tecla ENTER
void keyPressed() {
    if (keyCode == ENTER) go = !go;
}
```

4.  Execute e dê ENTER. Salve o projeto.

3. *Funções trigonométricas*

Da mesma forma que as formas geométricas nos permitem criar composições visuais variadas, algumas funções trigonométricas também o podem fazer. Algumas delas geram o desenho de formas ondulares, que dependendo de como sejam utilizadas, podem resultar em representações visuais bastante originais. As funções trigonométricas que calculam os senos, os co-senos e arcos de valores angulares (entre 0 e 360 graus) são um bom exemplo disso. As funções seno e co-seno retornam valores sempre entre -1.0 e 1.0. A figura abaixo, ilustra o comportamento dessas funções:

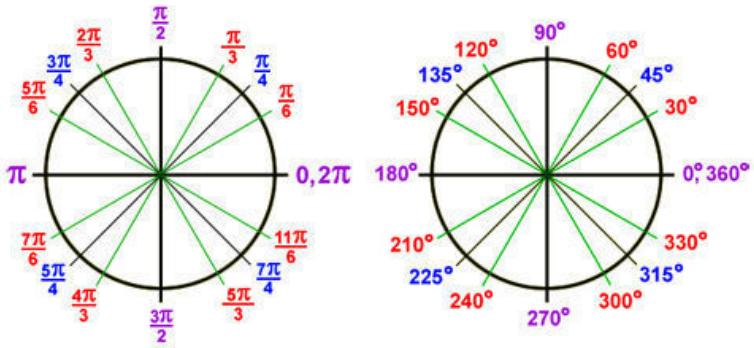


Comportamento das curvas sinusoidal e co-sinusoidal

As funcionalidades que temos a nossa disposição no processing são as: ***sin(), cos(),*** e ***arc().*** Para conseguirmos trabalhar com as funções trigonométricas **temos que utilizar graus convertidos em radianos**. Para convertermos as unidades e definirmos valores, temos as seguintes

constantes do sistema e funcionalidades a nossa disposição: ***PI, QUARTER_PI, HALF_PI, TWO_PI, radians(), degrees()***.

A correspondência entre os valores em radianos e graus é a seguinte: **PI** = 180 graus, **TWO_PI** = 360 graus, **QUARTER_PI** = 45 graus e **HALF_PI** = 90 graus. Logo, os valores a serem passados como parâmetros devem estar entre **0** e **6.28** radianos (**PI** = 3.14), ou seja, 0 e 360 graus. A figura abaixo ilustra a correspondência existente entre as duas unidades.



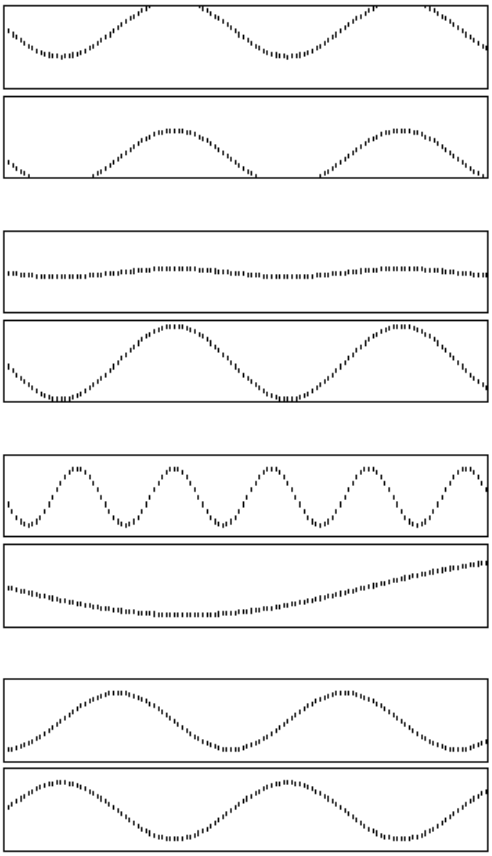
Correspondência entre radianos e ângulos

Portanto, com a utilização destas funcionalidades, conseguimos facilmente criar um conjunto de valores que **varia de forma cíclica e controlada**. Estes valores **podem ser utilizados** para controlar a posição de um determinado elemento de nossa composição visual, a sua cor, transparência ou qualquer outro atributo visual ou até de interacção de nosso programa ☺

Outro pormenor interessante, é que se consegue controlar a **posição**, **amplitude** e **suavidade** das curvas com grande facilidade. Isto permite atingir um excelente grau de flexibilidade. Vejamos analisar as seguintes expressões:

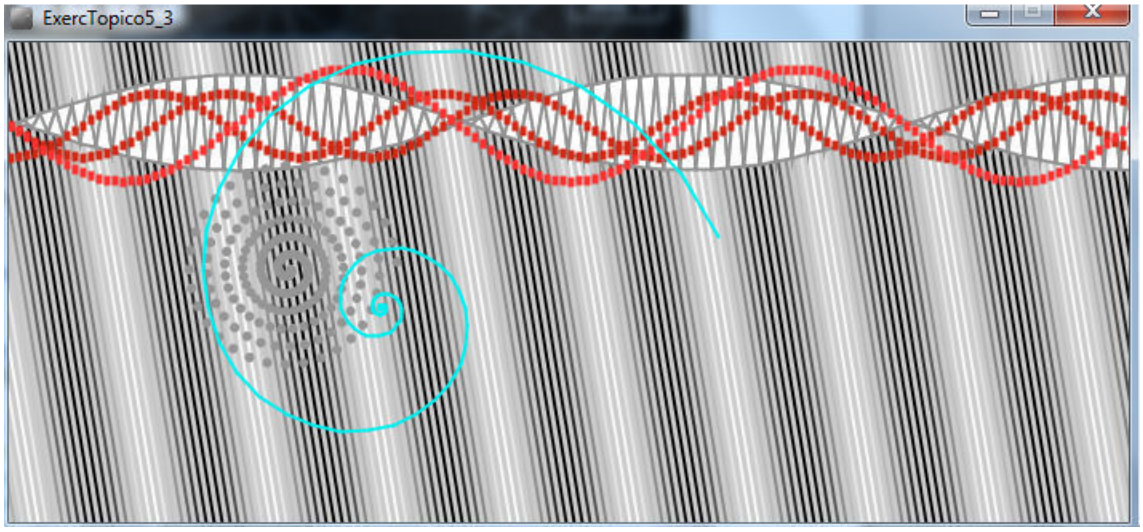
```
<valor> = <offset> + (sin ou cos(<angulo>) * <fator de escala>)

<angulo> = <angulo> + <incAngulo>
```

<valor> é o resultado obtido da expressão, que será igual ao valor da função sinusoidal ou co-sinusoidal no caso de *<offset>* ser igual a 0 e *<fator de escala>* a 1. *<offset>* desloca a curva mais para cima ou baixo, mudando a sua posição original. *<fator de escala>* consegue mudar a amplitude da onda, ou seja, se for superior a 1, a onda fica mais ampla (altura da onda é maior), em caso contrário, a onda fica menos ampla (altura é menor). *<angulo>* controla a frequência de variação da onda, e a torna mais ou menos "suave". Se aplicarmos continuamente o cálculo e formos incrementando o ângulo em cada vez, quanto mais alto for o incremento *<incAngulo>* utilizado, mais suave a forma da onda resulta, o oposto, se o incremento for muito pequeno. Alguns exemplos onde apenas um dos elementos de controle foi alterado:



Curvas com <*offset*> iguais a 25 e 75, <*fator de escala*> 5.0 e 45.0, <*incAngulo*> PI/12.0 e PI/90.0 e <*angulo*> HALF_PI e PI, respectivamente

Vamos percorrer um exemplo, onde utilizaremos de forma abrangente estas funcionalidades e constataremos os resultados visuais que obtemos em cada caso. Na imagem abaixo, algumas das formas visuais geradas em forma acumulativa:



1. Abra o **processing** e crie um programa de nome **ExercTopico5_3.pde**
2. Vamos começar por criar a o objecto ***FigurasTrigonometricas.*** Vamos adicionar mais um ficheiro para a classe, dando-lhe o nome igual ao desta. Dentro dele, vamos adicionar os atributos e métodos necessários:

```
class FigurasTrigonometricas {
// Atributos da classe
  float offset; // incremento para deslocamento
  float scaleVal;// incremento para fator de amplificação
  float anguloInc;// incremento para ângulo
  float angulo;// angulo a ser utilizado
  float raio;
// Construtor vazio
  FigurasTrigonometricas() {
  }

// Utilização da função seno para variar de forma cíclica
// de acordo com o comportamento da curva sinusoidal
// a tonalidade cinzenta da linha que é desenhada
  void desenhaEstilo1() {
    smooth();  // evita efeito escada
    strokeWeight(2); // espessura do traço

    offset = 126.0; // para controle da posição
    scaleVal = 126.0;// para controle da amplitude
    anguloInc = 0.42; // para controle da suavidade
    angulo = 0.0;// para conter o ângulo a cada iteração

    for (int x = -52; x <= width; x += 5) {
      float y = offset + (sin(angulo) * scaleVal);
      stroke(y);
      line(x, 0, x+50, height);
      angulo += anguloInc;
    }
  }
// Desenha uma malha triangular tendo como base a forma
// sinusoidal da curva
  void desenhaEstilo2() {
    offset = 50; // idem
    scaleVal = 30.0;
    anguloInc = PI/56.0;
    angulo = 0.0;
 // Desenha uma malha triangular tendo como base
 // os limites da curva sinusoidal
    beginShape(TRIANGLE_STRIP);
      for (int x = 4 ; x <= width+5; x += 5) {
        float y = sin(angulo) * scaleVal;
 // Calcula o resto da divisão de x por 2
 // se for zero, é par e y é incrementado de 50
 // caso contrário, é decrementado de 50
        if ((x % 2) == 0) {
                  vertex(x, offset + y); // vértice da malha
        } else {
                  vertex(x, offset - y);
                }
        angulo += anguloInc;// actualiza valor do ângulo
     }
     endShape();
  }
// Desenha uma onda sinusoidal e outra co-sinusoidal
// com pequenos retângulos
  void desenhaEstilo3() {
    stroke(210, 40, 20); // altera cor do traço
    smooth();
    offset = 50.0; //idem
    scaleVal = 20.0;
```

```
    anguloInc = PI/18.0;
    angulo = 0.0;
    for (int x = 0; x <= width; x += 5) {
      float y = offset + (sin(angulo) * scaleVal);
      fill(255); // com preenchimento
      rect(x, y, 2, 4); // retângulo da curva sinusoidal
      y = offset + (cos(angulo) * scaleVal);
      fill(0); // sem preenchimento
      rect(x, y, 2, 4); // retângulo da curva co-sinusoidal
      angulo += anguloInc;
    }
  }
// Desenha uma onda sinusoidal com pequenos retângulos
  void desenhaEstilo4() {
    stroke(255, 50, 50);
    smooth();
    fill(0);
    float offset = 50.0; // idem
    float scaleVal = 35.0;
    float anguloInc = PI/28.0;
    float angulo = 0.0;
    for (int x = 0; x <= width; x += 5) {
      float y = offset + (sin(angulo) * scaleVal);
      rect(x, y, 2, 4);
      angulo += anguloInc;
    }
  }
// Desenha uma padronagem tendo como base a função sinusoidal
// é utilizada uma elipse que é desenhada com ou sem contornos
// e com preenchimento a preto ou branco
// para complementar a padronagem, é utilizado um ponto que
// marca a localização da elipse
  void desenhaEstilo5(int cor, boolean op) {
    smooth();
    fill(cor, 20);// varia, consoante o passado...
    float scaleVal = 18.0; // idem
    float anguloInc = PI/28.0;
    float angulo = 0.0;
    for (int offset = -10; offset < width+10; offset += 5) {
      for (int y = 0; y <= height; y += 2) {
        float x = offset + (sin(angulo) * scaleVal);
        if (op) noStroke();
        ellipse(x, y, 10, 10);
        stroke(0);
        point(x, y);
        angulo += anguloInc;
      }
      angulo += PI;
    }
  }
// Desenha arcos cujo início e fim variam
// de i = 0 até 160 em graus radianos
  void desenhaEstilo6() {
    smooth();
    noFill();
    strokeWeight(10);// espessura do traço
    for (int i = 0; i < 160; i += 10) {
      stroke(i + 50, i, i + 30); //varia a cor do traço
      float begin = radians(i);// inicio do arco
      float end = begin + HALF_PI; // incrementa 90 graus
      arc(67, 37, i, i, begin, end);
```

