

# Processing São Paulo

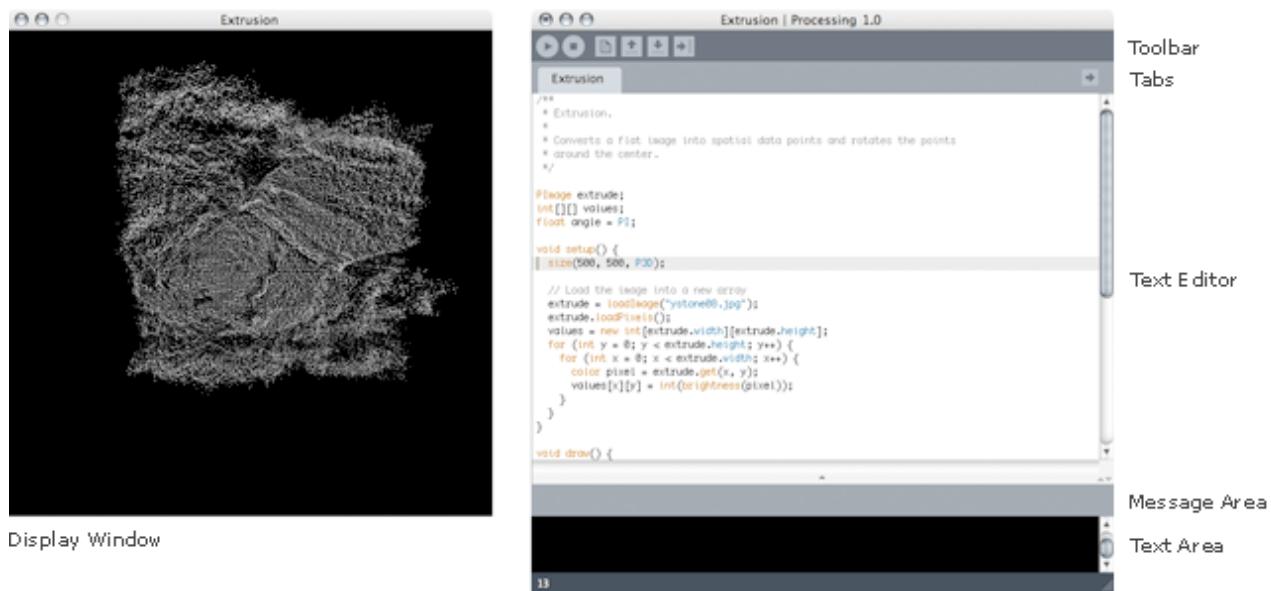
**Curso Básico : Processing**

**Módulos I - II**

# Módulo I

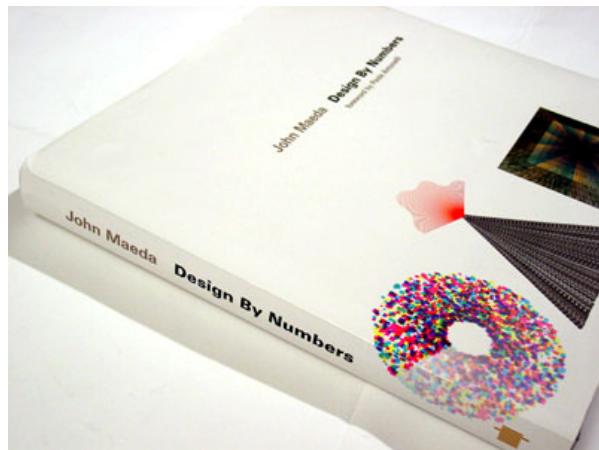
Neste primeiro módulo abordaremos a interface do Processing, figuras geométricas primitivas, atributos de desenho e uso de variáveis associadas a algumas operações aritméticas.

# 1 - Sobre o Processing



Ambiente de programação do Processing

Iniciado por [Casey Reas](#) e [Ben Fry](#) em 2001 , o projeto [Processing](#) é um ambiente de desenvolvimento baseado na linguagem de programação [Java](#) e é destinado ao ensino dos fundamentos da computação dentro de um contexto visual. Segundo Casey REAS, a idéia do projeto nasceu durante o curso ministrado pelo designer [John Maeda](#) (1997) para o [Aesthetics and Computation Group](#) (MIT). De fato, o Processing foi uma extensão do projeto [Design By Numbers](#) (DBN), software criado por Maeda para a formação de artistas interessados na disciplina emergente do “design computacional”. No período pré Web, Maeda foi o responsável pela difusão das técnicas computacionais orientadas para a geração de gráficos interativos.



Publicação "Design By Numbers" - John Maeda

Os programas feitos no Processing são chamados de ***sketches*** e podem ser armazenados numa pasta chamada *sketchbook*. O conceito de sketching é parecido com o do scripting, a diferença é que não trabalhamos numa linguagem interpretada como *JavaScript*, *ActionScript*, *Python* ou *Ruby*.

Um programa escrito numa linguagem compilada deve ser convertido num formato diferente antes de ser executado. O programa vai através de um processo que o modifica de uma linguagem legível aos humanos para uma linguagem de máquina. Esse processo é chamado de compilação.

Já um programa escrito numa linguagem interpretada é interpretado por outro programa enquanto ele é executado. Um interpretador é um programa que analisa cada linha do programa enquanto ele roda, determinando a sua continuidade. Muitas linguagens interpretadas são categorizadas como linguagem de *scripting*.

A linguagem Java possui aspectos da linguagem compilada e interpretada. Antes de um programa Java ser executado, ele é compilado em códigos binários que é executado no *Java Virtual Machine* (JVM). O [JVM](#) é software processador que age como uma memória intermediária entre o código binário e o processador físico do sistema.

Java foi escolhida como base para o Processing, pois representa um bom balanço entre performance e simplicidade de uso. O **IDE** (*Integrated Development Environment*) do Processing conta com um editor (texto) para programação e mais alguns recursos como um seletor de cores e inspetor de pastas. Além destes recursos, também existe a possibilidade de integrar as funções do Processing aos outros ambientes de programação Java como é o caso do [Eclipse](#).

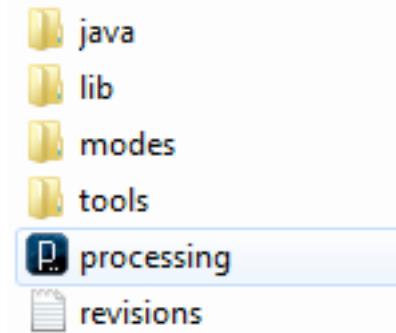
Atualmente o Processing conta com uma enorme comunidade de usuários criativos, pois além de ser gratuito, [open source](#) e rodar na maioria dos sistemas, oferece a possibilidade da geração de gráficos 2D, 3D, manipulação de mídias audiovisuais e integração com outros ambientes ou interfaces como o [Arduino](#) ou [Kinect](#).

Após uma década de desenvolvimento, o projeto está na versão 1.5 e conta com recursos de publicação para dispositivos móveis (Android), aceleração gráfica ([OpenGL](#)) e uma centena de [bibliotecas](#) de software ou extensões para aplicações específicas como a execução de vídeos e captura de imagens em tempo real.

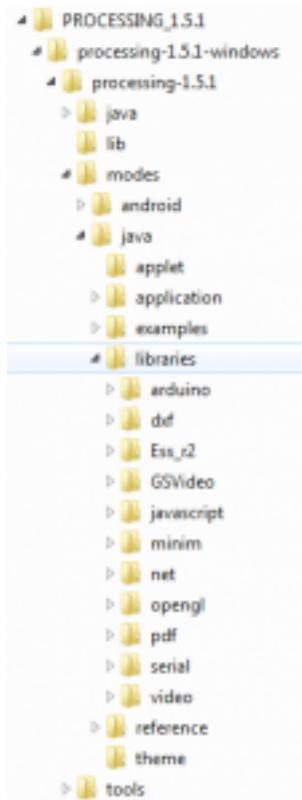
## 2 - Instalação e Configuração

O software pode ser baixado gratuitamente na área de [downloads](#) do website *Processing.org*. Existem versões para Linux, Mac OSX e Windows (incluindo JDK) – no caso do Windows, recomendamos que os iniciantes não escolham a versão *Without Java*. O Processing não necessita de uma instalação específica, bastando descompactar os arquivos na pasta de *Programas*, *Aplicações* ou qualquer outra pasta de seu computador.

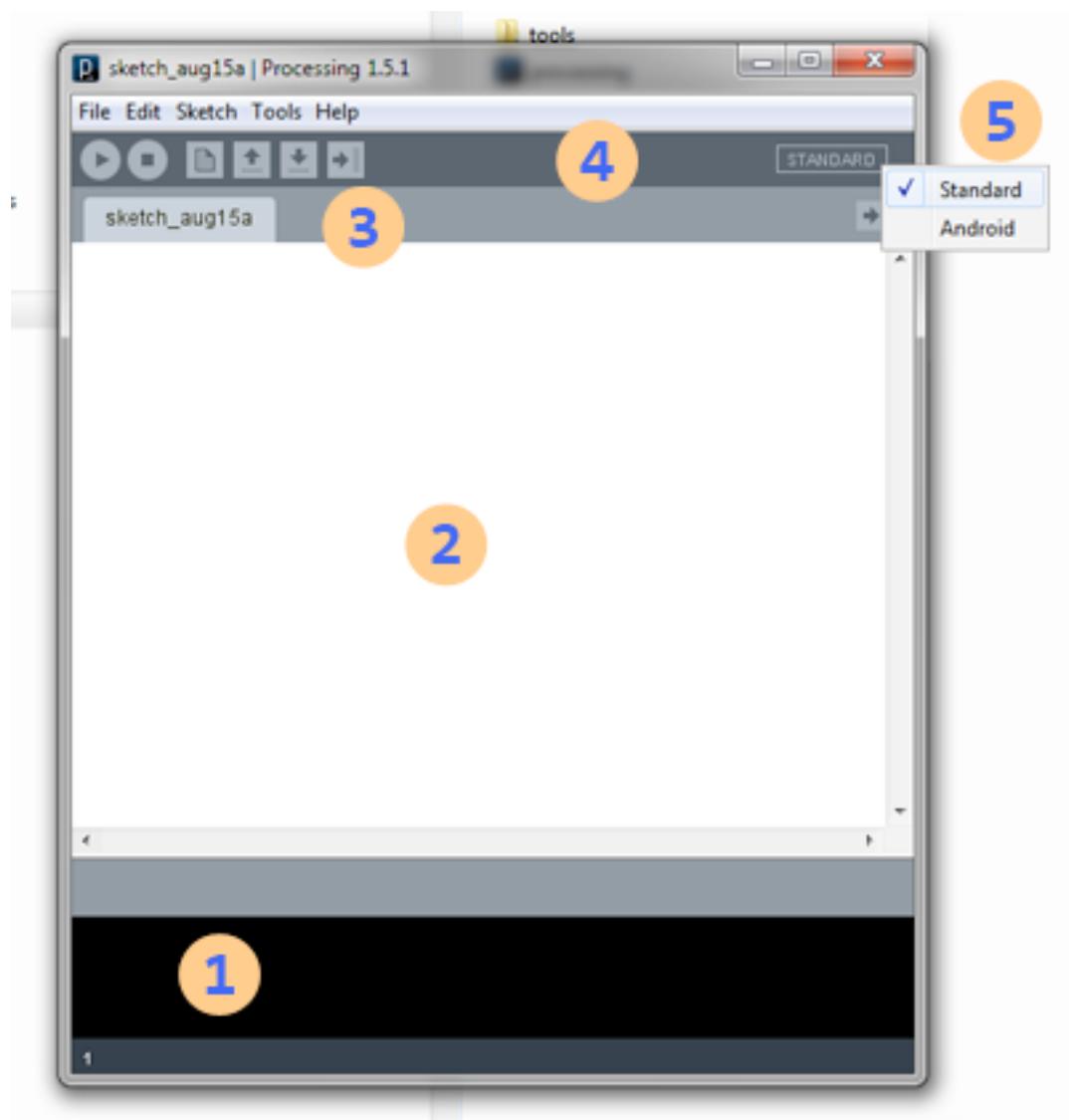
Ao descompactar a pasta encontramos a estrutura:



O Processing pode ser aberto diretamente bastando um click no arquivo executável. As outras pastas do pacote incluem bibliotecas e arquivos importantes para a compilação de seus futuros programas. Em outro tópico veremos mais sobre a pasta *libraries* que é o local onde podem ser incluídas bibliotecas e extensões desenvolvidas pela comunidade. O caminho a partir da raiz é: *modes>java>libraries* (versão 1.5.1)



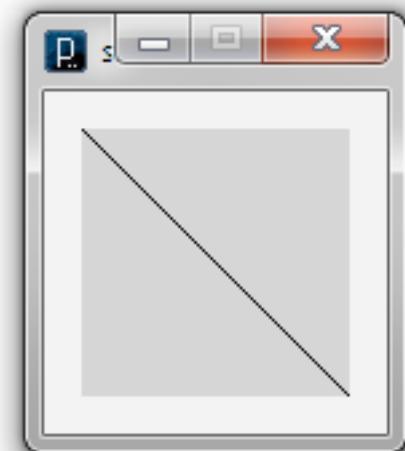
A interface do Processing é composta por duas janelas – a janela principal, contendo um editor onde podemos criar nossa programação e a janela de visualização destinada ao render de gráficos, textos, imagens e vídeos.



Interface do Processing (1.5.1)

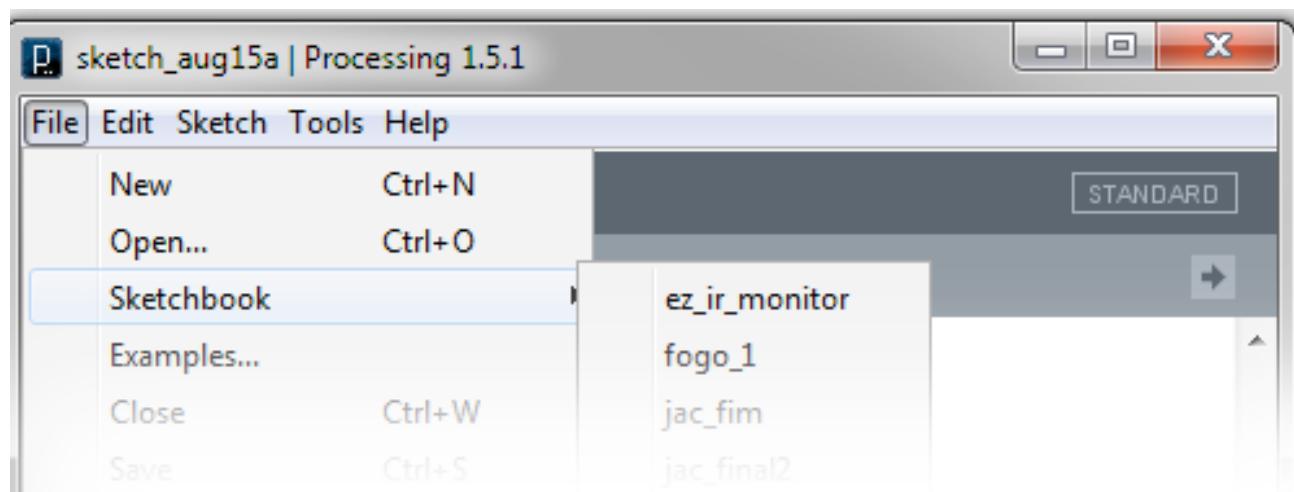
- 1 -Console (área de testes e mensagens de erros)
- 2- Editor (programação por texto)
- 3- Barra de objetos (cada etiqueta representa um novo objeto)
- 4- Barra de açãoe contendo botões: Run, Stop, New, Open, Save, Export
- 5 – Seleciona “modo” de programação – o programa pode ser exportado para Android

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



Janela visualização

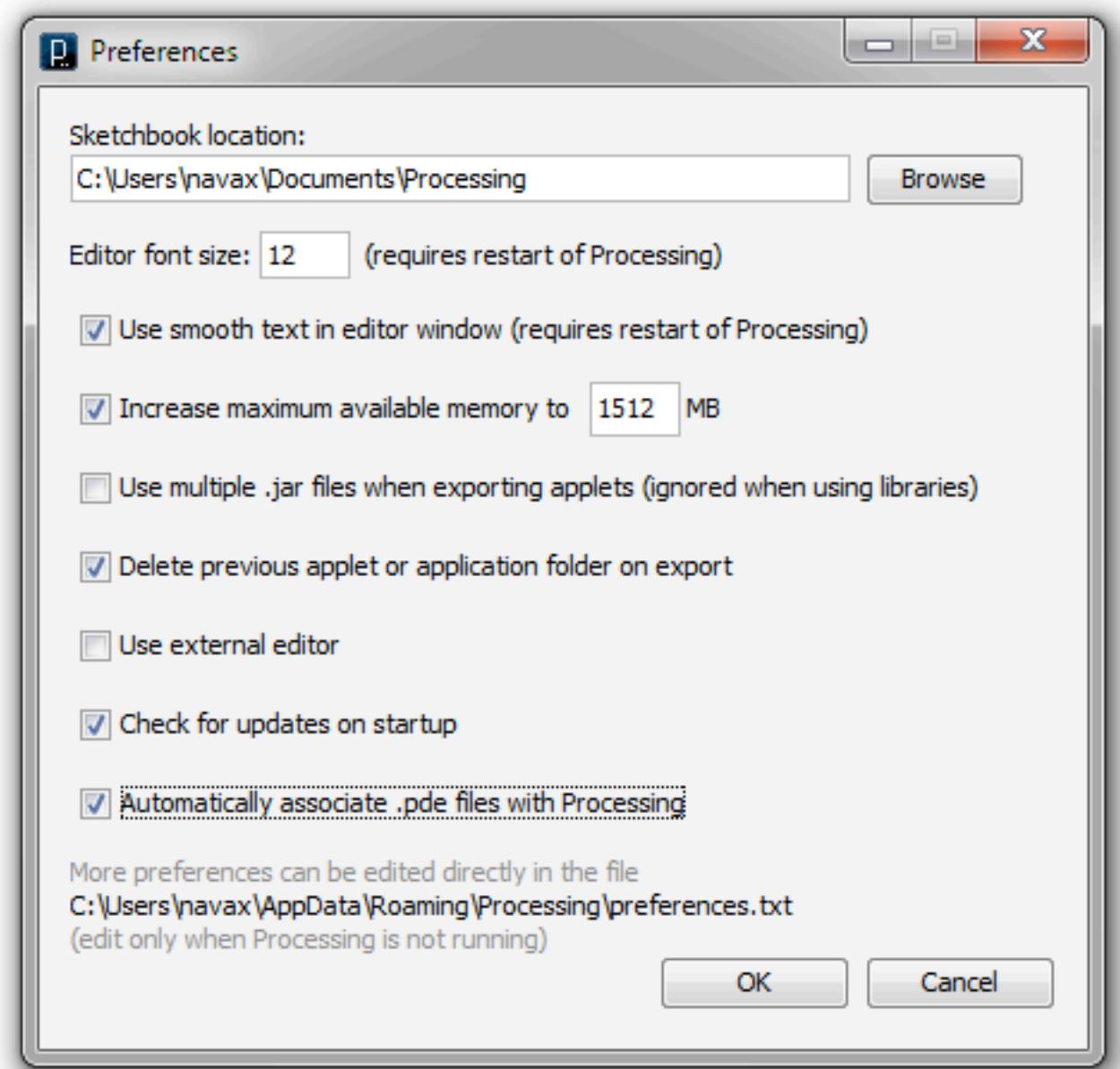
Os programas criados no ambiente do Processing são salvos numa área padrão chamada “**Sketchbook**” (pasta *documentos/processing*).



*Os arquivos 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.

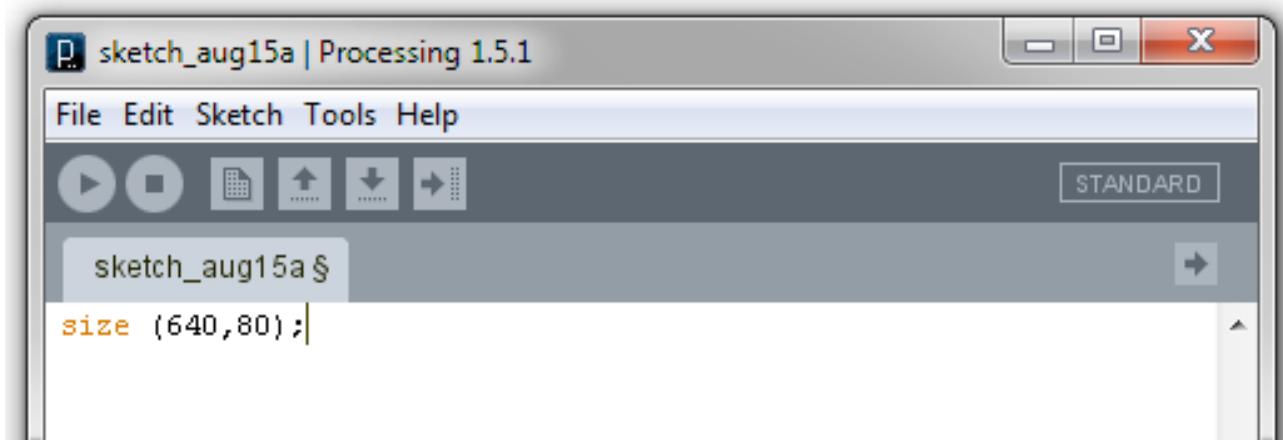
Como algumas aplicações exigem áreas de processo e armazenamento acima de 1Gb, devemos alterar a disposição deste espaço no campo *Increase maximum available memory* – também em *File>Preferences*.



Em preferências podemos alterar configurações sobre o editor, arquivos e gerenciamento de memória.

Os arquivos fontes do Processing possuem a extensão *.pde* e podem ser alterados em qualquer editor de texto. Já os arquivos 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) dos arquivos exportados 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

# 3 - Elementos da Linguagem

Neste tópico veremos alguns elementos básicos da sintaxe ou as regras que organizam a escritura do código de programação. Como na maioria das linguagens , a ordem de execução é computada sequencialmente, linha-a-linha.

Existe a possibilidade do programador inserir comentários textuais que são ignorados pelo compilador. Através dos comentários podemos apontar notas e lembretes sobre a própria estrutura do código.

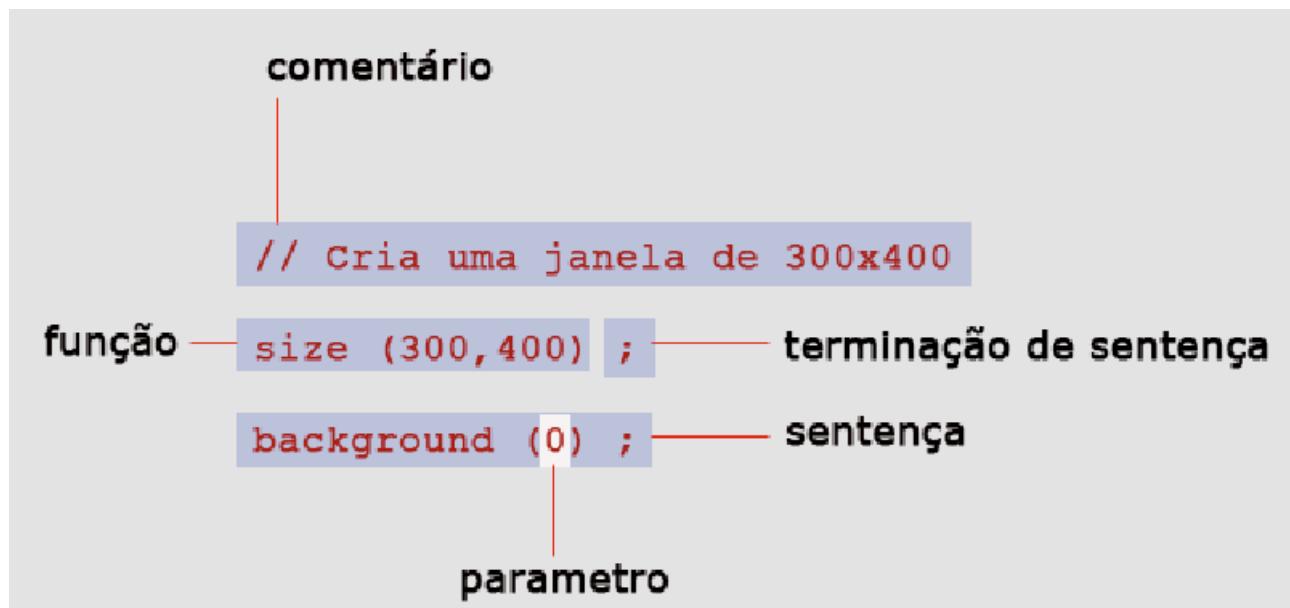
**Comentários (//)**// Duas barras são utilizadas para denotar um comentário  
// não deve existir espaço entre as barras

```
/*
```

Essa forma é usada para a escrita de comentários mais longos  
\*/

## Funções

As funções permitem o desenho de formas, seleção de cores, cálculos e a execução de outros tipos de ação. O nome de uma função geralmente começa com letra minúscula e é seguido por parênteses. Os **parâmetros** são os elementos separados por vírgula que ficam dentro dos parênteses. Algumas funções não possuem parâmetros enquanto outras podem conter muitos.



Na maioria dos scripts as sentenças são finalizadas pelo sinal ponto e vírgula. A função `size` contem 2 parametros – o primeiro é a largura da janela de display e o segundo define a altura

```
size (300,400);
```

Essa outra versão da função `background` contem apenas 1 parametro. Ele determina a quantidade de cinza do fundo que pode variar de 0 (preto) a 255 (branco)

```
background(0);
```

Além destas funções pré definidas pelo ambiente do Processing , existe a possibilidade de criarmos funções para fins específicos. É importante lembrar que o Processing diferencia caracteres maiúsculos e minúsculos (*case sensitive*). Por exemplo , escrever “Size” pode produzir um erro. Na digitação, a quantidade de espaços entre os elementos não prejudica a execução correta do código.

## Mensagens

As funções `print()` e `println()` podem ser usadas para imprimir dados durante a execução de um programa. As funções não enviam páginas para uma impressora, mas escrevem texto para o console do Processing.

O console também pode ser usado para mostrar o estado ou valor de uma variável, confirmar um evento ou verificar os dados enviados por um dispositivo externo.

```
println("10");
```

```
print ("Processing é bacana.");
```

Tanto `print` quanto `println` enviam textos para o console. A diferença é que `println` ainda acrescenta um salto de linha (*print new line*) para a impressão de uma nova mensagem. O resultado visualizado no console seria -

Além do envio de textos (caracteres entre aspas), podemos visualizar o resultado de **expressões** matemáticas.

```
println( (3+2)*-10+1 );
```

```
println (6>3);
```

```
print (15.0 +1.0);
```

Nesta expressões utilizamos **operadores** aritméticos para efetuar cálculos. Entre eles

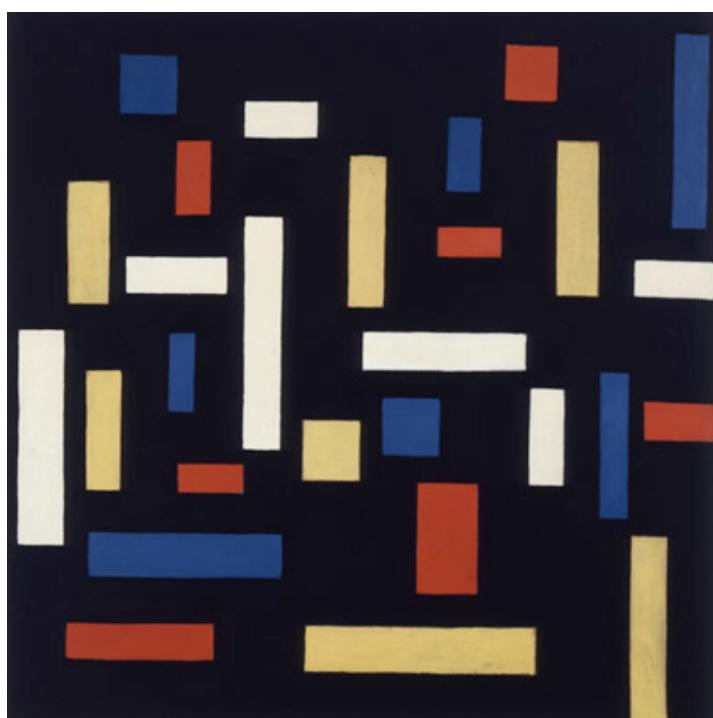
- + adição
- subtração
- \* multiplicação
- / divisão
- % módulo (resto de uma divisão)

## 4 - Elementos Geométricos

Na primeira metade do século XX, o uso das formas geométricas foi uma constante na obra dos artistas e designers formadores das vanguardas modernas. O rompimento com o espaço referencial figurativo tridimensional deu-se principalmente pela abstração geométrica articulada pela composição de elementos como o ponto, linha, plano e pelo arranjo distributivo das cores primárias.



Esquema de Theo Van Doesburg - 1917



Theo Van Doesburg - Composição VII (as três graças)



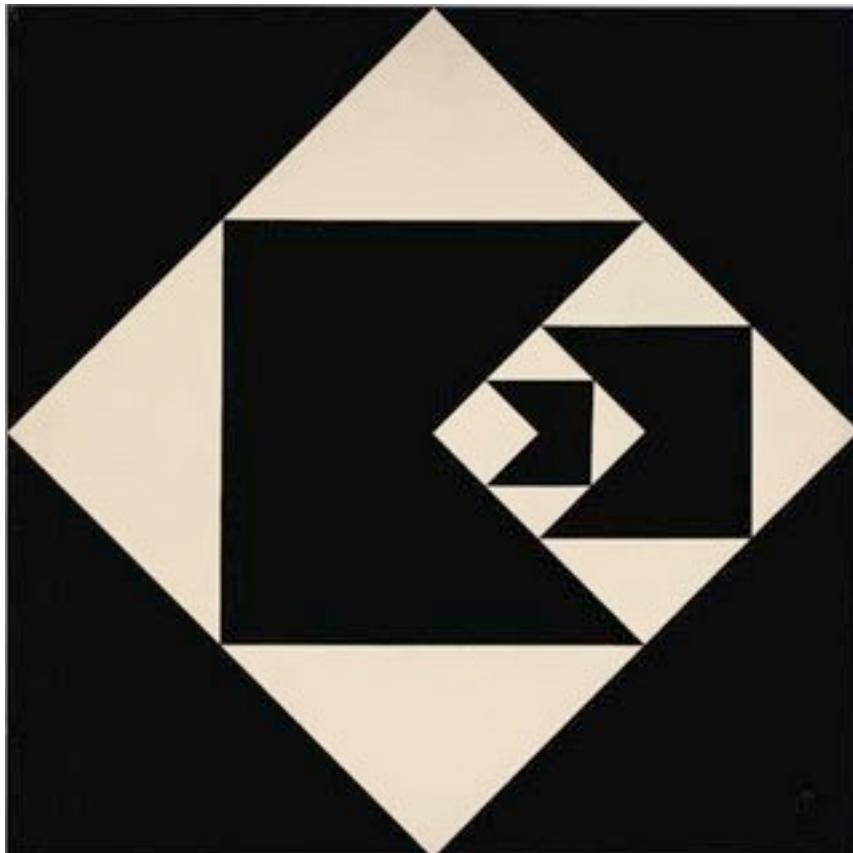
Piet Mondrian -Tableau 2 , 1922. Equilíbrio, unidade e redução estética marcaram o neoplasticismo como a corrente representante de uma nova "consciência racional"



O suprematismo elege a geometria como a essencia da expressividade interior - a figura carrega significados simbólicos e culturais.Kazimir Malevich - Suprematismo 3



Alexander Rodtchenko - Cartaz - Lendiz, 1925. Geometria aplicada na comunicação para as massas - construtivismo russo.



Geraldo de Barros - Função Diagonal. No Brasil a geometria foi explorada pelo artista, fotógrafo e designer Geraldo de Barros e outros integrantes do Grupo Ruptura.

## 4.1 Coordenadas e Primitivas

A tela do computador é composta de uma grade de milhões de **pixels**. Uma posição nesta grade corresponde a uma coordenada (**x**-horizontal e **y**-vertical). No Processing a origem destas coordenadas (0,0) está no **canto superior esquerdo da tela**. Logo, os valores das coordenadas aumentam de cima para baixo e da esquerda para direita.

Como foi visto, as dimensões da área de desenho do *sketch* são determinadas pela função **size()**.

### **size** (largura,altura)

```
// determinando uma área de 300×200 pixels
```

```
size (200,300);
```

Podemos desenhar algumas figuras geométricas primitivas como pontos, linhas e planos utilizando essas coordenadas.

### **point** (x,y)

```
// um ponto corresponde a um pixel desenhado na coordenada x,y
```

```
point (10,30);
```

```
// pontos com x,y identicos formam uma diagonal
```

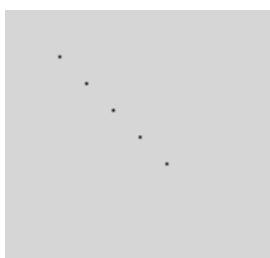
```
point (20,20);
```

```
point (30,30);
```

```
point (40,40);
```

```
point (50,50);
```

```
point (60,60);
```



```
// pontos negativos não são visíveis
```

```
point (-10,20);
```

Além de pontos podemos desenhar: linhas, triângulos, quadriláteros, retângulos, elipses e curvas Bezier.

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

```
// x1,y1 são coordenadas para o ponto inicial da linha e x2,y2 o para o ponto final
```

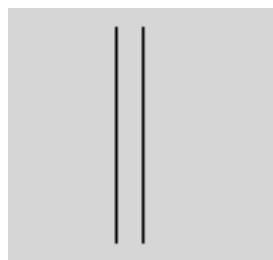
Neste código, traçamos uma linha que se inicia no ponto (10,30) e termina no ponto (90,30):

```
line (10,30,90,30)
```

```
// linhas verticais possuem x1 e x2 idênticos
```

```
line (40,10,40,90);
```

```
line (50,10,50,90);
```



```
// linhas horizontais possuem y1 e y2 idênticos
```

```
line (10,30,90,30);
```

```
line (10,40,90,40);
```



## { Prática }

1 - Desenhar duas linhas que partem do mesmo ponto.

2 - Desenhar duas linhas perpendiculares.

3 - Desenhar um grid 3×3.

**triangle** ( $x_1, y_1, x_2, y_2, x_3, y_3$ )

// os 3 pares de coordenadas definem os vértices do triângulo

*triangle (60,10,25,60,75,65);*

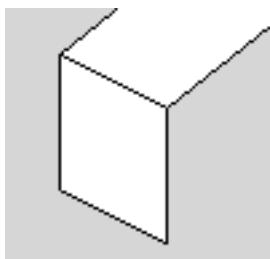


**quad** ( $x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4$ )

// os 4 pares de coordenadas definem os vértices do polígono

*quad (20,20,20,70,60,90,60,40);*

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



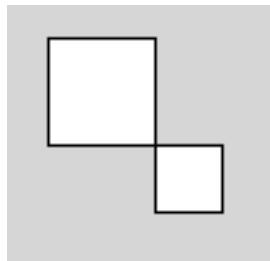
**rect** (*x, y, largura, altura*)

// os 2 primeiros parâmetros definem as coordenadas do vértice superior esquerdo

// o terceiro e quarto parâmetro definem a largura e altura do retângulo

```
rect (15,15,40,40);
```

```
rect (55,55,25,25);
```



**ellipse** (*x, y, largura, altura*)

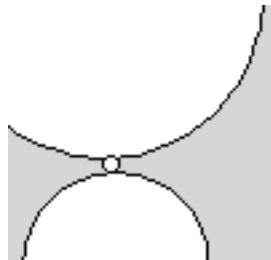
// os 2 primeiros parâmetros definem a localização do centro da elipse

// o terceiro e quarto parâmetro define a largura e altura da elipse

```
ellipse(35,0,120,120);
```

```
ellipse (38,62,6,6);
```

```
ellipse (40,100,70,70);
```



**bezier** (x1, y1, cx1, cy1, cx2, cy2, x2, y2)

// a função contem 8 parâmetros que determinam 4 pontos

// a curva é desenhada entre o primeiro e quarto ponto

// os pontos de controle são determinados pelo segundo e terceiro parâmetro

// podemos visualizar estes pontos

// basta desenhar círculos nas coordenadas dos pontos

// e linhas que ligam estes pontos

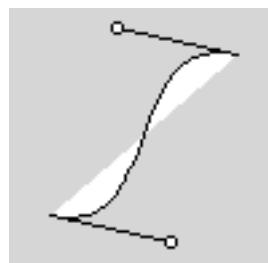
*bezier (85,20,40,10,60,90,15,80);*

*line (85,20,40,10);*

*ellipse (40,10,4,4);*

*line (60,90,15,80);*

*ellipse (60,90,4,4);*



*Curva bezier + linhas+elipses*

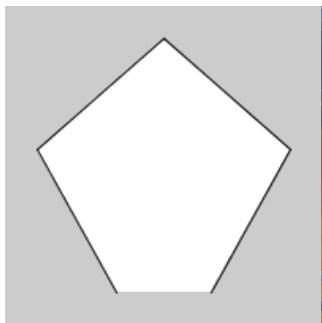
## 4.2 Vértices

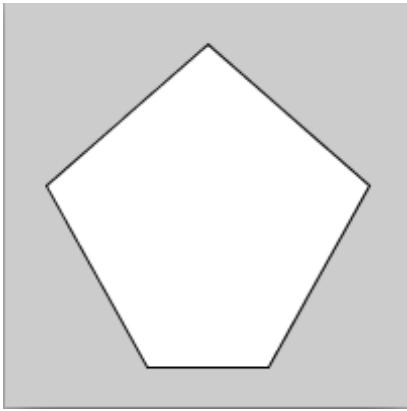
Para o desenho de formas mais complexas podemos usar uma série de coordenadas chamadas vértices. Um vértice é uma posição definida por uma coordenada (x,y).

