Processing 101

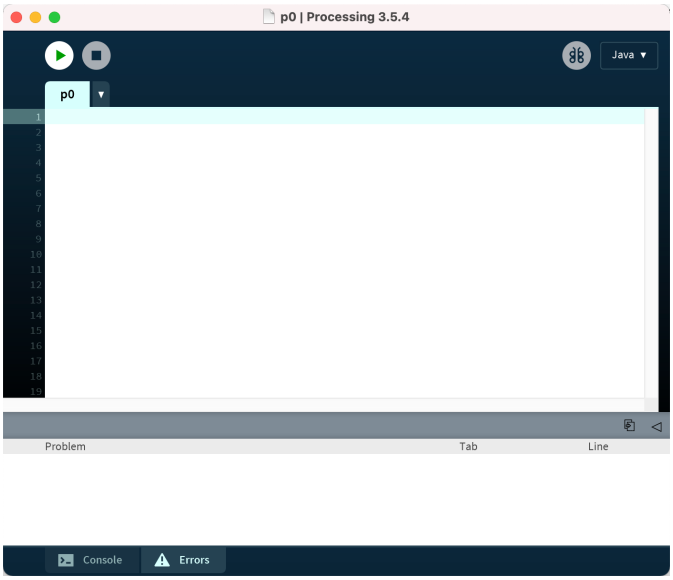
Pedro Guerreiro

Universidade do Algarve

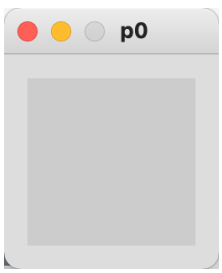
3 de maio de 2021

O primeiro programa

O primeiro programa Processing mais simples não poderia ser: basta não escrever nada:



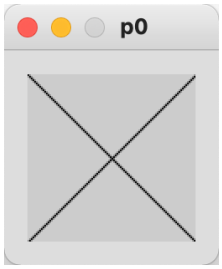
Corremos isto com o comando Run:



Surge uma janelita cinzenta, e pronto. Mas é o nosso programa sendo executado no computador. A única coisa que podemos fazer com ele é terminá-lo, fechando a janela.

Na verdade, aquele quadrado cinzento, dentro da janela, é uma área de 100 píxeis por 100 píxeis onde podemos desenhar "coisas". Para desenhar, temos de programar. Observe, como exemplo:

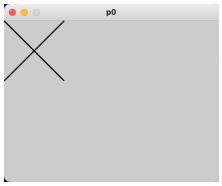
```
void setup()
{
  line(0,0, 100, 100);
  line(0, 100, 100, 0);
}
```



É um progresso em relação ao primeiro programa que não fazia nada.

Podemos conseguir uma janela de tamanho arbitrário, recorrendo à função `size`:

```
void setup()
{
  size(360, 270);
  line(0,0, 100, 100);
  line(0, 100, 100, 0);
}
```



Para isto se parecer mais com os programas que escreveremos a seguir, introduzimos a função `draw`, separando a *inicialização*, que é feita na função `setup`, do *desenho*, que é feito na função `draw`:

```
void setup()
{
  size(360, 240);
```

```
}

void draw()
{
  line(0,0, 100, 100);
  line(0, 100, 100, 0);
}
```

O desenho parece o mesmo, mas é conseguido de maneira diferente. No primeiro caso, o programa desenha o xis uma vez e mais nada. No segundo, o programa, desenha o xis 60 vezes por segundo. Como o xis é sempre o mesmo o desenho não se distingue do anterior.

Ainda assim, não seria má ideia "variabilizar" aquelas constantes, 0 e 100, para permitir desenhar xis de qualquer tamanho em qualquer parte da janela. Precisamos das coordenadas do ponto que corresponde ao canto superior esquerdo do xis e precisamos da medida do "lado" do xis:

```
float x1, y1;
float side;

void setup()
{
  size(360, 270);
  x1 = 0;
  y1 = 0;
  side = 100;
}

void draw()
{
  line(x1, y1, x1+side, y1+side);
  line(x1, y1+side, x1+side, y1);
}
```

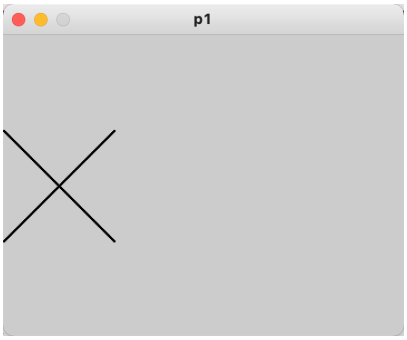
O efeito é o mesmo. Para variar, coloquemos o xis centrado verticalmente, encostado ao lado esquerdo da janela:

```
void setup()
{
  size(360, 270);
  side = 100;
  x1 = 0;
```

```
  y1 = (height - side) / 2;
}
```

Usamos aqui a variável `height`, do Processing, que dá a altura da janela. Sabemos que é 270, mas se amanhã mudarmos para outro valor, o xis continuará a meia altura, na janela.