```
    }
  }
  // Desenha uma espiral com elipses
  // em função das funções sinusoidal e co-sinusoidal
  void desenhaEstilo7() {
    fill(150);
    noStroke();
    smooth();
    float radius = 1.0;
    for (int deg = 0; deg < 360*6; deg += 11) {
      float angle = radians(deg);
      float x = 175 + (cos(angle) * radius);
      float y = 142 + (sin(angle) * radius);
      ellipse(x, y, 6, 6);
      radius = radius + 0.34;
    }
  }
// Desenha uma elipse na mesma, utilizando uma linha
// em lugar de elipses
  void desenhaEstilo8() {
    smooth();
    strokeWeight(2);
    stroke(0, 240, 240);
    raio = 0.15; // funciona como um fator de escala…
    float cx = 233; // Centro da elipse em x e y
    float cy = 166;
    float px = cx; // posição inicial da linha em x e y
    float py = cy;
    for (int deg = 0; deg < 360*5; deg += 12) {
      float angulo = radians(deg);
// A cada ciclo, a posição é ligeiramente afastada do centro
// da espiral. A variação cíclica e regular dada pelo seno e
// co-seno permite que o padrão de desenho se mantenha
      float x = cx + (cos(angulo) * raio);
      float y = cy + (sin(angulo) * raio);
      line(px, py, x, y);
      raio = raio * 1.05;
      px = x;
      py = y;
    }
  }
}
```

3. Definiremos o programa principal, criando apenas uma variável global para conter o objecto a ser instanciado. Utilizaremos um ***switch-case*** para consoante a tecla premida (de 0 a 9), um determinado método seja invocado. Para forçar a **execução cíclica do programa**, declaramos uma função ***draw()*** vazia ☺. Como na ***keyPressed()*** limpamos sempre o écran a cada execução, o **efeito acumulativo não está visível**, mas se quiséssemos, poderíamos tirar partido disso facilmente, apenas eliminando a limpeza da janela e adicionando alguma transparência as visualizações geradas ☺:

```
FigurasTrigonometricas figura;

void setup() {
  size(700, 300);
  figura = new FigurasTrigonometricas();// instanciação
}

void draw() {
}
```

```
void keyPressed(){
  background(255); // coloca o fundo a branco

// Caso as teclas 0, 1, 2, 3, 4, 5, 6, 7 ou 8 sejam premidas
  switch (key) {
    case '1': figura.desenhaEstilo1(); break;
    case '2': figura.desenhaEstilo2(); break;
    case '3': figura.desenhaEstilo3(); break;
    case '4': figura.desenhaEstilo4(); break;
    case '5': figura.desenhaEstilo5(255, true); break;
    case '6': figura.desenhaEstilo5(0, true); break;
    case '7': figura.desenhaEstilo5(255, false); break;
    case '8': figura.desenhaEstilo6(); break;
    case '9': figura.desenhaEstilo7(); break;
    case '0': figura.desenhaEstilo8(); break;
  }
}
```

4.  Execute e prima as teclas 0, 1…9. Salve o projeto.

4. *Transformações espaciais em 3D*

Em tópico anterior, abordamos as transformações espaciais em 2D: alteração de escala, translação e rotação. Em 2D, considerávamos que a nossa janela de visualização funcionava como uma espécie de folha de papel. Na orientação horizontal colocávamos o eixo do X e na orientação vertical, o de Y. Todas as transformações espaciais que fazíamos estavam restritas a essas duas dimensões (que poderiam ter uma orientação diferente em dada altura, em função de rotações que aplicássemos…).

As transformações espaciais em **3D obedecem o mesmo raciocínio**, porém, apresentam **uma dimensão adicional** a qual podemos também considerar: a profundidade, representada por Z. Todos os objectos que são representados visualmente possuem um **volume**, e não apenas uma área (situação típica no caso 2D). As funcionalidades são bastantes semelhantes a que vimos para as transformações espaciais em 2D, aparecendo agora, um terceiro eixo de orientação, o Z. A nossa janela de visualização passa não estar mais associada a uma folha de papel, em termos conceptuais, mas sim, a um **cubo ou prisma**.

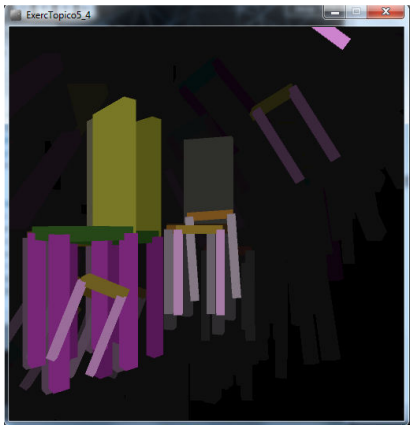
Por detrás da visualização em 3D em superfícies 2D (que é o caso da janela de visualização do computador…) estão importantes conceitos de **projecção geométrica** (geometria descritiva). Esses conceitos fundamentam e dão suporte matemático a visualização tridimensional de objectos em superfícies 2D. Duas projecções são utilizadas tradicionalmente na visualização em 3D no computador: a em **perspectiva** e a **ortogonal**.

A projecção em **perspectiva** é a que por defeito está configurada no processing quando você **ativa o 3D**. Essa projecção cria **visualizações mais realistas** e o **espaço volumétrico virtual da janela de visualização é associado a um prisma**. A projecção **ortogonal** tem que **ser activada expressamente** por você. Ela proporciona uma visualização **menos realista** e o espaço volumétrico virtual da janela é **associado com um cubo**.

Não vamos entrar em mais pormenores da visualização em 3D, pois será tema futuro desta disciplina. Por agora, iremos apenas explorar as transformações espaciais em 3D, tendo em conta essa característica especial do espaço de projecção/visualização. Para ativarmos o 3D no processing temos que passar a constante ***P3D*** como terceiro parâmetro da função ***size***.

A nível de funcionalidades, podemos utilizar as mesmas que já vimos, só **que agora incluindo uma terceira dimensão** na sua especificação. Existem algumas que são específicas para quando ativamos o P3D: ***rotateX***, ***rotateY*** e ***rotateZ*** (consulte o manual do processing online para ver a sintaxe de cada uma delas e exemplos de utilização). Existem também duas primitivas gráficas prontas para utilizar quando em 3D no processing: ***sphere()*** e ***cube()***.

Vamos executar um exemplo para compreender melhor as transformações em 3D. Iremos criar um objecto de nome ***Cadeira***. A cadeira será composta **por 6 cubos**. Utilizaremos as transformações em 3D para fazer a montagem desse objecto. Inicialmente, criaremos um cubo quadrado e fino, com auxílio da função ***box().*** De forma a garantirmos que todas as transformações seguintes que iremos aplicar aos demais cubos **são executadas de forma isolada** (sem efeitos acumulativos), vamos utilizar blocos de ***pushMatrix()-popMatrix()***.



Em seguida, vamos rodar 90 graus a nossa orientação espacial em X e desenhar um outro cubo idêntico, porém um pouco mais alto (y é maior). Esse cubo é posicionado perpendicular ao anterior, sendo que é transladado para cima e para trás, de forma a ficar devidamente posicionado na base do assento.

Por fim, vamos desenhar cada uma das pernas com cubos mais finos e longos. Para posiciona-los devidamente, transladamo-los para cada canto do cubo que definiu o assento, ou seja, nas localizações em x e y dadas por (25, 25), (-25, 25), (-25, -25) e (25, - 25). Temos que mover as pernas -50 em Z, de forma a ficarem devidamente posicionadas na base do assento.

Para obter alguma interacção, também vamos utilizar as transformações em 3D. A posição do rato faz com que o ponto de partida para o desenho da cadeira seja reposicionado para aquela localização. Dependendo de um valor aleatoriamente gerado, a cadeira passa a ser girada em torno de X, Y ou Z, de um valor em graus, também aleatoriamente calculado.

A cor de cada uma das partes da cadeira também é alterada com auxílio de ***random().***

Por fim, vamos explorar o efeito visual *fade out*. Para conseguir obtê-lo entre os redesenho das cadeiras a cada *frame*, desenhamos sempre um rectângulo quase totalmente preto (com pouca transparência) com dimensão igual a da janela de visualização, antes de desenhar novamente a cadeira. Com isso, conseguimos ter um efeito suave de transição entre o desenho das diversas cadeiras na janela de visualização ☺

1. Abra o **processing** e crie um programa de nome **ExercTopico5_4.pde**
2. Vamos começar por criar a o objecto ***Cadeira.*** Vamos adicionar mais um ficheiro para a classe, dando-lhe o nome igual ao desta. Dentro dele, vamos adicionar os atributos e métodos necessários:

```
class Cadeira {
  float [] corAssento; // para conter o rgb das partes
  float [] corPernas;
  float [] corEspaldas;
 // Construtor
  Cadeira(float [] cA, float [] cP, float[] cE) {
    corAssento = cA;
    corPernas = cP;
    corEspaldas = cE;
    noStroke();
  }
  // Métodos para atualizar as cores das partes
  void setCorAssento(float[] cA) { corAssento = cA; };
  void setCorPernas(float[] cP) { corPernas = cP; };
  void setCorEspaldas(float[] cE) { corEspaldas = cE; };

  // Método que desenha a cadeira com as cores atuais
  void desenha() {
     pushMatrix();
        fill(corAssento[0], corAssento[1], corAssento[2]);
        rotateX(PI/2);
     // Desenha assento
        box(55, 55, 10);
     // Desenha o encosto para as costas
        pushMatrix();
          fill(corEspaldas[0],    corEspaldas[1],    corEspaldas
[2]);
          rotateX(PI/2);
          translate(0, 50, 25);
          box(55, 100, 10);
        popMatrix();
    // Desenha as 4 pernas
        fill(corPernas[0], corPernas[1], corPernas[2]);
        pushMatrix();
          translate(-25, -25, -50);
          box(10, 10, 100);
        popMatrix();
        pushMatrix();
          translate(25, 25, -50);
          box(10, 10, 100);
        popMatrix();
        pushMatrix();
          translate(25, -25, -50);
          box(10, 10, 100);
        popMatrix();
        pushMatrix();
          translate(-25, 25, -50);
```

```
          box(10, 10, 100);
        popMatrix();
    popMatrix();
  }
}
```

3. Definiremos o corpo do programa principal. Vamos reduzir a taxa de refrescamento para ficar menos "pesado" o processamento. Não esqueça que por defeito, o volume contido na janela tem a forma de um prisma. Todos os objectos que sejam desenhados, são colocados dentro desse prisma invisível, com uma projecção em perspectiva:

```
// Variáveis globais
Cadeira cadeira;
float [] cA = {255, 153, 51}; // valores rgb
float [] cP = {255, 51, 255};
float [] cE = {255, 255, 51};

void setup() {
  size(500, 500, P3D); // ativa o 3D na visualização
  frameRate(24); // reduz a taxa de refrescamento
  background(0);
  cadeira = new Cadeira(cA, cP, cE); // instancia a cadeira
}

void draw() {
// Gera a opção para orientação da rotação de forma aleatória
  int orientacao = (int)random(1, 9);
// Pinta um retângulo quase preto para fazer o efeito fade out
// limpando a janela a cada execução de draw(),  sem causar
// efeito blinking entre as execuções...
  fill(0, 9);
  rect(0, 0, width, height);