Para criar uma forma primeiro usamos a função `beginShape()`, então especificamos uma série de pontos com a função `vertex()` e então fechamos a forma com `endShape()`. As formas criadas com `vertex()` são preenchidas com branco e traçadas por contornos em preto. Esse padrão pode ser alterado conforme veremos no tópico de atributos de desenho.

### **vertex** (x,y)

```
//traçando uma forma aberta
size (200,200);
smooth();
beginShape();
vertex (70,180);
vertex (20,90);
vertex(100,20);
vertex (180,90);
vertex( 130,180);
endShape();
```





```
//traçando a mesma forma fechada  
// utilizamos CLOSE como parametro para a função endShape()
```

```
size (200,200);  
smooth();  
beginShape();  
vertex (70,180);  
vertex (20,90);  
vertex(100,20);  
vertex (180,90);  
vertex( 130,180);  
endShape(CLOSE);
```

Veremos em outro tópico que ainda existe a possibilidade de desenharmos curvas utilizando vértices com as funções **curveVertex()** e **bezierVertex()**.

## { Prática }

4 – Criar um sketch com 500×500 pixels e construir uma composição contendo formas primitivas como ponto, linha, quadilátero, quadrado ,triângulo e algum polígono usando vértices.

## 5 - Ordem de desenho

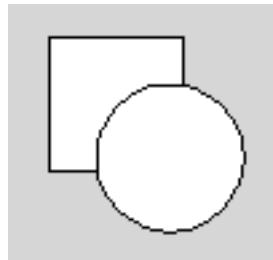
// se um retângulo é desenhado na primeira linha

// ele é mostrado antes de uma elipse que é desenhada na segunda linha de código

// revertendo a ordem colocamos o retângulo por cima

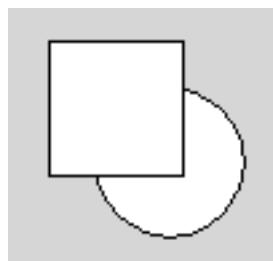
```
rect( 15,15,50,50);
```

```
ellipse (60,60,55,55);
```



```
ellipse (60,60,55,55);
```

```
rect (15,15,50,50);
```



# 6 - Atributos de desenho

## 6.1 Preenchimento e contorno

A função `fill()` altera o valor de preenchimento das figuras e `stroke()` altera o valor de contorno. Se nenhum valor de preenchimento é definido, o valor 255 (branco) é usado. No caso do contorno 0 preto é o valor default.

**fill** (cinza) // valores podem variar de 0 (preto) a 255 (branco)

**fill** (cinza, transparência) // os valores de transparência variam de 0 a 255

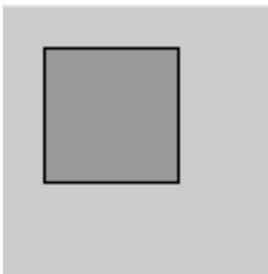
**noFill()** (sem preenchimento)

**stroke** (cinza)

**noStroke()** (sem contorno)

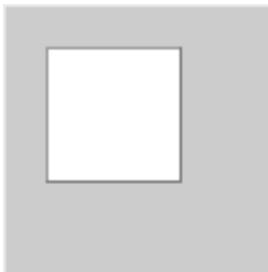
// desenha um retângulo com preenchimento cinza

```
fill(153);  
rect( 15,15,50,50);
```

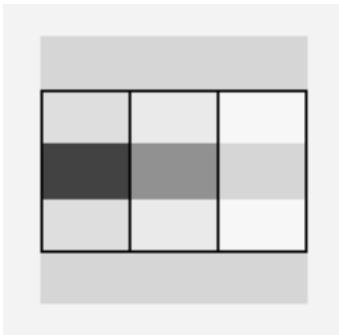


// desenha um retângulo branco com contorno preto (0) e transparência (100)

```
stroke(0,100);  
rect( 15,15,50,50);
```

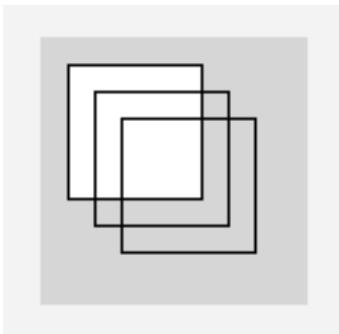


```
// desenha retangulos com niveis gradativos de transparência.  
fill(0);  
rect (0,40,100,20);  
fill (255,51);  
rect (0,20,33,60);  
fill (255,127);  
rect (33,20,33,60);  
fill (255,204);  
rect (66,20,33,60);
```



Obs: as funções *noFill()* e *noStroke()* interrompem o desenho de preenchimento e contorno automático (default).

```
rect (10,10,50,50);  
noFill(); // desabilita o preenchimento  
rect(20,20,50,50);  
rect (30,30,50,50);
```



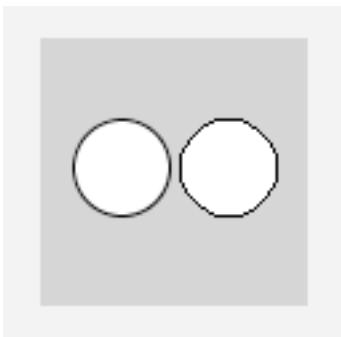
Os atributos de geometria também podem ser modificados.

As funções *smooth()* e *noSmooth()* habilita e desabilita a suavização (*antialiasing*). Uma vez usadas, toda as formas desenhadas posteriormente são afetadas.

## **smooth ()**

## **noSmooth()**

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



Os atributos das linhas podem ser controlados por strokeWeight() , strokeCap() e strokeJoin().

## **strokeWeight** (espessura)

```
smooth ();  
line (20,20,80,20);  
strokeWeight(6);  
line (20,40,80,40);  
strokeWeight(18);  
line (20,70,80,70);
```



## **strokeCap** (modo)

// strokeCap() requer apenas um parâmetro que pode ser os modos : ROUND, SQUARE ou PROJECT

```
smooth();  
strokeWeight(12);  
strokeCap(ROUND);  
line (20,30,80,30);  
strokeCap(SQUARE);  
line (20,50,80,50);  
strokeCap(PROJECT);  
line (20,70,80,70);
```



## **strokeJoin** (modo)

// strokeJoin() requer apenas um parâmetro que determina a forma como os segmentos de linhas se conectam. Estes modos podem ser BEVEL (cantos quadrados), MITER (default) ou ROUND (cantos arredondados).

```
smooth();  
strokeWeight(12);  
strokeJoin(BEVEL);  
rect(12,33,15,33);  
strokeJoin(MITTER);  
rect(42,33,15,33);  
strokeJoin(ROUND);  
rect(72,33,15,33);
```



## 6.2 Cores

Trabalhar com cores na tela do computador é diferente da aplicação de pigmentos sobre papel ou tela. Por exemplo, se adicionamos todas as cores juntas na tela do computador temos o branco como resultado. Na tela do computador misturamos as cores com adição de luz. No computador, a maneira mais comum de determinar uma cor é através do código RGB, pois ele determina a quantidade de vermelho, verde e azul. A intensidade de cada elemento de cor é usualmente especificada com valores entre 0 e 255.

**background** (r,g,b) (red(vermelho), green(verde), blue(azul)) - 0 a 255)

**fill** (r,g,b)

**fill** (r,g,b,alpha)

**stroke** (r,g,b)

**stroke** (r,g,b,alpha)

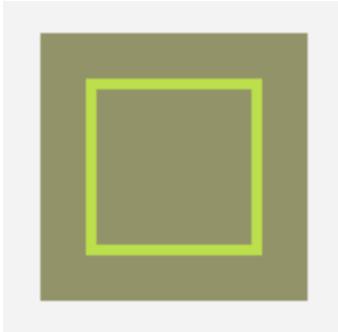
// desenhando um quadrado com preenchimento verde

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



// desenhando um quadrado com contorno verde sem preenchimento

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



No Processing, podemos selecionar o valor RGB de uma cor utilizando a ferramenta *Color Selector* que está no *Menu Tools*. O RGB do vermelho puro, por exemplo, é (255,0,0) ou #FF0000 (notação hexadecimal).

Além do RGB, podemos determinar a transparência (alpha) de uma cor utilizando o quarto parâmetro das funções.

```
// desenhando retangulos com alphas diferentes
```

```
background (116,193,206);
noStroke();
fill(129,130,87,102);
rect (20,20,30,60);
fill (129,130,87,204);
rect (50,20,30,60);
```



Em outro tópico veremos outro espaço de cor utilizando o código HSB (matiz, saturação e brilho).

## { Prática }

5 – Utilizando os atributos de desenho , alturar a espessura de linha, preenchimento e contornos das figuras desenhadas no exercício 4 – utilize o código feito anteriormente.

6 – Alterar as transparências e cores de preenchimento e contorno das figuras resultantes do exercício 5.

# 7 - Variáveis

As memórias físicas dos computadores possuem um sistema de endereçamento para armazenagem e recuperação de dados. Na programação, as **variáveis** correspondem a estes endereços e podem, por exemplo, guardar a posição de uma figura, o código de uma cor ou registrar mudanças de estados. A linguagem do Processing possibilita a configuração de diferentes tipos de dados, incluindo números, letras, palavras, cores, imagens, fontes e valores booleanos (verdadeiro, falso).

Na prática, as variáveis permitem a reutilização de uma informação durante a execução de um programa. A criação de uma variável possui 2 ações: a **declaração** de um **nome** para a identificação da variável e a atribuição de um **valor** para **inicializar** a variável. Adicionalmente, uma variável indica um tipo ou categoria de dado que ela pode armazenar. Algumas regras neste processo:

## 1 – Toda variável deve ser declarada (nome) antes de ser usada.

```
int largura;
```

Neste exemplo declaramos (criamos) uma variável chamada *largura*. Note o “*int*” antes do nome – isso quer dizer que esta variável só pode guardar números inteiros (*integer*). No Processing os tipos das variáveis são escritos antes do nome. A seguir, inicializamos a variável *largura* pela atribuição de um valor:

```
largura=30;
```

O sinal de atribuição (=) é utilizado para associarmos o número 30 à variável *largura*. Todas as vezes que a palavra *largura* for usada no programa, ela será substituída por 30. Portanto, o valor 30 é apenas um número armazenado pela variável. Podemos utilizar esta variável em outro momento do código. Por exemplo, *largura* pode integrar a função de criação de uma elipse:

```
ellipse (100,100,largura,50);
```

Isto quer dizer que criamos uma elipse cujo centro está na coordenada (100,100), possuindo 30pixels de largura (width) e 50 pixels de altura – lembre da sintaxe:

```
ellipse (x,y,width,height);
```

Além disto, não podemos atribuir um valor decimal para largura (ex: *largura=30.5*), pois a variável só pode armazenar tipos inteiros. A declaração de um conjunto de variáveis do mesmo tipo pode ser feita assim:

```
int x,y,z;  
x=-10;  
y=3;  
z=0;
```

Note que na primeira linha criamos as variáveis *x*, *y* e *z* do tipo inteiro (*int*). Posteriormente inicializamos cada uma delas com números inteiros. Veja exemplos de outros tipos permitidos:

### Float

Indicado para armazenar números decimais.

```
float x=10.20;
```

Observe que declaramos uma variável **x** do tipo **float** e , ao mesmo tempo, atribuimos o valor **10.20** (note o ponto decimal). Portanto, podemos declarar e inicializar uma variável numa mesma sentença.

### Boolean

Variáveis booleanas podem assumir apenas 2 valores: *true* (verdadeiro) ou *false* (falso). Esse tipo é geralmente utilizado quando precisamos conferir resultados de operações lógicas.

```
boolean resultado; // declaramos a variável resultado como booleana  
resultado=false; // inicializamos com o valor apropriado: false
```

## Color

Este é um tipo especial de variável encontrado no Processing. Com ele podemos armazenar códigos de cores e atribui-los a nomes arbitrários.

```
size (300,300);
noStroke();
color azulceleste;           // declara a variável “azulceleste”
color cinza;                 // declara a variável “cinza”
azulceleste= color (39,159,216); // inicializa azulceleste com color(39,159,216)
cinza= color(130);           // inicializa cinza com color(130)
fill (azulceleste);          // utiliza azulceleste como cor de preenchimento
rect (0,0,300,200);          // desenha retângulo superior
fill (cinza);                // altera cor de preenchimento para cinza
rect (0,200,300,100);         // desenha retângulo inferior
```

Perceba que usamos as variáveis em dois momentos do código, alternando as cores de preenchimento de azul para cinza. Neste caso, o que importa são os valores atribuídos , pois os nomes são escolhidos arbitrariamente. Mesmo assim, a escolha de um nome apropriado facilita a leitura do código, seria mais fácil entender *fill (cinza)* ou *fill (xyz)* ?

**2 – O nome de uma variável NÃO deve começar com um número ou palavra reservada como *true*, *int* ou *if* , entre outras palavras reservadas para o código.**

```
int 15alo=13; // gera um ERRO;
```

```
int int=10; // gera um ERRO;
```

Na área de programação existem **regras** de sintaxe, modos de escritura de uma determinada linguagem e convenções criadas pela comunidade de programadores. Por exemplo, no *actionScript* (script de programação do Adobe Flash) a declaração de variáveis obedece a uma outra sintaxe:

```
var largura:uint=30; // declaramos a variável largura e inicializamos com o valor 30.
```

Já as convenções de atribuição de nomes podem ser adotadas ou não em qualquer tipo de linguagem. Por exemplo, poderíamos declarar o nome “azulceleste” de uma outra forma

```
color azulCeleste=color (39,159,216);
```

A diferença é que “celeste” foi escrito com letra maiúscula. Isto facilita a leitura mas não altera a função. Porém, devemos tomar o cuidado de utilizar de forma consistente o nome e tipo da variável, caso contrário é erro na certa:

```
int valorMaximo;  
valormaximo=30; // ERRO!
```

Neste caso declaramos “valorMaximo” e tentamos utilizar “valormaximo” que ainda não foi declarada.

```
int valorMinimo;           // declara a variável valorMinimo  
valorMinimo=30;           // inicializa com o valor 30  
int valorMinimo=170;       // ERRO! Tenta declarar (int) valorMinimo novamente
```

Aqui acertamos o nome da variável, mas erramos quando tentamos declarar a variável novamente. O erro acontece , pois o programa não pode criar duas variáveis com o mesmo nome.

## 7.1 Variáveis do Sistema

O Processing possuem algumas variáveis reservadas que armazenam informações como as dimensões da janela de *display* ou posição do cursor (mouse).

Por exemplo as variáveis “width” e “height” sempre guardam as dimensões da janela determinada pela função *size()*. Por exemplo, executando o programa:

```
size (300,100);  
println (width);  
println(height);
```

Visualizamos no console os valores 300 e 100 que são as dimensões estipuladas por *size (300,100)*. Lembrando que a função *println* imprime mensagens no console do Processing.

Experimente esse código:

```
size (400,400);  
line (0,0,width,height);
```

E este:

```
size (350,70);  
line (0,0,width,height);
```

Qual a diferença? Apenas o tamanho da tela. Note que a linha traçada é sempre uma diagonal que começa na coordenada (0,0) e termina no canto inferior direito, coordenada gerada pelas dimensões máximas guardadas por “width” e “height”.

Concluindo, a declaração de uma variável é obrigatória, pois ela reserva o espaço de memória necessária para a execução correta do programa. Em linguagens como o Java, o programador não se preocupa tanto em gerenciar a quantidade de memória que foi alocada, pois esta linguagem possui mecanismos (*garbage collector*) para liberar parcelas de memória que não estão sendo utilizadas. Já em linguagens como C, a alocação e liberação destes espaços são mais críticos e cabe ao programador o gerenciamento da quantidade de memória a ser reservada e desalocada.

Aqui temos uma lista de tipos( intervalo de valores possíveis) de variáveis e a quantidade de memória ocupada:

<b>Tipo</b>	<b>Tamanho</b>	<b>Valores</b>
boolean	1 bit	true ou false
byte	8 bits	-128 a 127
char	16 bits	0 a 65535
int	32 bits	-2,147,483,648 a 2,147,483,647 (*)
float	32 bits	3,40282347E+38 a -3,40282347E+38 (**)
color	32 bits	16,777,216 cores

(\*) o intervalo depende da plataforma (32 ou 64 bits)

(\*\*) ponto flutuante

# 8 - Expressões Aritméticas

No Processing as propriedades de uma imagem podem ser calculadas por expressões aritméticas ou operações matemáticas sobre valores armazenados nas variáveis.

## 8.1 - Operadores

Os principais **operadores** e respectivas operações possíveis:

+ (adição)

- (subtração)

\* (multiplicação)

/ (divisão)

% (módulo)

++ (incremento)

-- (decremento)

+= (adição e atribuição)

-= (subtração e atribuição)

Neste exemplo verificamos como a operação pode ser aplicada em conjunto com uma variável.

```
int cinza=153;  
fill(cinza);  
rect (10,10,55,55);  
cinza+=102;           // equivalente a cinza=cinza+102  
fill(cinza);  
rect(35,20,55,55);
```

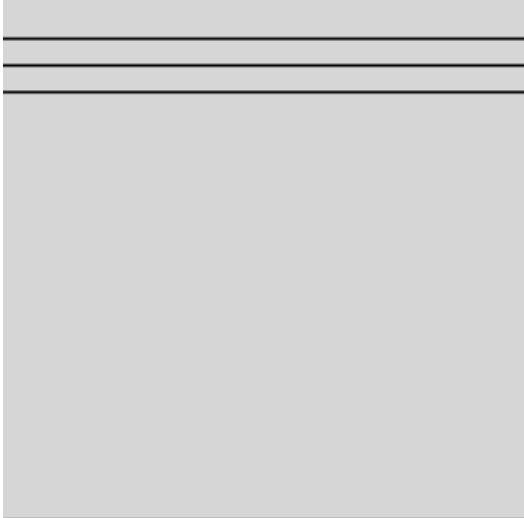
Inicialmente valor 153 é armazenado na variável “cinza” e aplicado para o preenchimento. No próximo passo o valor 102 é adicionado ao valor 153 e atualizando novamente a variável cinza, logo cinza recebe o valor 255.

Uma operação pode ser usada como elemento para uma função de desenho como line(), por exemplo:



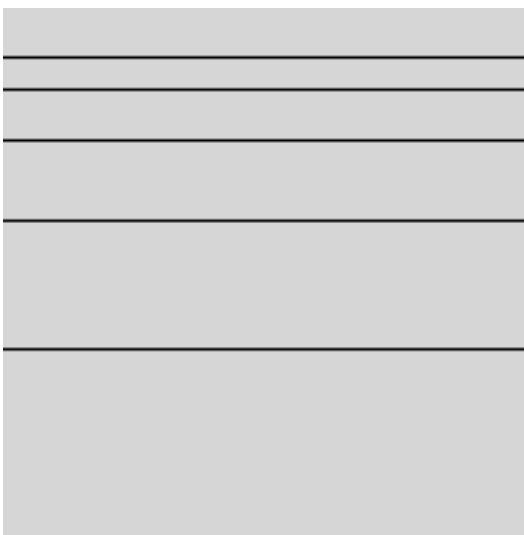
```
int a=8;  
int b=10;  
line (a,0,a, height);  
line (b,0,b, height);  
strokeWeight(4);  
line (a*b,0,a*b,height);
```

Ou neste exemplo, onde incrementamos regularmente o valor de y:



```
int y=20;  
line (0,y,width,y);  
y+=10;  
line (0,y,width,y);  
y+=10;  
line (0,y,width,y);  
y+=10;
```

Utilizando sucessivas multiplicações:



```
int y=20;  
line (0,y,width,y);  
y*=1.6;  
line (0,y,width,y);  
y*=1.6;  
line (0,y,width,y);  
y*=1.6;  
line (0,y,width,y);  
y*=1.6;  
line (0,y,width,y);  
y*=1.6;
```

## 8.2 - Expressões

A ordem das operações determina a prioridade dos cálculos e seus operadores. A multiplicação é executada sempre antes da adição. Para a mudança desta ordem, devemos isolar as operações prioritárias com os sinais de parenteses:

```
x= 3+4*5      // resulta em x= 23
```

```
y= (3+4)*5    // resulta em y =35
```

Os sinais ++ e -- são atalhos que facilitam o incremento e decremento unitário dos valores. Veja o exemplo:

```
int x=20;  
x++;          // incrementa o valor (x+1)  
println( x); // x deve valer 21;  
x--;          // decrementa o valor (x-1)  
println(x);  // x deve valer 20 novamente
```

## 8.3 - Funções Numéricas

A ordem das operações determina a prioridade dos cálculos e seus operadores. A multiplicação é executada sempre antes da adição. Para a mudança desta ordem, devemos isolar as operações prioritárias com os sinais de parenteses:

Como foi mostrado no tópico sobre geometria, podemos utilizar funções especialmente implementadas para a geração de gráficos como rect(), line() e point().

Existem outros tipos de funções que operam cálculos e **retornam resultados estritamente numéricos** Neste tópico introduzimos algumas:

**ceil ()** - calcula o número inteiro mais próximo que é **maior ou igual** ao valor seu parametro

```
int w= ceil (2.1) (w recebe o valor inteiro retornado pela função que é 3)  
int x= ceil (2.9) (x recebe o valor inteiro retornado pela função que é 3)  
int y= ceil (2.0) (y recebe o valor inteiro retornado pela função que é 2)
```

**floor ()** - calcula o número inteiro mais próximo que é **menor ou igual** ao valor seu parametro

```
int w= floor (2.1) (w recebe o valor inteiro retornado pela função que é 2)  
int x= floor (2.9) (x recebe o valor inteiro retornado pela função que é 2)  
int y= floor (2.0) (y recebe o valor inteiro retornado pela função que é 2)
```

**round ()** - calcula o número inteiro **mais próximo** do seu parametro.

```
int w= floor (2.1) (w recebe o valor inteiro retornado pela função que é 2)  
int x= floor (2.5) (x recebe o valor inteiro retornado pela função que é 3)  
int y= floor (2.0) (y recebe o valor inteiro retornado pela função que é 2)
```

Apesar do resultado ser um número inteiro, podemos atribui-lo a um valor float:

```
float w= round(2.1) ( w recebe 2.0)
```

A função **min()** determina o menor valor numa sequência de parametros e **max()** o maior valor. Essas funções podem receber 2 ou 3 parametros:

```
int a=min (10 , 2) (a=2)  
int b= min (-3,-34,-90) (b=-90)  
float c=max (23.1, 0.5, 3.4) (c=23.1)
```

# **Módulo II**

No segundo módulos estudaremos os princípios da animação 2d. As animações podem ser operadas por recursos matemáticos, transformações geométricas e função randômica.

## 9 -Animação

A animação consiste na disposição temporal de uma sequência de imagens no intuito de causar a ilusão do movimento. Além do processo tradicional na qual os “quadros” são desenhados e coloridos manualmente, existem ainda o *stop motion* (registro de objetos físicos) e computação gráfica entre outros.

Alguns softwares foram elaborados especificamente para a produção e publicação final de animações destinadas aos suportes e meios audiovisuais como o vídeo digital, cinema e TV. O [Toon Boom](#), por exemplo, oferece ferramentas de desenho vetorial, *timeline* e suporte para a sincronização de áudio e criação de personagens no estilo *cartoon*.

O Processing é um ambiente de programação de uso geral cujos [recursos](#) incentivam a produção de gráficos, tipografia, interatividade e mesmo animações 2D e 3D.

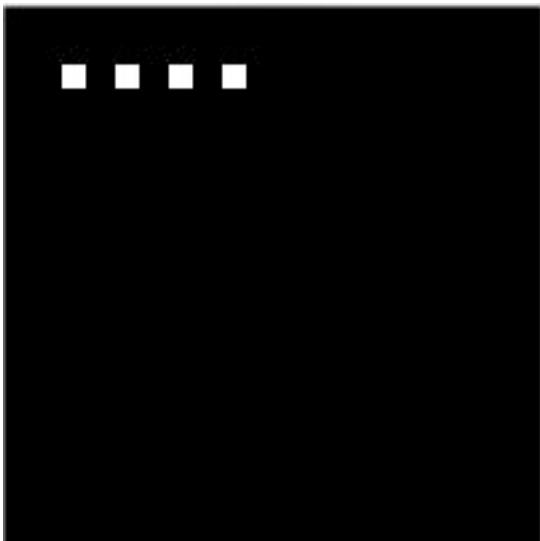
Em seu ambiente de desenvolvimento não existe uma interface gráfica semelhante à *timeline* do Toon Boom ou Adobe Flash. Apesar desta “limitação”, veremos que existem funções especiais como o *setup()* e *draw()* que possibilitam o controle do fluxo de execução de um programa, gerando repetições necessárias para a produção de animações ou movimentos diversos.

### As funções *setup()* e *draw()*

Como vimos em tutoriais anteriores, o posicionamento espacial de elementos geométricos, imagens ou textos foi feito por linhas de programação consecutivas. Por exemplo, se desejamos desenhar um quadrado em quatro posições diferentes, devemos repetir a função *rect()* , variando os parametros x ou y em cada posição desejada:

```
size (200,200);
background(0);
rect (20,20,10,10);
rect (40,20,10,10);
rect (60,20,10,10);
rect (80,20,10,10);
```

E este seria o resultado visual deste *script*:



Ao executarmos o código, notamos que não houve uma animação. Apesar de variarmos a posição horizontal (20,40,60 e 80) em cada função, conseguimos apenas o desenho do objeto em posições diferentes e ao mesmo tempo! Para conseguirmos a ilusão de movimento precisamos desenhar cada quadrado em instantes diferentes ou em *frames* diferentes.

Neste caso, a função *draw()* torna-se útil, pois ela funciona como uma espécie de *looping* dentro do qual uma ou mais instruções são repetidas indefinidamente a cada segundo.

No Processing, execute o código:

```
int px;  
  
void setup() {  
    size (200,200);  
    background(0);  
    noStroke();  
    px=0;  
}  
  
void draw() {  
    background(0);  
    px++;  
    rect (px,20,10,10);  
}
```

Apesar de ainda não apresentarmos um tópico específico sobre funções, basta saber no momento que a escrita de funções segue algumas regras, por exemplo em

```
void setup() {  
}
```

o termo **void** significa que a função não gera (retorna) um valor específico, como o resultado de uma operação matemática, por exemplo. O termo **setup** é o nome da função que deve ser seguido de ( ), assim como na função *background()*. A diferença aqui é que os sinais { } determinam o **escopo** da função, ou seja, quando a função é executada todo o código inserido neste trecho será também executado em bloco.

A estrutura inicial do código acima pode ser resumida assim:

```
void setup() {  
...  
}  
  
void draw() {  
...  
}
```

Este arranjo está presente na maioria dos sketches programados no Processing e representam duas funções distintas. Primeiro, todo o código aninhado dentro das chaves { } da função **setup()** são executados **somente uma única vez**. Então, as linhas

```
size(200,200);  
background(0);  
noStroke();  
px=0;
```

não precisam entrar no *looping* (repetição), pois são funções iniciais que preparam os atributos gerais do sketch como dimensão, cor de fundo e inicialização de variáveis.

Já as linhas,

```
background(0);  
px++;  
rect (px,20,10,10);
```

acontecem dentro do **escopo** {} da função **draw()**, isto é, as três linhas são executadas em bloco, infinitamente.

Vamos entender melhor como isso ocorre.

A intenção original é gerar o movimento horizontal do quadrado, portanto devemos aumentar os valores da coordenada x (lembre-se que o eixo horizontal corresponde à coordenada x e o vertical à coordenada y). Veja o código:

```
rect (px,20,10,10);
```

Perceba que no parâmetro referente à coordenada x utilizamos a variável px! Para que o valor da variável px aumente, devemos incrementá-la constantemente e isto é feito pela linha:

```
px++;
```

Como vimos no tutorial sobre *expressões aritméticas*, escrever px++ equivale à expressão px=px+1. Como esta expressão também está dentro do *looping* da função **draw()**, então a variável px será incrementada infinitamente – veja que o quadrado segue para fora da área de desenho, pois sua coordenada x recebe o valor armazenado na variável px e que aumenta constantemente.

Finalmente, note que a criação da variável px (declaração) foi escrita fora do escopo das funções **setup()** ou **draw()**. Se, por exemplo, esta variável fosse criada dentro da função **setup()**, provavelmente o sistema acusaria algum erro, pois a função **draw ()** não “recomhiceria” a variável px que, neste caso, é funcional apenas para o bloco de **setup ()**.

Vamos comentar o código para ficar mais legível:

```

int px; //declaramos uma variável px do tipo inteiro (integer)

void setup() { // função setup – só é executada uma vez
    size (200,200); // determina as dimensões do sketch
    background(0); // determina a cor de fundo do sketch (preto)
    noStroke(); // desativa desenho do contorno
    px=0; // inicializa a variável px com um valor (zero) posição inicial do
    quadrado
}

void draw() { // função draw – looping, executada infinitamente
    background(0); // aplicamos novamente a função background(0)
    px++; // incrementamos o valor de px (soma uma unidade a cada
           // repetição)
    rect (px,20,10,10); //desenhamos um retangulo ( coordenada x=valor atual de px,
           // coordenada y valor fixo=20, altura e largura=10)
}

```

A ilusão de animação é criada pelo incremento da variável px que substitui a coordenada x na função `rect(px, 20, 10, 10)`. Mas, existe outro truque aqui! Veja que na função `draw ()` existe ainda a chamada de `background(0)`. Se não usamos este recurso não perceberíamos a animação, pois cada quadrado desenhado ficaria permanentemente fixo em sua posição. Para que o efeito de movimento seja completo, devemos desenhar um quadrado, limpar o desenho com `background (0)` e tornar a desenhar a mesma figura em outra posição. Este processo está fundamentado na própria síntese da imagem em movimento, desde a origem do cinema e a exploração de um fenômeno chamado de [persistência retiniana](#).

No exemplo do código anterior, existe o apagamento total da imagem antes da mesma ser redesenhada pelo *looping*.

No Processing não contamos com uma ferramenta de “apagamento” semelhante à borracha do Photoshop. O rastro da animação pode ser produzido pela obstrução gradual dos desenhos, bastando gerar uma sobreposição contínua de camadas transparentes.

Para testar esse efeito, modifique e execute :

```
int px=0;
```

```
void setup( ) {  
    background(0);  
    size(200,200,P2D);  
    smooth();  
    noStroke();  
    px=0;  
}  
  
void draw( ) {  
    px++;  
    fill(0,5);  
    rect(0,0,width,height);  
    fill(255,0,0);  
    rect(px,100,20,20);  
    fill(0,255,0);  
    rect(100,px,20,20);  
}
```

Neste novo exemplo fizemos algumas modificações. Note que na função draw() não utilizamos mais a função background(0). Para a apagar os quadros, desenhamos um retângulo com a cor de preenchimento igual ao do background, porém a transparência do mesmo é de apenas 5. Veja o código:

```
fill(0,5);                                // preenchimento preto e transparência 5  
rect(0,0,width,height);                // desenha o retângulo com as mesmas dimensões  
  
//sketch
```

Logo após, mudamos a cor para vermelho com `fill(255,0,0)` para desenhar o primeiro retângulo animado e verde com `fill (0,255,0)` para o segundo retângulo que se desloca verticalmente (note que também colocamos a variável px no parâmetro referente à coordenada y em `rect(100,px,10,10)` ).

Igual ao exemplo anterior, background(0) poderia ter sido utilizado para uma animação simples. Como planejamos o efeito do “rastro”, optamos pela de acumulação das transparências dos quadros que se sobrepõem continuamente sobre as figuras coloridas desenhadas também no mesmo *looping*.

Um outro detalhe foi acrescentado na função `size (200,200,P2D)`. Existe um terceiro parâmetro para a função `size()` além das dimensões largura e altura:

```
size (width,height,MODE);
```

O elemento MODE determina o modo como o Processing executa os gráficos, podendo assumir os seguintes valores:

JAVA2D (default)  
P2D  
P3D  
OPENGL

Neste exemplo utilizamos o P2D para ganhos na velocidade de processamento (veja maiores [detalhes](#)).

## Alterando a Velocidade

A velocidade em que os quadros (frames) são redesenados pode ser alterada pela função `frameRate()`. Por exemplo, podemos dispor esta função logo no início do sketch, dentro do escopo de `setup()`:

```
void setup() {  
    size(200,200);  
    background(0);  
    frameRate(12); // desaceleramos a animação para 12 quadros por segundo  
}
```

O valor default para `frameRate()` é de 60 quadros por segundo. O fato de aumentarmos esse valor não significa que a animação seja executada muito mais rápido, pois o fator considera também o poder de processamento da máquina onde o programa está sendo executado. A alteração do valor indica apenas a taxa máxima de quadros que o programa deve tentar desenhar por unidade de tempo (segundos).

De outra forma, podemos produzir a ilusão de velocidade alterando o código e fazendo com que valores maiores ou menores sejam adicionados à variável responsável pelo posicionamento no espaço das coordenadas. Veja a alteração do exemplo anterior:

```
int px;  
int py; // devemos acrescentar a declaração da variável py  
  
void setup() {  
    background(0);  
    size(200,200,P2D);  
    smooth();  
    noStroke();  
    px=0; // determina valor inicial para px  
    py=0; // determina valor inicial para py  
}
```

```

void draw( ) {
    px+=2;           // incrementa px com duas unidades
    py+=4;           // incrementa py com quatro unidades
    fill(0,5);
    rect(0,0,width,height);
    fill(255,0,0);
    rect(px,100,20,20);   // px é utilizada na atualização da posição horizontal
    fill(0,255,0);
    rect(100,py,20,20);   // py é utilizada na atualização da posição vertical
}

```

Neste exemplo criamos duas variáveis (px e py) para o controle das velocidades individuais dos quadrados. Veja o código:

```

px+=2; // equivale a px=px+2
py+=4; // equivale a py=py+4

```

Aqui as variáveis são incrementadas com valores diferentes, logo a velocidade horizontal parece ser menor que a velocidade vertical do quadrado verde. Experimente este código no Processing e teste o resultado.

## { Prática }

7 - Utilizando formas geométricas primitivas, faça uma animação que altere cor e outros atributos como largura e altura.

# 10 -Transformações

As funções de transformações podem modificar a geometria dos objetos. Veremos alguns recursos para as transformações bidimensionais.