// A rotação na cadeira é alternada em orientação e
// é calculada de forma aleatória
  if (orientacao <= 3) rotateZ(radians(random(0, 360)));
  else if (orientacao <=6) rotateX(radians(random(0, 360)));
       else rotateY(radians(random(0, 360)));

// A cadeira é movida segundo a posição do rato na janela
  translate(mouseX, mouseY, 0);

// A cada execução atualiza a cor das partes da cadeira
// de forma aleatória
  cA[0] = random(0,255);
  cP[1] = cA[0];
  cE[2] = cA[0];
  cadeira.setCorAssento(cA);
  cadeira.setCorPernas(cP);
  cadeira.setCorEspaldas(cE);
  cadeira.desenha();
}
```

4.  Salve. Mova o rato na janela e veja a cadeira ser desenhada em várias posições, com a rotação ora em X, ora em Y e ora em Z ☺

Programação Criativa – Tópico 6

***Temática**: Curvas, criação de formas e de padrões visuais*
***Duração**: semanas 1, 2, 3 e 4*

Actividade 6: Aprender a utilizar funções para criar curvas, definir figuras dentro do código e criar formas padronizadas.

Competências a desenvolver:

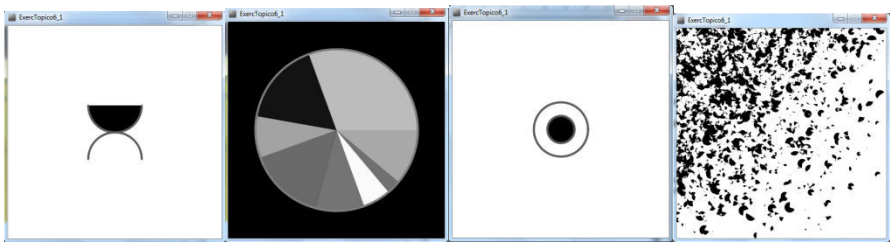
- Criação de curvas
- Criação de formas com ***beginShape-endShape***
- Implementação de padrões visuais

1. Curvas

Apesar de já termos utilizado muitas funções para desenho de formas geométricas, ainda não abordamos as curvas com maior detalhe. Em termos matemáticos, existem muitas formas de calcularmos curvas. O cálculo de curvas pode ser na maioria das vezes mais ou menos complexo. O **processing** nos oferece algumas funcionalidades que permitem com que criemos formas curvas com muita facilidade.

Nesta seção iremos experimentar a utilização das funções: ***arc***(), ***curve***() e ***bezier***(). A primeira permite com que desenhemos arcos de circunferências. A **curve**() deve ser utilizada quando queremos criar curvas pequenas, definidas por dois pontos. Ela utiliza algoritmos para desenho de curvas do tipo *Spline* (http://pt.wikipedia.org/wiki/Spline). A ***bezier***() permite desenhar curvas mais longas e com desenho suave, pois se baseia no algoritmo de *Bézier* (http://pt.wikipedia.org/wiki/Curva_de_B%C3%A9zier) para desenho de curvas. Tanto as curvas do tipo Spline como Bézier utilizam na sua abordagem pontos de controlo, que servem para definir referências espaciais por onde a curva deve mais ou menos passar.

O primeiro exemplo que vamos abordar, ilustra a utilização da função ***arc***() de diversas formas, criando efeitos visuais bastante diferentes entre si. Os parâmetros que ela utiliza são, respectivamente: coordenadas espaciais x e y, largura e altura do arco, ângulo de início e fim do arco. Vamos ver o código passo-a-passo:



1. Abra o **processing** e crie um programa de nome **ExercTopico6_1.pde**
2. Vamos começar por definir algumas variáveis globais, que vão nos ajudar na definição de parâmetros do arco em cada situação, sendo a explicação feita depois na parte do código onde elas são utilizadas:

```
// Variáveis globais
int op;
int x, y, w, h;
float radius;
```

```
int[]angs = {40, 10, 20, 35, 55, 30, 60, 110};
float lastAng;

void setup() {
  size(400, 400);
// width e height são variáveis pré-definidas do sistema
// e tem a altura e largura da janela, ou seja, 400 e 400
  x = width/2;
  y = height/2;
  w = 100;
  h = 100;
  op = 0;
}
```

3. Vamos criar quatro visualizações distintas, que vão ser alternadas, conforme o utilizador prime uma tecla qualquer de seu teclado. Para isso, utilizaremos a função pré-definida ***keyPressed*** do processing, que gere eventos de teclado, sendo invocada sempre que alguma tecla é premida. A cada passada, testa-se o valor contido na variável ***op*** e muda-se o seu valor. Com estruturas lógicas condicionais do tipo ***if-else***, avaliaremos todas as situações possíveis, trocando o valor de ***op*** quando necessário:

```
// Conforme se prime uma tecla, cicla as visualizações
void keyPressed() {
  if (op == 0) {
    op = 1;
  } else if (op == 1){
            op = 2;
          } else if (op == 2) {
                    op = 3;
                } else op = 0;
                        }
```

4. Em ***draw***() iremos controlar a execução de cada visualização. Para isso, utilizaremos uma outra estrutura lógica condicional mais prática para nós neste caso, a ***switch-case***. Consoante o valor depositado em ***op***, um dos quatro casos será executado. O primeiro arco será desenhado no centro da janela (x = 200, y = 150), com largura e altura de 100 pixels e 180º de abertura (de 0 a 180 graus, metade de baixo da elipse). O segundo, terá posição também central mas invertida ao anterior (x = 200, y = 250), mesma largura, altura e abertura (de 180 a 360 graus, metade de cima da elipse). O primeiro será preenchido com a cor ativa e o segundo, sem preenchimento. Vamos definir o primeiro caso:

```
void draw() {
  smooth(); // Desenho sem efeito-escada
  switch (op) {
    case 0: // caso op seja igual a 0
// Desenha 2 arcos invertidos (180° cd um)
      background(255);
      stroke(90); // cor do traço
      strokeWeight(4); // espessura do traço
      fill(0); // o 1° é preenchido
// definição do arco, com: posição x e y da elipse
// largura e altura do mesmo, ângulos de início e fim em radianos
      arc(width/2, height/2-50, 100, 100, 0, PI);
      noFill(); // o 2° não é preenchido
      arc(width/2, height/2+50, 100, 100, PI, PI*2);
```

```
            break;
        }
```

5.  e veja o que acontece…

6. Vamos adicionar outro caso para tratar quando ***op*** tem valor 1. Neste caso, desenharemos dois arcos com 360 graus, ou seja, serão 2 círculos, um dentro do outro, com posições x = 200 e y = 200 em ambos os casos. A altura e largura será de 50 para o primeiro e de 100 para o segundo:

```
        case 1:
          // Desenha 2 arcos concêntricos (360° cd um)
            background(255);
            stroke(90);
            strokeWeight(4);
            fill(0); // arco preenchido
            arc(width/2, height/2, 50, 50, 0, PI*2);
            noFill();// arco não preenchido
            arc(width/2, height/2, 100, 100, 0, TWO_PI);
            break;
```

7.  e veja o que acontece…terá que premir alguma tecla para alternar entre a primeira e segunda visualizações.

8. Vamos agora inserir o caso para ***op*** ser igual a 2. Neste caso, iremos fazer um gráfico do tipo *pizza*, construído com vários arcos, posicionados sempre no centro da janela, com alturas e larguras sempre de 300 pixels, porém, com ângulos de abertura variáveis. A utilização de um ciclo ***for*** permite que variemos o valor dos ângulos, criando vários arcos:

```
            case 2:
// Desenha vários arcos com aberturas diferentes
// (40°, 10°, 20°, 35°, 55°, 30°, 60° e 110° respectivamente)
                background(0);
                stroke(127);
                fill(0);
                radius = 150;
                lastAng = 0;
                for (int i=0; i < angs.length; i++){
// Preenchimento aleatório em tons cinza
                  fill(random(255));
                  arc(width/2, height/2, radius*2, radius*2, lastAng,
          lastAng+=radians(angs[i]));
                }
                break;
```

9.  e veja o que acontece…terá que premir alguma tecla para alternar entre as visualizações, ok?!

10. Agora vamos a definição do último caso, ou seja, quando ***op*** tem valor 3. Esta visualização é a mais "artística" de todas, pois todos os parâmetros do arco são definidos de forma aleatória, criando uma composição visual dinâmica e altamente variável. Utilizamos estruturas de repetição ***for***, duas aninhadas, para que consigamos

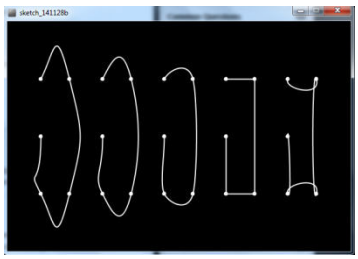
variar as posições espaciais x e y que são utilizadas como base para gerar as posições aleatórias:

```
            case 3:
// Desenha vários arcos em posições, amplitudes e com ângulos de
// abertura aleatórios
                  background(255);
                  noStroke();
                  radius = 10;
                  for (int i=0; i<=width; i+=radius/2){
                    for (int j=0; j<=height; j+=radius/2){
                        float size = (random(radius*2));
                        fill(255);
                       arc(random(i),     random(j),     size,     size,
random(PI*2), random(PI*2)); // todos os parâmetros são aleatórios…
                        fill(0);
                        arc(random(i),     random(j),     size,     size,
            random(PI*2), random(PI*2));
                      }
                  }
                  break;
```

11.  e veja o que acontece…

A função ***curve***() utiliza o algoritmo de *Catmull–Rom spline* (veja detalhes em http://en.wikipedia.org/wiki/Centripetal_Catmull%E2%80%93Rom_spline). A função utiliza oito parâmetros como entrada no caso 2D, e doze no caso 3D. Esses parâmetros definem posições espaciais, ou melhor, pontos, por onde a curva deve passar (proximidade). Na realidade, os quatro pontos, funcionam como pontos de controle. Para conseguirmos fazer uma curva mais longa e detalhada, temos que invocar várias vezes ***curve***() passando pontos convenientes, de forma a garantir que o início de uma ocorre no final de outra. Mais a frente veremos outro modo de criar curvas com este algoritmo, porém utilizando o recurso de criação de formas.

Vamos fazer mais um exemplo para ilustrar a utilização desta função. Neste exemplo, criaremos cinco curvas distintas, sendo cada uma criada com quatro segmentos de curvas. Utilizaremos outra função que pode ser aplicada em conjunto com a ***curve***():***curveTightness***. Esta última, consegue moldar a forma da curva com maior ou menor curvatura conforme o valor do parâmetro passado. Se ele for 1.0, a curva passa a ser desenhada como um segmento de reta. Para ficar claro como ***curveTightness*** afeta ***curve***(), cada curva será desenhada com valores diferentes. Além disso, serão desenhados como pequenas bolinhas, os pontos que são interpretados como sendo os de controle da curva que é desenhada.



1. Crie um programa de nome **ExercTopico6_2.pde.** Neste programa, não há necessidade de utilizar a função ***draw***(), pois a visualização é estática (permanece a mesma durante toda a execução). Isto equivale a ter uma função ***setup***() implícita ☺. Vamos declarar todas as variáveis que iremos utilizar para que o programa funcione como queremos e parametrizar o funcionamento de algumas funcionalidades:

```
            size(600, 400);
            background(0);
            stroke(255);
            strokeWeight(2);
            smooth();
// Valor da largura de cada curva, 50 pixels
            int curveWdth = 50;
// 5 curvas vão ser desenhadas, logo, são 5 espaços verticais
// reservados para cada uma delas na janela de visualização
            int cols = 5;
// Calcula o espacejamento ideal entre cada curva de forma a não se
// sobreporem, neste caso (600 - 50*5)/(5 + 1) = 58
            int xPadding = (width-curveWdth*cols)/(cols+1);
            int x = xPadding; // x é inicializado com esse valor, 58
```

2. Utilizaremos uma classe do java para nos ajudar a criar vectores/*arrays*: ***PVector***. Como os pontos espaciais são *arrays* de 2 elementos (estamos em 2D), essa classe é muito prática para organizarmos os nossos dados. Além disso, utilizaremos um ciclo ***for*** para desenhar as cinco curvas distanciadas entre si com um valor igual a sua largura somado ao valor contido em ***xPadding***. Isso garante que elas não se sobreponham no desenho. Por fim, utilizaremos a função ***curve***(), alimentando-a com os cinco pontos dinamicamente calculados a cada ciclo e denotando com ***ellipse***() quem são os pontos de controle das curvas. Cada curva é composta por quatro curvas na realidade:

```
            for (int i=-2; i<3; i++){ // Repete-se 5 vezes…
// esta função causa com que a curva fique mais ou menos
// linearizada. Com 1.0, ela torna-se uma reta. Com outros
// valores, ela se deforma de forma variada...
              curveTightness(i);
// Define os ptos de controle das curvas
              PVector p0 = new PVector(x, 100);
              PVector p1 = new PVector(x+curveWdth, 100);
              PVector p2 = new PVector(x+curveWdth, 300);
              PVector p3 = new PVector(x, 300);
              PVector p4 = new PVector(x, 200);
// A cada repetição do for, x é incrementado de 50
              x+=curveWdth+xPadding;
// Desenha as curvas com a função curve
// desenha sem preenchimento as curvas (teste comentar...)
              noFill();
              curve(p4.x, p4.y, p0.x, p0.y, p1.x, p1.y, p2.x, p2.y);
              curve(p0.x, p0.y, p1.x, p1.y, p2.x, p2.y, p3.x, p3.y);
              curve(p1.x, p1.y, p2.x, p2.y, p3.x, p3.y, p4.x, p4.y);
              curve(p2.x, p2.y, p3.x, p3.y, p4.x, p4.y, p0.x, p0.y);
// Desenha os ptos de controle das curvas como bolinhas
              fill(255);
              ellipse(p0.x, p0.y, 5, 5);
              ellipse(p1.x, p1.y, 5, 5);
```

```
  ellipse(p2.x, p2.y, 5, 5);
  ellipse(p3.x, p3.y, 5, 5);
  ellipse(p4.x, p4.y, 5, 5);
}
```

12.  e veja o que acontece…Teste comentando o ***noFill***(). Verá que as curvas são desenhadas não como linhas, mas preenchidas…

Por fim, vamos falar um pouco sobre a função ***bezier***(). Ela também utiliza quatro pontos como entrada, porém a forma como os interpreta, é diferente da ***curve***(). Na função ***bezier***() existem dois pontos âncora (o primeiro e último pontos) e dois pontos de controle (os dois pontos do meio). Neste caso, a curva obrigatoriamente passa pelos pontos âncora e utiliza os de controle como referência espacial para traçado da curva (proximidade). Portanto, se substituirmos no código acima ***curve***() por ***bezier***() veremos que o resultado visual é bem diferente, apesar de esta função também desenhar curvas. Não iremos percorrer nenhum exemplo só com a função ***bezier***(), pois ela já foi abordada em tópico anterior. Iremos ver uma utilização mais avançada de ***curve***() e ***bezier***() na seção seguinte.

2. Criação de formas

Um outro artifício poderoso que podemos utilizar para criar formas (regulares ou livres) é a utilização das funcionalidades ***beginShape***()-***endShape***(). Essas duas funções podem ser utilizadas para delimitar blocos de código que definem uma determinada forma/objecto gráfico. As formas/objectos são definidas vértice a vértice ou ponto a ponto (conforme o que se esteja a criar, curvas, segmentos de reta, polígonos, etc.) em 2D ou 3D.

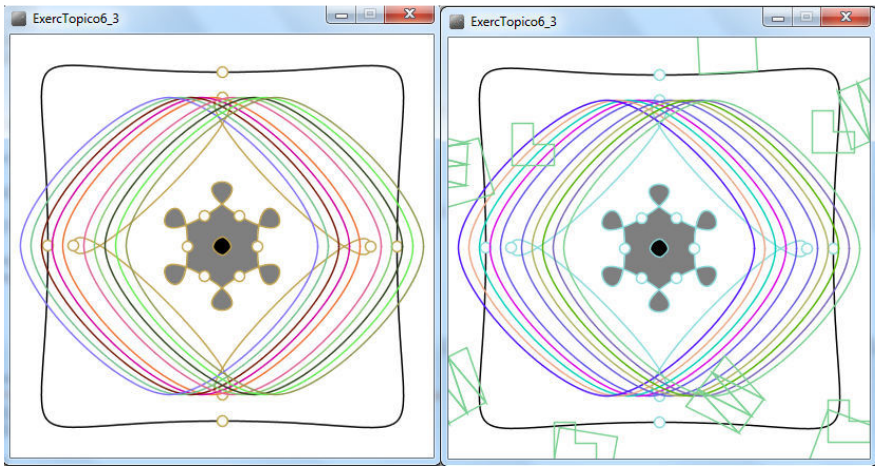
Os pontos/vértices são definidos com a utilização das funcionalidades: ***vertex***, ***curveVertex*** ou ***bezierVertex***. Portanto, as linhas de código que existem num bloco ***beginShape()-endShape()*** são na sua maioria compostas por uma destas três funcionalidades. **Não é possível** a execução de transformações espaciais ou ainda alterar a configuração do aspecto de primitivas (cor, espessura, etc.), dentro de um bloco ***beginShape()-endShape()***.

O ***beginShape*** pode receber um dos seguintes parâmetros de configuração: **POINTS, LINES, TRIANGLES, TRIANGLE_FAN, TRIANGLE_STRIP, QUADS**, ou **QUAD_STRIP**. De acordo com o parâmetro indicado, a forma que será criada obedecerá uma determinada lógica no seu desenho. Por exemplo, se for **POINTS**, significa que a forma será desenhada apenas com pontos enquanto o **QUAD_STRIP** indica que ela será uma malha geométrica, quadrangular. Se nada lhe for passado, os vértices serão interpretados como pertencendo a um polígono irregular.

A ***endShape***() pode receber como parâmetro o valor **CLOSE**, indicando o final da definição da forma.

Uma utilização bastante eficaz dos blocos ***beginShape-endShape*** pode se obter com a utilização de uma classe. Nessa ótica, a classe poderia conter vários métodos, cada um dedicado a uma determinada forma, por exemplo ☺

Para ilustrarmos de forma abrangente a utilização deste bloco, iremos fazer um programa que desenha uma figura de fundo com base em curvas e depois, ao clicar o rato, são desenhadas algumas outras figuras, na posição em que o rato estava. Na primeira figura, boa parte dela é definida utilizando o bloco ***beginShape-endShape***, enquanto as demais figuras criadas ao clicar o rato sobre a janela, são totalmente definidas dentro de blocos ***beginShape-endShape***. Criaremos duas classes para organizar o código e facilmente reutilizar o que é definido dentro do bloco ***beginShape-endShape.*** Iremos aplicar algumas transformações 2D de forma isolada nas figuras, de forma a se perceber como pode se aplicar de forma focada as acções nas partes da figura que são definidas pelos blocos. Vamos ao exemplo:



1. Crie um programa de nome **ExercTopico6_3.pde.** Neste programa, já há necessidade de utilizar a função ***draw***(), pois precisamos de interacção com o rato, logo, o programa tem que entrar em ciclo. Vamos começar por declarar cada objecto. Inicialmente, vamos definir o mais simples, que vai conter os polígonos. Daremos o nome de ***Poligonos*** a classe. Ela vai ter um construtor vazio, pois não é necessário que nada de especial ao ser instanciada. Note-se que estamos desenhando figuras simples, mas poderiam ser qualquer outra coisa ☺

```
class Poligonos {
  Poligonos() { // Construtor vazio
  }
  // Desenha um poligono em forma de L
  void formaA() {
    beginShape();
      vertex(20, 20);
      vertex(40, 20);
      vertex(40, 40);
      vertex(60, 40);
      vertex(60, 60);
      vertex(20, 60);
    endShape(CLOSE);
  }

  // Desenha um retângulo
  void formaB() {
    beginShape();
      vertex(30, 20);
      vertex(85, 20);
      vertex(85, 75);
```

```
      vertex(30, 75);
    endShape(CLOSE);
  }

  // Desenha uma malha poligonal formada por triângulos
  void formaC(){
    beginShape(TRIANGLE_STRIP);
      vertex(30, 75);
      vertex(40, 20);
      vertex(50, 75);
      vertex(60, 20);
      vertex(70, 75);
      vertex(80, 20);
      vertex(90, 75);
    endShape();
  }
}
```

2. Vamos definir a outra classe que conterá as formas curvas. Daremos o nome de ***Curvas*** a classe. Essa classe é mais complexa que a anterior. Utilizamos tanto a função ***curve***() para desenhar as curvas como o bloco ***beginShape-endShape*** para definir de forma mais precisa uma curva, além de mais extensa (o número de pontos pode ser qualquer e não apenas quatro, como é com ***curve***). Utilizamos ainda uma formulação matemática para desenhar curvas *spline*, definida em ***curveEllipse***. Por fim, a função ***curveTightness*** é utilizada para deformar as curvas de forma conveniente:

```
class Curvas {
  float radius = 165;
  float angle = 0;
//círculo externo
  float[]cx = new float[4];
  float[]cy = new float[4];
//círculo do meio
  float[]cx2 = new float[4];
  float[]cy2 = new float[4];
  Curvas() {
    strokeWeight(1.5);
    smooth();
    desenha();
  }

  void desenha() {
    for (int i =0; i<4; i++){
//elipse externa
      cx[i] = width/2+cos(radians(angle))*radius;
      cy[i] = height/2+sin(radians(angle))*radius;
//elipse central
      cx2[i] = width/2+cos(radians(angle))*(radius*.85);
      cy2[i] = height/2+sin(radians(angle))*(radius*.85);
      angle+=360.0/4.0;
    }
//curva externa
    curveTightness(-3);
    curve(cx[3], cy[3], cx[0], cy[0], cx[1], cy[1], cx[2],
cy[2]);
    curve(cx[0], cy[0], cx[1], cy[1], cx[2], cy[2], cx[3],
cy[3]);
```

```
    curve(cx[1], cy[1], cx[2], cy[2], cx[3], cy[3], cx[0],
cy[0]);
    curve(cx[2], cy[2], cx[3], cy[3], cx[0], cy[0], cx[1],
cy[1]);
//curva central
    curveTightness(2);
    noFill();
    formaA(); // invoca o método que desenha uma das formas
    for (int i=0; i<4; i++){
      fill(255);
      ellipse(cx[i], cy[i], 10, 10);
      ellipse(cx2[i], cy2[i], 10, 10);
    }
//Curva central
    curveEllipse(6, width/2, height/2, radius*.2, -8, 127,
true);
//curva interna
    curveEllipse(8, width/2, height/2, radius*.05, 0, 0,
false);
  }

  void formaA(){
    noFill();
    stroke(random(255), random(255), random(255));
    beginShape();
      curveVertex(cx2[3], cy2[3]);
      curveVertex(cx2[0], cy2[0]);
      curveVertex(cx2[1], cy2[1]);
      curveVertex(cx2[2], cy2[2]);
      curveVertex(cx2[3], cy2[3]);
      curveVertex(cx2[0], cy2[0]);
      curveVertex(cx2[1], cy2[1]);
    endShape();
  }

  void formaB(int pts, float[]cx, float[]cy){
    beginShape();
      for (int i=0; i<pts; i++){
        if (i==0){
          curveVertex(cx[pts-1], cy[pts-1]);
        }
        curveVertex(cx[i], cy[i]);
        if (i==pts-1){
          curveVertex(cx[0], cy[0]);
          curveVertex(cx[1], cy[1]);
        }
      }
    endShape(CLOSE);
  }