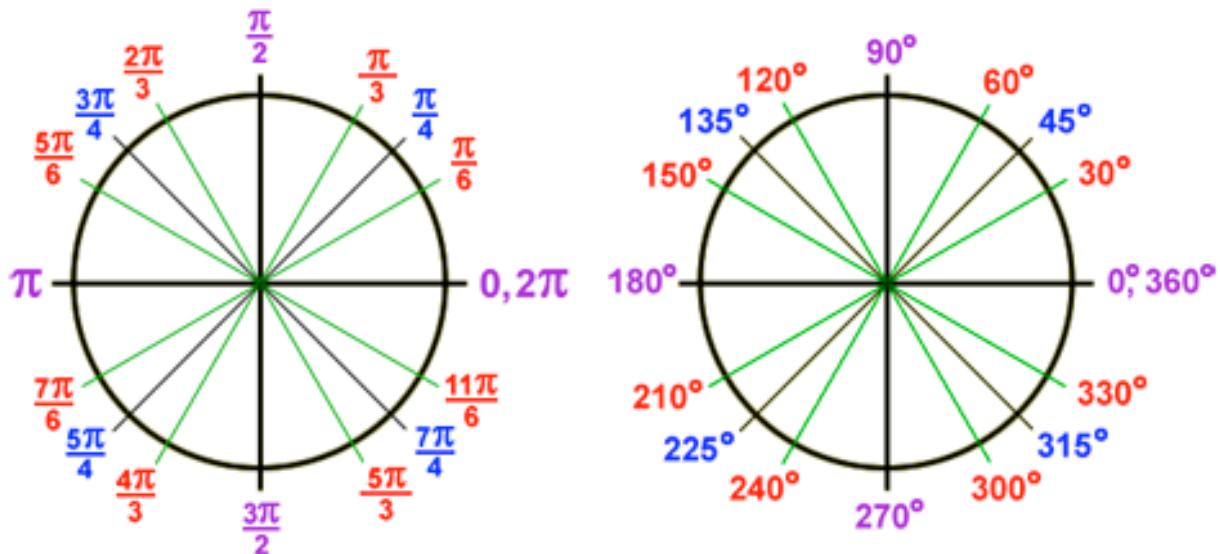
## 10.1 - Rotação

Rotaciona os sistema de coordenadas fazendo com que as formas sejam desenhadas num ângulo específico.

**rotate** (*ângulo*)

O angulo de rotação toma radianos como unidade para rotacionar as formas em torno do ponto (0,0) do sketch. O processo de rotação é cumulativo, isto é, se aplicarmos duas rotações de  $\pi/4$  teremos um angulação final de  $\pi/2$ .

Veja a correspondência entre ângulos registrados em radianos e graus:



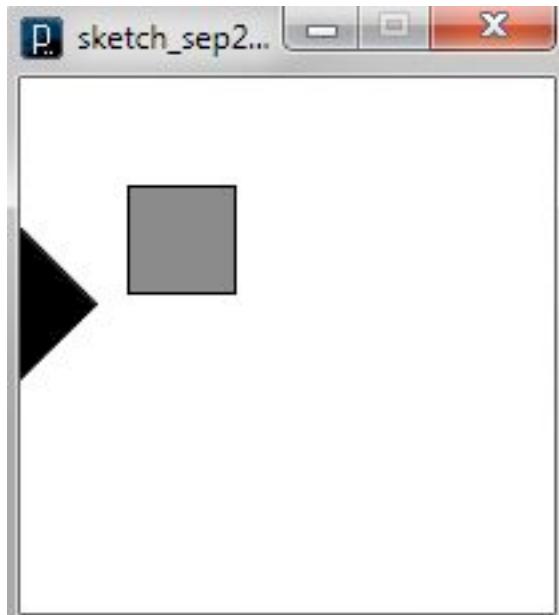
No Processing, o ângulo de rotação deve ser especificado em radianos. Para facilitar, existe uma função matemática ***radians()*** que converte valores especificados em graus para radianos. Neste exemplo, primeiro convertemos o valor da variável ângulo (igual a 45 graus) para seu equivalente em radianos e logo após realizamos a rotação.

```
int angulo=radians(45);  
rotate (angulo);
```

Vejamos como funciona o esquema de rotação no Processing. Planejamos rotacionar em 45 graus um quadrado de 40 pixels de lado, posicionado na coordenada (40,40):

```
size (200,200);  
smooth();  
fill(255,0,0);  
rect (40,40,40,40);  
rotate (radians(45));  
rect (40,40,40,40);
```

Este seria o resultado do sketch:



Note que desenhamos dois quadrados, o primeiro (cinza) sem rotação e o segundo (preto) na mesma posição , porém rotacionado 45 graus. Para entender melhor, veja este gráfico desenhado por [J. David Eisenberg](#):

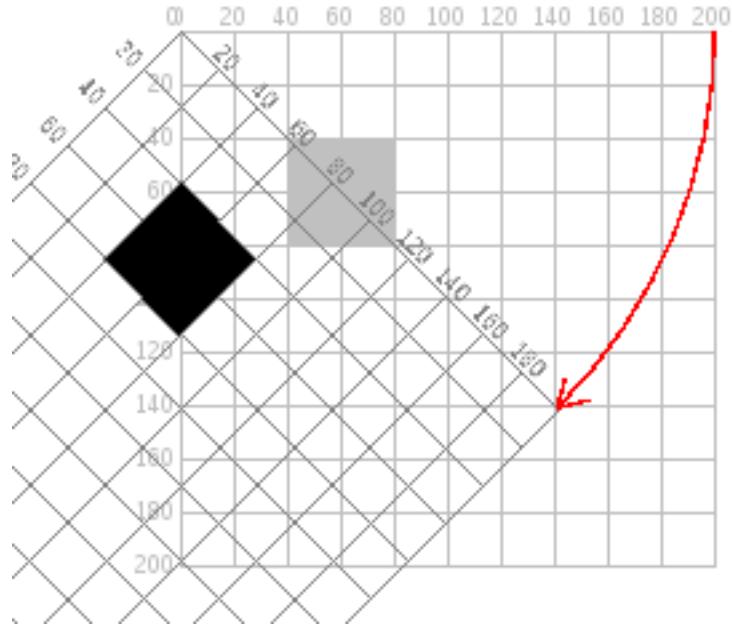


Gráfico criado por J David Eisenberg - <http://www.processing.org/learning/transform2d/>

Perceba que o segundo quadrado não rotacionou em torno do seu próprio centro de registro. A função `rotate()` rotaciona todo o plano das coordenadas e , desta forma, o quadrado se desloca para a esquerda, pois gira junto com o plano cujo o centro é o ponto (0,0). Existe uma forma “correta” de rotacionar uma figura em torno de um ponto que corresponda ao seu próprio centro; veremos isto após o estudo da transformação de translação.

## 10.2 - Translação

A translação é uma transformação ou deslocamento do plano das coordenadas. Assim como a rotação, **não deslocamos as formas** em si, mas sim o plano ou “papel” onde foram desenhadas. Veja o gráfico:

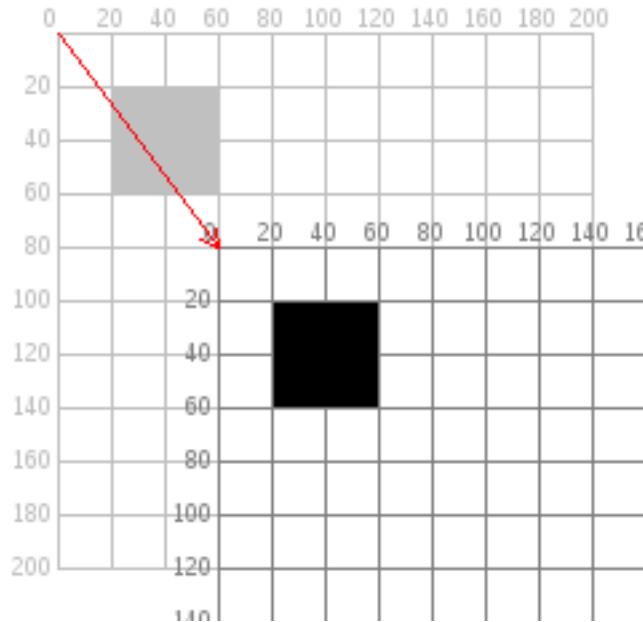


Gráfico criado por J David Eisenberg - <http://www.processing.org/learning/transform2d/>

Neste esquema, apesar da translação, perceba que o quadrado permanece na posição (20,20), relativa ao grid. A função de movimento deve ser escrita da seguinte forma:

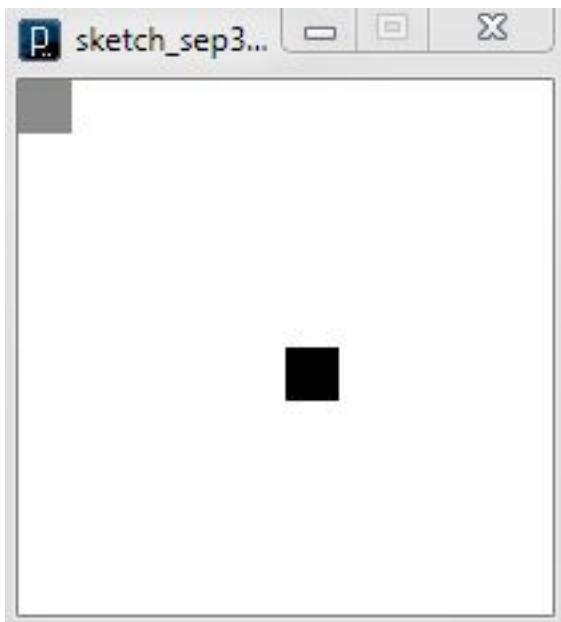
**translate** (x,y);

Os parâmetros x e y são as coordenadas para onde o novo ponto de registro (0,0) do plano de desenho foi deslocado.

Teste o código:

```
size (200,200);
smooth();
background(255);
noStroke();
fill(120);
rect(0,0,20,20);
fill(0);
translate(100,100);
rect(0,0,20,20);
```

Este é o resultado:

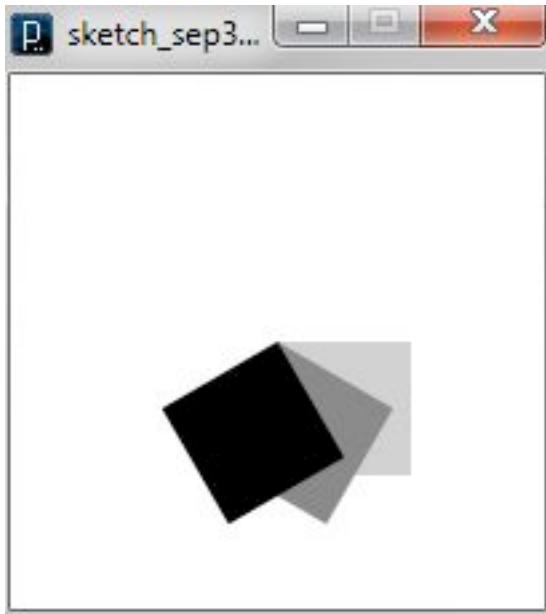


Primeiro o quadrado cinza foi desenhado normalmente na posição (0,0). Depois, mudamos a cor de preenchimento para preto e realizamos um movimento de translação, deslocando a origem do plano de desenho para a posição (100,00). Finalmente desenhamos o segundo quadrado, mas veja que agora ele está na posição deslocada, mesmo que seja desenhado exatamente igual ao primeiro. Apesar dos parâmetros de posicionamento da segunda figura estarem indicados como (0,0) em `rect(0,0,20,20)`, o desenho foi para o centro do sketch devido ao deslocamento anterior efetuado no plano desenho com `translate(100,100)`.

Talvez seja mais fácil apenas alterar a posição do quadrado inicial, porém a função `translate()` torna-se útil quando necessitamos deslocar **várias figuras ao mesmo tempo**.

## 10.3 Combinando as transformações

Como vimos no exemplo da rotação, as figuras giram em torno do **ponto de origem** (0,0) do plano sobre o qual foram desenhadas. Com a combinação da translação e rotação, podemos coincidir este ponto com as coordenadas de registro das próprias figura. Veja o exemplo:



O quadrado mais claro foi desenhado sem rotação no ponto (100,100) , ou centro do sketch. Adotamos este ponto como centro de rotação dos outros dois quadrados. Para que isso ocorra, precisamos primeiro fazer uma translação do plano para esta coordenada (100,100). Na sequencia aplicamos funções de rotação logo antes de desenhar cada uma das figuras. Teste o código:

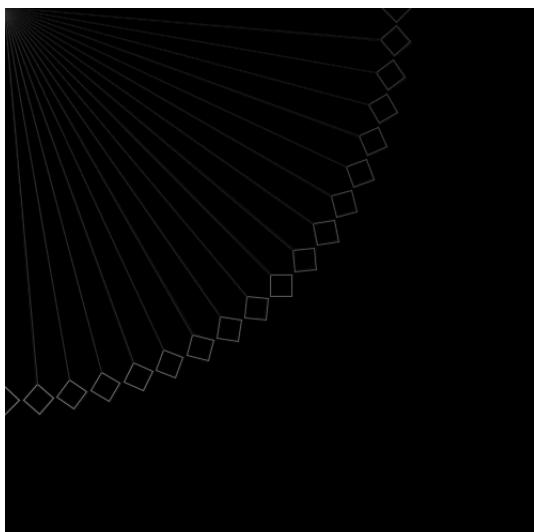
```
size (200,200);
smooth();
background(255);
noStroke();
fill(200);
rect(100,100,50,50); // o ponto (100,100) é o ponto de origem do primeiro quadrado
fill(120);
translate(100,100); // desloca o plano para o ponto (100,100)
rotate(radians(30)); // rotaciona o plano trinta graus em torno desta nova posição
rect(0,0,50,50); // desenhamos o segundo quadrado no ponto (0,0) deslocado
fill(0);
rotate(radians(30)); // rotaciona novamente o plano em 30 graus
rect(0,0,50,50); // o quadrado preto sofre o efeito acumulativo das 2 rotações
```

Como no exemplo anterior, perceba que os quadrados cinza-escuro e preto foram desenhados na coordenada (0,0). Mas agora eles também estão no ponto (100,100), pois o plano foi antecipadamente deslocado com *translate* (100,100). A transformação de deslocamento alterou o centro de rotação das figuras que a agora coincidem com o centro do primeiro quadrado desenhado.

Vamos utilizar a técnica de animação para visualizar estas combinações de uma forma mais dinâmica:

```
int angulo;  
  
void setup() {  
    size (500,500);  
    smooth();  
    background(0);  
    stroke(255);  
    noFill();  
    angulo=0;  
    frameRate(40);  
}  
  
void draw() {  
    fill(0,12);  
    stroke(255);  
    rect (0,0,width,height);  
    angulo+=5;  
    rotate (radians(angulo));  
    rect (250,250,20,20);  
    stroke(100);  
    line (0,0,250,250);  
}
```

Testando esse código, temos:

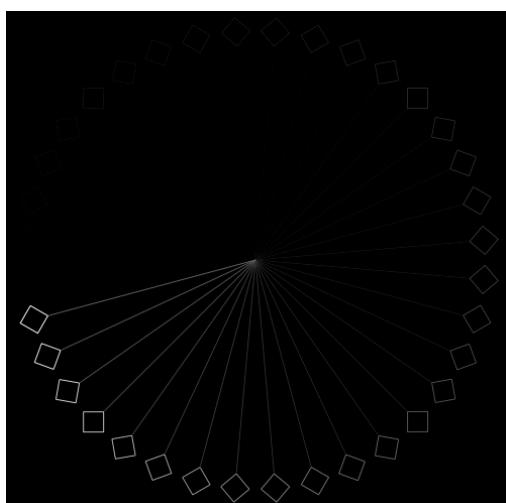


Pode não parecer, mas nesta animação o quadrado está sendo desenhado **sempre** na posição (250,250)! Mas note que ele se desloca sempre em relação ao ponto (0,0) do sketch (desenhamos a linha aqui apenas para reforçar a visualização deste conceito). Na verdade o que está sendo rotacionado é a própria “folha do desenho” com a função *rotate(radians(angulo))*.

Veja que a variável *angulo* está sendo incrementada em 5 unidades constantemente dentro do looping na linha **angulo+=5**. Com a repetição estabelecida dentro de *draw()*, o valor armazenado em ângulo aumenta gradativamente, resultando na acumulação de rotações para o plano. A linha é igualmente rotacionada, apesar das coordenadas fixas de desenho (0,0,250,250).

Na próxima animação experimentamos uma combinação de transformações:

```
int angulo;  
  
void setup() {  
    size (500,500);  
    smooth();  
    background(0);  
    stroke(255);  
    noFill();  
    angulo=0;  
    frameRate(40);  
}  
  
void draw() {  
    fill(0,20);  
    stroke(255);  
    rect (0,0,width,height);  
    angulo+=10;           // aumentamos a velocidade angular  
    translate (250,250);  // deslocamos o plano das coordenadas  
    rotate (radians(angulo));  
    rect (150,150,20,20);  
    stroke(100);  
    line (0,0,150,150);  
}
```



Aqui fizemos pequenas e expressivas alterações. A função *translate* (250,250) desloca o plano para o centro do sketch. Logo, o quadrado deve girar em torno deste novo *pivot*, mas sempre mantendo a posição ou distância relativa 150 com *rect* (150,150,20,20). A linha parte também deste novo centro, por isso conservamos seu ponto inicial em (0,0) e o final na mesma posição do quadrado (150,150).

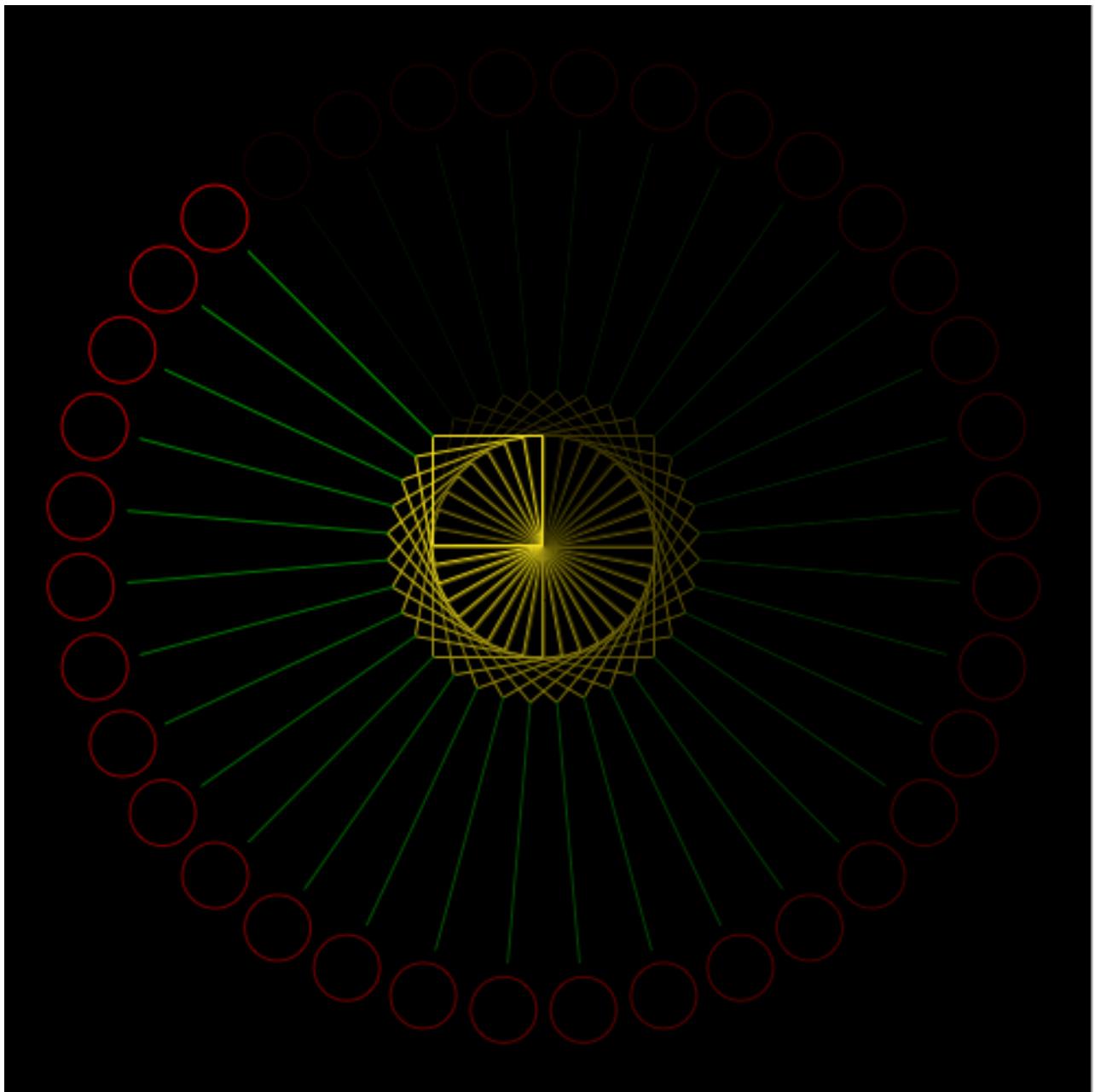
Finalmente, vejamos a combinação de *translate* e *rotate* para a rotação do quadrado em torno de seu próprio centro:

```
int angulo;

void setup() {
  size (500,500);
  smooth();
  background(0);
  stroke(255);
  noFill();
  angulo=0;
  frameRate(60);
}

void draw() {
  fill(0,12);
  stroke(255);
  rect (0,0,width,height);
  angulo+=10;
  translate (250,250);
  rotate (radians(angulo));
  stroke(247,226,32);
  rect (0,0,50,50);           // aumentamos o tamanho do quadrado e centro
  stroke(0,150,0);
  line (50,50,135,135);     // eixo da linha deslocado 50 pixels
  stroke(200,0,0);
  ellipse(150,150,30,30);   // elipse próxima ao final da linha
}
```

Além da alteração do atributo cor, a principal mudança aqui foi o posicionamento do quadrado. Após a translação (idêntica ao exemplo anterior), o centro de rotação da figura coincide agora com o registro (0,0) do plano com *rect*(0,0,50,50). O ponto inicial da linha foi levemente deslocado com relação a este centro , 50 pixels com *line* (50,50,135,135). Acrescentamos o desenho de uma elipse cujo centro fica próximo ao ponto final da linha (150,150).



## 10.4 Escala

A transformação de escala amplia o sistema de coordenadas fazendo com que as figuras sejam desenhadas maiores. A função pode ser descrita em dois formatos:

**scale** (*tamanho*)

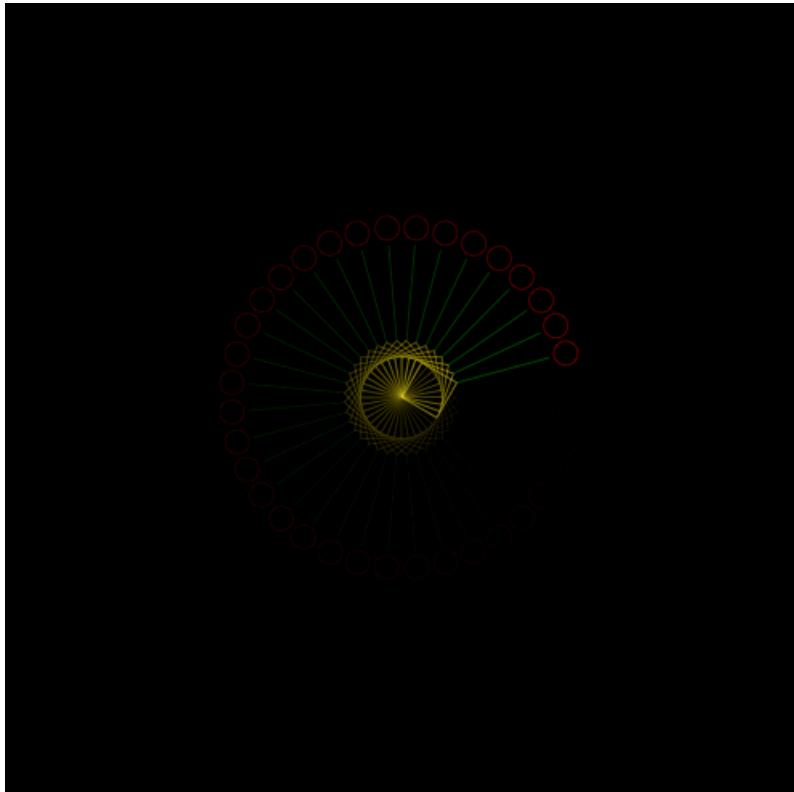
**scale** (*tamanho x, tamanho y*);

No primeiro modo a escala é alterada em todas dimensões, e no segundo modo as escalas do eixo x e y são alteradas separadamente. Os parâmetros de escala são definidos em termos de porcentagem representadas em decimais. Por exemplo 200% corresponde a 2.0, 0.7 equivale a 70%. Utilizando o exemplo anterior, vamos reduzir a escala da animação a metade do tamanho original:

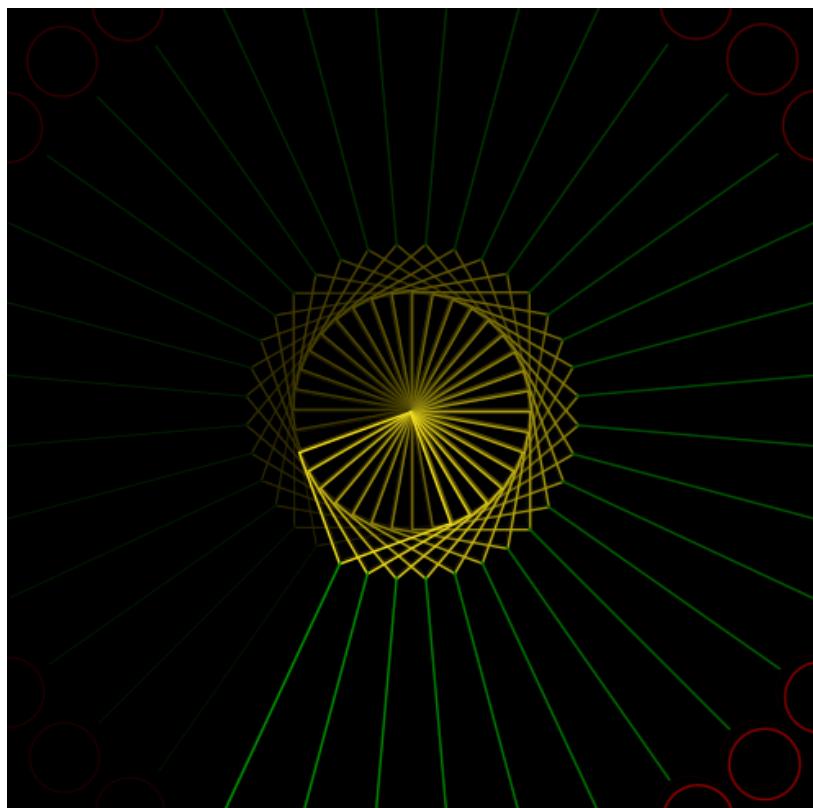
```
int angulo;

void setup() {
size (500,500);
smooth();
background(0);
stroke(255);
noFill();
angulo=0;
frameRate(60);
}

void draw() {
fill(0,12);
stroke(255);
rect (0,0,width,height);
angulo+=10;
translate (250,250);
rotate (radians(angulo));
scale(0.5);           // altera a escala após transformações
stroke(247,226,32);
rect (0,0,50,50);
stroke(0,150,0);
line (50,50,135,135);
stroke(200,0,0);
ellipse(150,150,30,30);
}
```



Quando utilizamos combinações de transformações é importante verificar a ordem de aplicação. Neste exemplo, a transformação de escala foi aplicada logo após rotate(). A inversão da ordem deve gerar outros resultados visuais, pois alteramos as coordenadas e orientações do plano. Um efeito interessante de “zoom in” pode ser criado apenas pelo incremento do parametro de scale():



Este é o código:

```
int angulo;  
float zoom; // declara variável zoom  
  
void setup() {  
    size (500,500);  
    smooth();  
    background(0);  
    stroke(255);  
    noFill();  
    angulo=0;  
    zoom=0; // inicializa variável zoom  
    frameRate(60);  
}  
  
void draw() {  
  
    fill(0,12);  
    stroke(255);  
    rect (0,0,width,height);  
    angulo+=10;  
    zoom+=0.001; // zoom é incrementada no looping  
    translate (250,250);  
    rotate (radians(angulo));  
    scale(zoom); // o parâmetro recebe valor atualizado de zoom  
    stroke(247,226,32);  
    rect (0,0,50,50);  
    stroke(0,150,0);  
    line (50,50,135,135);  
    stroke(200,0,0);  
    ellipse(150,150,30,30);  
}
```

Criamos uma variável *zoom* cujo valor será incrementado gradativamente (+ 0.001) a cada repetição da função *draw()*. A escala aumenta conforme o incremento de zoom em *scale(zoom)*. Ainda é possível o controle de transformações agrupadas por figuras individuais, veremos isso em matrizes de transformação.

# 11 - Matrizes de Transformação

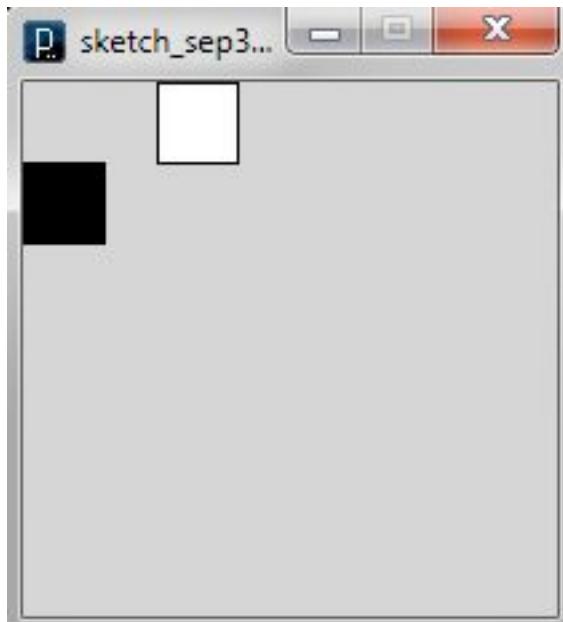
Podemos fazer uma analogia entre o sistema de transformações do Processing e as transformações aplicadas nas camadas (*layers*) utilizadas em pacotes gráficos como o Photoshop . Quando uma transformação é aplicada sobre uma camada , todos os objetos desenhados nesta camada são afetados pela transformação. Felizmente, existe um artifício que facilita a aplicação de alterações em camadas isoladas pelo código.

Na álgebra, a matriz (matrix) é a entidade matemática utilizada no cálculo das transformações geométricas como a [rotação](#) ou [translação](#). No Processing, podemos considerar uma matriz enquanto uma camada (*layer*) , ou área transformativa gerenciada por duas funções:

**pushMatrix()**

**popMatrix()**

Continuando a analogia das camadas, a função *pushMatrix()* é responsável pela adição de uma nova matriz e *popMatrix()* pela remoção da última camada criada na pilha (**stack**) de camadas. Assim como nas camadas, as transformações somente afetarão os objetos desenhados sobre a última matriz adicionada. Veja o exemplo:



```

size (200,200);
pushMatrix();      // adiciona nova “camada” para transformação
translate(50,0);
rect (0,0,30,30);
popMatrix();       // remove “camada” de transformação
fill(0);
rect(0,30,30,30);

```

Neste *sketch*, a transformação `translate(50,0)` afetou apenas o quadrado desenhado entre as funções `pushMatrix()` e `popMatrix()`. Se retirarmos estas duas funções, o segundo quadrado (preto) também será deslocado 50 pixels no eixo x, pois pertence a mesma camada geral de transformações.

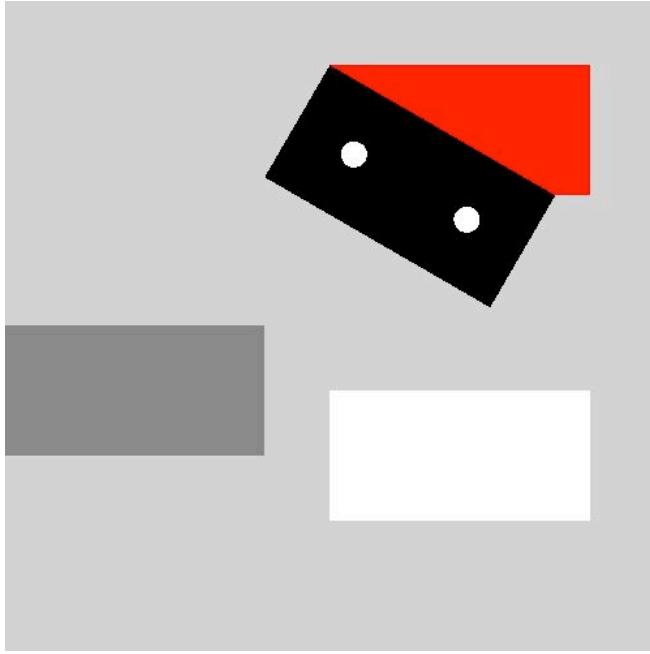
Resumindo, podemos dizer que `pushMatrix` “reserva” uma área para transformações, enquanto `popMatrix()` remove esta área. Uma regra importante para o uso destas funções é a seguinte: **a função `pushMatrix()` não pode ser usada sem `popMatrix()` e vice versa.**

Veja o código:

```

size (500,500);
noStroke();
fill (255,0,0);
pushMatrix();           // insere primeira camada de transformação
    translate(250,50);
    rect (0,0,200,100);
pushMatrix();           // insere segunda camada de transformação
    fill(0);
    rotate (radians(30));
    rect (0,0,200,100);
    fill(255);
    ellipse (50,50,20,20);
    ellipse (150,50,20,20);
popMatrix();            // remove segunda camada de transformação
    rect (0,250,200,100);
popMatrix();            // remove primeira camada de transformação
    fill(120);
    rect (0,250,200,100);

```



A primeira função `pushMatrix()`, assinalada em verde, adiciona uma camada para a transformação do deslocamento `translate(250,0)`. Logo depois, temos a adição de uma segunda área para a rotação com outra `pushMatrix()`, assinalada em laranja, seguida de `rotate(radians(30))`.

O código isolado entre `pushMatrix()` e `popMatrix()`, assinalados em laranja desenha um retângulo preto e duas elipses. Percebem que estas figuras sofrem as transformações inseridas tanto pela primeira função (verde) quanto da segunda (laranja).

O retângulo vermelho, construído por `rect(0,0,200,200)`, sofre apenas o deslocamento, visto que é desenhado logo após a adição da primeira matriz (verde). O retângulo branco ainda está dentro da primeira camada (verde), antes do último `popMatrix()` e, portanto, sofre a transformação exercida por `translate(250,0)`.

O último retângulo (cinza) não sofre nenhuma transformação, pois esta fora das áreas reservadas pelas funções.

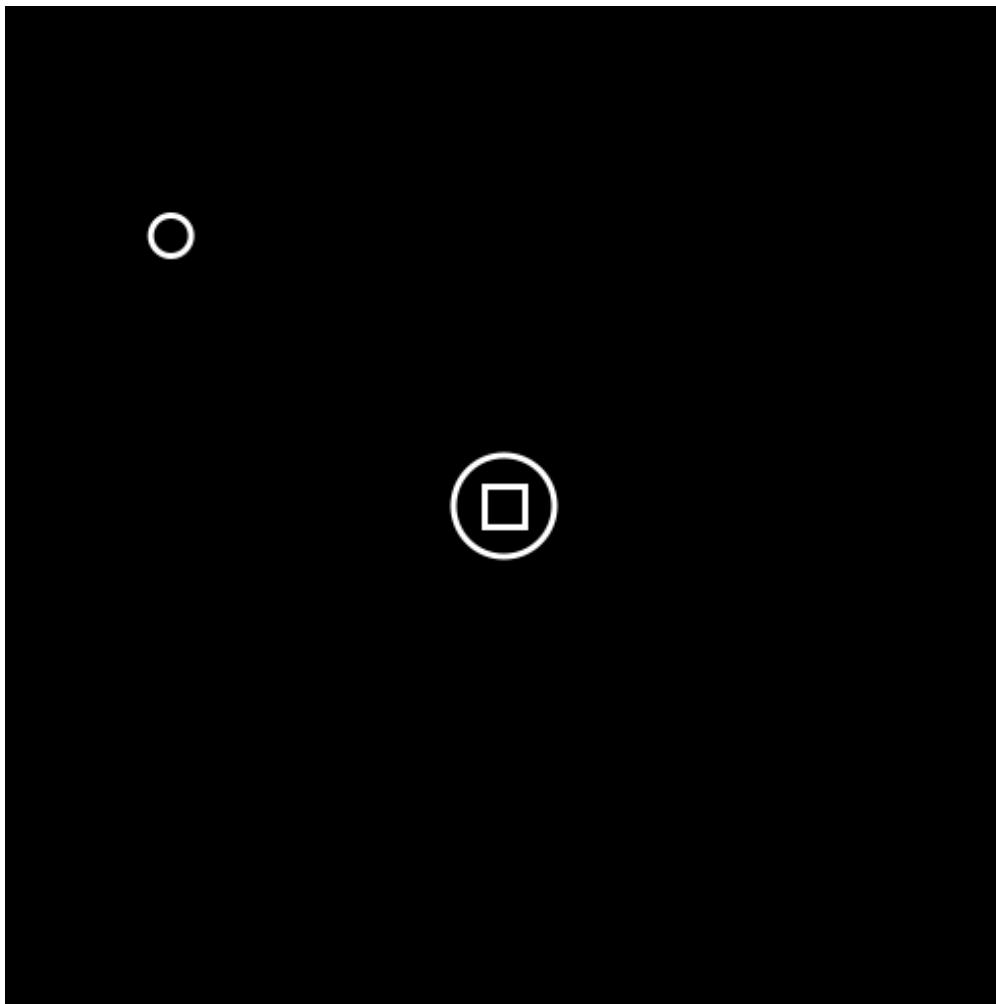
Vamos executar esse conceito de matrizes numa animação. Teste o código:

```
int angulo=0;

void setup() {
  size (500,500);
  background(0);
  noFill();
  smooth();
  strokeWeight(3);
  stroke(255);
  angulo=0;
}

void draw() {
  angulo+=1;
  background(0);
  pushMatrix();
  translate(250,250);
  ellipse(0,0,50,50);
  pushMatrix();
  rotate (radians(angulo));
  ellipse(150,150,20,20);
  popMatrix();
  rect(-10,-10,20,20);
  popMatrix();
}
```

Com o auxílio de *push* e *popMatrix* (laranja), criamos a camada de rotação apenas para a figura *ellipse(150,150,20,20)*. Todas as figuras sofrem a ação de *translate (250,250)*, pois estão inseridas na área criada por *pop/pushMatrix* (verde). Veja que a coordenada do quadrado central é (-10,-10) , dez pixels a esquerda e acima do ponto inicial (0,0) da camada - deslocada 250 pixels para a direita e para baixo por *translate (250,250)*.



## { Prática }

8 - Faça uma animação combinando os três tipos de transformações: rotação, translação e escalonamento. Antes de construir o código faça um planejamento, desenhando um storyboard visual descrevendo tipos de movimentos (variações possíveis), posicionamentos, formatos e cores.

9 - Utilizando o código anterior, isole as transformações de alguns elementos (utilize a técnica de `popMatrix()` e `pushMatrix()`)

## 12 -Funções Randômicas

Softwares como Processing possuem funções que simulam a aleatoriedade. Os números randomicos podem conferir um certo grau de “caos” ou desordem na estrutura das composições, causando efeitos ou resultados estéticos complexos. Vejam estas reproduções das pinturas de [Piet Mondrian](#) e [Jackson Pollock](#):

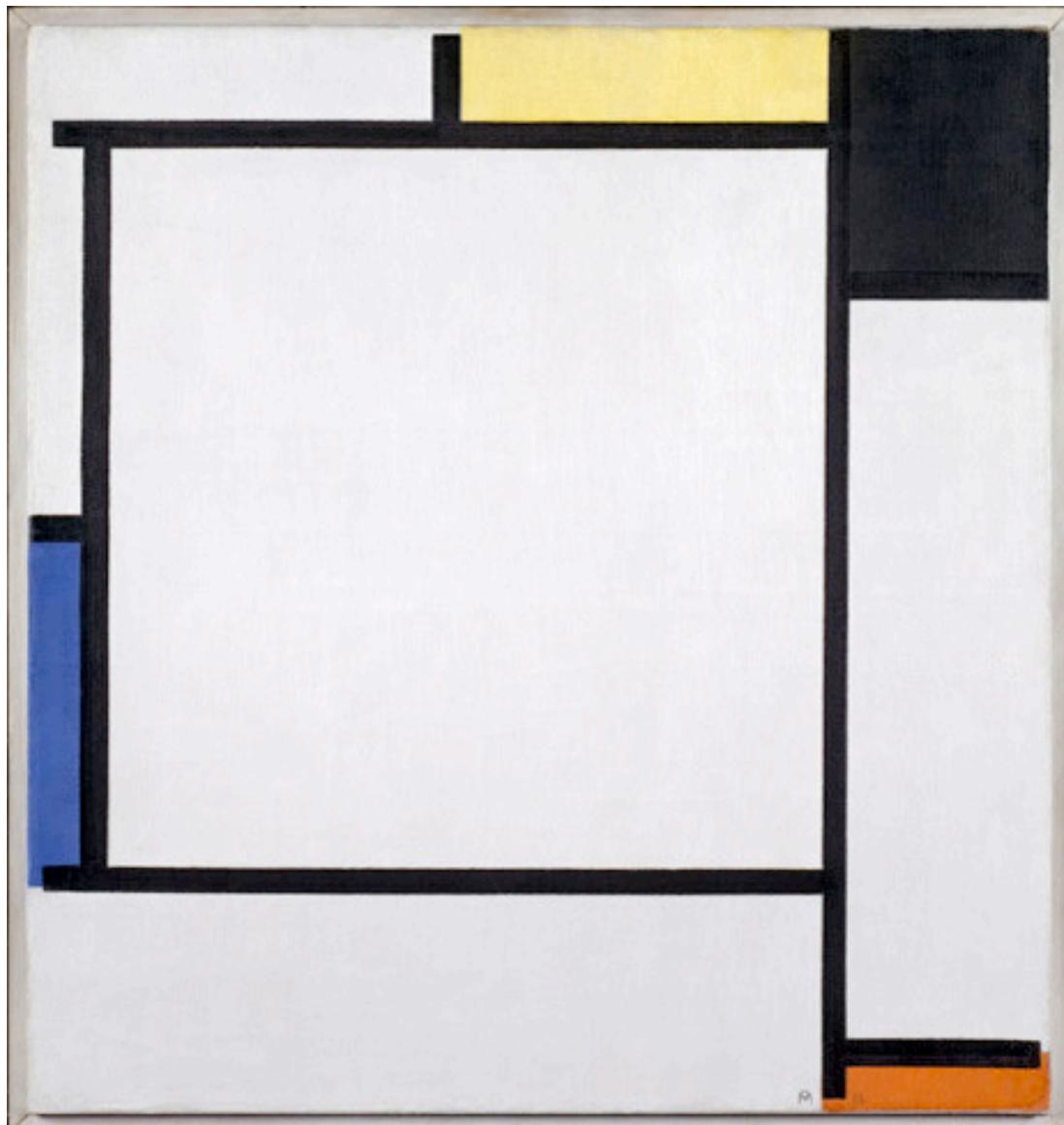


Tableau 2 (Piet Mondrian -1922)

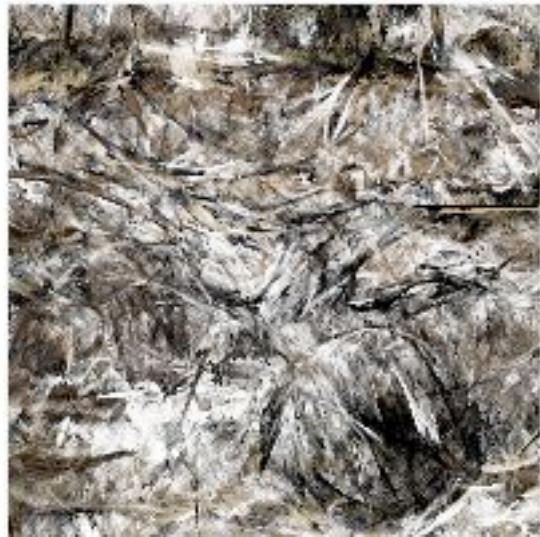


Number 1 (Jackson Pollock - 1949)

É fácil constatar que a pintura de Mondrian denota um aspecto estrutural ordenado e bem construído, tanto pelo uso das linhas ortogonais quanto pela distribuição cromática. Aparentemente, a incerteza e o acaso não entraram na produção ou projeto desta obra.

Já Pollock contou com uma parcela de acaso nesta pintura. Aqui o artista utilizou o seu *drip painting*, método pelo qual despejava quantidades de tinta líquida sobre a tela esticada no chão. Apesar da aparente confusão, o efeito de profundidade e movimento de padrões são inconfundíveis na obra deste representante do expressionismo abstrato. Alguns matemáticos compararam a estrutura de seus quadros com os famosos [fractais](#).

Sabemos que no universo da computação os algoritmos desempenham a função organizativa da programação, isto é, são regras precisas construídas por lógica pura. Neste caso, para obtermos resultados imprevistos, contamos com certas funções matemáticas que fornecem séries aleatórias de números. Veja esta sequencia de imagens:



Intersection.Aggreegate - Jared Tarbell



Intersection.Aggreegate - Jared Tarbell



Intersection.Aggreegate - Jared Tarbell

Estas imagens foram geradas por um código feito no *Processing* pelo artista Jared Tarbell - veja a execução em tempo real no site [Complexification](#). Apesar de apresentarem a mesma escala de cores, as abstrações são diferentes entre si e, de modo geral, lembram as pinturas de Pollock. A série foi gerada pela mesma matriz (código) , ordenada pela mesma regra, porém varia pela introdução das funções randômicas que orientam a duração e movimentação das “pinceladas”. Vejam esta outra imagem de Tarbell:



Substrate - Jared Tarbell

O aspecto geométrico assemelha-se ao processo racional de Mondrian. Novamente, o algoritmo promove a construção, elegendo fatores aleatórios para a posição das linhas e suas subdivisões. O fato da imagem ser “menos caótica” não exclui a possibilidade do uso de algumas variáveis randômicas. Veja aqui uma parcela do código responsável pelo desenho acima – note o destaque para a função **random( )**:

```
void findStart() {  
// pick random point  
int px=0;  
int py=0;  
// shift until crack is found  
boolean found=false;  
int timeout = 0;  
while ((!found) || (timeout++>1000)) {  
px = int(random(dimx));  
py = int(random(dimy));  
if (cgrid[py*dimx+px]<10000) {  
found=true;  
}  
}
```

Os valores aleatórios gerados pela função *random()* são criados a partir de números extraídos do relógio interno(*clock*) do computador e ainda são processados por funções matemáticas. Assim, podemos dizer que estes valores são pseudo-randômicos, pois podem ser repetidos ou recalculados por processos determinísticos previsíveis. De qualquer maneira, a percepção desta previsibilidade não é verdadeira na escala do ponto vista humano. No Processing, geramos essa “impressão” sobre a grandeza aleatória com as seguintes funções:

**random ( valor );**  
**random (valor 1, valor 2);**

Por exemplo, se temos o código:

```
float x=random (10);
```

A função gera um número inesperado que fica no intervalo entre 0 (zero) e o parâmetro, no caso 10. Logo, a variável x pode valer números não inteiros (float) que variam entre este intervalo. No outro formato, temos:

```
float x=random (5,10);
```

Neste caso, contamos com dois parâmetros para a função e aqui x pode valer qualquer número entre 5 e 10. Vamos a um exemplo mais prático:

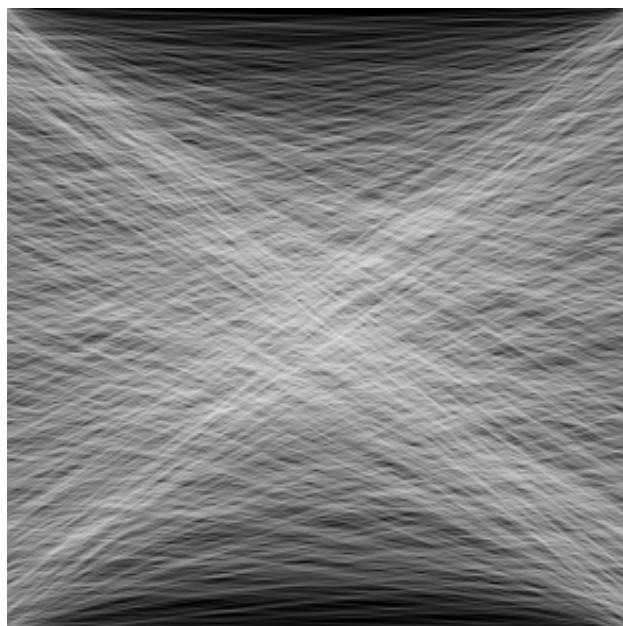
```
line (0 , random(100) , 300, random(100) );
```

Lembrando que para desenharmos uma linha temos uma função cujos parâmetros são coordenadas do ponto inicial (x1,y1) e coordenadas do ponto final da linha (x2,y2):

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

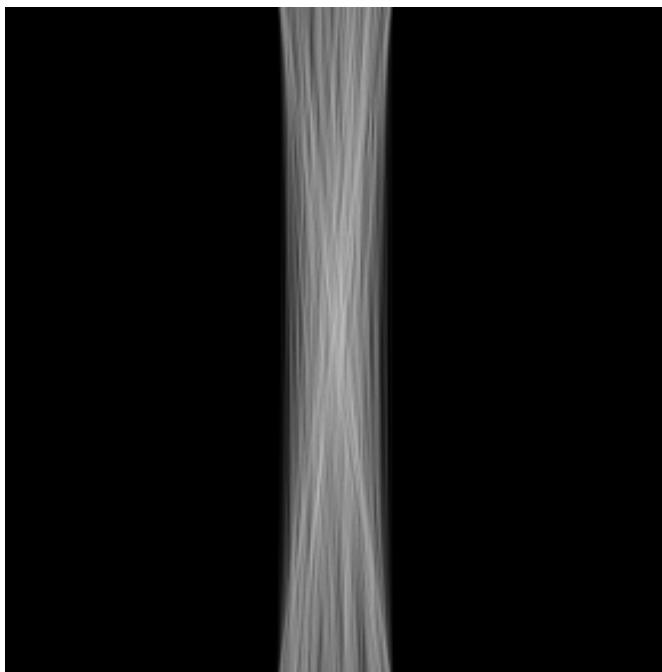
Então, no exemplo, escolhemos números aleatórios para o y1 e y2. Temos como resultado uma linha diagonal ou horizontal que começa na coordenada (0, random(100) ) e termina no ponto (300,random(100) ). O resultado pode ser mais interessante se inserirmos esta função na repetição :

```
void setup() {  
    size (300,300);  
    background(0);  
    smooth();  
    stroke (255,20);  
}  
  
void draw () {  
// background(0);  
    line (0 , random(300) , 300, random(300) );  
}
```



A outra maneira de utilização de *random()* utiliza dois parâmetros. Veja a modificação do código anterior:

```
void setup() {  
    size (300,300);  
    background(0);  
    smooth();  
    stroke (255,10);  
}  
  
void draw () {  
// background(0);  
    line (random(125,175) ,0 , random(125,175) , 300 );  
}
```



Aqui a função limita o intervalo de números aleatórios para as coordenadas da linha:

```
line (random(125,175) ,0 , random(125,175) , 300 );
```

A posição x1 inicial da linha e final x2 variam apenas entre os pontos 125 e 175, determinando o desenho de um feixe central de traços.

## { Prática }

10 - Experimente a função *random()* para gerar movimentos e transformações aleatórias.

# 13 - Funções

Até agora utilizamos funções específicas da linguagem tais como size(), background() ou random() que nos retorna valores aleatórios. Existe ainda a possibilidade de criarmos funções adaptadas às necessidades particulares. As funções tornam nosso código mais conciso e organizado, pois agrupa e concentra instruções que podem ser reutilizadas diversas vezes durante a execução de um programa.

## 13.1 Declaração e Parâmetros

Vamos iniciar com um exemplo prático, criando uma função responsável pelo cálculo da média aritmética de dois números inteiros chamados **N1** e **N2**. Sabemos que a fórmula necessária para a resolução deste cálculo é  $M=(N1+N2)/2$ , ou, a média ( $M$ ) é igual a soma dos números ( $N1+N2$ ) dividida por 2 (/2).

Para a implementação devemos definir essa função:

```
media (int n1, int n2)
```

A declaração ainda não está completa, mas devemos entender o conceito sobre parâmetros. Como vimos, a função *line()* necessita de quatro valores que determinam as coordenadas de dois pontos: *line (0,0,500,500)*. Da mesma forma a função **média** também necessita de alguns parâmetros. Neste caso note que utilizamos o tipo **int** ANTES de cada parâmetro, pois queremos calcular a média de dois números INTEIROS.

Agora precisamos implementar a funcionalidade, ou as instruções necessárias para a realização do cálculo:

```
void media (int n1, int n2) {  
    float soma=n1+n2;  
    float m=soma/2;  
    println (m);  
}
```

Nossa função soma é composta por um bloco contendo três linhas de código aninhadas dentro das chaves {}:

```
float soma=n1+n2;  
float m=soma/2;  
println(m);
```

Primeiro, armazenamos a soma de N1 e N2 na variável **soma**, criada **localmente** para esse bloco. Essa variável é utilizada no cálculo final, quando será dividida por 2 (soma/2).

Segundo, o resultado da operação (soma/2) é armazenado na variável **m**. Assim como soma, essa variável é declarada como **float**, pois o resultado pode ser um número não inteiro.

Terceiro, o resultado é mostrado no console pela instrução `println(m)`.

Quais seriam os valores de N1 e N2?

Onde estão?

Esta é uma simples demonstração da flexibilidade determinada pelas funções. Acabamos de construir uma função para o cálculo da média entre dois números inteiros quaisquer, ou seja, os valores não são incluídos na implementação. Para utilizarmos a função basta o código:

```
void setup() {  
    media (5,4);  
}  
  
void media (int n1, int n2) {  
    float soma=n1+n2;  
    float m =soma/2;  
    println (m);  
}
```

E o resultado será impresso no console (mensagem): **4.5 !**

Neste exemplo não usamos a função `draw()`, pois não desenhamos ou executamos qualquer tipo de repetição. Lembre-se que `setup()` só é executado uma única vez, constituindo um bom local para a chamada isolada da função.

## 13.2 Argumentos

Na instrução *media* (5,4) fizemos a **chamada** da função *media()* , enviando os **argumentos** 5 e 4 para completar os **parâmetros** necessários (int n1, int n2) definidos pela função. Se experimentamos:

```
void setup() {  
    media (5,4);  
    media (1.5, 4);  
}  
  
void media (int n1, int n2) {  
    float soma=n1+n2;  
    float m =soma/2;  
    println (m);  
}
```

Esse código produz uma mensagem de erro do tipo:

The method media (int,int) in the type <nome do sketch> is not applicable for the arguments (float,int)

Isso ocorre porque tentamos enviar um argumento não inteiro (1.5) para um parâmetro definido como inteiro (int n1, int n2), ou seja, uma regra a ser lembrada é que **os argumentos devem ser do mesmo tipo definido pelo parâmetro**.

## 13.3 Retorno da função

Notamos que a partícula **void** foi posicionada logo antes da função *media()*:

```
void media(int n1, int n2) {  
    ...  
}
```

Isto quer dizer que a função não retorna nenhum valor. Vimos que a função *round()*, por exemplo, retorna um valor inteiro. Logo, na expressão:

```
int w= floor (2.1);
```

a variável w recebe 2.0 como resultado, pois *round(2.1)* retorna um valor inteiro.

Na função *media()* que acabamos de criar, a variável ***m*** recebe o resultado do cálculo, mas seu valor não pode ser acessado de qualquer outra parte do código, experimente:

```
void setup() {  
    media (5,4);  
    println (m);  
}  
  
void media (int n1, int n2) {  
    float soma=n1+n2;  
    float m =soma/2;  
}
```

Deslocamos a instrução *println (m)* para dentro de *setup()*. Apesar de ***m*** ter sido declarada e inicializada com o resultado da média, ainda assim não podemos imprimir seu valor, mesmo após a chamada de *media (5,4)*. Como resultado, temos a mensagem de erro:

Cannot find anything named "m"

O sistema simplesmente ignorou a presença da variável, isso ocorre porque ***m*** foi declarada dentro do **escopo da função *media()***.

**Variáveis declaradas dentro de um bloco definido por uma função são válidas ou acessadas pelo intervalo (escopo) determinado pela função. Tais variáveis são armazenadas localmente e eliminadas logo após sua utilização (término da execução do bloco).**

Veremos dois métodos para contornar a situação. O **primeiro** diz respeito aos pontos de entrada e saída de uma função. Em nosso exemplo, podemos considerar os parâmetros ***n1*** e ***n2*** como portas pelas quais podemos injetar valores (argumentos) dentro da função. Essas informações serão processadas por instruções específicas como cálculo da média, por exemplo.

O resultado do processo pode ser externado pela instrução ***return***, isto é, uma forma de saída das informações processadas pelo bloco. Veja o exemplo:

```
void setup() {  
    println (media (5,4));  
}  
  
float media (int n1, int n2) {  
    float soma=n1+n2;  
    float m =soma/2;  
    return m;  
}
```

Perceba que alteramos o código e trocamos **void** por **float**. Desta forma definimos que a função *media()* retornará um valor do tipo não inteiro, fazendo com que o sistema reserve uma área de memória compatível. O retorno do valor calculado e armazenado em **m** será dado pela instrução:

```
return m;
```

Assim podemos acessar esse valor diretamente como foi feito em **println( media(5,4) )**. Neste caso, estamos imprimindo justamente o valor retornado , ou resultado do processo armazenado em **m**. Devemos observar que o tipo da variável retornada deve ser o mesmo declarado para o retorno da função:

```
float media (int n1, int n2) {  
    ...  
    ...  
    float m= soma/2;  
    return m;           // deve retornar float  
}
```

No segundo método não utilizaremos retorno de valor. Para isso, vamos declarar a variável **m** fora do escopo das funções *setup()* e *media()*:

```
float m;
```

```
void setup() {  
    m=0;  
    media (5,4);  
    println(m);  
}
```

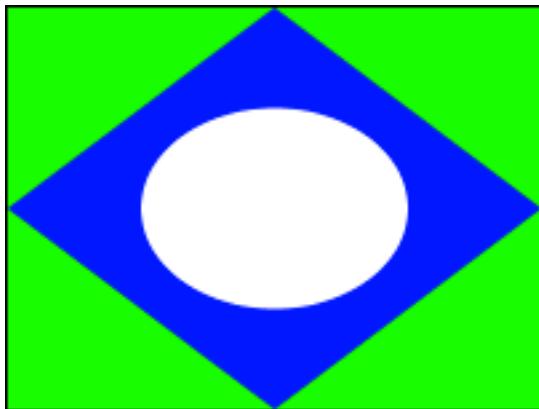
```
void media (int n1, int n2) {  
    float soma=n1+n2;  
    m =soma/2;  
}
```

Declarando uma variável fora de um bloco ou função esta poderá ser acessada a qualquer momento, até que a execução do programa seja finalizada. Desta forma, seu valor pode ser alterado, pois agora a variável é declarada **globalmente**. Note que a inicialização da variável (*m=0*) foi realizada dentro de *setup()* e a função *media()* não retorna nenhum valor (*void*). Dentro da função, o valor da variável foi alterado (*m=soma/2*) e pode ser exibido logo após a chamada *media(5,4)* em *setup()*.

Existem vantagens e desvantagens nesse método. Utilizamos variáveis **globais** (fora das funções) quando precisamos acessá-las de constantemente. A desvantagem é que alocamos parcelas de memória de maneira persistente. Já as variáveis **locais** são utilizadas dentro de funções ou laços de repetição (veremos adiante). Esse tipo de variável é utilizadas apenas na execução de instruções delimitadas pelo bloco.

## 13.4 Criando uma função para desenho

Vamos construir uma função, utilizando os conhecimentos sobre variáveis, formas, atributos de desenho e função randômica. Como proposta inicial, projetamos a seguinte figura, que será desenhada por uma função:



Isolando as formas que compõem a figura, podemos obter algumas pistas sobre os **parâmetros** que serão necessários para a implementação da função:

- a figura é composta por um **retângulo** (deve ser posicionado em algum ponto das **coordenadas** e possui propriedades como **largura**, **altura** e **cor** de preenchimento);
- a figura é composta por um **quadrilátero** cujos **vértices** estão localizados nos pontos médios que dividem os lados do retângulo descrito;
- a figura é composta por uma **elipse** posicionada no **centro** do retângulo
- a elipse possui largura e altura proporcionais às dimensões do retângulo
- existe uma ordem para a composição dos elementos (retângulo, quadrilátero e elipse)

Analisando a figura desta maneira, podemos estipular um **algoritmo** que constituirá o corpo da função:

- escolhemos uma cor de preenchimento;
- desenhamos um retângulo de largura e altura variáveis;
- posicionamos o retângulo num ponto específico do espaço;
- calculamos os pontos médios dos lados a partir das propriedades do retângulo (posição e dimensões);
- escolhemos uma cor de preenchimento;
- utilizando as coordenadas dos pontos médios, desenhamos um quadrilátero;
- escolhemos uma cor de preenchimento;
- desenhamos uma elipse cuja posição deve ser calculada utilizando os pontos medios e as dimensões proporcionais às dimensões do retângulo;

A construção do quadrilátero e da elipse está vinculada às propriedades do retângulo, portanto estabelecemos a posição e dimensões do retângulo como parâmetros da função.

Vamos declarar a função “bandeira”:

```
void bandeira (float posx, float posy, float largura, float altura) {  
    ...  
}
```

A função conta com quatro parâmetros do tipo *float*:

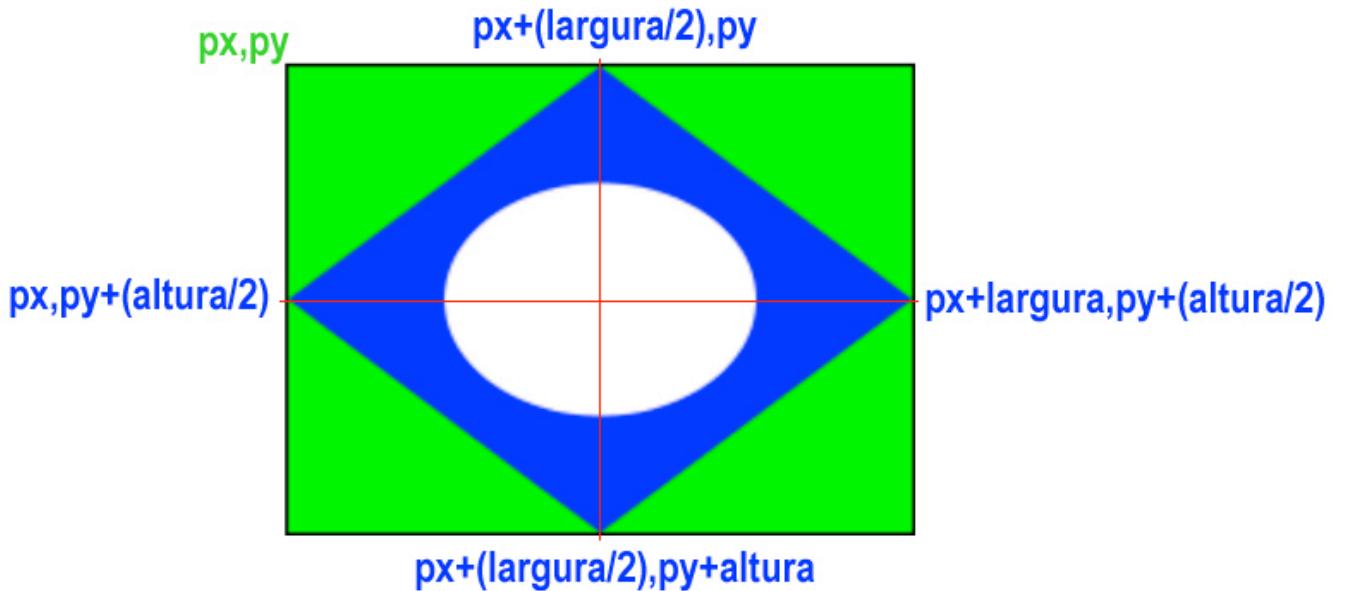
**posx e posy**: coordenadas para posicionamento;

**largura e altura**: coordenadas para dimensões;

Utilizaremos esses valores para o desenho do retângulo, seguindo o algoritmo descrito acima:

```
void bandeira (float posx, float posy, float largura, float altura) {  
    fill (0,255,0);  
    rect (posx,posy,largura,altura);  
}
```

Precisamos calcular os pontos médios dos lados do retângulo para a construção do quadrilátero. Para isso criamos duas variáveis (**alturamedia** e **larguramedia**) locais que armazenam a divisão das dimensões dadas. Veja o gráfico:



Vamos implementar o desenho do quadrilátero, conforme coordenadas do gráfico:

```
void bandeira (float posx, float posy, float largura, float altura) {  
    fill (0,255,0);  
    rect (posx,posy,largura,altura);  
    float alturamedia=altura/2;  
    float larguramedia=largura/2;  
}
```

Vamos recordar que a função **quad()** necessita de oito parâmetros representantes das posições de seus vértices:

```
void bandeira (float posx, float posy, float largura, float altura) {  
    fill (0,255,0);  
    rect (posx,posy,largura,altura);  
    float alturamedia=altura/2;  
    float larguramedia=largura/2;  
    fill (0,0,255);  
    quad ( posx, posy+alturamedia, posx+larguramedia ,posy, posx+largura, posy  
    +alturamedia, posx+larguramedia, posy+altura );  
}
```

Finalmente implementamos a elipse utilizando as variáveis alturamedia e larguramedia para determinar a altura e largura:

```
void bandeira (float posx, float posy, float largura, float altura) {  
    fill (0,255,0);  
    rect (posx,posy,largura,altura);  
    float alturamedia=altura/2;  
    float larguramedia=largura/2;  
    quad ( posx, posy+alturamedia, posx+larguramedia ,posy, posx+largura, posy  
    +alturamedia, posx+larguramedia, posy+altura );  
    fill(255);  
    ellipse (posx+larguramedia,posy+alturamedia,larguramedia,alturamedia);  
}
```

A construção da composição pode ser chamada dentro da função `setup()`:

```
void setup() {  
    size (500,500);  
    background(0);  
    noStroke();  
    smooth();  
    bandeira (250,250,200,150);  
  
}  
  
void bandeira (float posx, float posy, float largura, float altura) {  
    fill (0,255,0);  
    rect (posx,posy,largura,altura);  
    float alturamedia=altura/2;  
    float larguramedia=largura/2;  
    quad ( posx, posy+alturamedia, posx+larguramedia ,posy, posx+largura, posy  
+alturamedia, posx+larguramedia, posy+altura );  
    fill(255);  
    ellipse (posx+larguramedia,posy+alturamedia,larguramedia,alturamedia);  
}
```

Podemos ainda construir mais de uma figura, utilizando o looping da função `draw()` e `random()` para determinar argumentos aleatórios:

```
void setup() {  
    size (500,500);  
    background(0);  
    noStroke();  
    smooth();  
}  
  
void draw() {  
    bandeira (random(width), random(height) random (300),random(300));  
}  
  
void bandeira (float posx, float posy, float largura, float altura) {  
    fill (0,255,0);  
    rect (posx,posy,largura,altura);  
    float alturamedia=altura/2;  
    float larguramedia=largura/2;  
    quad ( posx, posy+alturamedia, posx+larguramedia ,posy, posx+largura, posy  
+alturamedia, posx+larguramedia, posy+altura );  
    fill(255);  
    ellipse (posx+larguramedia,posy+alturamedia,larguramedia,alturamedia);  
}
```

Note o uso das constantes **width** e **height** como argumentos para `random()`. Esses valores representam a largura e altura total do sketch, garantindo que a posição sorteada de cada retângulo esteja dentro de toda a extensão do sketch.

# 14 - Exportar as animações

As animações geradas no Processing podem ser salvas em tempo real no formato vídeo digital (padrão Quicktime).

Neste código desenhamos com o mouse:

```
void setup() {  
background(204);  
}  
  
void draw() {  
ellipse(mouseX, mouseY, 20, 20);  
}
```

Para salvar cada momento da interação devemos usar uma biblioteca especial **processing.video.\***, veja o novo código:

```
import processing.video.*;  
  
MovieMaker mm;  
  
void setup() {  
size(640, 480);  
mm = new MovieMaker(this, width, height, "drawing.mov",  
30, MovieMaker.MOTION_JPEG_A, MovieMaker.BEST, 1);  
  
background(204);  
}  
  
void draw() {  
ellipse(mouseX, mouseY, 20, 20);  
mm.addFrame();  
}  
  
void keyPressed() {  
if (key == ' ') {  
mm.finish();  
}  
}
```

Explicando o código:

Você notou que utilizamos alguns recursos de interação com mouse e teclado.  
Não se preocupe com esse código, pois abordaremos processos de interação em outro módulo.

Primeiro importamos a biblioteca necessária para a geração do vídeo **com import processing.video.\***. Criamos uma variável (mm) do tipo **MovieMaker** que armazena o novo vídeo que será gravado:

**MovieMaker mm;**

Dentro da função **setup()** configuramos o parâmetro desta varável:

```
mm = new MovieMaker(this, width, height, "drawing.mov",
30,MovieMaker.MOTION_JPEG_A, MovieMaker.BEST,1);
```

Note que os parâmetros determinam a largura, altura, nome do filme, frames por segundo (fps), codec para quicktime, qualidade e keyframeRate. Neste exemplo, estamos exportando um vídeo chamado “desenho.mov” com as dimensões iguais a do size (640,480). Cada formato de vídeo digital tal como o [Quicktime](#) ou AVI possuem diferentes formas de compactação ([codec](#)). Em nosso exemplo, estamos usando o MOTION\_JPEG\_A , mas existem outros ideais para gráficos vetoriais como o ANIMATION.

A qualidade de compactação pode ser modificada como no parâmetro BEST e o intervalo de keyframes no último parâmetro setado em 1. Tanto a qualidade quanto o keyframerate influenciam no tamanho final do arquivo. Para maiores detalhes sobre codec e qualidade vejam em :

<http://processing.org/reference/libraries/video/MovieMaker.html>

Na função **draw()** temos ainda a linha **mm.addFrame();** que é responsável pela cópia dos pixels da janela do Processing no vídeo que está sendo gravado.

Finalmente dentro da função **keyPressed()** temos **mm.finish()** que finaliza e salva a gravação no formato Quicktime (.mov). Para verificar a gravação, pressionamos a tecla de espaço e verificamos a pasta do sketch onde o vídeo deve estar gravado.

Um último detalhe: para o funcionamento do sketch é necessária a instalação do Quickime (download no site da [Apple](#))

## { Prática }

10 - Experimente exportar alguma animação randômica (utilize a função **keyPressed()** para interromper o processo).

# **Módulo III**

# 14 - Interação

A interação com o ambiente do Processing ocorre pela captura de eventos como a movimentação do mouse, clique ou o acionamento de alguma tecla. Existem ainda outros dispositivos físicos especiais que permitem a leitura de outros tipos de inputs como os sensores de movimento, temperatura, pressão e mesmo de imagens de uma simples webcam. Neste tópico veremos eventos exclusivos de mouse e teclado.

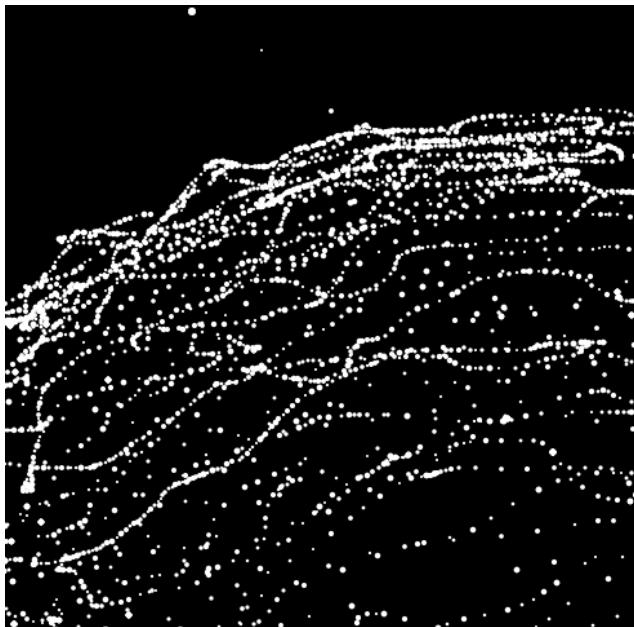
## 14.1 Propriedades do Mouse

Se imaginamos que existe um objeto virtual associado ao mouse físico então seria possível que existam áreas de memória (variáveis) que armazenam a posição (coordenadas) registradas pelo mouse físico. As *propriedades* são estas variáveis que guardam certos valores ou atributos associados a um objeto. No Processing, estas são as propriedades relativas ao mouse:

**mouseX**  
**mouseY**  
**pmouseX**  
**pmouseY**

Iniciamos com a utilização destas propriedades. Veja o código:

```
void setup() {  
    size (500,500);  
    background(0);  
    smooth();  
    noStroke();  
    noCursor();  
}  
  
void draw() {  
    float diam=random(2,5);  
    ellipse(mouseX,mouseY,diam,diam);  
}
```



Neste *sketch* criamos uma ferramenta simples de desenho. Na função **draw()** , a variável *diam* armazena um valor aleatório que varia entre 2 e 5. Este valor randômico é utilizado para determinar a largura e altura da circunferência desenhada na função **ellipse()**. As posições do mouse armazenadas automaticamente nas variáveis *mouseX* e *mouseY* determinam a posição vertical e horizontal dos círculos desenhados no *looping*.