// Desenho de elipses utilizando uma função spline
  void curveEllipse(int pts, int x, int y, float radius,
float tightness, int fillCol, boolean isNodeVisible){
    float[]cx = new float[pts];
    float[]cy = new float[pts];
    float angle = 0;
    for (int i=0; i<pts; i++){
      cx[i] = x+cos(radians(angle))*(radius);
      cy[i] = y+sin(radians(angle))*(radius);
      angle+=360.0/pts;
    }
```

```
    curveTightness(tightness);
    fill(fillCol);
    formaB(pts, cx, cy);
// desenha os pontos de controle
// alguns pontos não são para desenhar
if (isNodeVisible){
      fill(255);
      for (int i=0; i<pts; i++){
        ellipse(cx[i], cy[i], 10, 10);
      }
    }
  }
}
```

3. Para terminar, temos que definir o programa principal, onde instanciamos as classes e exemplificamos como podemos atuar de forma focada nas figuras que se encontram definidas dentro de métodos das classes em blocos ***beginShape-endShape***. Aplicamos algumas transformações 2D, alterando posições e girando algumas dessas figuras e com isso, criando uma visualização diferente da anterior. Mais uma vez, as alterações poderiam ser muitas outras, sendo a nossa imaginação o limite:

```
Curvas curva;
Poligonos poligono;
void setup(){
  size(400, 400);
  background(255);
  curva = new Curvas();
  poligono = new Poligonos();
// Isolamos as transformações 2D para repetir o desenho da
// curva definida em formaA
// repete-se a direita e esquerda, com cor aleatória
  pushMatrix();
  for (int i = 0; i <= 4; ++i) {
      translate(-10, 0);
      curva.formaA();
  }
  popMatrix();
  pushMatrix();
  for (int i = 0; i <= 4; ++i) {
      translate(10, 0);
      curva.formaA();
  }
  popMatrix();
}
void draw() {
}
// Invocada quando premimos o botão do rato
void mousePressed() {
  translate(mouseX, mouseY);
// desenha o polígono definido em forma
  poligono.formaA();
// isola as transformações 2D aplicadas nas demais formas
  pushMatrix();
    rotate(random(PI));
    poligono.formaB();
    rotate(random(PI));
    poligono.formaC();
  popMatrix();
}
```

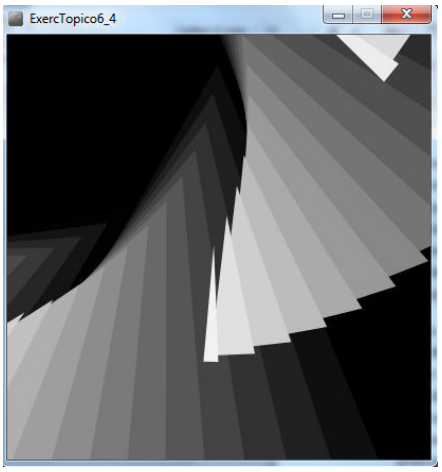
4.  e veja o que acontece…clique o rato sobre a janela e veja aparecer as outras formas…

3. Criação de padrões visuais

Muitos padrões visuais, que esteticamente causam uma impressão harmoniosa, podem ser obtidos a partir da repetição de formas básicas. Utilizar transformações geométricas (2D ou 3D) como a rotação, translação ou alteração de escala, juntamente com alguma função ou expressão que controla a variação de parâmetros de posição dos elementos visuais de forma incremental e gradual, permite que facilmente se consiga criar padrões. Por exemplo, as funções matemáticas que calculam curvas (co-senos, senos, p. ex.), retas ou ainda geram geometria fractal, são bons exemplos de funções que ajudam a conseguir gerar padrões.

Nesta seção iremos analisar 3 exemplos de criação de padrões visuais com base nessas considerações, e com isso, ilustrar a potencialidade do efeito visual resultante. Vamos ainda lançar mão de um subterfúgio muito versátil da programação: a recursividade. Ela é obtida quando dentro de uma função (ou método) fazemos chamada a ela mesma. Um exemplo clássico é dado no cálculo do factorial de um número. Em programação, este cálculo é tipicamente resolvido com uma função recursiva, em que a cada chamada dela própria, o próximo valor é calculado e repassado para a chamada seguinte. A recursividade apesar de versátil traz alguns problemas de gestão memória, pois se uma função se chamar um número demasiado de vezes, poderá causar o arrebentamento da memória e consequentemente, do próprio programa ☹

O primeiro exemplo vai desenhar uma figura geométrica básica: um triângulo. A cada passada, é executada uma transformação geométrica 2D de rotação, ou seja, em cada ciclo de execução, temos a matriz de transformação sendo actualizada, girando de ***spin*** (variável de memória) graus tudo o que vai ser desenhado de seguida na cena. Após a rotação, os próprios vértices têm o seu valor alterado, ou seja, as suas componentes x e y são incrementadas ou decrementadas de ***shift*** ou ***shift/2***. Com isso, consegue-se criar um padrão visual interessante, que poderia ser utilizada com qualquer outra figura geométrica regular, por exemplo. Vamos ao exemplo, passo-a-passo:



1. Crie um programa de nome **ExercTopico6_4.pde.** Neste programa, não há necessidade de utilizar a função ***draw***(), pois vamos utilizar uma função recursiva, que se repetirá infinitamente, até a condição limite ser alcançada. Inicialmente definimos as variáveis de memória globais para conter os vértices do triângulo (***p***), o fator de incremento/decremento para deslocar a posição (***shift***), o ângulo de rotação (***spin***), o fator de incremento da tonalidade cinza para pintar o triângulo (***fade***) e a cor que pinta (***fillCol***). Tud isso é declarado no programa principal:

```
//Gira o triângulo como um padrão
Point[]p = new Point[3];
float shift = 2;
float fade = 0;
float fillCol = 0;
float spin = 0;
```

2. ***Point*** é uma classe criada para gerir pontos espaciais. Portanto, crie a classe também com esse nome, anexada ao projeto:

```
class Point {
  int x;
  int y;
  int z;
  Point (int xcoord, int ycoord)
  {
    x = xcoord;
    y = ycoord;
    z = 0;
  }
  Point (int xcoord, int ycoord, int zcoord)
  {
    x = xcoord;
    y = ycoord;
    z = zcoord;
  }
}
```

3. Voltando ao programa principal, vamos agora declarar a função ***setup***(). Nela, inicializaremos os valores das variáveis globais. Note que a posição inicial do triângulo é dada pelo tuplo {(1, 390), (390, 390), (200, 1)}, que define os seus 3 vértices:

```
void setup(){
  size(400, 400);
  background(0);
  smooth();
// fade = 255/(400/2.0/2.0) = 2.5
  fade = 255.0/(width/2.0/shift);
// spin = 360.0/(400/2.0/2.0) = 3.6
  spin = 360.0/(width/2.0/shift);
  p[0] = new Point(1, height-1); // (1, 390)
  p[1] = new Point(width-1, height-1); // (390, 390)
  p[2] = new Point(width/2, 1); // (200, 1)
  noStroke();
  triBlur();
}
```

4. Por fim, declaramos a função triBlur() e dentro dela, executamos sequencialmente os seguintes passos: colore, recalcula o tom cinza para a próxima iteração, gira a cena de 3.6 graus, desenha o triângulo, porém, actualiza o valor de cada vértice. Por fim, testa se a

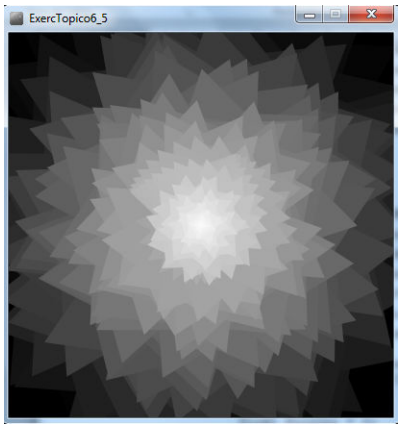
coordenada x do primeiro vértice tem valor ainda inferior 200, se tiver, chama-se a si própria mais uma vez, senão, para o processamento.

```
void triBlur(){
  fill(fillCol);
  fillCol+=fade;
  rotate(spin);
  triangle(p[0].x+=shift, p[0].y-=shift/2,
  p[1].x-=shift, p[1].y-=shift/2, p[2].x, p[2].y+=shift);

  if(p[0].x<width/2){
// chamada recursiva
    triBlur();
  }
}
```

5.  e veja o que acontece…não esqueça de salvar!

No segundo exemplo, vamos apenas alterar alguns pormenores do código anterior e verificar as alterações visuais que conseguimos. Neste exemplo, iremos alterar o valor do fator de deslocamento a ser aplicado aos vértices do triângulo para 1.0 (***shift***). Além disso, a própria posição inicial do 1º triângulo é diferente, sendo totalmente centralizada em relação a janela. Para garantir isso, além do próprio valor das coordenadas *x* e *y* dos vértices serem definidas com valores apropriados, executa-se uma translação para posicionar toda a cena (e consequentemente iniciar a matriz de transformação) e ter como referência espacial de arranque o centro da janela.

Por fim, a nível de rotação, são executadas duas (tente comentar a 2ª e veja o efeito). A primeira é idêntica ao exemplo anterior. A segunda utiliza a variável local ***rot*** que é incrementada a cada iteração da função ***triBlur()*** com o valor que está em ***spin*** convertido para graus em radianos. O limite de chamadas a ***triBlur()*** de forma recursiva ocorre enquanto a coordenada *x* do primeiro vértice tiver um valor inferior a zero.



1. Crie um programa de nome **ExercTopico6_5.pde.** Utilize o código do exemplo anterior, efectuando as alterações que estão a vermelho.