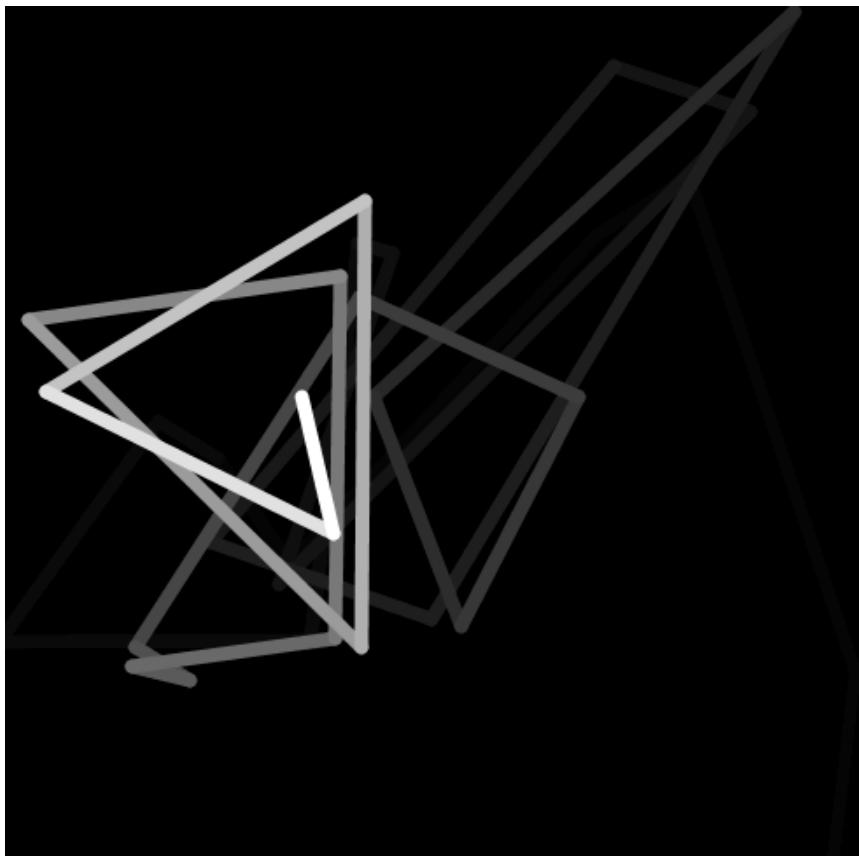
Perceba que o cursor (seta) do mouse foi desabilitada pela função **noCursor()** escrita dentro do **setup()**. Utilizamos apenas uma variável aleatória (*diam*) para determinar tanto a largura quanto a altura da elipse , pois queremos uma circunferência perfeita.

Em muitas aplicações interativas surge a necessidade de verificar a velocidade de deslocamento do mouse para a obtenção de resultados dinâmicos. Por exemplo, uma aplicação de ilustração pode registrar a velocidade do gesto para simular a pressão do pincel ou lápis, variando o visual da espessura do traço ou “pincelada”.

Vamos recordar que velocidade pode ser traduzida pela variação de deslocamento com relação ao tempo decorrido da ação (espaço/tempo). No Processing, o **tempo** é mensurado em **frames** e o **espaço** em **pixels** (coordenadas xy). As propriedades **pmouseX** e **pmouseY** facilitam muito esse cálculo da velocidade, pois registram as posições do mouse em frames (instantes) anteriores.

Veja o exemplo:

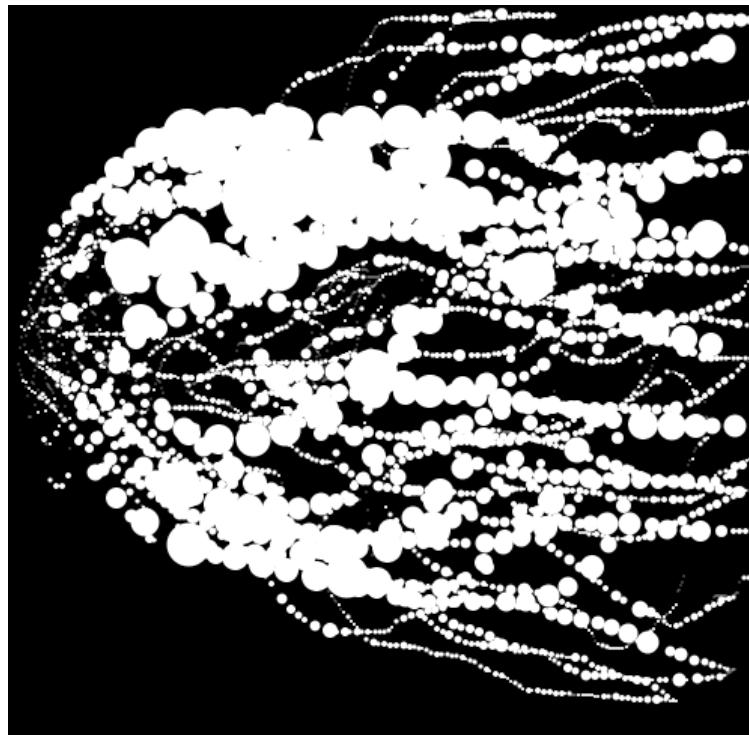
```
void setup() {  
    size (500,500);  
    background(0);  
    strokeWeight(8);  
    stroke(255);  
    smooth();  
    noCursor();  
    frameRate(10);  
}  
  
void draw() {  
    background(0);  
    fill (0,30);  
    rect(0,0,width,height);  
    line (mouseX,mouseY,pmouseX,pmouseY);  
}
```



A função *line()* necessita de dois pontos. O primeiro ponto é retirado da posição atual do mouse (*mouseX*,*mouseY*). O segundo ponto corresponde a posição do mouse em outro momento (*pmouseX*,*pmouseY*). Teste o código clicando e arrastando o mouse e perceba que a extensão do traço varia conforme a velocidade, quanto mais rápido o gesto maior é a linha.

Agora, vamos modificar o exemplo:

```
void setup() {  
    size (500,500);  
    background(0);  
    smooth();  
    noStroke();  
    noCursor();  
}  
  
void draw() {  
    println ("pmouseX= "+pmouseX);  
    println ("mouseX= "+mouseX);  
    if (pmouseX>0 && mouseX>0) {  
        float diam=abs(pmouseX-mouseX);  
        ellipse(mouseX,mouseY,diam,diam);  
    }  
}
```



Os tamanhos dos círculos variam conforme a quantidade de deslocamento (distância percorrida pelo mouse) calculado em:

```
float diam=abs(pmouseX-mouseX);
```

Subtraímos a posição final do mouse da posição inicial e esta diferença pode gerar um número negativo. Por exemplo, num instante o mouse pode estar na posição x=200 e no frame seguinte está na posição x=100. O resultado da subtração (100-200) é -100 (100 negativo) e esse valor não pode ser aplicado para a altura ou largura na função `ellipse()`.

No Processing, a função matemática `abs()` converte qualquer número em seu valor absoluto, ou seja, converte para o valor positivo. Logo, `abs(100-200)` equivale a 100 (positivo). Veja [aqui](#) mais detalhes sobre esta função.

A linha `if (pmouseX>0 && mouseX>0)` determina uma instrução condicional que estudaremos mais adiante. Basta entender que o cálculo e o desenho somente serão executados apenas se a posição horizontal do mouse for maior que zero.

## 14.2 Utilizando Eventos

Os **eventos** são ocorrências específicas disparadas pela intervenção externa do usuário ou mesmo do próprio sistema. No Processing estes eventos externos são reconhecidos e gerenciados pelas seguintes função especiais (handlers):

<b>mouseButton()</b>	- identifica qual botão foi pressionado (LEFT,CENTER,RIGHT)
<b>mouseClicked()</b>	- identifica se algum botão foi pressionado e liberado
<b>mouseDragged()</b>	- identifica se o mouse se movimenta enquanto botão estiver pressionado
<b>mouseMoved()</b>	- ocorre quando o mouse é movimentado
<b>mousePressed()</b>	- ocorre quando o botão é pressionado uma vez
<b>mouseReleased()</b>	- ocorre quando o botão é liberado
<b>cursor()</b>	- habilita o cursor do mouse
<b>noCursor()</b>	- desabilita cursor do mouse
<b>delay()</b>	- força uma pausa por um período de tempo (milisegundos)
<b>redraw()</b>	- Executa o bloco dentro de <code>draw()</code> apenas uma vez
<b>exit()</b>	- interrupção do programa

Aproveitando um código do tópico anterior utilizaremos o mouse para construir uma ferramenta de desenho:

```
void setup() {  
    size (500,500);  
    background(0);  
    smooth();  
    noStroke();  
    noCursor();  
}  
  
void draw() {  
}  
  
void mouseDragged() {  
    float diam=random(2,5);  
    ellipse(mouseX,mouseY,diam,diam);  
}
```

Nesta modificação, o desenho dos círculos só ocorre enquanto algum botão do mouse estiver pressionado, pois as instruções estão dentro de mouseDragged(). Neste caso, a função draw() fica sem utilidade – veja que as ações estão associadas ao evento de mouse.

## 14.3 Controlando o looping

Até agora vimos que as animações ocorrem automaticamente graças ao fluxo de processamento contínuo gerado pela função draw(). Existem duas funções que podem desativar e reativar esse fluxo: loop() noLoop() Utilizaremos estas duas funções associadas aos eventos de mouse para interromper e reiniciar uma animação simples:

```
void setup() {  
    size (500,500);  
    background(0);  
    smooth();  
    noStroke();  
    fill (150,30);  
}  
  
void draw() {  
    rect (random(width),random(height),random(10,50),random(10,100));  
}  
  
void mousePressed() {  
    noLoop();  
}  
  
void mouseReleased() {  
    loop();  
}
```

Este sketch desenha uma série de retângulos transparentes com posições e dimensões randômicas. A função *mousePressed()* captura o evento de botão e ativa a função *noLoop()* que interrompe o fluxo. O fluxo só pode ser reativado com a função *loop()* que está incluída na função *mouseReleased()*. Logo, se pressionamos algum botão do mouse a animação é interrompida e só é retomada se soltamos o botão. De outra forma, podemos controlar a execução do looping *draw()* utilizando um evento de mouse e a função ***redraw()***:

```
float x=0;
void setup() {
    size(200, 200);
    noLoop();
}

void draw() {
    background(204);
    line(x, 0, x, height);
}

void mousePressed() {
    x += 1;
    redraw();
}
```

Nesse sketch, desativamos a execução de *draw()* logo no início com *noLoop()*, dentro da função *setup()*. Utilizamos o evento *mousePressed()* para incrementar um contador que é usado como argumento para a função *line()*. Além disso, o evento força um redesenho (update) da área de display com *redraw()*.

Além da função *frameRate()* , podemos forçar um atraso na execução de *draw()* pelo uso de função ***delay()***:

```
int pos = 0;

void draw() {
    background(204);
    pos++;
    line(pos, 20, pos, 80);
    delay(2000);           // 2000 milisegundos= 2 segundos
}
```

Nesse sketch, a atualização do desenho é feita em intervalos de 2 segundos. A instrução *delay(2000)* é inserida dentro de *draw()*, pois deve ser executada em cada looping (redesenho).

A função **exit()** deve ser utilizada somente quando necessitamos interromper o programa totalmente, causando o fechamento da janela do sketch:

```
float x=0;
void setup() {
    size(200, 200);
}

void draw() {
    background(204);
    x++;
    line(x, 0, x, height);
}

void mousePressed() {
    exit();
}
```

## 14.4 Utilizando o teclado

As teclas também podem ser utilizados como dispositivo de interação. Os eventos associados são:

- keyPressed()** – identifica se alguma tecla foi pressionada
- keyReleased()** – identifica se alguma tecla foi liberada

Utilizaremos o evento de teclado para apagar toda a tela:

```
void setup() {  
    size (500,500);  
    background(0);  
    smooth();  
    noStroke();  
    fill (150,30);  
}  
  
void draw() {  
    rect (random(width),random(height),random(10,50),random(10,100));  
}  
  
void mousePressed() {  
    noLoop();  
}  
  
void mouseReleased() {  
    loop();  
}  
  
void keyPressed() {  
    background(0);  
}
```

No final do código acrescentamos a função *keyPressed()*, que é responsável pela captura de eventos. Se alguma tecla for pressionada, a instrução *background(0)* é executada , apagando a área de desenho.

# 15 - Decisões

As instruções condicionais permitem um controle maior sobre o fluxo ou a ordem de execução, pois direcionam a decisão sobre qual linha de código deve ou não ser executada. Vamos a um exemplo prático: queremos imprimir a mensagem “Teste” **dez** vezes. Essa seria a lógica do algoritmo:

Descrição do algoritmo:

Iniciar o programa

O contador inicia com valor um.

Se o contador não chegar a dez eu imprimo a palavra teste e incremento o contador.

Se o contador for maior que 10 eu encerro programa.

Para que o programa saiba quantas vezes a palavra foi impressa precisamos de uma variável para acumular a contagem – chamamos esta variável de **contador**. Durante o processo a variável é incrementada (acrescentamos um valor). Se o valor acumulado chegar a 10 paramos a execução. Esta é a representação numa linguagem próxima aquelas utilizadas em programação:

Descrição do programa em pseudocódigo:

INICIO

Contador=1

PASSO 1 {

SE (contador<=10)

ENTÃO

escrever “teste”

contador=contador+1

IR para PASSO 1

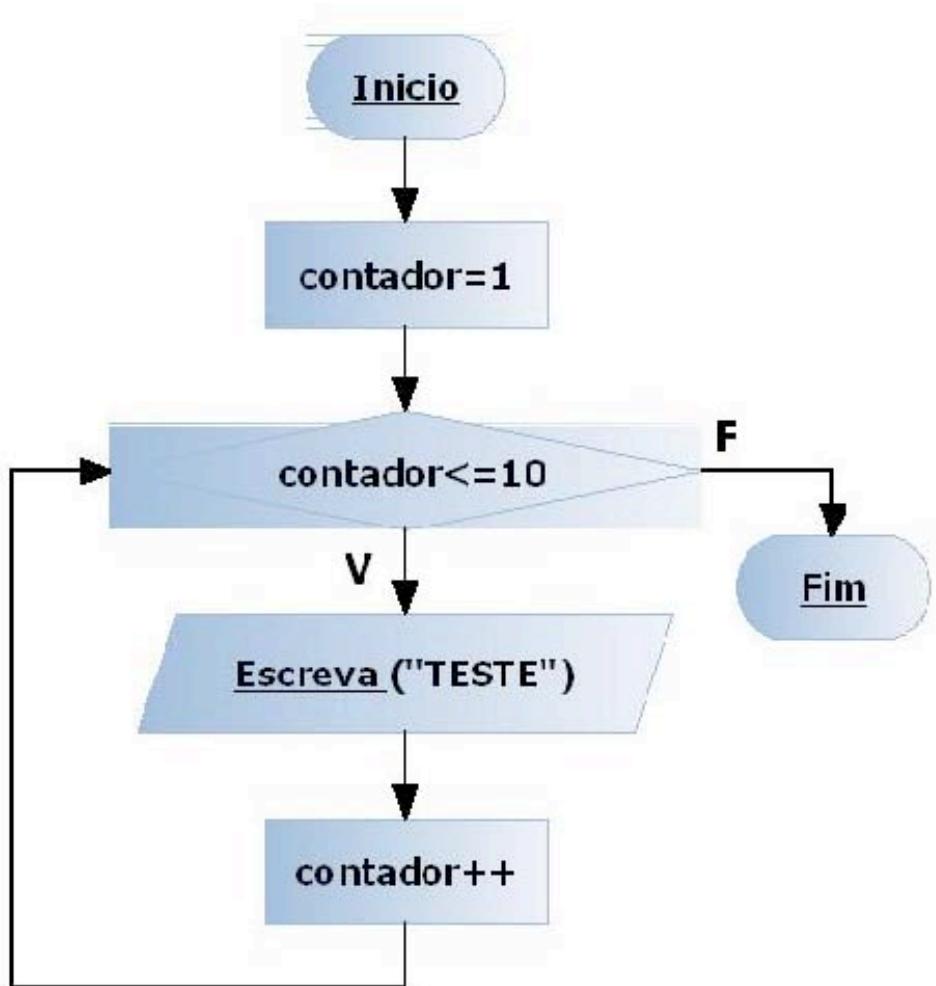
SENÃO

IR para FINAL

}

FINAL

Note que existe um teste para verificar se a variável ainda não atingiu o nível máximo pretendido (10). SE (contador<=10,) verifica se a variável contador é **menor ou igual** ( $\leq$ ) a 10. Isso quer dizer que o resultado desta avaliação deve ser **verdadeiro** para a continuidade do fluxo e impressão da mensagem. O fluxograma é uma ferramenta para a representação gráfica deste algoritmo:



Se o valor do contador for maior que zero então a cláusula (condição)  $\text{contador} \leq 10$  é falsa (F) estado que leva ao Fim da execução.

## 15.1 Estrutura Condicional (if...else)

No Processing e na maioria das linguagens de programação a estrutura condicional é escrita da seguinte forma:

```
if (condição) {  
    instrução
```

...

...

```
    instrução;
```

```
}
```

A instrução IF é utilizada para testar se a CONDIÇÃO é verdadeira e , neste caso, as instruções aninhadas dentro de { } são executadas. Caso contrário, se a CONDIÇÃO não for verdadeira, então nenhuma dessas intruções são executadas. Neste exemplo,

```
if (x>30) {  
    println("fim");  
}
```

Se o valor de uma variável x for maior que 30 então imprimir uma mensagem “fim”. Vamos adaptar o algoritmo inicial para a linguagem do Processing. No fluxograma, perceba que existe um direcionamento do fluxo (seta) que leva a uma repetição da condicional IF após o incremento da variável.

No código, implementamos a **repetição** visto que o teste está dentro da função **draw()**:

```
int contador;           // declaramos a variável contador

void setup() {
  contador=1;
}

void draw() {           // dentro do looping de draw()
  if (contador<=10) { // testamos a condição SE contador é menor ou igual a 10

    println ("teste"); // se a condição for verdadeira imprimimos “teste”
    contador++;        // e incrementamos o contador

  } else {             // SENÃO (se a condição não é mais verdadeira - desvio para

    println("fim");   // imprimir “fim”

  }
} // fecha draw()
```

Distribuímos o código pelas funções **setup()** e **draw()**. Inicialmente criamos a variável **contador**. Em **setup()** , iniciamos a variável **contador** com valor igual a 1. Na função **draw()** testamos o valor de **contador** e se for menor OU igual a dez, imprimir mensagem “**teste**” e, logo após, incrementa variável com **contador++**. Finalmente **ELSE** (senão) , ou a se a condição não for mais verdadeira ( maior que dez) imprime **fim**.

Note que aparece um novo elemento no código: ELSE. Esta instrução oferece uma saída alternativa caso a condição não seja mais verdadeira, traduzindo:

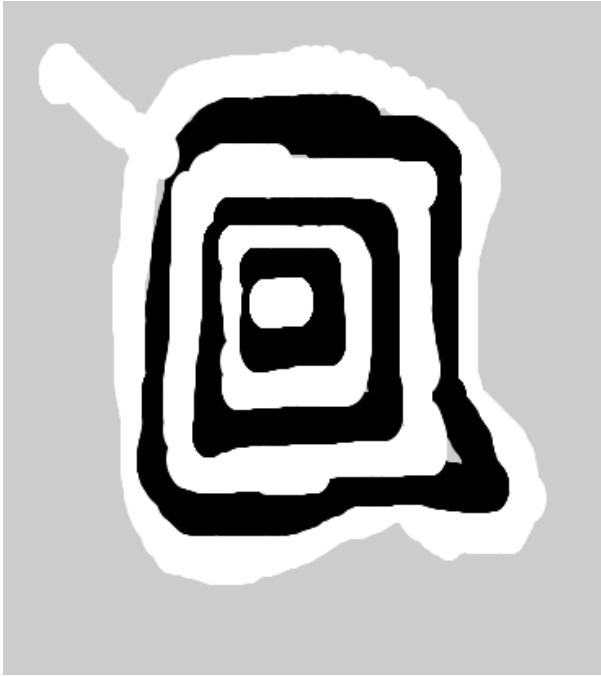
```
SE (condição) {  
....  
} SENÃO {  
....  
}
```

No Processing:

```
if (condição) {  
....  
} else {  
.... }
```

Vamos para um exemplo mais prático utilizando condicionais e eventos de mouse. Como foi visto, a função `mouseButton` identifica qual botão do mouse foi pressionado e pode valer LEFT, RIGHT ou CENTER (esquerda, direita ou centro). Utilizaremos as condicionais para testar qual botão foi pressionado e desenhar círculos com tons diferentes :

```
void setup() {  
    size(500,500);  
    noStroke();  
    smooth();  
}  
  
void draw() {  
    ellipse (mouseX,mouseY,20,30);  
}  
  
void mousePressed() {  
    if (mouseButton == LEFT) {  
        fill(0);  
    }  
  
    if (mouseButton == RIGHT) {  
        fill(255);  
    }  
  
    if (mouseButton==CENTER) {  
        fill(126);  
    }  
}
```



Nenhum segredo nesta função draw() que desenha uma elipse na posição atualizada do mouse. Em mousePressed () avaliamos o valor de mouseButton com o auxílio de três testes condicionais. Por exemplo:

```
if (mouseButton == LEFT) {  
    fill(0);  
}  
}
```

Aqui testamos se o botão esquerdo (LEFT) foi pressionado. Se isto for verdadeiro então mudamos a cor de preenchimento para preto em fill(0). As outras condicionais fazem o mesmo tipo de teste variando apenas a escolha do botão (RIGHT,CENTER) e respectivas tonalidades branco e cinza.

Existe uma outra forma de escrever a sequencia de condicionais encadeando os testes com a forma IF ELSE:

```
void mousePressed() {  
    if (mouseButton == LEFT) {  
        fill(0);  
    } else if (mouseButton == RIGHT) {  
        fill(255);  
    } else {  
        fill(126);  
    }  
}
```

Na modificação acima existe um teste inicial e os testes seguintes são encadeados restando uma última alternativa com else { fill(126) }. O encadeamento poderia ser maior, veja o formato:

```
if (condição) {  
    ...  
} else {  
    ...  
} // fim do teste
```

## 15.1 Operadores Relacionais

Na expressão relacional (`mouseButton == LEFT`) o sinal (`==`) corresponde a uma comparação de **equivalência** e não a uma igualdade (`=`). Ou seja, não afirmamos que `mouseButton` é igual a `LEFT` mas SE `mouseButton` é o mesmo que `LEFT`. Este tipo de sinal utilizado nas expressões relacionais é chamado de operador. Veja a lista de operadores possíveis:

- > maior que
- < menor que
- $\geq$  maior ou igual que
- $\leq$  menor ou igual que
- `==` equivalente a
- `!=` não equivalente a

As expressões podem gerar dois resultados: TRUE (verdadeiro) e FALSE (falso). Exemplo:

`3>5` retorna false

`3>=5` retorna false

`3==5` retorna false

`5!=5` retorna false

Vamos utilizar alguns desses operadores para controlar o fluxo de execução de uma animação:

```
int posx;  
void setup() {  
    size(500,500);  
    noStroke();  
    smooth();  
    posx=0;  
}  
  
void draw() {  
    background(0);  
    if (posx<width) {  
        ellipse (posx,300,30,30);  
        posx++;  
    } else {  
        posx=0;  
    }  
}
```

A variável **posx** foi criada para determinar a posição horizontal do desenho. O círculo só é desenhado se **posx<width**, pois não permitimos que a figura ultrapasse o limite direito (width) do sketch. Ao contrário, se o limite for ultrapassado (condição é falsa) , então ELSE altera posx para seu valor inicial igual a zero e a condição volta a ser satisfeita (verdadeira).

Existe uma forma reduzida para escrever a estrutura **if...else** utilizando o operador **?:** , veja o exemplo:

```
float valor= random(50);  
float resultado= (valor<25) ? 0 : 255;
```

Nestas instruções, sorteamos um valor entre 0 e 50 e a variável resultado recebe 0 (zero) se a condição (valor<2) for verdadeira, ou (else) recebe 255 se a condição for falsa.

A condicional:

**resultado = condição ? expressão 1 : expressão 2**

equivale a:

```
if (condição) {  
    resultado=expressão 1;  
} else {  
    resultado= expressão 2;  
}
```

## 15.2 Operadores Lógicos

Existem situações nas quais precisamos testar mais de uma condição simultaneamente. Neste caso, podemos utilizar os operadores (símbolos) que correspondem aos conceitos lógicos de E (and), OU (or) e NÃO (not):

**&&** (AND)

**||** (OR)

**!** (NOT)

A seguir, apresentamos uma tabela de combinações de expressões lógicas e suas respectivas avaliações (CONSIDERE V=VERDADEIRO e F=FALSO) :

V && V = V

V && F = F

F && F = F

V || V = V

V || F = V

F || F = F

! V = F (inversão)

! F = V (inversão)

Considere o código:

```
int resultado;
int a=10;
int b=20;
if ( (a>5) && (b<100) ) {
    resultado=0;
} else {
    resultado=1;
}
println (resultado);
```

Na execução desse código temos a impressão do algarismo zero (valor guardado na variável resultado), pois o valor da variável **a** é **maior** que 5 e b é também **menor** que 100.

Neste exemplo, utilizamos o operador **&&** (AND) para verificar se as expressões (**a>5**) E (**b<100**) são verdadeiras simultaneamente. Caso uma das expressões for falsa, então (else) o valor de resultado passa a valer 1.

*Vamos verificar um exemplo prático:*

```
float vx, vy;

void setup () {
    size (400,400);
    background(0);
    fill(255,5);
    noStroke();
    vx=random(width);
    vy=random(height);
    frameRate(120);
}

void draw() {

    if ((vx>width) || (vy>height) || (vx<0) || (vy<0)) {

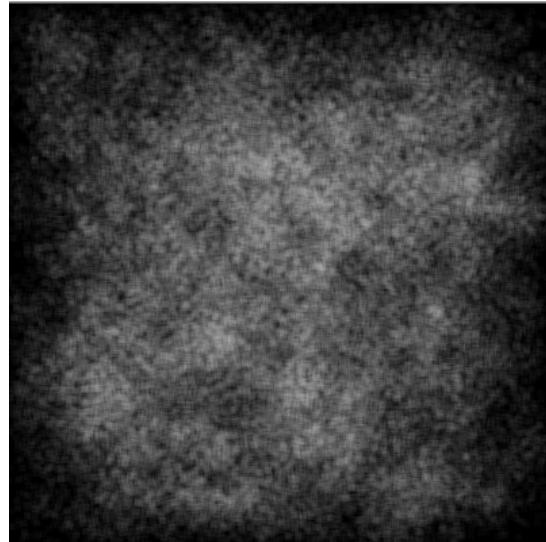
        vx=random(width);
        vy=random(height);

    } else {

        vx+=random(-10,10);
        vy+=random(-10,10);
        ellipse (vx,vy,5,5);

    }
} // última chave fecha função draw()
```

Após executar esse sketch por um intervalo de tempo, temos o resultado:



Este desenho é construído por rastros que são compostos por elipses posicionadas em coordenadas aleatórias. Note que inicialmente criamos as variáveis **vx** e **vy** as quais foram inicializadas com valores aleatórios em *setup()*:

```
vx=random(width);  
vy=random(height);
```

Os argumentos *width* e *height* garantem que os valores sorteados estejam dentro do intervalo determinado pela largura e altura do sketch. Na função *draw()*, impletamos o algoritmo (regra):

*1 - se os valores de vx e vy (coordenadas da elipse) ultrapassarem os limites do sketch (altura e largura) , sortear outros valores que estejam dentro do intervalo de terminado por width e height.*

*2- senão, se os valores de vx e vy estivrem ainda no intervalo, some nas variáveis valores randômicos que possam variar no intervalo -10 e 10. Desenhe uma elipse na nova posição dada por vx e vy.*

Na implementação da primeira regra utilizamos os **operadores lógicos** para a verificação das posições:

```
if ((vx>width) || (vy>height) || (vx<0) || (vy<0)) {
```

```
    vx=random(width);  
    vy=random(height);
```

A estrutura condicional IF serve para o teste das expressões:

**(vx>width)** - se variável **vx** é **maior** que a largura do sketch

**(vx<0)** - se variável **vx** é **menor** que a largura do sketch (coordenada x negativa)

**(vy>height)** - se variável **vy** é maior que a altura do sketch

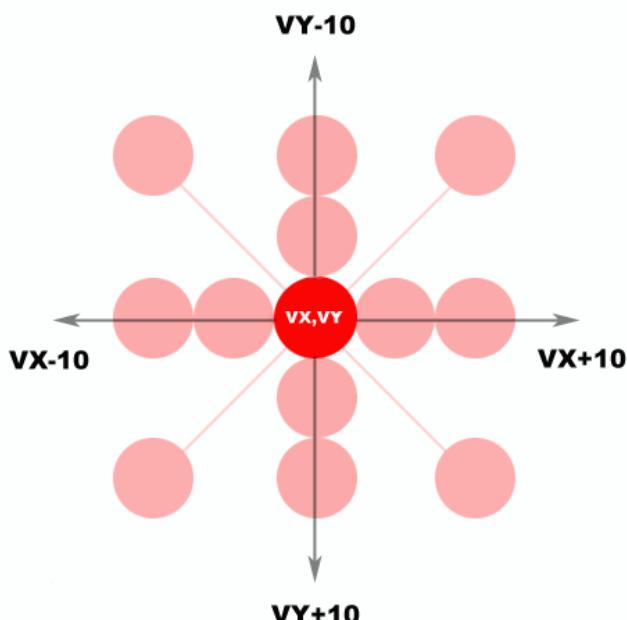
**(vy< 0)** - se variável **vy** é menor que a altura do sketch (coordenada y negativa)

Foi utilizado o operador **||** (OR) , pois qualquer uma das expressões não pode ser verdadeira (TRUE). Se alguma das expressões for avaliada como verdadeira, logo o valor da variável está fora do intervalo e desta forma sorteamos outro valor válido para as variáveis.

Caso contrário, se nenhuma expressão for verdadeira, permitimos o desenho (regra 2) , continuado pelo ELSE (senão) da condicional:

```
} else {  
    vx+=random(-10,10);  
    vy+=random(-10,10);  
    ellipse (vx,vy,5,5);  
}
```

As variáveis são adicionados (ou subtraídos) valores que podem variar no intervalo entre -10 e 10. Esse intervalo garante que a próxima elipse possa ser desenhada em outra direção ou sentido:



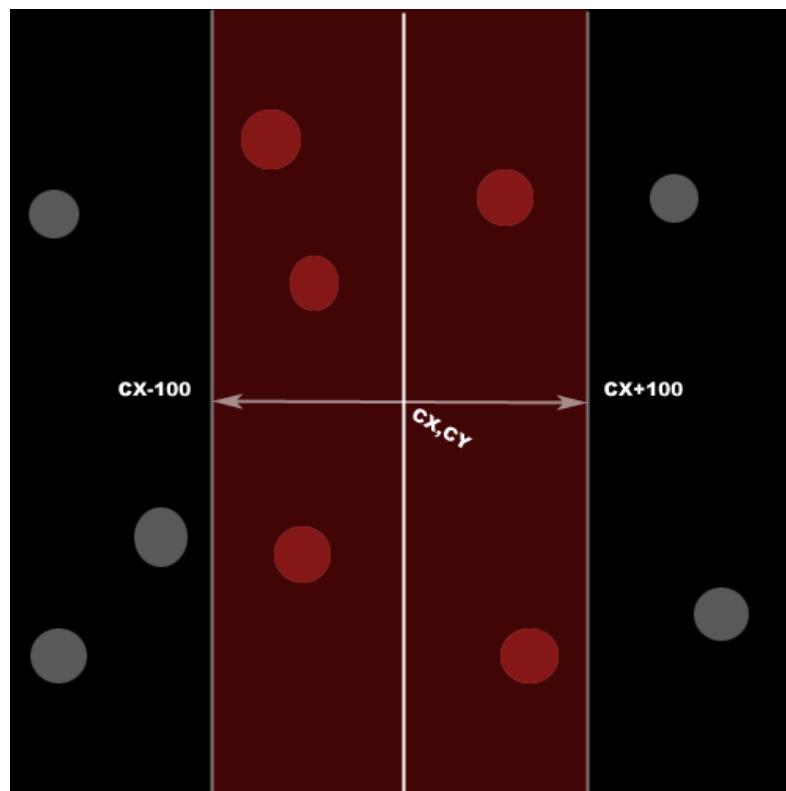
Vamos utilizar os operadores lógicos para delimitar uma área onde as elipses serão desenhadas com outra cor. Escolhemos uma faixa central do sketch, ou seja, um intervalo que parte do eixo vertical central do sketch. Para isso, criaremos duas outras variáveis **cx** e **cy** que armazenarão as coordenadas do centro do sketch:

```
float vx, vy;  
float cx, cy; // declaramos cx e cy  
  
void setup () {  
    size (400,400);  
    background(0);  
    fill(255,5);  
    noStroke();  
    vx=random(width);  
    vy=random(height);  
    cx=width/2; // inicializamos cx e cy com a metade da largura e altura  
    cy=height/2;  
    frameRate(120);  
}  
}
```

Acrescentaremos mais uma regra ao algoritmo:

3 - As elipses que estiverem num intervalo de 100 pixels a partir do centro do sketch serão desenhadas na cor vermelha.

Veja o diagrama:



Vamos implementar essa nova regra ao sketch anterior, utilizando as variáveis **cx** e **cy** criadas. Para verificar se a elipse será desenhada neste intervalo central, incluiremos outro teste condicional:

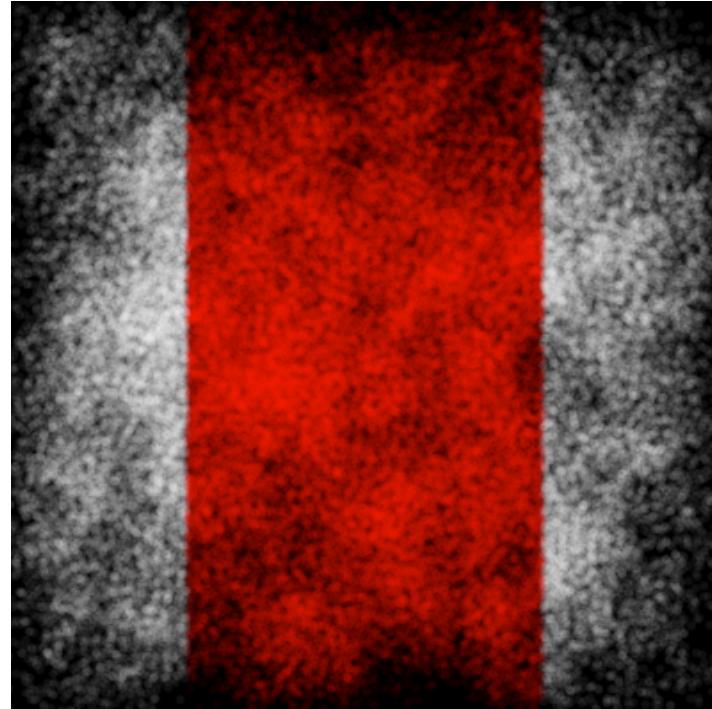
**Se**  $vx > (cx - 100)$  E  $vx < (cx + 100)$ , então desenhe elipses vermelhas **SENÃO** desenhe elipses brancas. O código é modificado, incluimos a condicional DENTRO da condicional anterior:

```
float vx, vy;  
float cx, cy;
```

```
void setup () {  
    size (400,400);  
    background(0);  
    fill(255,5);  
    noStroke();  
    vx=random(width);  
    vy=random(height);  
    cx=width/2;  
    cy=height/2;  
    frameRate(120);  
}  
  
void draw() {  
  
    if ((vx>width) || (vy>height) || (vx<0) || (vy<0)) {  
  
        vx=random(width);  
        vy=random(height);  
  
    } else {  
        vx+=random(-10,10);  
        vy+=random(-10,10);  
        if ( (vx>(cx-100)) && (vx<(cx+100)) ) { // inserimos um novo teste  
            fill (255,0,0,5); // para a seleção da cor  
        } else { // antes de desenhar  
            fill (255,5);  
        }  
        ellipse (vx,vy,5,5);  
    }  
}  
} // última chave fecha função draw()
```

Note o uso de **&& (AND)** na condicional. Este operador é útil, pois queremos testar se as duas expressões (**vx > (cx - 100)** e **vx < (cx + 100)**) são verdadeiras simultaneamente, indicando que as coordenadas **vx** e **vy** estão dentro do intervalo de 100 pixels para a direita ou esquerda do centro vertical definido por **cx**.

Veja o resultado do novo sketch:



Percebemos nitidamente os limites demarcados por nosso algoritmo. No próximo passo, vamos diluir um pouco este limite randomizando o valor de 100 pixels.

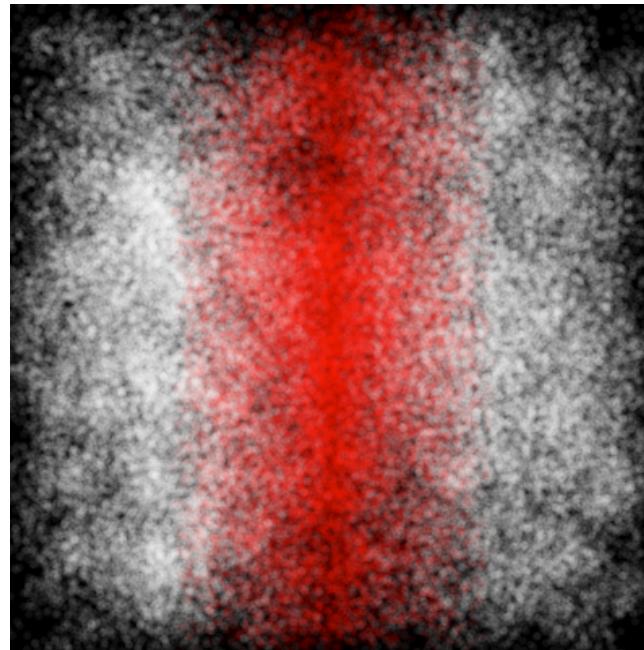
Altere

```
if ( (vx>(cx-100)) && (vx<(cx+100)) ) {  
    fill (255,0,0,5);  
} else {  
    fill (255,5);  
}
```

Para:

```
if ( (vx>( cx-random(100) )) && (vx<( cx+random(100) )) ) {  
    fill (255,0,0,5);  
} else {  
    fill (255,5);  
}
```

Veja o resultado:



Agora os limites ficaram mais diluídos, pois estipulamos valores que podem variar de 0 a 100, a partir do centro, utilizando `random(100)`. A impressão de “pulverização” da estrutura acontece também pelo uso de transparência nas cores com `fill (255,0,0,5)` e `fill (255,5)`. Outros parâmetros como largura e altura da elipse podem ser randomizados e essa é uma alteração que pode gerar efeitos mais orgânico. Vamos criar duas variáveis locais (dentro do bloco `draw()`) que armazenarão os valores randômicos os quais podem ser aplicados na determinação das dimensões da elipse.

Altere:

```
if ((vx>width) || (vy>height) || (vx<0) || (vy<0)) {  
  
    vx=random(width);  
    vy=random(height);  
  
} else {  
    vx+=random(-10,10);  
    vy+=random(-10,10);  
    if ( (vx>(cx-random(100))) && (vx<(cx+random(100))) ) {  
        fill (255,0,0,5);  
    } else {  
        fill (255,5);  
    }  
    ellipse (vx,vy,5,5);  
}
```

Para:

```
if ((vx>width) || (vy>height) || (vx<0) || (vy<0)) {  
    vx=random(width);  
    vy=random(height);  
}  
else {  
    float px=random(-10,10);  
    float py=random (-10,10);  
    vx+=px;  
    vy+=py;  
  
    if ( (vx>(cx-random(150))) && (vx<(cx+random(150))) ) {  
        fill (255,0,0,5);  
    } else {  
        fill (255,5);  
    }  
    ellipse (vx,vy,abs(px*5) ,abs(py*5));  
}
```

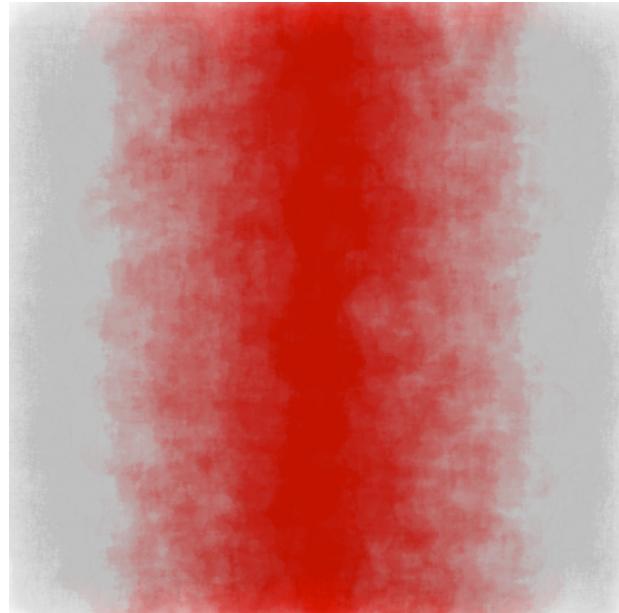
Dentro de *setup()* , altere também:

```
size (400,400,P2D);  
background(255);
```

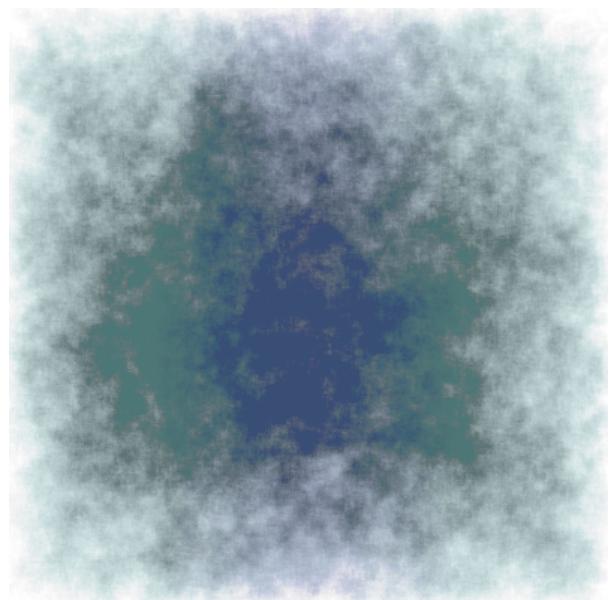
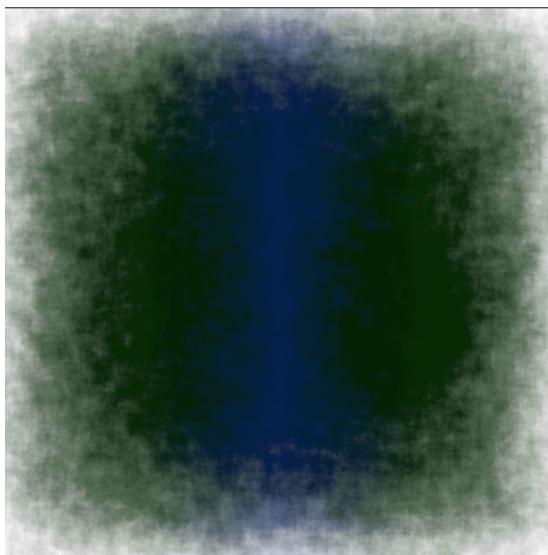
O argumento **P2D** na função **size()** indica um outro tipo de renderização gráfica, fornecendo melhores resultados na mistura de transparências e maior velocidade na execução da saída gráfica.

Antes de utiizar os valores aleatórios para modificar a posição das elipses, armazenamos os mesmos nas variáveis **px** e **py**. Os valores dessas mesmas variáveis serão utilizados para dimensionar as elipses em **ellipse(vx, vy, abs(px\*5), abs(py\*5))**. Perceba que **multiplicamos** os valores por 5, pois queremos ampliar um pouco as dimensões que agora podem valer de 5 a 50. A função **abs ()** garante a conversão de valores negativos em valores absolutos.

Veja o resultado:

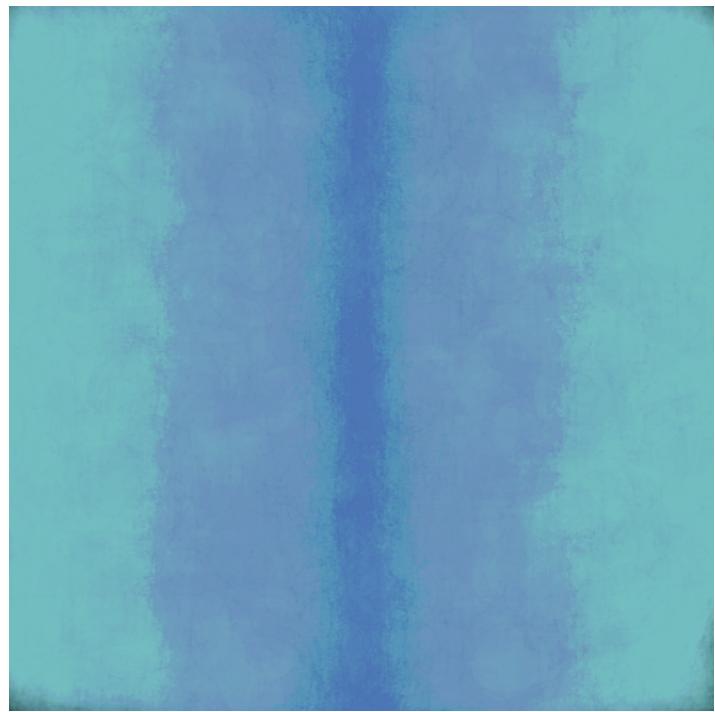


Comparando com o sketch anterior, observamos uma área de intervalo onde as cores se misturam. Podemos obter outros resultados visuais apenas variando os valores das planejados pelo algoritmo como, transparéncia, coloração, dimensões e intervalo entre as posições das elipses:



Segue o código da figura:

```
float vx, vy;  
int cx,cy;  
  
void setup () {  
    size (400,400,P2D);  
    background(255);  
    fill(255,5);  
    noStroke();  
    smooth();  
    vx=random(width);  
    vy=random(height);  
    cx=width/2;  
    cy=height/2;  
    frameRate(120);  
}  
void draw() {  
    if ((vx>width) || (vy>height) ||(vx<0) || (vy<0)) {  
        vx=random(width);  
        vy=random(height);  
    } else {  
        float px=random(-10,10);  
        float py=random(-10,10);  
        vx+=px;  
        vy+=py;  
        if ((vx<(cx+random(200))) && vx>(cx-random(200))) {  
            fill(119,156,232,8);  
        } else {  
            fill (156,237,239,8);  
        }  
        ellipse (vx,vy,abs(py*5),abs(px*5));  
    }  
}
```



## 15.2 Estrutura Condicional (switch...case)

A estrutura *switch...case* funciona da mesma forma que *if...else*, porém são mais convenientes quando temos três ou mais condições ou alternativas. Esta é a estrutura:

```
switch (expressão)
{
    case valor :
        instruções;

    case valor :
        instruções;

    case valor :
        instruções;

    default:
        instruções;
}
```

O resultado da **expressão** é armazenado em **valor** e deve ser avaliado pelas condicionais *case*. Os tipos de dados retornados pela **expressão** devem ser primitivos, isto é, dados que podem ser convertidos para valores **inteiros** como *byte*, *char* e *int*.

No próximo exemplo veremos um código que desenha figuras geométricas diferentes ou muda a cor do *background* de acordo com a tecla pressionada.

As teclas utilizadas são:

**R ou r** - desenha retângulo vermelho;

**L ou I** - desenha linha branca;

**E ou e** - desenha elipse verde;

**Seta para cima (UP)** - apaga background com cor branca;

**Qualquer outra tecla** (default) - apaga background com preto;

A expressão a ser avaliada é o valor (*keyCode*) retornado pela função *keyPressed()*.

Veja o exemplo:

```
void setup() {
  size(400,400);
  background(0);
  noStroke();
  rectMode(CENTER);
}

void draw() {

}

void keyPressed() {
  switch (keyCode)
  {
    case 'R':
    case 'r':
      noStroke();
      fill(255,0,0);
      rect (width/2,height/2,100,100);
      break;

    case 'E':
    case 'e':
      noStroke();
      fill(0,255,0);
      ellipse (width/2,height/2,100,100);
      break;

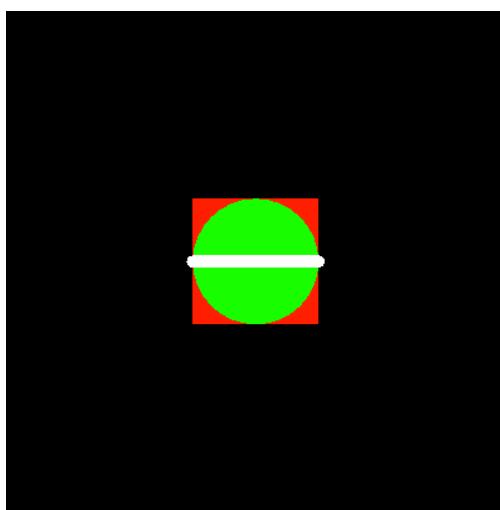
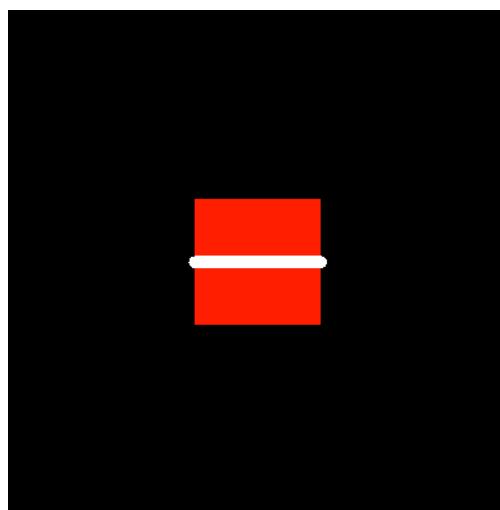
    case 'L':
    case 'l':
      stroke (255);
      strokeWeight(10);
      line((width/2)-50,height/2,(width/2)+50,height/2);
      break;

    case UP:
      background(255);
      break;
  }
}
```

A condicional ***switch()*** é inserida dentro da função `keyPressed()` , pois queremos avaliar, ou mudar o estado do desenho, **apenas** quando pressionamos uma tecla. Desta forma, não é necessário o uso da função `draw()`.

A instrução ***break*** é colocada em cada avaliação, fazendo com que o fluxo de avaliações seja interrompido caso alguma tecla seja identificada.

A última instrução, ***default***, é opcional e determina alguma instrução que deve ser executada caso nenhuma das condições anteriores sejam satisfeitas.



# 16 - Repetições

As estruturas [iterativas](#) são utilizadas para a redução de um código repetitivo. Veja as instruções abaixo:

```
line (0,10,200,10);
line (0,20,200,20);
line (0,30,200,30);
line (0,40,200,40);
line (0,50,200,50);
line (0,60,200,60);
line (0,70,200,70);
line (0,80,200,80);
```

...

A sequencia desenha uma série de linhas horizontais pela repetição da função *line()* e variação apenas das posições verticais (y). Imaginem que para desenharmos um grid de 200 linhas teríamos que repetir essas instruções duzentas vezes! Podemos construir esta repetição de uma forma mais compacta com o uso de algumas estruturas de repetição, como veremos a seguir.

## 16.1 Repetição While

A estrutura *while()* executa uma série de instruções continuamente enquanto (while) uma dada expressão for verdadeira:

```
while (expressão) {
    instruções;
}
```

A expressão utilizada na estrutura deve ser atualizada e avaliada constantemente. Se a condição permanecer *verdadeira* indefinidamente então a repetição continuará. Esse comportamento não é desejável , pois a estrutura *while()* suspende qualquer outro processamento do programa enquanto o looping não for finalizado, o que pode causar travamentos do ambiente.

Por exemplo (**não execute esse código**):

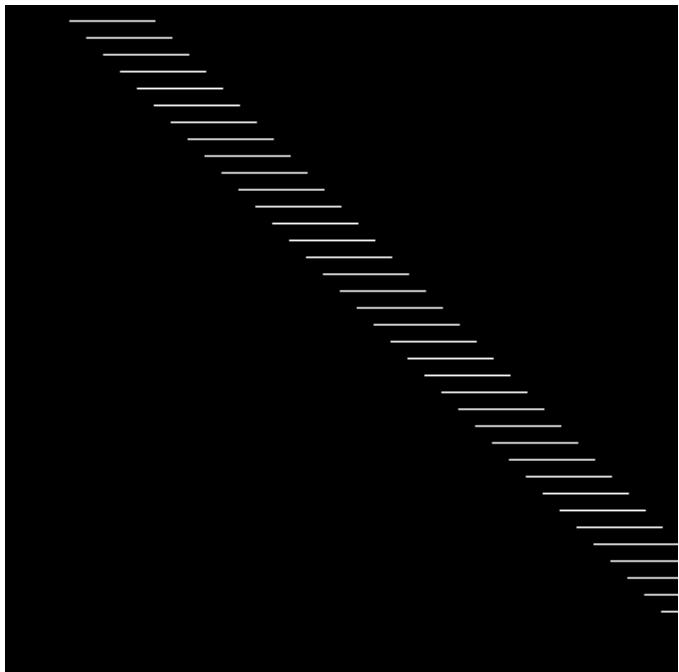
```
int i=0;
while (i<10) {
  line (0,0,200,200);
}
```

Aqui temos um looping infinito isso porque a **expressão** ( $i < 10$ ) sempre será **verdadeira**. Em nenhuma instrução do código o valor da variável **i** (igual a zero) foi alterado, e **i** sempre será menor que 10.

Veja este outro código (agora pode executar):

```
int i=0;
size (400,400);
background(0);
stroke(255);

while (i<400) {
  line (i+30, i, i+80, i);
  i+=10;
}
```



Diferente do primeiro código, neste estamos incrementando a variável **i** a cada looping de **while()**. Logo a expressão ( $i < 400$ ) é **atualizada**, a repetição continuará **enquanto** o valor de **i** for **menor** do que 400.

O valor de **i** está sendo utilizado também para o desenho e o deslocamento das linhas. As coordenadas x e y dos pontos (argumentos) que constituem cada linha mudam de valor a toda vez que a variável **i** é adicionada de 10 na expressão  **$i+=10$** .

Neste sketch percebemos que é possível desenhar uma série de formas sem a escritura de inúmeras instruções. A seguir veremos como realizar essas repetições de uma forma mais controlada e segura.

## 16.2 Laço FOR

A instrução FOR faz parte de uma estrutura **iterativa** que é um dispositivo útil no encurtamento de linhas de código repetitivo. A vantagem desta instrução é a possibilidade de inclusão da inicialização, cláusula condicional e atualização de uma variável na mesma expressão:

```
for (inicialização; teste; atualização) {  
    instruções;  
}
```

Observe o sketch e compare com o que foi utilizado em `while()`, no tópico anterior.

```
size (400,400);  
background(0);  
stroke(255);  
  
for ( int i=0; i<400; i+=10) {  
    line (i+30, i, i+80, i);  
}
```

O resultado visual é exatamente o mesmo do exemplo anterior. Vamos examinar as expressões utilizadas em `for ()`:

**int i =0;** : Aqui declaramos e inicializamos a variável i do tipo inteiro. Esta é uma variável local, isto é, só poderá ser acessada dentro do bloco determinado pelo laço.

**i<400;** : Este é o teste que deve ser avaliado a cada looping do laço. As instruções serão executadas apenas enquanto a condição for verdadeira.

**i+=10;** : Após a realização do teste, a variável é atualizada (aqui é incrementada de 10) e pode ser utilizada (ou não) pelo bloco de instruções.

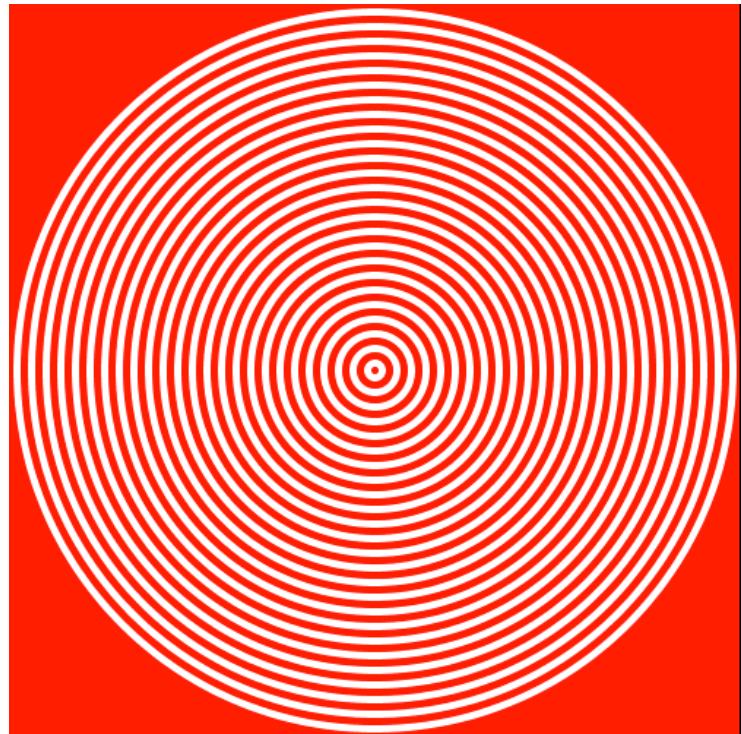
A sequência de execução é clara:

- variável é *inicializada com um valor do tipo adequado*;
- a expressão é *avaliada*;
- se a expressão *for verdadeira*, a variável é *atualizada e o bloco é executado e a avaliação é retomada*;
- se a expressão *for falsa*, o laço é *terminado*.

Outro exemplo:

```
size (500,500);
background(255,0,0);
stroke(255);
strokeWeight(5);
smooth();
noFill();

for (int i=10; i<500;i+=20) {
  ellipse (width/2,height/2,i,i);
}
```

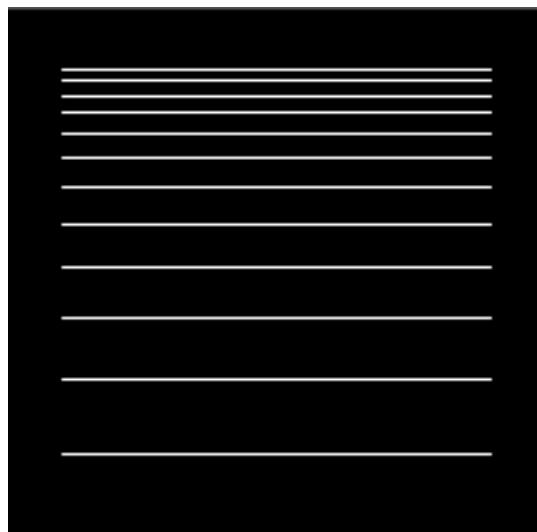


Aqui desenhamos uma série de elipses concéntricas ( $width/2$ ,  $height/2$ ) . Note que a variável **i** é **inicializada** com o valor 10, isto porque o valor de **i** é utilizado no bloco para dimensionar a largura e altura de cada elipse, na instrução `ellipse (width/2, height/2, i , i)`.

Com a estrutura FR podemos gerar vários padrões de imagem, seja pelo uso de diferentes expressões matemáticas na atualização ou inserção de condicionais nos blocos. Veja os exemplos:

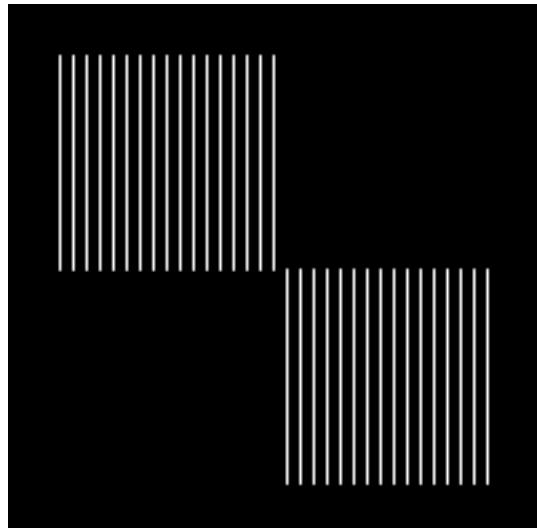
```
size (200,200);
background(0);
stroke(255);
smooth();
noFill();

for (float y=200;y>20;y-=1.2) {
  line (20,y,180,y);
}
```



```
size (200,200);
background(0);
stroke(255);
smooth();
noFill();
```

```
for (int x=20; x<=180;x+=5) {
  if (x<=100) {
    line (x,20,x,100);
  } else {
    line (x,100,x,180);
  }
}
```



## 16.3 Repetições Aninhadas

Existem ocasiões em que precisamos executar repetições de uma outra série de iterações, como é o caso de matrizes bidimensionais. Por exemplo, aqui temos uma repetição linear realizada com o laço FOR:

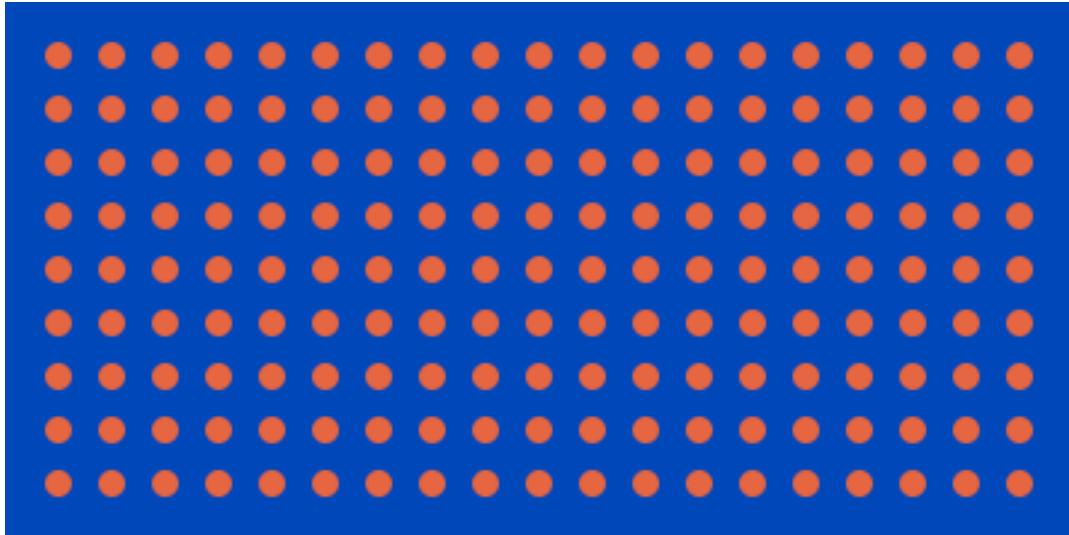


```
size (400,200);
background(22,69,183);
noStroke();
fill (227,99,66);
smooth();

for (int x=20; x<width; x+=20) {
    ellipse (x,20,10,10);           // ellipse (x,y,altura,largura)
}
```

Construimos uma linha de elipses pela repetição e variação (incremento) da coordenada **x** de cada figura. Veja que isto ocorre graças à atualização da variável **x**, que é aplicada na função **ellipse (x,20,10,10)** ( o valor da coordenada y é sempre fixo = 20).

E no caso dessa figura? Como seria o código para construí-la?



Aqui temos uma matriz bidimensional. Como repetimos a linha nove vezes, deveríamos copiar o código da estrutura FOR nove vezes?

Já vimos que a utilidade de uma estrutura de iteração é justamente reduzir a quantidade de código repetitivo. A solução é construção de uma repetição para cada elemento da linha, utilizando um **aninhamento** de laços.

Veja o código da matriz:

```
size (400,200);
background(22,69,183);
noStroke();
fill (227,99,66);
smooth();

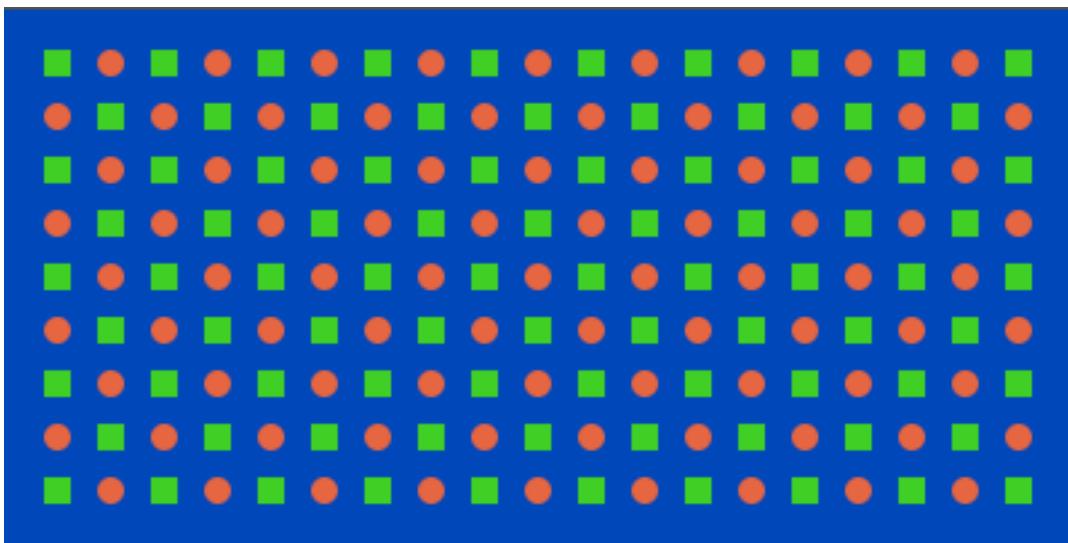
for (int y=20; y<height; y+=20) {
    for (int x=20; x<width; x+=20) {
        ellipse (x,y,10,10);
    }
}
```

Perceba que o laço responsável pelo desenho de uma linha de elipses foi aninhado dentro de outro laço **externo**. Agora, laço **interno** será repetido tantas vezes até que a condição ( $y < 200$ ) seja falsa, isto é, quando as linhas forem repetidas por toda extensão vertical (altura) do sketch.

O desenho de cada elipse é feito pela instrução:

**ellipse (x, y, 10,10);**

Os laços aninhados são ideais para a construção de padrões de repetição visual. Veremos no próximo sketch um exemplo do uso de uma estrutura condicional associada à uma estrutura iterativa. Veja o desenho:



Observe que agora existem elipses intercaladas com quadrados, seguindo a mesma estrutura da matriz anterior. Essa alternância entre as formas é semelhante àquela existente entre os números pares e ímpares como, por exemplo, na sequência 0,1,2,3,4,5,6...

E m cada linha da matriz, as figuras são posicionadas em intervalos regulares devido ao incremento ( $x+=20$ ). Apesar disso, no novo código, utilizaremos uma outra variável denominada **contador** que será incrementada a cada avanço da repetição. Inicialmente a variável será inicializado com zero e , desta forma, ela armazena valores crescentes como na sequência 0,1,2,3,4,5,6...

Em cada repetição, usaremos uma condicional (if) para testar se **contador** armazena um valor par ou ímpar. Com esse dispositivo, podemos imaginar um algoritmo para alternar o desenho de figuras diferentes:

- 1- incrementar a variável **contador**;
- 2- Verificar se **contador** armazena um número par;
- 3- Se sim , ENTÃO desenhe uma elipse4 -SENÃO, se armazena um número ímpar, desenhe um retângulo

Para verificar se o conteúdo de **contador** é par ou ímpar, usamos o operador % (módulo). Este operador retorna o resto de uma divisão qualquer, por exemplo: (10%5); retorna zero, pois o resto da divisão de 10 por 5 é zero.

Em nosso algoritmo , queremos saber se um número é par (regra 2). Para isso criamos a condicional:

```
if (contador%2) == 0 {  
    // desenha uma elipse  
} else {  
    // desenha um quadrado  
}
```

Se o resto da divisão (contador/2) for zero, quer dizer que a variável contador armazena um número par ou, caso contrário (else), ímpar se a expressão (contador%2) for falsa.

Veja o código:

```
size (400,200);  
background(22,69,183);  
noStroke();  
  
rectMode (CENTER);  
smooth();  
int contador=0;  
  
for (int y=20; y<200; y+=20) {  
    for (int x=20;x<400;x+=20) {  
        contador++;  
        if ((contador%2)==0) {  
            fill (227,99,66);  
            ellipse (x,y,10,10);  
        } else {  
            fill (59,209,57);  
            rect (x,y,10,10);  
        }  
    }  
}
```

## 16.3 Repetições e draw()

Os laços podem ser atualizados pelo looping principal do sketch originado pela função `draw()`. Faremos uso deste recurso para alterar o estado visual de uma matriz, seja por interações com o mouse ou animações.