```
Point[]p = new Point[3];
float shift = 1.0;
float fade = 0;
```

```
float fillCol = 0;
float rot = 0;
float spin = 0;
void setup(){
  size(400, 400);
  background(0);
  smooth();
  fade = 255.0/(width/2.0/shift);
  spin = 360.0/(width/2.0/shift);
  p[0] = new Point(-width/2, height/2);//(-200, 200)
  p[1] = new Point(width/2, height/2);// (200, 200)
  p[2] = new Point(0, -height/2); // (0, -200)
  noStroke();
  translate(width/2, height/2); // centro da janela
  triBlur();
}
void triBlur(){
  fill(fillCol);
  fillCol+=fade;
  rotate(spin);
// uma variação é criada com essa rotação adicional
  rotate(rot+=radians(spin));
  triangle(p[0].x+=shift, p[0].y-=shift/2, p[1].x-=shift,
    p[1].y-=shift/2, p[2].x, p[2].y+=shift);
  if(p[0].x<0){
// chamada recursiva
    triBlur();
  }
}
```

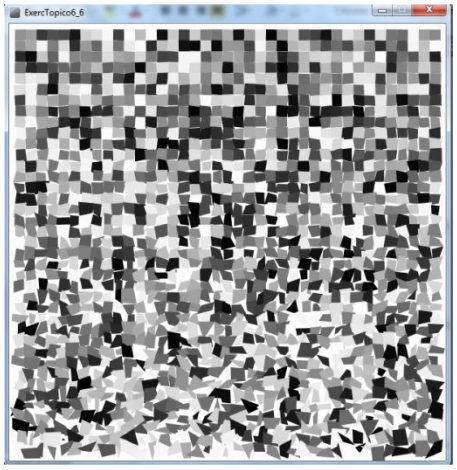
2.  e veja o que acontece…

Como último exemplo de criação de padrões com formas básicas, vamos utilizar pequenos quadrados que gradualmente serão desenhados de forma irregular. Utilizaremos a função ***quad (x1, y1, x2, y2, x3, y3, x4, y4)*** para desenhar os quadrados irregulares. O desenho ocorrerá linha a linha, sendo que a cada nova linha, um fator mais acentuado de deformação (que corresponde a posição dos vértices menos alinhadas umas com as outras) vai ser gerado. Dois laços de repetição são utilizados. O primeiro controla a posição vertical em que a nova fileira de quadrados será desenhada, enquanto o segundo, controla a quantidade de quadrados que é desenhada em cada linha. Para cada quadrado é reservada uma área com altura e largura iguais a 15 pixels.

De forma a garantir que a próxima fileira de quadrados inicia sempre no canto esquerdo da janela e na linha que é dada por *k * 15* pixels (calculado no ciclo mais externo), é executada uma translação com ***translate(0, quadH*k)*** logo no início do ciclo mais externo. Para evitar o efeito acumulativo das transformações causadas por esse ***translate*** e os demais que ocorrerão no ciclo mais interno, executa-se o *reset* da matriz de transformação a cada iteração do ***for*** mais externo, utilizando ***resetMatrix()***.

Dentro do ciclo for mais interno também se executa uma outra translação. Essa serve para ir posicionando os quadrados ao longo da linha que está sendo desenhada. Cada novo quadrado é deslocado 15 *pixels* para a direita do anterior executando-se um ***translate(quadW, 0).***

Para terminar, de forma a se causar a irregularidade no quadrado, os seus vértices são actualizados a cada iteração com o valor retornado pela função r. Esta função retorna um valor aleatório que é calculado numa faixa entre ***-i*randShift*** e ***i*randShift***. Conforme o valor de i aumenta, aumenta a faixa de valores possíveis de serem gerados. Como i varia de 0 a um valor menor que ***height-quadH*** (ou seja, 600 – 15 = 585), temos que o valor retornado por ***r(i)*** irá de zero até um máximo aleatório obtido entre -585 *.2 e 585 * .2. Isso, evidentemente, causará um efeito de irregularidade que será percebido de forma gradual, como se fosse uma desintegração progressiva do padrão inicialmente definido. Vamos ao código!



1. Crie um programa de nome **ExercTopico6_6.pde.** Vamos definir as variáveis globais que utilizaremos:

```
            float randShift = .2; // fator para aleatoriedade
// distância horizontal para encaixar o quadrado
            int quadW = 15; int quadH = quadW;
// array com valores que definiram as coordenadas dos vértices
// respectivamente -15/2, -15/2, 15 e 15
            float[]q = { -quadW/2, -quadH/2, quadW, quadH };
```

2. Vamos definir a função que calcula o valor aleatório para incrementar a posição dos vértices:

```
            float r(int i){
                return random(-i*randShift, i*randShift);
            }
```

3. Vamos por fim especificar os ciclos dentro de ***setup***(), e garantir que é feito o desenho progressivo dos quadrados, linha a linha, e em cada linha, coluna a coluna:

```
            void setup() {
              size(600, 600);
              background(255);
              smooth();
              noStroke();
// Ciclo mais externo para controlar o salto linha a linha
              for (int i=0, k=1; i<height-quadH; i+=quadH, k++){
// reset da matriz de transformação para evitar o acumular destas…
                resetMatrix();
// Posiciona no canto esquerdo extremo da janela, na linha atual
                translate(0, quadH*k);
// Ciclo interno que controla a posição coluna a coluna
```

```
                for (int j=0; j<width-quadW; j+=quadW){
// move 15 pixels para a direita
                  translate(quadW, 0);
                  fill(random(0, 255));
// r(k) é invocada para calcular os vértices
                  quad(q[0]+r(k), q[1]+r(k),q[0]+q[2]+r(k), q[1]+r(k),
                      q[0]+q[2]+r(k), q[1]+q[3]+r(k),
                      q[0]+r(k), q[1]+q[3]+r(k));
                 }
                }
            }
```

4.  e veja o que acontece…

Programação Criativa – Tópico 7

***Temática**: Efeito animado*
***Duração**: semanas 5, 6, 7 e 8*

Actividade 7: Aprender a explorar a animação utilizando a detecção de colisão e gravidade (entre outros) na criação de visualizações criativas.

Competências a desenvolver:

- Saber implementar efeitos animados
- Aplicar efeitos dinâmicos básicos

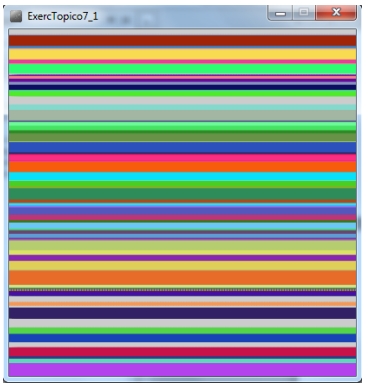
1. Animação

Efeitos animados que ocorrem sem ter nenhum controlo direto do utilizador, podem ser aplicados para dar um toque mais criativo a visualização oferecida. Esses efeitos podem ser obtidos de várias formas. Por exemplo, podemos aplicar equações básicas da física mecânica para com isso simularmos o comportamento de um corpo em queda livre, ou ainda em colisão com algum outro. Podemos utilizar soluções muito mais simples, como por exemplo apenas controlar o movimento dentro da janela tendo como base algum conjunto de condições definidas por nós mesmos: utilizar os limites da janela de visualização ou outra posição previamente definida, ou ainda o tempo decorrido.

Neste último tópico iremos percorrer algumas das aproximações que podem ser utilizadas para se obter o efeito animado na visualização. Não será totalmente abrangente a abordagem, mas elucidativa o suficiente para você posteriormente conseguir perceber outra solução de animação.

Vamos ao primeiro exemplo. O objectivo é desenhar uma série de quadrados coloridos que vão sendo desenhados gradualmente na horizontal, da esquerda para a direita, criando um efeito visual de fitas. Eles são desenhados coluna a coluna, num total de 100 em cada coluna. Cada quadrado possui uma posição vertical, dimensão e cor diferente. Inicialmente (em ***setup***) são preenchidos os quatro vectores para armazenar os parâmetros (posição e largura) para os 100 quadrados, e suas respectivas cores. Tudo é criado de forma completamente aleatória. Posteriormente, em ***draw***() é controlada a maneira como as colunas da janela vão ser preenchidas. Em cada uma, são desenhados os 100 quadrados com os parâmetros e cores previamente carregados. A cada iteração, é executado um ligeiro deslocamento para a direita da posição inicial para o desenho do quadrado, sendo incrementado o valor da variável *speed* a posição em *x*, ou seja, o desenho repete-se, agora para a próxima coluna da janela. Como ***draw***() repete-se indefinidamente, a dada altura o desenho dos quadrados prossegue para além do limite direito da janela (o que não é muito "esperto"…). A inserção de um ***if-else*** a testar o limite, resolveria essa situação, por exemplo.

Neste exemplo fazemos uso do tipo ***color***, que é um tipo pré-definido no **processing** e serve para conter atributos de variáveis que contêm dados de cor. Dê uma olhada em https://processing.org/reference/color_datatype.html para perceber melhor como este tipo funciona ☺



1. Abra o **processing** e crie um programa de nome **ExercTopico7_1.pde**
2. Vamos começar por definir algumas variáveis globais, que vão conter os parâmetros de posição, largura e cor dos quadrados, além do valor do deslocamento de cada coluna a ser desenhada. Note que utilizamos o tipo color que é pré-definido no **processing** e serve para conter atributos de cor:

```
// Variáveis globais
int x;
float speedX = 3.0;
int shapes = 100;
// Cria 3 vetores de 100 elementos do tipo float
float[]y = new float[shapes];
float[]w = new float[shapes];
float[]h = new float[shapes];
// Cria 1 vetor de 100 elementos do tipo color
color[]colors = new color[shapes];
```

3. Vamos inicializar os vectores com os parâmetros para cada um dos 100 quadrados e respectivas cores. Tudo será definido aleatoriamente. O vector *y* conterá as posições verticais na janela, *w* a largura (um máximo de 17 pixels e um mínimo de 2, no caso do valor gerado ser zero) e *h* a altura, que será sempre igual a w para garantir termos um quadrado:

```
void setup(){
  size(400, 400);
  frameRate(30);
  noStroke();
  // Preenchimento dos vetores com valores aleatórios
  for (int i=0; i<shapes; i++){
    y[i]=random(height);
    w[i]=random(15)+2;
    h[i]=w[i];
    colors[i]=color(random(255), random(255), random(255));
  }
}
```

4. Em ***draw***() iremos desenhar seguidamente coluna a coluna, criando o efeito de fitas horizontais de largura e cores diferentes. Experimente alterar o valor de *speedX* para 20 e veja o que acontece:

```
void draw(){
  for (int i=0; i<shapes; i++){
    fill(colors[i]);// pinta de uma dada cor
    rect(x, y[i], w[i], h[i]); // desenha quadrados
```

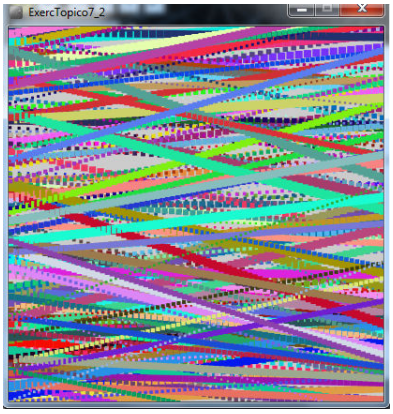
```
}
x+=speedX;// avanço horizontal dos quadrados
```

5.  e veja o que acontece…

Como poderíamos alterar o exemplo anterior para fazer com que as fitas, em lugar de serem sempre rectas e horizontais, fossem ligeiramente inclinadas para cima ou para baixo, resultando num efeito visual como o ilustrado abaixo? Além disso, em lugar de todas as fitas avançarem a mesma velocidade, fazer com que algumas fossem mais rápidas que outras?

Para criar essa situação, teremos em cada iteração para desenhar uma nova coluna na janela, incrementar não só a componente *x* da posição do quadrado, mas também a *y*. Para conseguirmos o efeito de subir ou descer, bastará garantirmos que o valor a adicionar em *y* é positivo ou negativo (causando a inclinação da recta variar). . Note que se quiséssemos criar um efeito *zig-zag*, teríamos que garantir que o valor em *y* era ora incrementado e ora decrementado ☺. Para fazer com que uma "fita" atinja o limite esquerdo da janela mais rápido, bastará fazermos com que o valor de incrementos em *x* sejam diferentes, ou seja, quão maior, mais rápido isso acontecerá.

Precisaremos de mais algumas variáveis (para os incrementos em *x* e em *y* no preenchimento de cada coluna, e para a posição em *x* de cada quadrado) e tudo novamente sendo carregado com valores aleatórios, devidamente definidos em termos de valores mínimos e máximos.



1. Crie um programa de nome **ExercTopico7_2.pde.** Neste programa nós iremos apenas efetuar as modificações necessárias para obter o resultado acima, desenhando agora 200 quadrados em lugar de 100. O código está a vermelho para evidenciar as alterações:

```
// Variáveis globais
int shapes = 200;
// Vetores adicionais para controlar os incrementos
// em x e y dos quadrados coluna a coluna
float[]speedX = new float[shapes];
float[]speedY = new float[shapes];
float[]x = new float[shapes];
float[]y = new float[shapes];
float[]w = new float[shapes];
float[]h = new float[shapes];
color[]colors = new color[shapes];
void setup(){
```