Primeiro, vamos usar as propriedades do mouse (`mouseX`,`mouseY`) para causar uma mudança de estado nas fiduras desenhadas na matriz. Teste o código:

```
void setup() {
size (600,600);
background(22,69,183);
noStroke();
fill (227,99,66);
rectMode (CENTER);
smooth();
}

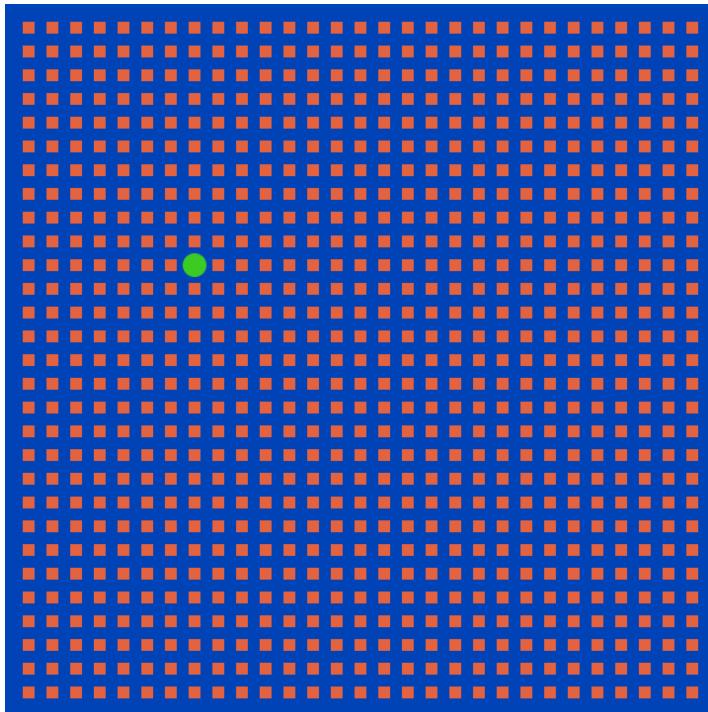
void draw() {
background(22,69,183);
for (int y=20; y<height; y+=20) {
for (int x=20;x<width; x+=20) {

if ( (mouseX<(x+5)) && (mouseX>(x-5)) && (mouseY>(y-5)) && (mouseY<(y+5)) ) {
fill(59,209,57);
ellipse (x,y,40,40);
} else {
fill (227,99,66);
rect (x+d,y+d,2*d,2*d);
}
}
}
}
```

Estruturamos o código setando as propriedades iniciais do sketch dentro de `setup()` e os laços para a montagem da matriz dentro de `draw()`. As repetições são as mesmas utilizadas no sketch da matriz, apenas inserimos uma condicional para a verificação da posição do mouse, ainda com ou auxílio do operador lógico `&&` (AND). As condições :

```
mouseX<(x+5);
mouseX>(x-5);
mouseY>(y-5);
mouseY<(y+5);
```

devem ser verdadeiras simultaneamente para verificar se o mouse está no intervalo espacial de alguma figura que está no ponto (x,y). Caso as cláusulas sejam verdadeiras, será desenhada uma elipse no ponto coincidente com o mouse ou, caso contrário, será desenhado um quadrado. Este teste ocorre constantemente, visto que a construção da matriz (laços) estão dentro do looping draw(). Veja o resultado



Agora vamos alterar a posição dos elementos utilizando a função `dist()` , que retorna o valor da distância entre dois pontos. No caso, calcularemos a **distância** entre o local apontado pelo mouse e o ponto que está sendo calculado pelos laços, veja:

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

A função `dist()` retorna um valor do tipo float e necessita de quatro argumentos correspondentes às coordenadas dos pontos dos quais a distância será calculada:

`float dm=(dist (x,y,mouseX,mouseY))*0.02;`

A variável `dm` receberá o valor da distância que ainda está sendo multiplicado por 0.02 para uma atenuação. Veja como essa variável é utilizada, na modificação do código anterior:

```

void setup() {
size (600,600);
background(22,69,183);
noStroke();
fill (227,99,66);
rectMode (CENTER);
smooth();
}

void draw() {
background(22,69,183);
for (int y=20; y<height; y+=20) {
for (int x=20;x<width; x+=20) {

if ( (mouseX<(x+5)) && (mouseX>(x-5)) && (mouseY>(y-5)) && (mouseY<(y+5)) ) {
fill(59,209,57);
ellipse (x,y,40,40);
} else {
float dm=(dist (x,y,mouseX,mouseY))*0.02;
fill (227,99,66);
rect (x+dm,y+dm,10,10);
}
}
}
}
}

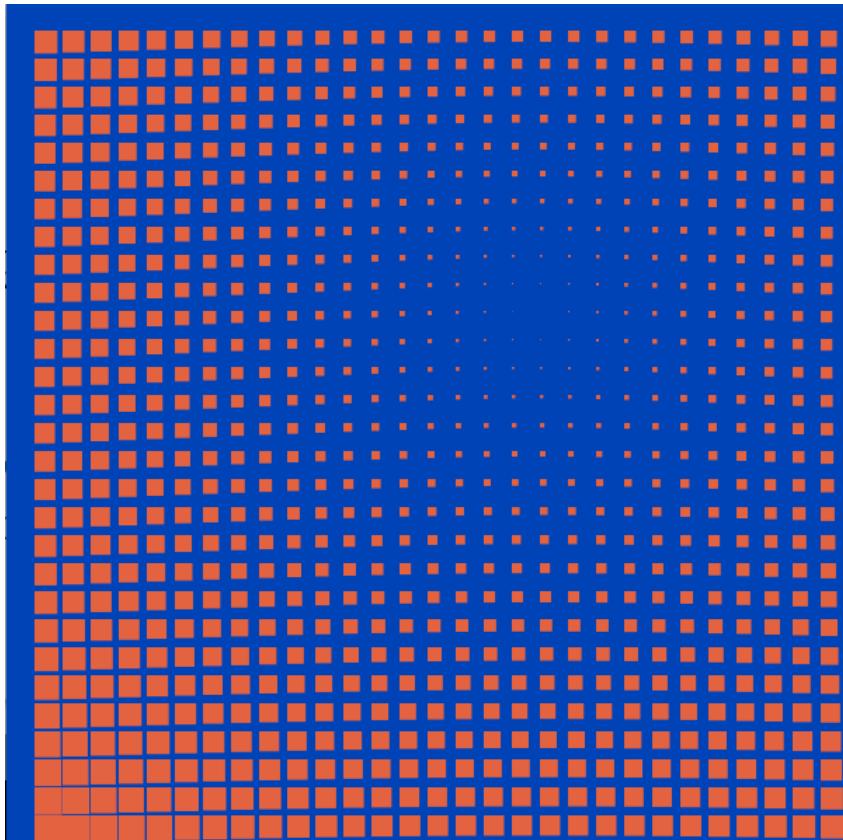
```

Na instrução `rect (x+dm,y+dm,10,10)`, provocamos um deslocamento dos elementos da matriz em função da distância calculada (`dm`) entre o mouse ao ponto (x,y) do elemento. Quanto mais distante o ponto estiver do mouse, menos o elemento correspondente será deslocado.

Podemos modificar as **dimensões** dos elementos proporcionalmente à distância , apenas modificando a linha para:

```
rect (x+dm,y+dm,2*dm, 2*dm);
```

Veja o resultado da interação :



## 16.3 Cor e repetição

A função `colorMode(modo)` altera o modo como o Processing interpreta informações sobre coloração. O parâmetro **modo** pode assumir o código **RGB** ou **HSB**.

O modo HSB (*hue, saturation, brightness*) corresponde ao matiz de uma cor como amarelo, vermelho, verde, sua diluição ou saturação e ao brilho.

Para definir o modo HSB, podemos usar a função:

```
colorMode (HSB,360,100,100);
```

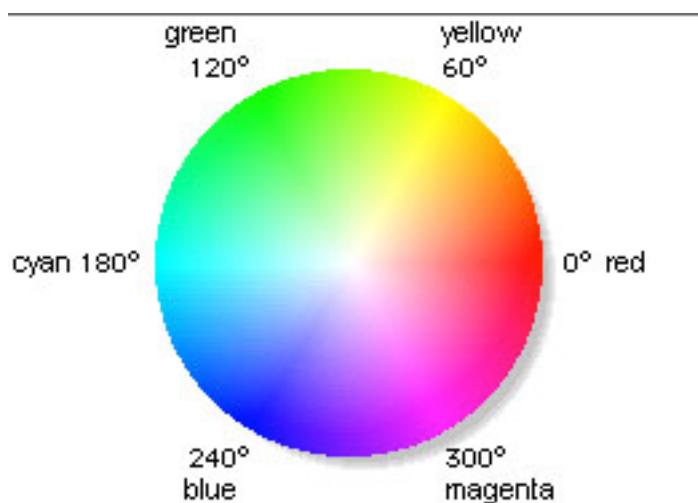
Além de definir o padrão HSB, os parâmetros seguintes equivale a:

360 - indica que usaremos o círculo cromático para escolha da cor (HUE)

100 - indica que usaremos a saturação na escala de porcentagens.

100 - indica que indicaremos o brilho na escala de porcentagem.

A figura abaixo mostra o círculo de cores indicando a posição angular de cada matiz:



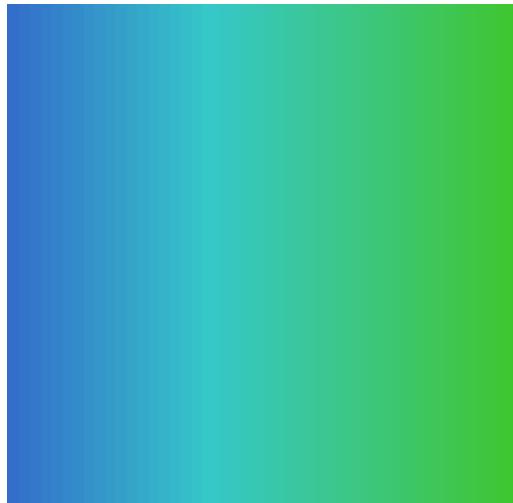
Vamos utilizar o modo HSB para desenhar um quadrado amarelo. O amarelo está posicionado n em 60 graus, então:

```
size (500,500);
background(0);
colorMode(HSB,360,100,100);
fill (60,100,100);
rect (100,100,50,50);
```

Na instrução `fill(60,100,100)` selecionamos cor amarela indicando 60 para a posição e 100% de saturação e brilho.

Podemos usar um laço de repetição para criar um gradient de cores , variando o valor do parâmetro referente à posição da cor (HUE). Neste sketch desenhos uma sequência de linhas que assumem cores graduais que iniciam azul (220) e terminam na área da cor verde (120):

```
size (500,500);
background(0);
colorMode(HSB,360,100,100);
for (int i=0;i<500;i++) {
  float novacor=220-(i*0.2);
  stroke (novacor,70,80);
  line (i,0,i,500);
}
```



A formação do gradiente ocorre pela justaposição das linhas verticais. A variação do matiz é gerada pela expressão  $220-(i*0.2)$  que utiliza a atualização da variável  $i$  como fator que multiplica a taxa (0.2) de aumento.

# 17 - Lista e Matriz

Na área de computação é comum o armazenamento de uma série de informações para a criação de banco de dados compostos por listas ou conjuntos de valores indexados. O processamento de imagens e gráficos computacionais exigem armazenamento e cálculos baseados em estruturas informacionais especiais como é o caso da matriz. Assim como em outras linguagens de programação, o Processing conta com o elemento **Array** para efetuar esse tipo de armazenamento. Veremos adiante os possíveis usos desta estrutura.

## 16.3 Array

Em termos gerais, um array é uma estrutura capaz de armazenar um conjunto de valores do mesmo tipo. No exemplo:

```
int [] valores={10,40,50,120,1}
```

a variável **valores** é um array que armazena números inteiros. Cada elemento do array é identificado por uma série de índices numéricos que iniciam em zero. O primeiro elemento está na posição [0], o segundo na posição [1] e o último na posição [n-1] onde n corresponde ao número total de elementos (comprimento do array). Analisando o exemplo, temos:

```
valores[0] =10;  
valores[1]=40;  
valores[2]=50;  
valores[3]=120;  
valores[4]=1;
```

O comprimento (*length*) é 5, pois armazena cinco números inteiros. Os índices ficam entre colchetes [ ] e indicam a posição do valor armazenado dentro do array.

## 16.4 Declarando um array

Um array deve ser declarado como uma variável qualquer e [ ] deve ser usado logo após a indicação do tipo:

```
float [] x;
```

Aqui declaramos o array x que pode armazenar elementos do tipo float.

Após a declaração, devemos criar a estrutura com a instrução ***new***, indicando a quantidade de elementos que a estrutura pode armazenar:

```
float [] x;  
x= new float [5];
```

Agora sabemos que o array **x** pode armazenar 5 números do tipo float. É possível declarar e criar um array na mesma linha de instruções:

```
float [] x= new float[5];
```

Após ser declarado e criado, o array pode receber valores a qualquer instante do código. Podemos inserir os elementos um a um:

```
float [] x= new float[5];  
x[0] =1.2;  
x[1]=2.0;  
x[2]=35.7;  
x[3]=1.9999;  
x[4]=0.34;
```

Ou em apenas uma expressão, declarando e inicializando o array com os valores:

```
float [] x={1.2, 2.0, 35.7, 1.9999, 0.34};
```

Note que se ao tentarmos acessar um elemento que está fora dos limites do array ocorrerá um erro. Por exemplo:

```
println (x[5]);
```

Esta instrução causa o erro *ArrayIndexOutOfBoundsException*, pois estamos acessando o elemento com índice **[5]** e o array possui 5 elementos apenas (lembre-se que o último elemento do array corresponde ao índice **[comprimento-1]**, no exemplo o último elemento é indentificado por [4]).

Para verificarmos o comprimento de um array basta acessar a função ***length***:

```
println (x.length); // imprime 5, que é o tamanho do array x
```

## 16.5 Acessando elementos do array

Para inserir um elemento numa determinada posição do array basta indicar o valor e índice entre [ ]. Por exemplo:

```
float sorteio[ ]=new float[50];
sorteio[33]=random (500);
```

No exemplo, criamos o array sorteio com a dimensão total de 50 elementos do tipo float. Logo após, inserimos um valor randômico na posição [33] do array. Mas se queremos inserir números aleatórios em todas as posições do array, deveríamos escrever 50 linhas de instruções, uma para cada elemento? Como visto, a **repetição** por laço pode resolver o problema, reduzindo a quantidade de código repetitivo. Vamos utilizar a essa estrutura de repetição para inserir valores em todas as posições do array:

```
float [ ] sorteio;
sorteio=new float[50];

for (int i=0;i<50;i++) {
    sorteio[i]=random(500);
}
```

O array **sorteio** é declarado e criado com a capacidade para 50 elementos. Utilizamos um laço para inserir valores em cada posição do array. Em **for (int i=0; i<30;i++)** a variável **i** é inicializada com zero, pois queremos acesso desde a primeira posição do array que é [0]. A repetição é realizada enquanto **i** for menor que o número de elementos do array igual a 50. Utilizamos o valor atualizado de **i** para acessar o índice de cada elemento:

```
sorteio[ i ]=random(500);
```

Observe que **i** não pode ser igual 50, pois os índices do array vão de [0] a [49]. De outra forma, podemos usar **length** no lugar do valor 50:

```
for (int i=0;i<sorteio.length;i++);
```

Para realizarmos uma **leitura** ou recuperação de todos os elementos do array, utilizamos o mesmo procedimento, logo após a inserção dos valores, acessando seus índices com a variável **i**:

```
float [ ] sorteio;
sorteio=new float[50];

for (int i=0;i<50;i++) {      // insere valores nas posições do array
    sorteio[i]=random(500);
}

for (int i=0;i<50;i++) {      // imprime todos os elementos do array
    println( sorteio[i]);
}
```

O uso da repetição nos permite resgatar qualquer série de posições armazenadas no array. Para isso, basta determinar a posição inicial de leitura e o limite ou intervalo a ser resgatado. No próximo exemplo, vamos ler o intervalo de valores que se inicia na posição 25 do array e termina na posição 40:

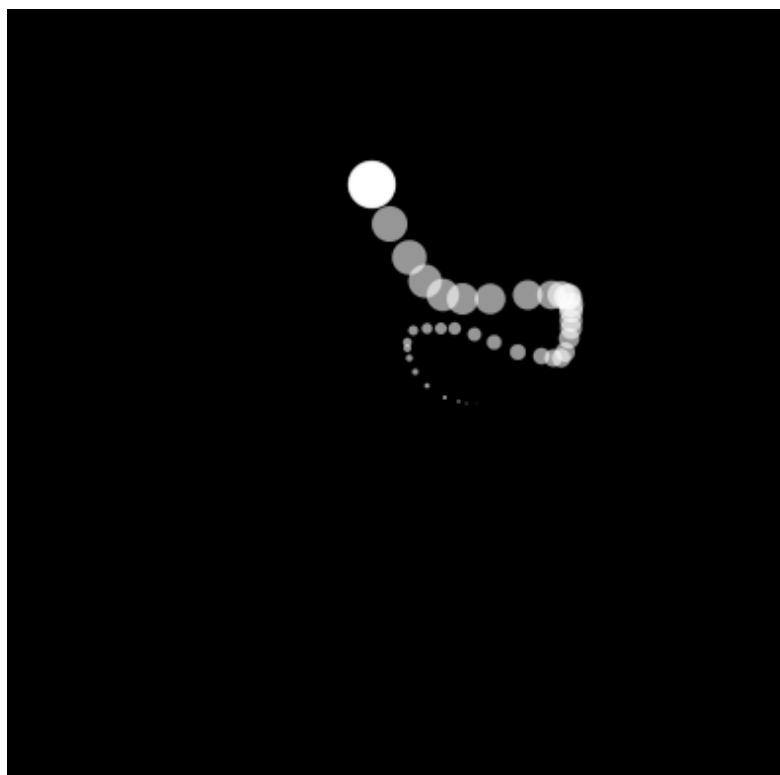
```
for (int i=25; i<=40; i++) {      // imprime o intervalo entre o elemento [25] e [40]
    println( sorteio[i]);
}
```

É possível ainda, ler os valores numa ordem inversa, ou seja, partindo do último elemento e retornando até o primeiro:

```
or (int i=49; i>=0; i--) {      // imprime o intervalo entre o último e primeiro elemento
    println( sorteio[i]);
}
```

A variável **i** recebe inicialmente a última posição do array [49] e regide a contagem pela atualização em **i --** (decremento). A condição de repetição (**i>=0**) determina que o índice seja um número **menor ou igual a zero**, pois o primeiro elemento está no índice [0].

Vamos usar este recurso para armazenar as posições do mouse em um array. Nesse sketch desejamos que uma série de elipses forme um rastro construído pela movimentação do mouse. Cada elipse é desenhada numa posição ocupada pelo mouse num determinado instante. Portanto, devemos construir uma lista de armazenagem das posições.



*Uma limitação da estrutura array impede o acréscimo de novos elementos durante a execução do programa. Como foi visto, devemos definir uma dimensão fixa logo na criação do array. Vamos criar dois arrays que serão responsáveis pelo armazenamento das posições (x e y) do mouse:*

```
int num=50;  
  
int [] x=new int[num];  
int [] y=new int [num];
```

A variável **num** torna o código mais flexível, pois basta mudar o seu valor para alterar as dimensões dos arrays **x** e **y** que armazenarão as posições espaciais do mouse. No exemplo, estamos reservando 50 posições. Como queremos uma animação (rastro), vamos estruturar o código com as funções **draw()** e **setup()**:

```
int num=50;  
  
int [] x=new int[num];  
int [] y=new int [num];  
  
void setup() {  
    size (400,400);  
    noStroke();  
    smooth();  
    fill (255,150);  
}  
}
```

Na função **draw()** faremos a leitura das posições do mouse, pois precisamos de uma atualização constante. Os valores serão armazenados nos arrays **x** e **y**, e posteriormente recuperados para o desenho das elipses.

Os valores dados por **mouseX** e **mouseY** devem ser acrescentados ao array de forma serial. A posição do mouse mais recente será armazenada sempre na última posição do array. Como uma espécie de empilhamento, em cada repetição do laço, cada posição do array receberá o valor armazenado no próximo elemento (**x[i]=x[i+1]**) garantindo o registro sequencial:

```
void draw() {  
background(0);  
for (int i=0;i<num-1;i++) {  
    x[i]=x[i+1];           // valores armazenados são deslocados  
    y[i]=y[i+1];  
}  
x[num-1]=mouseX;      // valores são armazenados sempre na última posição do array  
y[num-1]=mouseY;  
}
```

Ainda dentro do laço, temos o código que realiza a leitura dos valores armazenados:

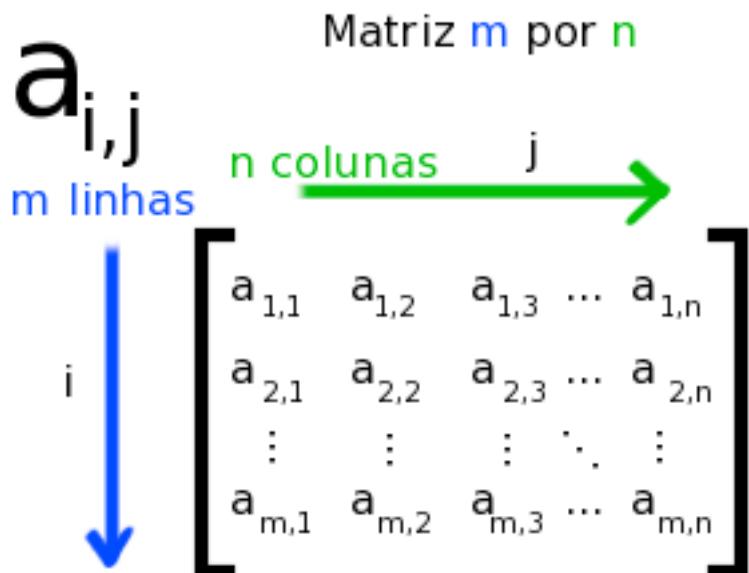
```
for (int i=num-1;i>=0;i--) {  
    ellipse(x[i],y[i],i/2.0,i/2.0);  
}
```

Numa ordem inversa, a leitura do array é feita a partir da última posição ( $i=num-1$ ) até a primeira ( $i\leq 0$ ). Para cada elemento do array (posições do mouse armazenadas) temos o desenho de uma elipse cuja posição é igual a uma coordenada `mouseX`, `mouseY`. As dimensões de cada elipse variam de acordo com o índice da posição do array, isto é, quanto mais recente o valor (índice maior) maior será o tamanho da elipse `ellipse(x[i],y[i],i/2.0,i/2.0)`. Este é o código final:

```
int num=50;  
  
int[] x=new int[num];  
int[] y=new int[num];  
  
void setup() {  
    size (400,400);  
    noStroke();  
    smooth();  
    fill(255,150);  
    frameRate(20);  
}  
  
void draw() {  
    background(0);  
    for (int i=0;i<num-1;i++) {  
        x[i]=x[i+1];  
        y[i]=y[i+1];  
    }  
    x[num-1]=mouseX;  
    y[num-1]=mouseY;  
    for (int i=num-1;i>=0;i--) {  
        ellipse(x[i],y[i],i/2.0,i/2.0);  
    }  
}
```

## 16.6 Arrays bidimensionais (matriz)

Em matemática, uma matriz ( $m \times n$ ) é uma tabela de **m** linhas e **n** colunas de símbolos:



Em termos computacionais, uma matriz pode ser construída pelo armazenamento de uma série de arrays, por exemplo:

Matriz = 1 0 0 0 0  
          0 1 0 0 0  
          0 0 1 0 0  
          0 0 0 1 0  
          0 0 0 0 1  
          0 0 0 0 1

Essa é uma matriz quadrada contendo 6 linhas x 6 colunas. Podemos traduzir essa matriz numa série de arrays que serão armazenadas em um único array:

```
array M= { {1, 0, 0 ,0 ,0, 0},  
          {0, 1, 0 ,0 ,0, 0},  
          {0, 0, 1 ,0 ,0, 0},  
          {0, 0, 0 ,1 ,0, 0},  
          {0, 0, 0 ,0 ,1, 0},  
          {0, 0, 0 ,0 ,0, 1}  
        }
```

O array M contém 6 arrays que, por sua vez, armazenam 6 elementos cada um.

Utilizando o conhecimento sobre arrays, vamos declarar o array matriz:

```
int [ ] [ ] matriz= { {1,0,0,0,0,0}, {0,1,0,0,0,0}, {0,0,1,0,0,0}, {0,0,0,1,0,0}, {0,0,0,0,1,0},  
{0,0,0,0,0,1} };
```

Note que cada elemento (posição) do **array matriz** armazena um outro array (unidimensional). Para acessar um elemento qualquer da matriz , basta indicar os índices do elemento (**linha e coluna**). Por exemplo:

```
int elemento= matriz [2][2];  
println(elemento); // imprime valor da variável elemento (1)
```

Neste código estamos acessando o valor armazenado na posição **[2][2]** , que equivale ao elemento posicionado na **terceira** coluna da **terceira** linha (lembre-se que os índices do array começam com zero). Dependendo da dimensão da matriz, o código pode se tornar repetitivo. Vamos usar repetições por laço para inicializar a matriz:

```
int [ ] [ ] matriz=new int[6][6];  
  
for (int i=0;i<6;i++) {  
    for (int j=0;j<6;j++) {  
        if (i==j) {  
            matriz[i][j]=1;  
        } else {  
            matriz[i][j]=0;  
        }  
    }  
}
```

O primeiro passo foi declarar o array:

```
int [ ] [ ] matriz=new int[6][6];
```

Aqui declaramos o array bidimensional **matriz** cuja dimensão é 6x6. Logo após criamos laços aninhados para inicializar as posições com valores (0 e 1):

```
for (int i=0; i<6; i++) {  
    for (int j=0; j<6; j++) {  
  
    }  
}
```

O primeiro laço é referente ao número de linhas da matriz e para cada elemento (*i*) haverá um laço que varia em seis posições equivalentes às colunas (*j* vai de 0 a 5). Em resumo, para acessar um elemento da matriz devemos:

*matriz[ i ][ j ] = valor;*      *i=linha j=coluna*

onde o índice dado por *i* equivale à linha e o índice *j* (laço interno) equivale à coluna da matriz. Perceba que na matriz:

1	0	0	0	0	0	[0][0]=1
0	1	0	0	0	0	[1][1]=1
0	0	1	0	0	0	[2][2]=1
0	0	0	1	0	0	[3][3]=1
0	0	0	0	1	0	[4][4]=1
0	0	0	0	0	1	[5][5]=1

Os índices com valores idênticos (*i=j*) recebem o valor 1 e, por essa razão, criamos uma condicional para verificar qual valor deve ser inserido, dependendo da relação entre *i* e *j* (operador ==):

```
if (i==j) {  
    matriz[ i ][ j ]=1;      // índices iguais, posição recebe 1  
} else {  
    matriz[ i ][ j ]=0;      // senão (índices diferentes)  
}
```

Após a criação da matriz, podemos utilizar a mesma estrutura de repetições para imprimir os valores armazenados (acesso ao array):

```
for (int i=0;i<6;i++) {  
    for (int j=0;j<6;j++) {  
        if (j==0) {  
            println();  
        }  
        print (matriz[i][j]+ " ");  
    }  
}
```

Imprimos o valor da posição [ i ] [ j ] em `print (matriz[i][j]+ " ")`. O comando `print` imprime os valores na mesma linha. Utilizamos uma condicional (`j==0`) para verificar se o código está iniciando mais um laço interno, equivalente à varredura das colunas e , caso isso for verdadeiro, chamamos a função `println()` para pular uma linha de impressão. O resultado é:

```
1 0 0 0 0  
0 1 0 0 0  
0 0 1 0 0  
0 0 0 1 0  
0 0 0 0 1  
0 0 0 0 1
```

Para um código mais flexível, caso tivermos a necessidade de construir uma matriz de qualquer dimensão, vamos usar duas variáveis para determinar as dimensões (linha x coluna) da matriz:

```
int l; // determina linhas  
int c; // determina colunas
```

```
int I =10;  
int c=2;  
  
int [ ] [ ] matriz=new int[I][c];
```

```
for (int i=0;i<I;i++) {  
    for (int j=0;j<c;j++) {  
        if (i==j) {  
            matriz[i][j]=1;  
        } else {  
            matriz[i][j]=0;  
        }  
    }  
}
```

```
for (int i=0;i<I;i++) {  
    for (int j=0;j<c;j++) {  
        if (j==0) {  
            println();  
        }  
        print (matriz[i][j]+ " ");  
    }  
}
```

No exemplo, costruimos uma matriz (10x2) e o resultado da impressão é:

```
1 0  
0 1  
0 0  
0 0  
0 0  
0 0  
0 0  
0 0  
0 0  
0 0
```

Note que apenas as duas primeiras linhas recebem o valor 1. Isto ocorre porque estas são as duas únicas posições que satisfazem a condicional ( $i==j$ ). São elas:

```
matriz[0][0]  
matriz[1][1]
```

## 16.7 Criando uma matriz

No próximo sketch, utilizaremos uma matriz bidimensional para armazenar valores correspondentes às posições e dimensões de várias elipses. Para ser desenhada, cada elipse necessita de quatro atributos: coordenada x, coordenada y, largura e altura. Podemos pensar que um array é o suficiente para armazenar esses valores:

```
{x,y,largura,altura}
```

Cada elemento deste array unidimensional equivale a um parâmetro específico:

[0] = x;

[1]=y;

[2]=largura;

[3]=altura;

Mas se vamos desenhar várias elipses então precisamos de um array para cada elipse:

```
{x,y,largura,altura}
```

...

...

Está parecendo que este é um trabalho para um array bidimensional. Cada array (conjunto) de atributos deve corresponder a uma linha do array bidimensional.

Determinando o número de elipses a serem desenhada saberemos o número de linhas da matriz e o número de colunas é o mesmo que a quantidade de atributos.

```
int numElipses=20;  
float [ ] [ ] matrizElipses= new float [numElipses] [4];
```

Declaramos o número de elipses que desejamos desenhar e a **dimensão** da matriz que guardará seus atributos é **numElipses x 4**. Para iniciar o o código, vamos estipular que as posições (randômicas) das elipses devem ser limitadas pelo sketch. As dimensões (altura e largura) podem variar entre 30 e 70, por exemplo.

Veja o sketch para a contrução da matriz:

```
int numElipses;  
float [ ] [ ] matrizElipses;  
  
size (500,500);  
background (255);  
noStroke();  
smooth();  
fill (80,169,227,100);  
  
numElipses=30;  
matrizElipses=new float [numElipses][4];  
  
for (int i=0;i<numElipses;i++) {  
    matrizElipses[i][0]=random(width);  
    matrizElipses[i][1]=random(height);  
    matrizElipses[i][2]=random(30,70);  
    matrizElipses[i][3]=matrizElipses[i][2];  
}  
}
```

Primeiro declaramos a variável **numElipses** e o array bidimensional **matrizElipses**. Logo após, temos a determinação dos atributos de desenho e a inicialização das variáveis:

```
numElipses=30; // determinação do número de elipses  
matrizElipses=new float [numElipses][4]; // array com dimensões 30 x 4
```

O laço vai construir o array onde o número de linhas varia com a quantidade de elipses (30). Para cada repetição alimentamos as colunas (4 elementos) com os valores randômicos estipulados:

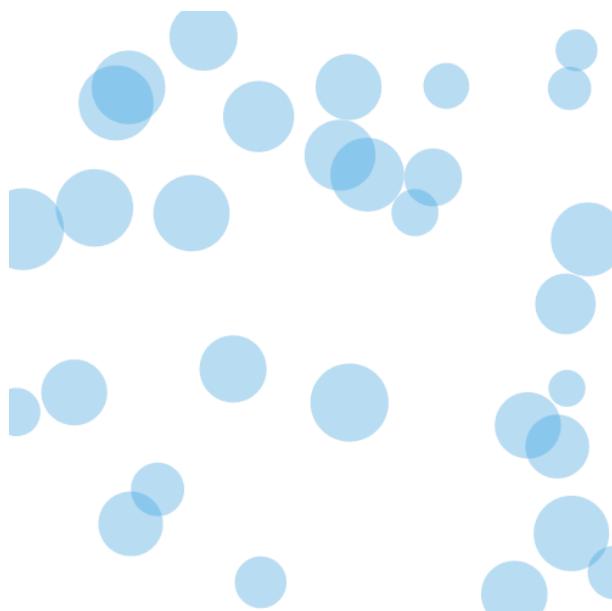
```
matrizElipses[i][0]=random(width);           // posição x  
matrizElipses[i][1]=random(height);          // posição y  
matrizElipses[i][2]=random(30,70);            // largura  
matrizElipses[i][3]=matrizElipses[i][2];       // altura = largura
```

Para cada elemento **i** (linha), teremos 4 elementos (colunas) correspondentes aos atributos da elipse. As posições [0] [1] [2] [3] armazenam esses valores!

As repetições deste laço servem para o simples armazenamento inicial de valores aleatórios. Como no exemplo anterior, construiremos um laço para acessar esses valores e desenhar as elipses:

```
for (int i=0;i<numElipses;i++) {  
    ellipse(matrizElipses[i][0], matrizElipses[i][1], matrizElipses[i][2], matrizElipses[i][3]);  
}
```

Novamente, este outro laço varia de acordo com o número de elipses estipulado pela variável **numElipses** (linhas do array). Os argumentos para a função **ellipse()** são resgatados no array construído no passo anterior.



Vamos organizar o código criando duas funções que irão agrupar os códigos para construir o array e exibir as elipses:

```
void criaArray (int nel) {  
    matrizElipses=new float [nel][4];  
    for (int i=0;i<nel;i++) {  
        matrizElipses[i][0]=random(width);  
        matrizElipses[i][1]=random(height);  
        matrizElipses[i][2]=random(30,70);  
        matrizElipses[i][3]=matrizElipses[i][2];  
    }  
}
```

A função **criaArray()** tem um parâmetro (**int nel**) que receberá como argumento o número de elipses que servirá para a determinação do comprimento do array e , consequentemente, o número de vezes que o laço será repetido. Vale observar que **nel** é uma variável do tipo inteiro e válida apenas para o bloco da função (local).

A segunda função será responsável apenas para o desenho das elipses:

```
void desenhaElipses (int nel) {  
    for (int i=0;i<nel;i++) {  
        ellipse(matrizElipses[i][0], matrizElipses[i][1], matrizElipses[i][2], matrizElipses[i][3]);  
    }  
}
```

Aqui criamos novamente uma variável chamada **nel** para a realização das repetições. Da mesma maneira que a anterior, precisamos enviar um argumento inteiro para referenciar a quantidade de elipses cujos atributos foram armazenados na construção do array.

Para a finalização do código, as chamadas destas funções e envio dos argumentos serão inseridas nas funções **setup()** e **draw()**.

```

int numElipses=30;           // inicializamos a variável numElipses
float [ ] [ ] matrizElipses;

void setup() {
    size (500,500);
    background (255);
    noStroke();
    smooth();
    fill (80,169,227,100);
    criaArray(numElipses); // chama a função para a construção do array
}

void draw() {
    background(255);
    desenhaElipses(numElipses); // chama a função para o desenho das elipses
}

void criaArray (int nel) {
    matrizElipses=new float [nel][4];
    for (int i=0;i<nel;i++) {
        matrizElipses[i][0]=random(width);
        matrizElipses[i][1]=random(height);
        matrizElipses[i][2]=random(30,70);
        matrizElipses[i][3]=matrizElipses[i][2];
    }
}

void desenhaElipses (int nel) {
    for (int i=0;i<nel;i++) {
        ellipse(matrizElipses[i][0], matrizElipses[i][1], matrizElipses[i][2], matrizElipses[i][3]);
    }
}

```

A função ***criaArray()*** é chamada apenas uma única vez, pois está dentro de ***setup()***. Já o desenho das elipses é realizado continuamente pela chamada de ***desenhaElipses()*** dentro de ***draw()***. Podemos implementar um evento de mouse para recriar o array e sortear outras posições e dimensões. Acrescente ao código a função:

```
void mousePressed() {  
    criaArray(numElipses);  
}
```

E ainda podemos gerar um número aleatório de elipses pela randomização da variável ***numElipses***. Modifique a função anterior para:

```
void mousePressed() {  
    numElipses=round(random(10,80));  
    criaArray(numElipses);  
}
```

A cada vez que o mouse for pressionado, sorteamos um número entre 10 e 80 que , por sua vez, servirá como **argumento** para a chamada da função em ***criaArray(numElipses)***. A partir deste evento, um novo array de atributos será criado e as elipses continuarão a serem desenhadas pela função ***desenhaElipses(numElipses)*** , dentro de ***draw()***.

## 16.8 Atualizando uma matriz

No exemplo anterior, vimos como uma matriz pode ser útil para o armazenamento e indexação de um conjunto de valores. O sketch poderia ser construído de outra forma, sem o uso de uma matriz, apenas pela geração de valores aleatórios dos atributos e imediato desenho de elipses dentro de uma repetição ***draw()***.

Agora veremos a real utilidade da matriz que é o estabelecimento de uma memória de estoque capaz de ser acessada atualizada para a mudança de estados , como acontece nas animações de vários elementos conectados entre si. Por exemplo, vamos atualizar as posições das elipses tomando como base os atributos armazenados na matriz.

Para a animação,vamos aplicar uma técnica semelhante àquela utilizada anteriormente quando movimentamos as elipses em sentidos aleatórios:

```
vx+=random(-10,10);  
vy+=random(-10,10);
```

Os valores da atriz, adicionando os valores randomicos a cada repetição do laço.  
Modifique a função draw() para:

```
void desenhaElipses (int nel) {  
  
    for (int i=0;i<nel;i++) {  
  
        matrizElipses[i][0]+=random(-10,10);  
  
        matrizElipses[i][1]+=random(-10,10);  
  
        ellipse(matrizElipses[i][0],matrizElipses[i][1],matrizElipses[i][2],matrizElipses[i][3]);  
  
    }  
  
}
```

Os elementos matrizElipses[i][0] e matrizElipses[i][1] correspondem às coordenadas x e y da elipse e são modificadas individualmente pela soma de valores aleatórios.

Exemplo com noise():

```
float xnoise,ynoise;  
  
xnoise=random(5);  
ynoise=random(5);  
  
void desenhaElipses (int nel) {  
  
    for (int i=0;i<nel;i++) {  
  
        matrizElipses[i][0]+=random(-10,10);  
        matrizElipses[i][1]+=random(-10,10);  
        //matrizElipses[i][0]+=noise(xnoise)*random(-10,10);  
        //matrizElipses[i][1]+=noise(ynoise)*random(-10,10);  
        ellipse(matrizElipses[i][0],matrizElipses[i][1],matrizElipses[i][2],matrizElipses[i][3]);  
  
    }  
  
    xnoise+=0.1;  
    ynoise+=0.1;  
}
```

# Módulo V

# 17 - Imagens

O Processing possibilita o carregamento e manipulação de imagens digitais. Com o uso de um tipo especial de variável (**PImage**) , podemos armazenar uma série de informações contidas nos arquivos dos tipos JPEG, PNG ou GIF, que devem ser disposos na pasta “data”, que por sua vez está localizada dentro da pasta reservada para o sketch atual.

## 17.1 Importando Imagens

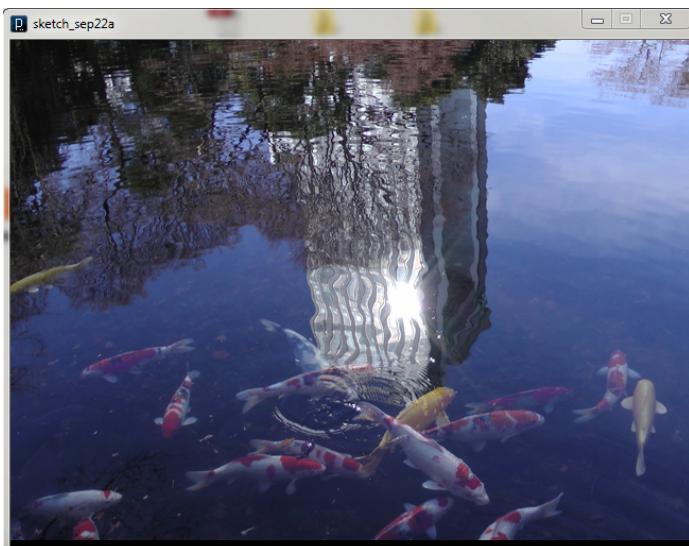
Para o **import** da imagem, basta arrastar o arquivo escolhido (jpg, png ou gif) para a área a janela de edição de texto no Processing. Desta forma, o sistema automaticamente incluirá a imagem no sketch.

Um outro método é selecionar a opção “**Add File**” no menu do Processing, navegar para a localização da imagem e selecioná-la clicando em “Open”. Posteriormente, verifique se a imagem foi corretamente adicionada ao sketch pressionando CTRL+K e certifique-se que o arquivo está salvo na pasta “data”. Uma vez armazenada, a imagem pode ser carregada numa variável. O código seguinte carrega a imagem contida no arquivo figura.png , exibindo-a na coordenada (0,0) do display.

```
PImage img;  
img=loadImage("figura.png");  
image (img,0,0);
```

Verificamos na primeira linha a declaração da variável img do tipo **PImage** . Esta variável foi criada para guardar uma imagem carregada pela função **loadImage()** . Note que o nome do arquivo escolhido deve estar entre aspas: “figura.png”. Finalmente a imagem é exibida na tela pela função **image()** . Os parâmetros usados por essa função são o nome da variável que armazena a imagem (img) e a posição (coordenadas) na tela.

Uma vez carregada a imagem também pode ser redimensionada pela função **image()** . Por exemplo, essa imagem capturada foi digitalizada e exportada no formato JPEG e redimensionada para 640 pixels de largura e 480 de altura.

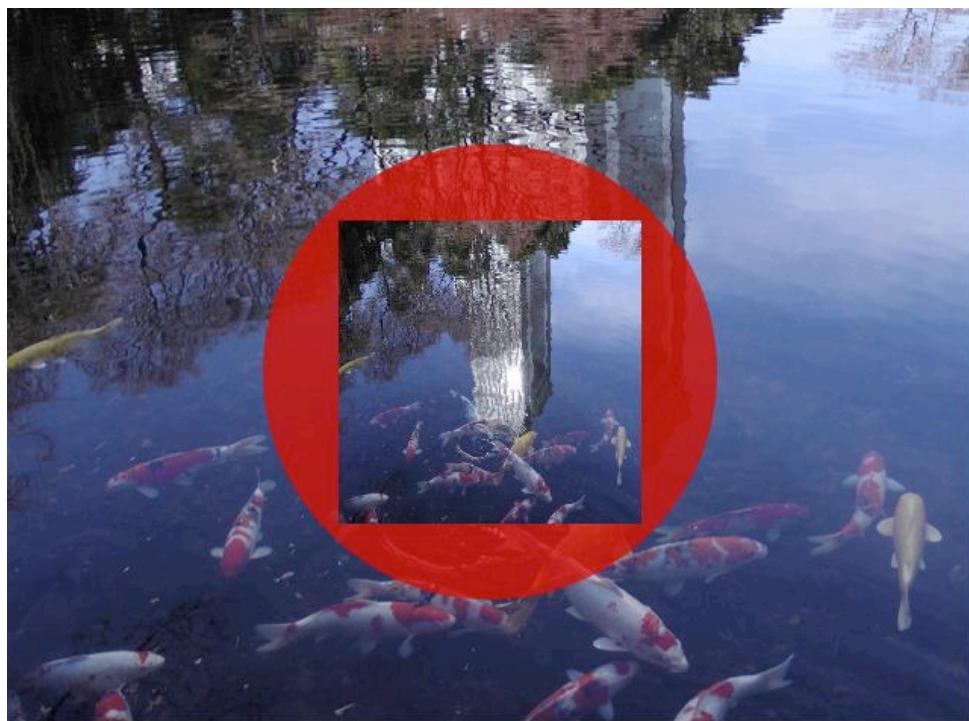


Este é o código para importar a imagem no Processing:

```
size (640,480);
background(0);
PImage img;
img=loadImage("carpas.jpg");
image (img, 0,0);
```

Podemos reutilizar a variável **img** e redimensionar a imagem novamente com a função **image()**:

```
size (640,480);
background(0);
noStroke();
smooth();
PImage img;
img=loadImage("carpas.jpg");
image (img,0,0);
fill(229,0,0,200);
ellipse(320,240,300,300);
image (img, 220,140,200,200);
```



Alteramos o código acrescentando a figura de um círculo vermelho desenhado no centro da tela (320,240). A função `image (img,240,140,200,200)` mostra a mesma imagem armazenada em `img`, porém agora ela está localizada na coordenada (220,140) e seu novo tamanho em pixels é 200×200 (altura e largura). Portanto os parâmetros para `image ()` também podem ser:

`image (x,y,largura,altura);`

## 17.2 Tingindo Imagens

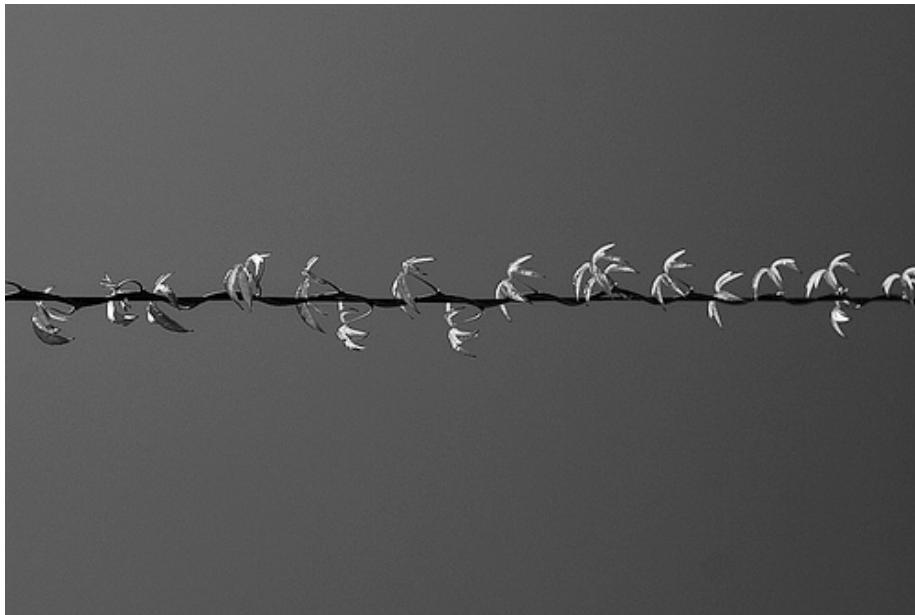
As imagens podem ser tingidas com uma cor especificada da mesma maneira como usamos a função `fill()`. Neste caso, utilizamos a função `tint()` para colorir a imagem, alterada pelos parâmetros cinza, `rgb` e transparência.

Neste exemplo, a imagem da fotógrafa [Lygia Nery](#) foi repetida três vezes. As imagens laterais foram redimensionadas com a função `image()` e coloridas com `tint()`.

```
size (640,426);
background(0);
PImage img;
img=loadImage("pessoa.jpg");
image (img,0,0);
tint(74,203,79);
image (img,490,0,150,213);
tint(196,49,49);
image (img,490,213,150,213);
```



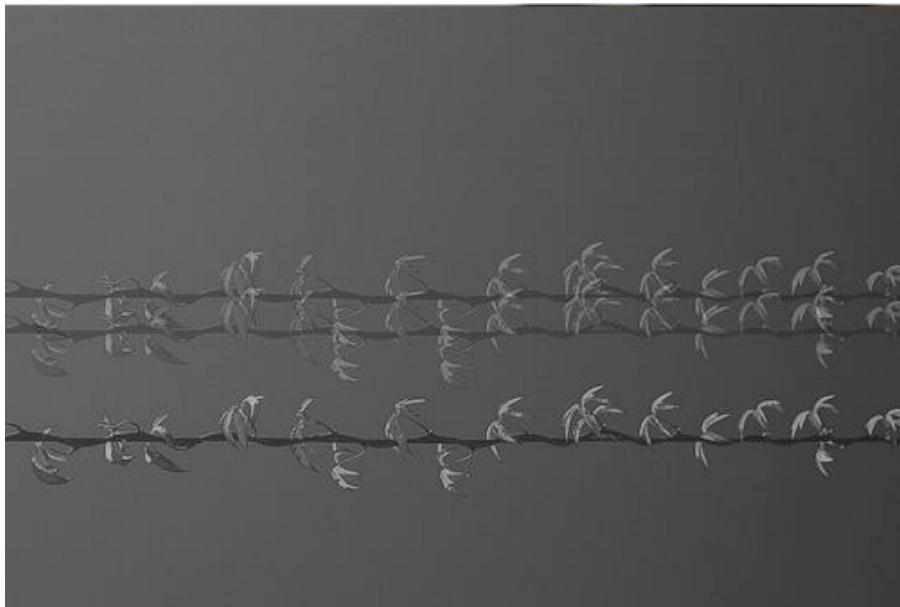
Também podemos conseguir efeitos interessantes com a manipulação da transparência . Observe esta imagem original do fotógrafo [Rogério Simões](#):



Vamos modificá-la , sobrepondo várias cópias da mesma imagem e alterando apenas o posicionamento e opacidade. Para preservar a coloração original e variar a transparência, basta manter a coloração com a cor branca e variar o parâmetro da opacidade . Veja o código utilizado:

```
size (500,334);
background(0);
PImage img;
img=loadImage("simples.jpg");
image (img,0,0);
tint(255,120);
image (img,0,20);
image (img,0,80);
```

Aqui usamos `tint(255,120)` antes de mostrar as duas cópias deslocadas. O fundo cinza da imagem original contribui para o efeito suave de transição entre as camadas.



Veja esta outra alteração:

```
size (500,334);
background(0);
PImage img;
img=loadImage("simples.jpg");
image (img,0,0);
tint(79,160,224,200);
image (img,0,-20,1000,600);
```



A segunda imagem foi colorida com opacidade alterada pelo código `tint(79,160,224,200)`. Note que o efeito gráfico de “pixelização” foi dado pelo redimensionamento da imagem: o código `image(img,0,-20,1000,600)` expandiu a largura e altura originais da imagem para fora dos limites estipulados em `size(500,334)`. Desta forma, foi preciso o repositionamento da figura, alterando a posição y da imagem para fora do sketch (y vale -20).

## 17.3 Pixels

Uma imagem (bitmap) é uma matriz (grid) de pixels na qual cada elemento é um número que especifica uma cor. Veremos como esses valores podem ser lidos e alterados com as funções `get()` e `set()`.

A função `get()` permite a leitura da cor de qualquer pixel da área do sketch (display). Ela também possibilita a captura total ou parcial da área de pixels que compõe a tela. Existem três versões ou maneiras de se utilizar a função:

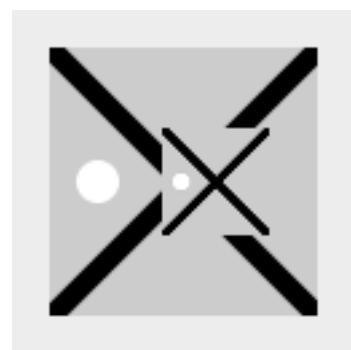
`get()`

`get(x,y)`

`get(x,y,largura,altura)`

Neste exemplo, faremos um desenho que posteriormente será capturado pela função `get()` (ex 35-02):

```
smooth();
strokeWeight(8);
line(0, 0, width, height);
line(0, height, width, 0);
noStroke();
ellipse(18, 50, 16, 16);
PImage cross = get(); // Captura a tela inteira
image(cross, 42, 30, 40, 40); // Redimensiona para 40x40
```



Os pixels capturados pela função **get()** foram armazenados na variável cross (do tipo PImage) e apresentados por image(cross,42,30,40,40). Apenas uma parcela da tela pode ser capturada (**exemplo 35-03**):

```
strokeWeight(8);
line(0, 0, width, height);
line(0, height, width, 0);
PImage slice = get(0, 0, 20, 100); // Captura parcela da tela
set(18, 0, slice);
set(50, 0, slice); // A função ser reposicionada a captura
```



A segunda versão get(x,y) retornam valores que devem ser armazenados num variável do tipo color. Os valores podem ser usados para setar parâmetros de preenchimento e traço (**exemplo 35-06**):

```
PImage trees;
void setup() {
  size(100, 100);
  noStroke();
  trees = loadImage("topangaCrop.jpg");
}

void draw() {
  image(trees, 0, 0);
  color c = get(mouseX, mouseY);
  fill(c);
  rect(50, 0, 50, 100);
}
```



Neste exemplo, a função **get(mouseX,mouseY)** retorna o valor (cor) correspondente à posição (pixel) apontada pelo mouse. Este valor é armazenado na variável **c** e posteriormente utilizado como parâmetro de preenchimento (fill c) do retângulo desenhado em rect(50,0,50,100). Ver exemplo 35-07.

Cada variável do tipo *PImage* possui sua função *get()* e isso possibilita a captura de parcelas de pixels independentes daqueles mostrados na tela (exemplo 35-8):

```
PImage trees;  
trees = loadImage("topanga.jpg");  
stroke(255);  
strokeWeight(12);  
image(trees, 0, 0);  
line(0, 0, width, height);  
line(0, height, width, 0);  
PImage treesCrop = trees.get(20, 20, 60, 60);  
image(treesCrop, 20, 20);
```



A função **set ()** permite a alteração dos pixel que compõem a tela:

**set (x,y,cor)**  
**set (x,y,image)**

O terceiro parâmetro pode indicar a cor da posição indicada por x e y, ou utilizar uma imagem que será copiada nas coordenadas, como no exemplo (35-11):

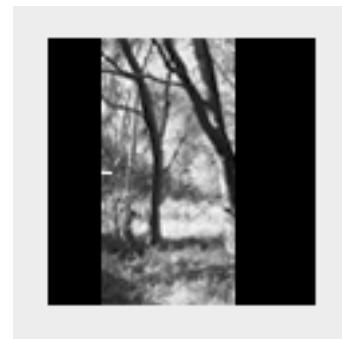
```
PImage trees;
```

```
void setup() {  
size(100, 100);  
trees = loadImage("topangaCrop.jpg");  
}  
  
void draw() {  
int x = constrain(mouseX, 0, 50);  
set(x, 0, trees);  
}
```



Assim como get() , as imagens possuem sua própria função set() que permite a escritura de pixel da própria imagem separadamente (exemplo 35-12):

```
PImage trees;  
trees = loadImage("topangaCrop.jpg");  
background(0);  
color white = color(255);  
trees.set(0, 50, white);  
trees.set(1, 50, white);  
trees.set(2, 50, white);  
trees.set(3, 50, white);  
image(trees, 20, 0);
```



No exemplo, mudamos a cor de alguns pixels da imagem para branco.

## 17.4 Filtrando os pixels

O Processing possibilita transformações dos pixels de uma imagem ou tela pela aplicação de filtros como THRESHOLD, GRAY, INVERT, POSTERIZE, BLUR, OPAQUE, ERODE ou DILATE ([http://www.processing.org/reference/filter\\_.html](http://www.processing.org/reference/filter_.html)):

**filter (modo)**

**filter (modo, nível)**

Neste exemplo aplicamos o filtro THRESHOLD em uma imagem com o parâmetro nível setado para 0.3. Neste caso, cada pixel com o valor de cinza maior que 30 por cento do brilho total será alterado para branco e pixels abaixo deste nível serão alterados para preto (exemplo 39-01):

```
PImage img = loadImage("topanga.jpg");  
image(img, 0, 0);  
filter(THRESHOLD, 0.3);
```



Neste exemplo variamos o parâmetro para a função **BLUR** (**exemplo 39-03**):

```
float fuzzy = 0.0;
```

```
void setup() {  
    size(100, 100);  
    smooth();  
    strokeWeight(5);  
    noFill();  
}
```

```
void draw() {  
    background(204);  
    if (fuzzy < 16.0) {  
        fuzzy += 0.05;  
    }  
    line(0, 30, 100, 60);  
    filter(BLUR, fuzzy);  
    line(0, 50, 100, 80);  
}
```



Aplicando filtro INVERT na área de uma imagem (**exemplo 39-04**):

```
PImage img = loadImage("forest.jpg");  
image(img, 0, 0);  
img.filter(INVERT);  
image(img, 50, 0);
```



## 17.5 Mesclando os pixels

A função `blend( )` realiza a composição de pixels em diferentes modos:BLEND,ADD, SUBTRACT, DARKEST,LIGHTEST, DIFFERENCE, EXCLUSION, MULTIPLY, SCREEN, OVERLAY, HARD\_LIGHT, SOFT\_LIGHT, DODGE e BURN. As formas desta função são:

**`blend (x,y,altura,largura,dx,dy,dlargura,daltura,modo)`**

**`blend (image,x,y,altura,largura,dx,dy,dlargura,daltura,modo)`**

x,y,altura e largura : local e área da área a ser copiada dx,y,dlargura,daltura: parâmetros referentes à área de destino

modo: ver descrições em: [http://www.processing.org/reference/blend\\_.html](http://www.processing.org/reference/blend_.html)

Para a mistura de duas imagens, ao invés da tela de display, uma segunda imagem pode ser utilizada como fonte. Se as regiões das áreas (fonte e destino) tiverem tamanhos diferentes, os pixels serão ajustados à região de destino.. Neste exemplo ocorre o blend entre uma imagem e o desenho, usando o modo DARKEST (**exemplo 39-06**):

```
PImage img = loadImage("topanga.jpg");
background(0);
stroke(255);
strokeWeight(24);
smooth();
line(44, 0, 24, 80);
line(0, 24, 80, 44);
blend(img, 0, 0, 100, 100, 0, 0, 100, 100, DARKEST);
```



A função blend() pode mesclar duas imagem sem afetar a janela de display (**exemplo 39-07**):

```
PImage img = loadImage("forest.jpg");
PImage img2 = loadImage("airport.jpg");
img.blend(img2, 12, 12, 76, 76, 12, 12, 76, 76, ADD);
image(img, 0, 0);
```



## 17.6 Copiando os Pixels

A função copy() permite a cópide de regiões de pixels entre imagens:

**copy (x,y,largura,altura,dx,dy,dlargura,daltura,modo)**

**copy (imagemFonte ,x,y,largura,altura,dx,dy,dlargura,daltura,modo)**

No próximo utilizaremos a coordenda y do mouse como parâmetro para cópia de uma região de pixels de uma imagem na área de display (**exemplo 39-10**):

```
PImage img1, img2;
void setup() {
  size(100, 100);
  img1 = loadImage("forest.jpg");
  img2 = loadImage("airport.jpg");
}
void draw() {
  background(255);
  image(img1, 0, 0);
  int my = constrain(mouseY, 0, 67);
  copy(img2, 0, my, 100, 33, 0, my, 100, 33);
}
```



A função `copy()` pode ser associada a um objeto `PImage`, permitindo que regiões da imagem possam ser copiadas em outras, ou na mesma `image` (exemplo 39-11):

```
PImage img = loadImage("tower.jpg");
img.copy(50, 0, 50, 100, 0, 0, 50, 100);
image(img, 0, 0);
```



## 18 - Processamento de Imagens

No Processing, cada imagem é armazenada num **array unidimensional** de cores. Quando a imagem é exibida na tela, cada elemento do array é desenhado como um pixel. o número (comprimento) de elementos deste array é determinado pela multiplicação da largura pela altura da imagem:

$$\text{comprimento array (pixels)} = \text{largura} \times \text{altura da imagem}$$

O primeiro elemento do array equivale ao pixel no canto superior esquerdo da imagem, e a última posição ao pixel posicionado no canto inferior direito da imagem. A altura e altura da imagem é usada para mapear a posição de cada elemento no array unidimensional na sua posição bidimensional na tela (x,y).

O array **pixels[ ]** armazena um valor de cor para cada pixel da janela de display. A função `loadPixels()` deve ser chamada antes que `pixels[ ]` seja usado. Após ao processamento do array devemos usar a função `updatePixels()` para a atualização.

Para calcular a posição de qualquer pixel no array usamos a fórmula:

$$\text{posição no array} = (y * \text{largura}) + x$$

Por exemplo, se queremos mudar a cor de um pixel localizado na posição (25,50) de uma imagem , usamos:

```
loadPixels();
pixels[50*largura+25] = color(0);
updatePixels();
```

Para a conversão contrária, dividimos a posição do pixel no array pela largura da imagem (ou da tela) para obter a **coordenada y**, e utilizamos o módulo entre e posição e largura para obter a **coordenada x**:

**coordenada y = posição no array/largura**

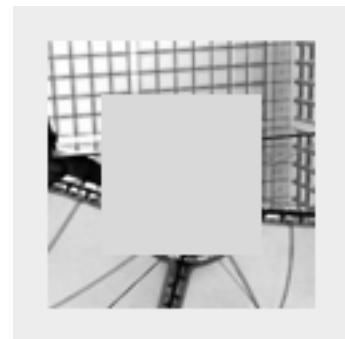
**coordenada x= posição no array % largura**

Por exemplo, para alterar a cor de um pixel que está na posição pixels[5075] do array:

```
int y= 5075 / largura;  
int x= 5075 % largura;  
set (x,y, color(0));
```

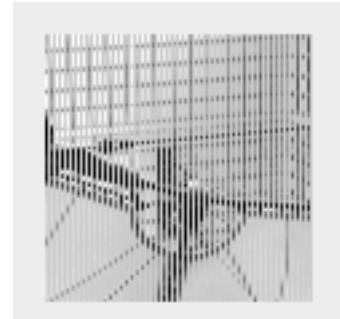
Na manipulação de muitos pixels simultaneamente, o uso de pixels[ ] é mais eficiente do que as funções get() e set(). Neste exemplo, utilizamos as coordenadas do mouse para acessar o array de pixels (**exemplo 40-05**):

```
PImage arch;  
  
void setup() {  
    size(100, 100);  
    noStroke();  
    arch = loadImage("arch.jpg");  
}  
  
void draw() {  
    background(arch);  
    int mx = constrain(mouseX, 0, 99);  
    int my = constrain(mouseY, 0, 99);  
    loadPixels();  
    color c = pixels[my*width + mx];  
    fill(c);  
    rect(20, 20, 60, 60);  
}
```



Podemos acessar e alterar o array de pixels de uma determinada imagem (**exemplo 40-7**):

```
PI mage arch = loadImage("arch.jpg");
int count = arch.width * arch.height;
arch.loadPixels();
loadPixels();
for (int i = 0; i < count; i += 2) {
pixels[i] = arch.pixels[i];
}
updatePixels();
```



Neste exemplo a varável count armazena o total de pixels da image, multiplicando a largura pela altura:

```
int count = arch.width * arch.height;
```

A função **arch.loadPixels()** prepara a imagem apenas para a leitura.

Já **loadPixels()** prepara a área de display que posteriormente será alterada e atualizada por **updatePixels()**.

A variável count é utilizada no **laço** para a leitura de cada pixel da imagem. O array **pixels[i]** corresponde aos pixels da área de display , e recebe os pixels da imagem selecionados pelo **índice i**:

```
for (int i = 0; i < count; i += 2) {
pixels[i] = arch.pixels[i];
}
```

Note que a variável i é atualizada pelo incremento **i+=2** garantindo que seja selecionada apenas a metade de pixels contidos na imagem armazenada na variável arch.

## 18.1 Gerando uma imagem

A função `createImage()` cria um buffer vazio de pixels.

### `createImage (largura,altura, formato)`

O formato da imagem pode ser RGB ou ARGB (alpha). Neste exemplo criamos um buffer de imagem que recebe os pixels de duas imagens diferentes:

```
size (1024,1024);
//
PImage solred=loadImage("sun_red.png");
PImage solblue=loadImage("sun_blue.png");
//
solred.loadPixels();
PImage comp=createImage(solred.width,solred.height,RGB);
//
int count=solred.width*solred.height;
//

for (int i=0;i<count;i++) {
    if (i%2==0) {                                // seleciona imagem para índices pares ou ímpares
        comp.pixels[i]=solred.pixels[i];
    } else {
        comp.pixels[i]=solblue.pixels[i];
    }
}
comp.updatePixels();
image (comp,0,0);
```

# 19 - Captura de Imagens

Utilizaremos a biblioteca para controle de video do Processing para a captura de imagens através de uma camera. Para o uso de recurso usaremos a importação da biblioteca:

```
import processing.video.*;
```

A seguir, criaremos uma variável ( **cam** ) do tipo **Camera** que receberá o fluxo de imagens enviados pela camera:

```
Capture cam;
```

A conexão (inicialização) da variável cam é dado por:

```
cam = new Capture(this, 320, 240); // 320 e 240 determinam dimensões da captura
```

A leitura ou captura deve ser feita dentro da função draw() para a constante atualização das imagens, utilizando a função *read()*:

```
void draw() {  
    if (cam.available() == true) {  
        cam.read();  
        image(cam, 320, 240);  
    }  
}
```

No próximo exemplo, veremos o código completo para a captura e armazenamento numa variável do tipo `PImage`. Através da função `arrayCopy()` podemos fazer uma transferência direta do array de pixels capturados pela camera para um buffer de pixels:

### **arrayCopy (fonte,destino)**

```
import processing.video.*;  
  
Capture cam;  
PImage cap;  
  
void setup() {  
    size (640,480);  
    background(0);  
    cam=new Capture(this,160,120);  
    cap=createImage(160,120,480,RGB);  
    cap.loadPixels();  
    cam.loadPixels();  
}  
  
void draw() {  
    if (cam.available()==true) {  
        cam.read();  
        arrayCopy(cam.pixels,cap.pixels);  
        image (cap,mouseX,mouseY);  
    }  
    cap.updatePixels();  
}
```

ver exemplo **slitscan**

# 20 - Texto e tipografia

Para o uso de tipografia no Processing, devemos converter qualquer **fonte** para o formato **VLW**. Neste padrão cada letra é transformada numa imagem , possibilitando a renderização de qualquer tipo de texto.

Para converter a fonte:

- 1. Selecione a opção “Create Font” no menu “Tools”.**
- 2. Selecione uma fonte na lista de fontes e clique “OK”.**
- 3.**

Após a seleção, a fonte é gerada e copiada na pasta data do sketch atualmente utilizado. Isso pode ser verificado selecionando a opção **“Show sketch Folder”** no menu **“Sketch”**. Para ser utilizada, a fonte deve ser carregada numa variável do tipo **PFont**:

```
PFont fonte;  
fonte=loadFont ("Arial-Black-48.vlw");  
textFont(fonte);  
fill(0);  
text("PROCESSING",20,30);
```

A função **loadFont()** é responsável pelo carregamento da fonte **Arial-Black-48.vlw** na variável “fonte”. A função **textFont()** seleciona a fonte vigente e a função **text()** imprime o texto “PROCESSING” na posição x=20 e y=30.

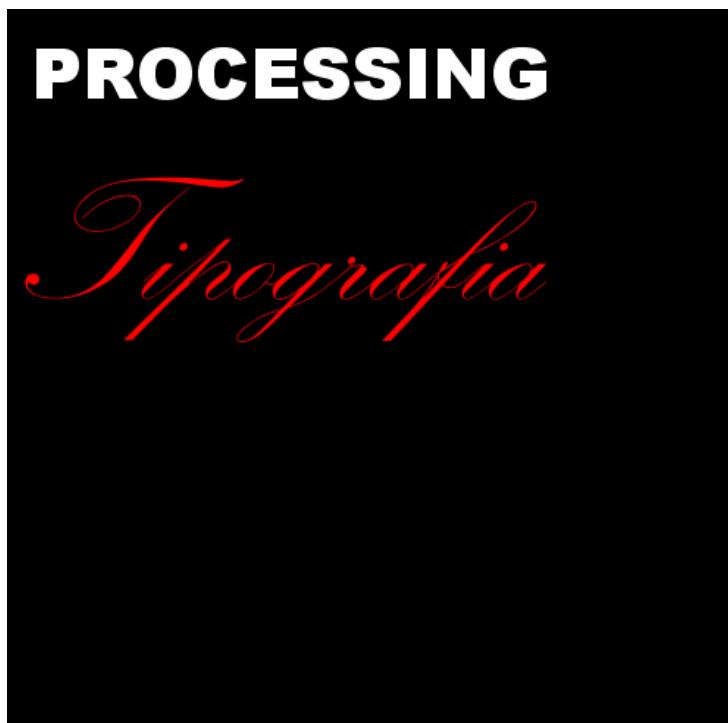
```
PFont fonte;  
fonte=loadFont ("Arial-Black-48.vlw");  
textFont(fonte);  
fill(0);  
text("PROCESSING",20,30,100,100);
```

A função **text** também pode usar mais dois outros parâmetros para dimensionar a largura e altura do texto:

```
text ("PROCESSING",20,30,100,100);
```

Podemos utilizar mais de uma fonte no mesmo sketch, bastando carregar os arquivos em suas respectivas variáveis:

```
PFont fonte1,fonte2;  
size (500,500);  
background(0);  
fonte1=loadFont ("Arial-Black-48.vlw");  
fonte2=loadFont ("KunstlerScript-124.vlw");  
fill(255);  
textFont(fonte1);  
text("PROCESSING",20,30,500,100);  
fill(255,0,0);  
textFont(fonte2);  
text("Tipografia",10,200);
```



## 20.1 Atributos de texto

Algumas funções podem controlar qualidades específicas como tamanho, espaçamento e alinhamento.

A função `textSize()` determina o tamanho da fonte em unidades de pixels.

```
PFont fonte1;  
size(400,400);  
background(0);  
fonte1=loadFont("Arial-Black-48.vlw");  
fill(255);  
textFont(fonte1);  
textSize(48);  
text("PROCESSING",20,90);  
textSize(124);  
text("PROCESSING",20,200);
```



## 20.2 Strings

A String é um tipo de variável adequada para o armazenamento de palavras e sentenças.:

```
String frase1=" Meu nome é “;
```

Podemos criar um array de strings:

```
String [] nomes = {"Paulo", "João", "Maria", "Volusia"};
```

As strings podem ser combinadas com o operador + :

```
String frase="Meu nome é ";
String[] nomes={"Paulo","João","Maria","Volusia"};
int i=round(random(nomes.length-1));
println (frase+nomes[i]);
```

O tipo String possui funções para a extração ou comparação de sequências de caracteres. Por exemplo, a função **substring()** retorna uma nova string que é parte da original:

```
String nome="Processing";
println (nome.substring(2)); // Imprime “ocessing”;
println (nome.substring(3,6)); // imprime “cess”;
```

Para comparação de duas strings utilizamos a função **equals()**, que pode retornar true ou false caso as strings forem idênticas ou diferentes:

```
String p1="Processing";
String p2="processyng";

println (p1.equals(p2)); // retorna false
```

No próximo exemplo, vamos usar um array para armazenar tipos de fontes diferentes que serão aplicados no texto de maneira randômica:

```

PFont fonte1,fonte2,fonte3;
PFont [] fontes={fonte1,fonte2,fonte3};
String [] frase1={"beba","babe","caco","cola"};
String [] frase2={"    ","coca","cola","caco","cloaca"};
float timer;

void setup() {
size (410,330);
background(227,30,30);
smooth();
fontes[0]=loadFont("Times-Bold-48.vlw");
fontes[1]=loadFont("Arial-ItalicMT-48.vlw");
fontes[2]=loadFont("Impact-48.vlw");
fill(255);
timer=0;
}

void draw() {
float segundos=round(millis()/1000);
if (segundos-timer>2) {
background(227,30,30);
for (int j=0;j<350;j+=50) {
textFont(fontes[round(random(fontes.length-1))]);
text(frase1[round(random(frase1.length-1))],10,j);
textFont(fontes[round(random(fontes.length-1))]);
text(frase2[round(random(frase1.length-1))],150,j);
textFont(fontes[round(random(fontes.length-1))]);
text(frase2[round(random(frase1.length-1))],300,j);
}
timer=segundos;
}
}

```

# 21 - Exportando Imagens

Os gráficos gerados no ambiente do Processing podem ser salvos nos formatos PDF, tiff, jpeg ou PNG. A qualidade ou resolução está associada ao tamanho ou size() determinado para o sketch. Neste turotial, veremos como exportar uma sequência de imagens (arquivos PDF) geradas em tempo real.

Salvando sequência de imagens (animação) no formato PDF:

Neste exemplo, faremos uma animação simples baseada na movimentação do mouse. Vamos lembrar que o efeito de animação pode ser feito com a função draw() , bastando variar alguns parâmetros dos objetos gráficos como posição, cor, transformações ou dimensões:

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

void draw() {
  background (255);
  stroke (0,20);
  strokeWeight(20);
  line (mouseX,0,width-mouseY,height);
}
```

Experimentando esse sketch, podemos observar a movimentação da linha conforme o movimento do mouse. Agora, acrescentamos um código (em negrito) que possibilita o "save" de vários frames dessa movimentação no formato PDF:

```
import processing.pdf.*;

boolean salvar=false;
void setup() {
  size (600,600);
}

void draw() {
  if (salvar==true) {
    beginRecord(PDF, "linha-###.pdf");
  }

  background (255);
  stroke (0,20);
```

```
strokeWeight(20);
line (mouseX,0,width-mouseY,height);
if (salvar==true) {
endRecord();
salvar=false;
}
}

void mousePressed() {
salvar=true
}
```

Explicando o código passo a passo:

Com este novo código, salvamos um arquivo no formato PDF a cada vez que pressionamos o botão do mouse. Os nomes dos arquivos salvos possuem uma numeração sequencial. Execute o código, pressione o mouse algumas vezes e verifique na pasta “data”, dentro do sketch (ctrl-K). Vamos explicar o código, primeiro temos antes da função setup():

```
import processing.pdf.*;
boolean salvar=false;
```

Para o “save” no formato PDF devemos importar a biblioteca (processing.pdf). Logo após criamos uma variável do tipo “boolean” (ela pode valer apenas true ou false) que será usada para a verificação se o mouse foi pressionado ou não. Agora, dentro da função draw():

```
if (salvar==true) {
beginRecord(PDF, "linha-###.pdf");
}
```

if (salvar==true) verifica (condicional IF) se a variável salvar está valendo true (VERDADEIRO) . Se sim , permite a gravação de um frame com a instrução beginRecord. Preste atenção neste código (“linha.###.pdf”), pois é aqui que determinamos o nome do arquivo a ser salvo.

No exemplo, serão salvos arquivos pdfs , numa sequência parecida com esta:

linha-001.pdf

linha-150.pdf

linha-320.pdf

...

Depois que qualquer desenho for feito, desabilitamos a gravação com endRecord() e preparamos o código para a próxima gravação com salvar=false. A linha salvar=false é importante, pois sem ela não podemos verificar se o mouse foi pressionado ou não.

```
if (salvar==true) {  
endRecord();  
salvar=false;  
}
```

Note que a verificação do estado da variável “salvar” (se true ou false) será testado constantemente, visto que o código condicional if (salvar==true está dentro do looping draw(). Com a função mousePressed() , a variável “salvar” volta a valer true, permitindo novamente a verificação de if (salvar==true):

```
void mousePressed() {  
salvar=true  
}
```

Em qualquer momento podemos selecionar um instante da animação para ser gravado num determinado arquivo pdf. O uso da função draw() junto com mousePressed() permite uma ação de “ liga (salvar=false) e desliga (salvar=true)”.

Teste o seu próprio código de animação , bastando substituir a porção responsável pelo desenho dentro da função draw():

```
background (255);  
stroke (0,20);  
strokeWeight(20);  
line (mouseX,0,width-mouseY,height);
```

Lembre que você pode usar a função random () para a geração de resultados aleatórios.