```
  size(400, 400);
  frameRate(30);
  noStroke();
  // Preenchimento aleatório de cada vetor
  for (int i=0; i<shapes; i++){
    x[i]=0;
    y[i]=random(height);
    w[i]=random(2, 10);
    h[i]=w[i];
    colors[i]=color(random(255), random(255), random(255));
    // O incremento em x vai de 5 a 10 pixels
    speedX[i] = random(5, 10);
    // O incremento em y vai de -2 a 2 (inclinação variável da reta)
    speedY[i] = random(-2, 2);
  }
}
void draw(){
  // Desenha coluna a coluna, 200 quadrados
  for (int i=0; i<shapes; i++){
    fill(colors[i]);
    rect(x[i], y[i], w[i], h[i]);
    // Haverão quadrados a avançar mais rápido que outros
    x[i]+=speedX[i];
    // Haverão quadrados a descer, subir ou ficar na horizontal
    y[i]+=speedY[i];
  }
}
```

2.  e veja o que acontece…

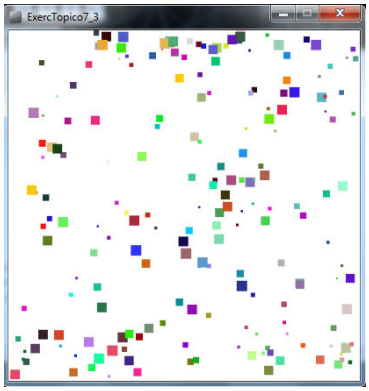
Nos exemplos anteriores, ocorre sempre uma situação indesejável. Primeiro, o processo de desenho ocorre de forma infinita, mesmo que já atingidos os limites máximos da janela de visualização. Isto é um bom exemplo de má utilização de recursos, pois o programa fica exigindo um processamento contínuo, embora, em termos visuais, já não seja possível ver nada. Isso só seria admissível se tivéssemos, por exemplo, algum evento que estivesse a ser monitorado (por exemplo, o rato, o teclado ou outro dispositivo de interacção), para com base nele, algo específico ocorrer no nosso programa. Logo, neste caso concreto, a detecção de colisões com os limites da janela seria desejável, para evitar essa sobrecarga desnecessária.

Por sua vez, a detecção de colisões também acaba por ser uma forma de animação, pois ela automatiza o comportamento dos objectos gráficos que tenhamos em nossa cena. Há muitas formas de fazê-la, sendo a colisão de partículas, um exemplo clássico de sua aplicação.

Tendo isso em conta, vamos rever o programa anterior de maneira a ele contemplar a situação de colisão com os limites da janela de visualização. Mais ainda: iremos introduzir o conceito de *time out*, que é muito útil quando queremos limitar o tempo de execução de nossos processos.

No exemplo a seguir, que é uma extensão do anterior, iremos desenhar uma série de quadrados a partir do centro da janela. Conforme a iteração ocorre, esses quadrados irão se mover em direcções e velocidades diferentes (para cima, para baixo, para esquerda ou direita). Quando eles atingem algum limite da janela, o que fazemos é simplesmente trocar o sinal de seus incrementos na horizontal e vertical. Logo, um quadrado que tenha incremento -2 em *x* e que atinja a borda esquerda da janela, terá o seu incremento passado para 2. Se ele tiver o seu incremento em *y* igual a 5 e atingir a borda inferior da janela, passará a ter esse valor a -5. Com

isso, conseguimos que os quadrados “reajam” ao toque com os limites da janela, andando para trás, frente, cima ou para baixo. O resultado final é uma animação contínua e automática.



1. Crie um programa de nome **ExercTopico7_3.pde.** Como pode notar, ele é muito parecido programa anterior na sua parte inicial, só exigindo a inclusão de uma nova variável, para controlo do tempo de execução. Além disso, passamos também a ter incrementos negativos em x, para causar movimentos não só para direita, mas também para a esquerda de alguns quadrados. A vermelho as diferenças:

```
int shapes = 200;
float[]speedX = new float[shapes];
float[]speedY = new float[shapes];
float[]x = new float[shapes];
float[]y = new float[shapes];
float[]w = new float[shapes];
float[]h = new float[shapes];
color[]colors = new color[shapes];
int timeLimit = 15;
void setup(){
  size(400, 400);
  frameRate(30);
  noStroke();
  // Preenche aleatoriamente
  for (int i=0; i<shapes; i++){
    x[i]=width/2;// Todos estão no centro da janela no
início
    y[i]=height/2;
    w[i]=random(2, 12); // Largura e alturas entre 2 a 12
pixels
    h[i]=w[i];
    colors[i]=color(random(255), random(255), random(255));
   // Os quadrados avançam para esquerda ou direita
   // com velocidades diferentes
    speedX[i] = random(-5, 5);
   // Os quadrados vão para cima ou para baixo
   // com velocidades diferentes
    speedY[i] = random(-2, 2);
  }
}
```

2. A grande diferença está no controlo de como as colisões são detectadas e alteram a direcção do movimento do quadrado, dentro de ***draw***(). Vamos utilizar uma função que calcula o tempo decorrido em milésimos de segundo:

```
void draw(){
```

```
  background(255);
  for (int i=0; i<shapes; i++){
    fill(colors[i]);
    rect(x[i], y[i], w[i], h[i]);
    x[i]+=speedX[i];
    y[i]+=speedY[i];
// Verifica as situações de colisão com as bordas da janela
    if (x[i] > width-w[i]){ // borda direita
        x[i] = width-w[i];
        speedX[i]*=-1;
      }
      else if (x[i] < 0){ // borda esquerda
          x[i] = 0;
          speedX[i]*=-1;
        }
          else if (y[i] > height-h[i]){ // borda inferior
              y[i] = height-h[i];
              speedY[i]*=-1;
            }
            else if (y[i] < 0){ // borda superior
              y[i] = 0;
              speedY[i]*=-1;
            }
  }
// Verifica o tempo decorrido, e para se atingi-lo
  if (millis() >= timeLimit*1000){
    noLoop();
  }
}
```

4.  e veja o que acontece…

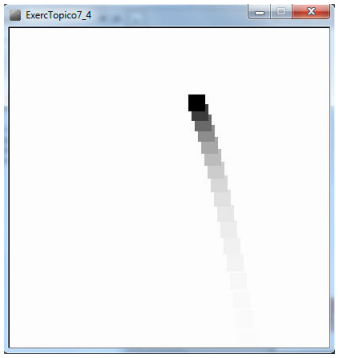
5. Por fim, para criarmos um efeito de rastro nos quadrados conforme eles se movem, basta substituir ***background(255)*** pelo desenho de um rectângulo com dimensões iguais a da janela de visualização, pintado de branco, porém com alguma transparência (alfa a 40%):

```
fill(255, 40);
rect(0, 0, width, height);
```

6.  e veja o que acontece…

Podemos também criar efeitos de colisão levando em conta a gravidade ou ainda o atrito e inércia. Para isso temos que fazer com que a velocidade com que a posição em *x* e *y* variem de acordo. Por exemplo, se o quadrado tiver uma velocidade em *y* variando gradualmente para mais e para menos, conseguimos simular o efeito da gravidade. Se considerarmos que quando o quadrado toca os limites da janela este deve sofrer alguma desaceleração em *y*, basta que o incremento passe a diminuir de valor gradualmente, até **zero**. Portanto, com algumas adaptações, o programa anterior consegue simular a gravidade e o amortecimento e fricção que ocorrem quando o quadrado bate nas bordas.

Vamos ver mais dois exemplos. O primeiro considera apenas o efeito da gravidade na componente *y* (vertical da posição). É criada uma variável para conter o valor atribuído a gravidade, e depois, ela é incrementada ao valor da velocidade na vertical. Como a velocidade ora fica positiva, ora negativa (dependendo de qual borda o quadrado atingiu), o resultado é uma velocidade variável, ora crescente (após atingir a borda inferior), ora decrescente (após atingir a borda superior). A nível de deslocamento na horizontal, não é feita nenhuma alteração ao que já estava, apenas fazendo a inversão de sentido do movimento horizontal do quadrado.



1. Crie um programa de nome **ExercTopico7_4.pde.** Como irá notar, ele é uma adaptação do programa anterior, com algumas pequenas alterações, para simular a gravidade (em vermelho as partes novas):

```
            float speedX, speedY;
            float x, y, w, h;
// variável para conter o valor da aceleração
            float gravity;
            void setup(){
              size(400, 400);
              x=width/2;
              w=20;
              h=w;
              fill(0);
              speedX = 4;
// Define o valor da aceleração
              gravity = .9;
            }
            void draw(){
              fill(255, 60);
              rect(0, 0, width, height);
              fill(0);
              rect(x, y, w, h);
              x+=speedX;
// Conforme as iterações ocorrem, a velocidade vai sendo incrementada
// continuamente com o valor da gravidade
              speedY+=gravity;
              y+=speedY;
// Verifica a colisão com as bordas da janela
              if (x > width-w){ // borda direita
                  x = width-w;
                  speedX*=-1;
                }
                else if (x < 0){ // borda esquerda
                    x = 0;
                    speedX*=-1;
```

```
        }
        else if (y > height-h){ // borda inferior
            y = height-h;
            speedY*=-1;
          }
          else if (y < 0){ // borda superior
              y = 0;
              speedY*=-1;
            }
}
```

2.  e veja o que acontece…

Neste outro exemplo, temos em consideração o amortecimento e fricção que deve ocorrer no quadrado, conforme ele bate na borda inferior da janela de visualização. Isso causa com que gradualmente o quadrado perca velocidade, pois as forças causadas pela fricção e amortecimento, causam a redução contínua das velocidades horizontal e vertical do quadrado.

1. Crie um programa de nome **ExercTopico7_5.pde.** Faça as alterações indicadas a vermelho:

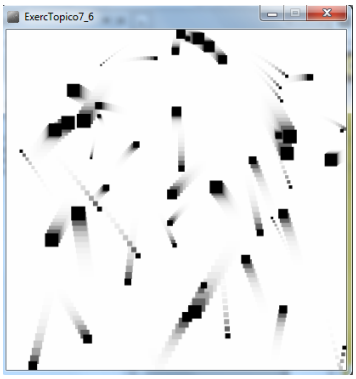
```
float speedX, speedY;
float x, y, w, h;
// aceleração da gravidade
float gravity;
// fricção e amortecimento
float damping, friction;
void setup(){
  size(400, 400);
  x=width/2;
  w=20;
  h=w;
  fill(0);
  speedX = 4;
  // define parâmetros básicos de dinâmica
  gravity = .5;
  damping = .8;
  friction = .9;
}
void draw(){
  fill(255, 60);
  rect(0, 0, width, height);
  fill(0);
  rect(x, y, w, h);
  x+=speedX;
  speedY+=gravity;
  y+=speedY;
  // Verifica as colisões nas bordas da janela
    if (x > width-w){
        x = width-w;
        speedX*=-1;
      }
        else if (x < 0){
          x = 0;
          speedX*=-1;
        }
          else if (y > height-h){
            y = height-h;
            speedY*=-1;
            // Cria o efeito de amortecimento e fricção
```

```
            speedY*=damping;
            speedX*=friction;
          }
            else if (y < 0){
              y = 0;
              speedY*=-1;
            }
    }
```

2.  e veja o que acontece…

Um outro efeito interessante é o de explosões ou jactos. A ideia é sempre a partir de uma determinada posição, uma série de objectos surgir e se propagarem. Dependendo do efeito desejado, podemos fazer com que esses objectos se movam todos em todas as direcções, ou apenas um conjunto destas. Com isso, conseguimos simular um *spray*, jacto de água ou rebentamento de um fogo-de-artifício, por exemplo. Os programas que temos abordado até agora, com as devidas adaptações, podem perfeitamente gerar este efeito.

O exemplo seguinte irá mostrar uma "chuva" de quadrados, que ocorre a partir da zona central, no topo da janela de visualização. Conforme os quadrados são criados, eles "saltam" para baixo, com velocidades diferentes e com trajectórias ligeiramente diferentes umas das outras. Os quadrados também têm uma taxa de crescimento variado, ou seja, para conseguir o efeito "borrifo", os quadrados não "nascem" sempre em mesmo número, porém, número crescente até chegar ao máximo definido. Outro detalhe é que a nível de colisões, eles só batem contra as bordas direita, esquerda e inferior da janela de visualização. O rasto do deslocamento completa a noção visual de *spray*. Vamos ao código.



1. Crie um programa de nome **ExercTopico7_6.pde.** Na mesma, iremos utilizar vários vectores para conter os parâmetros de localização e dimensão dos 200 quadrados, além da sua velocidade, e grandezas dinâmicas de seu comportamento (gravidade, fricção e amortecimento):

```
int shapes = 200;
float[]w = new float[shapes];
float[]h = new float[shapes];
float[]x = new float[shapes];
float[]y = new float[shapes];
float[]xSpeed = new float[shapes];
float[]ySpeed = new float[shapes];
float[]gravity = new float[shapes];
float[]damping = new float[shapes];
float[]friction = new float[shapes];
```

2. Também definiremos variáveis para conter o incremento de nascimento de quadrados a cada iteração, o contador, definir a intensidade do borrifo. Depois disso, inicializamos todas essas variáveis, na maior parte, de forma aleatória, para garantir um comportamento variável de cada quadrado:

```
// controla a taxa de aparecimento de quadrados
float shapeCount;
float birthRate = .25;
// Os quadrados são "borrifados" com uma dada intensidade
float sprayWidth = 5;
void setup(){
  size(400, 400);
  noStroke();
  //inicializa com valores aleatórios
  for (int i=0; i<shapes; i++){
    x[i] = width/2.0;
    w[i] = random(2, 17);
    h[i] = w[i];
    xSpeed[i] = random(-sprayWidth, sprayWidth);
    gravity[i] = .1;
    damping[i] = random(.7, .98);
    friction[i] = random(.65, .95);
  }
}
```

3. Por fim, em ***draw***(), definisse um ciclo for que irá ser executado cada vez um número de vezes correspondente ao valor de *shapeCount*. Como este valor vai sendo gradualmente, significa que inicialmente só 1 quadrado é desenhado, depois 2, 3,.., até 200. Como o ciclo for vai sempre se repetindo dentro de ***draw***(), a cada iteração, os valores de velocidade vão sendo actualizados e consequentemente, as posições *x* e *y* de cada quadrado. Lentamente, cada um vai ficando mais lento, por causa da fricção e amortecimento, até parar:

```
void draw(){
//Desvanecimento da trajetória do quadrado borrifado
  fill(255, 100);
  rect(0, 0, width, height);
  fill(0);
// Controla a quantidade de quadrados borrifados
  for (int i=0; i<shapeCount; i++){
    rect(x[i], y[i], w[i], h[i]);
    x[i]+=xSpeed[i];
    ySpeed[i]+=gravity[i];
    y[i]+=ySpeed[i];
// Detecção das colisões com as bordas
    if (y[i]>=height-h[i]){ // borda inferior
      y[i]=height-h[i];
      // bounce
      ySpeed[i]*=-1.0;
// reduz a velocidade vertical após bater com o fundo
      ySpeed[i]*= damping[i];
// reduz a velocidade horizontal após bater com o fundo
      xSpeed[i]*=friction[i];
    }
    if (x[i]>=width-w[i]){ // borda direita
      x[i]=width-w[i];
      xSpeed[i]*=-1.0;
    }
    if (x[i]<=0){ // borda esquerda
```

```
      x[i]=0;
      xSpeed[i]*=-1.0;
    }
  }
// atualiza o total de quadrados a serem desenhados
  if (shapeCount<shapes){
    shapeCount+=birthRate;
  }
};
```

3.  e veja o que acontece…

Repare que poderia ser qualquer objecto, inclusive 3D…as direcções poderiam ser outras além destas e com isso, o efeito visual final iria drasticamente variando ☺

A aceleração é outro efeito dinâmico interessante que podemos atribuir a um objecto. Fazer com que um ou mais elementos da cena, acelerem ou desacelerem conforme eles estão em determinadas localizações da janela ou na proximidade de outros objectos, permite com que simulemos a existência de aceleração.

O exemplo a seguir ilustra exactamente isso. Vamos criar um círculo, cujo comportamento será o de seguir a posição actual do rato na janela. Entretanto, ele não seguirá com velocidade constante, pois quanto mais perto ele estiver da posição actual do rato, mais devagar ele se move até parar na posição. Para isso, teremos que calcular a posição do círculo de forma incremental variada, ou seja, inicialmente verificamos o quanto distante da posição actual do rato ele está (*deltaX* e *deltaY*) e depois, vamos gradualmente actualizando a posição dele com um determinado incremento variável, isto é, ele vai diminuindo a medida que a posição do círculo se aproxima da do rato. Vejamos o código:

1. Crie um programa de nome **ExercTopico7_7.pde.** Definamos variáveis globais para conter a localização do círculo, além do fator de desaceleração, e depois inicializemos a localização do círculo:

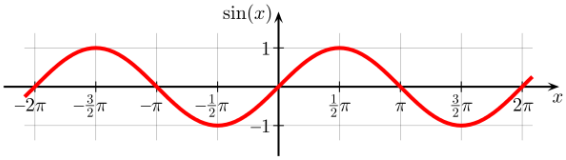
```
float x, y;
float easing = .05;
void setup(){
  size(400, 400);
  x = width/2; // no centro da janela
  y = height/2;
  smooth();
}
```

2. Em ***draw***(), coloca-se o fundo meio transparente, para ver o rastro do círculo e aplica-se a correcção da posição deste, consoante a posição actual do rato:

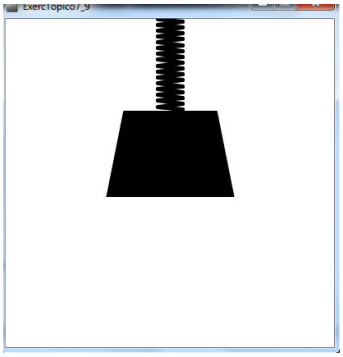
```
void draw(){
  fill(255, 40);
  rect(0, 0, width, height);
// Encontra a distância entre o rato e a posição
// atual do círculo
  float deltaX = (pmouseX-x);
  float deltaY = (pmouseY-y);
// Faz com que se aproxime, mas de forma gradual
  deltaX *= easing;
  deltaY *= easing;
  x += deltaX;
```

```
    y += deltaY;
    ellipse(x, y, 15, 15);
  }
```

Para terminar este tópico, iremos ainda ver um outro efeito animado interessante, que é o de uma mola. Uma mola tem um comportamento que pode ser calculado com o auxílio da função seno ou co-seno, pois ambas geram valores que conseguem traduzir o comportamento oscilante da mola. A figura abaixo ilustra a curva de uma função sinusoidal. O único pormenor, é que conforme o tempo passa, a amplitude do movimento reduz-se, ou seja, a onda vai ficando cada vez menos intensa, até chegar a um ponto que ela torna-se nula. A rapidez com que isso acontece depende do amortecimento, ou seja, o quão rápido a mola perde energia e pára. O amortecimento depende do peso que faz com que a mola oscile ou ainda do tipo de material do qual a compõe.



Tendo em conta estes pontos, vamos ver o código que simula o comportamento de uma mola, tendo um peso pendurado nela. Iremos desenhar 32 segmentos de mola, cada um com 15 *pixels* de largura. O ângulo (*angle*) de abertura entre os segmentos irá iniciar com valor zero, e gradualmente será incrementado com o valor de *frequency*. Isto significa que no início a mola terá a sua distensão máxima, pois o co-seno de zero é 1, que multiplicado pela *amplitude* de 26, resulta no valor mais elevado para *y*. Nas próximas iterações, o co-seno seguirá diminuindo e aumentando várias vezes, entretanto a amplitude irá sempre decrescer (é seguidamente multiplicada por 0.987, ou seja, *damping*). Com isso, consegue-se o efeito de amortecimento do movimento, pois a coordenada *y* irá também oscilar e gradualmente reduzir. Por fim, outro detalhe interessante é o desenho da própria mola. Ela é um *zig-zag* e portanto, dentro do ciclo em que ela é desenhada (função *createSpring*) é necessário avaliar se o próximo segmento de mola a ser desenhado deve iniciar da direita para esquerda ou o oposto. Com o teste do resto da divisão de *i* por 2 (*i%2*) consegue-se criar essa alternância facilmente, ou seja, se o resultado for zero (é par), opta-se por desenhar da direita para esquerda, caso contrário, o oposto. Tratamento diferenciado é dado ao último segmento da mola, pois este só vai até a metade. Note que o cálculo continua, embora a mola atinja a inércia a dado momento. É também possível fazer novamente a mola mexer com o clicar do rato (reinicia o processo, carregando os valores de arranque nas variáveis de controlo do cálculo). Vamos ao programa:



1. Crie um programa de nomes **ExercTopico7_8.pde.** Definamos variáveis globais e inicializemos tudo em ***setup***(). Vamos invocar uma função de nome ***setSpring***() que irá configurar o resto da variáveis:

```
float x, y;
int w = 150, h = 100;
float angle, frequency = 5.0;
float amplitude, damping = .987;
int springSegments = 32, springWidth = 15;
void setup(){
  size(400, 400);
  x = width/2.0-w/2.0;
  smooth();
  strokeWeight(5);
  fill(0);
  setSpring();
}
void setSpring(){
  y = 100;
  angle = 0;
  amplitude = 26.0;
}
```

2. Vamos agora definir as funções que desenham os segmentos de mola e calculam o seu movimento, traduzindo-se na coordenada *y* que será utilizada para definir os segmentos de reta:

```
void startSpring(){
  // Calcula o comportamento da mola
  y += cos(radians(angle))*amplitude;
  amplitude*=damping;
  angle+=frequency;
  if (mousePressed){ // quando premido, reinicia
    setSpring();
  }
}

void createSpring(){
  // Desenha o peso
  quad(x+20, y, x+w-20, y, x+w, y+h, x, y+h);
  // Desenha a mola
  for (int i=0; i<springSegments; i++){
    // último segmento
    if (i==springSegments-1){
      line(x+w/2+springWidth,(y/springSegments)*i,x+w/2,
(y/springSegments)*(i+1));
    }
    else {
// demais segmentos, desenhados com inclinação e orientação alternadas
      if (i%2==0){ // verifica se é par
        line(x+w/2-springWidth,(y/springSegments)*i,
x+w/2+springWidth, (y/springSegments)*(i+1));
      }
      else { // se for ímpar
        line(x+w/2+springWidth,(y/springSegments)*i,x+w/2-
springWidth, (y/springSegments)*(i+1));
      }
    }
  }
}
```

3. Para terminar, inserimos *draw*() com as chamadas das funções necessárias:

```
void draw(){
  background(255);
  createSpring();
  startSpring();
}
```

4.  e veja o que acontece…tende alterar a frequência ou ainda o ângulo de início e veja o resultado!

Para concluir este tópico, gostaria de chamar a atenção de que existem muitas outras fórmulas da física mecânica que podem perfeitamente serem utilizadas para criarmos efeitos visuais interessantes. A deformação de corpos ou ainda a colisão entre objectos, são outros pontos de interesse que deve procurar explorar. Além disso, não esqueça que tudo pode ter uma versão em 3D ou ainda ser reescrito numa orientação orientada a objectos, tornando o seu programa muito mais versátil e abrangente.