Há também a variável `width`, que dá a largura da janela. Usá-la-emos daqui a pouco.



Animação

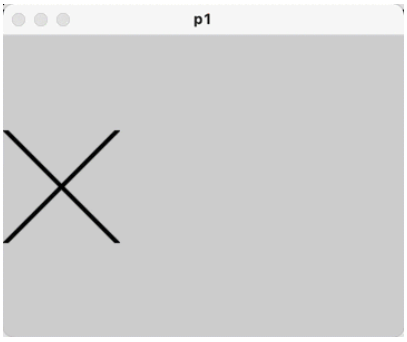
Se mudarmos ligeiramente a posição do xis de cada vez que desenhamos na função `draw`, conseguiremos uma animação. Observe, na função `draw`:

```
void draw()
{
  line(x1, y1, x1+side, y1+side);
  line(x1, y1+side, x1+side, y1);
  x1++;
}
```

Visto que a coordenada horizontal do canto superior esquerdo é incrementada, 60 vezes por segundo, antecipamos que o xis se descola horizontalmente para a direita. Sendo a largura 360, o xis desaparecerá pelo lado direito ao fim de 6 segundos:

Obtemos uma animação, mas talvez não a que desejávamos[1]:

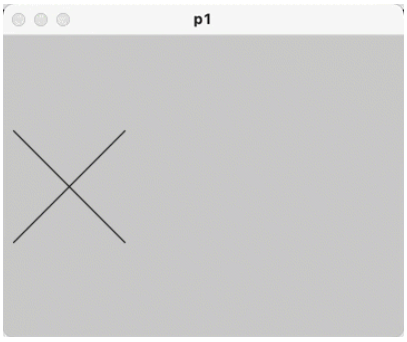
[1] Nos documentos pdf, as figuras que representam animações não estão animadas. Nas versões html e



O que acontece é que a cada sexagésimo de segundo um novo xis é desenhado, mas o anterior não é apagado. Falta um ingrediente na função `draw`: chamar a função `background`, à cabeça da função `draw`, para, por assim dizer, "limpar" o desenho, pintando a janela com uma dada cor de fundo, após o que o programa desenha de novo:

```
void draw()
{
  background(200);
  line(x1, y1, x1+side, y1+side);
  line(x1, y1+side, x1+side, y1);
  x1++;
}
```

Agora sim:



O valor 200 usado como argumento na função `background` representa o nível de cinzento usado: um valor mais baixo daria um cinzento mais escuro; um valor mais alto daria um cinzento mais claro. Aliás, 0 é preto e 255 é branco.

---

ePub, sim. No entanto, estas animações são feitas à base de *gifs*, e portanto são menos suaves do que as que observamos ao correr os programas no computador.

Alternativamente, em vez de usar variáveis para as coordenadas do canto do xis, podemos usar funções, que calculam os valores das coordenadas "em função" do tempo que decorreu desde que o programa arrancou.

A função `draw` é chamada automaticamente 60 vezes por segundo. A variável `frameCount`, gerida pelo sistema de execução, conta o número de vezes que a função `draw` foi chamada. Logo, podemos usá-la para registar a passagem do tempo, em sexagésimos de segundo.

Portanto, substituímos as variáveis `x1` por uma função:

```
float x1(int t)
{
  return t;
}

float y1;
float side;

void setup()
{
  size(360, 270);
  side = 100;
  y1 = (height - side) / 2;
}

void draw()
{
  background(200);
  float x1 = x1(frameCount);
  line(x1, y1, x1+side, y1+side);
  line(x1, y1+side, x1+side, y1);
}
```

O efeito é o mesmo.

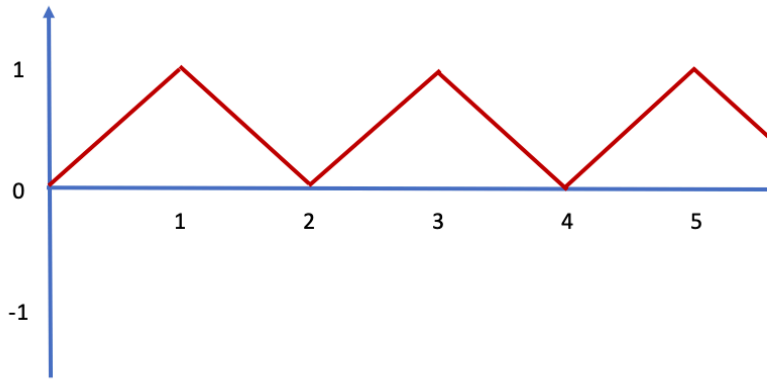
Na animação, o xis começa à esquerda e sai pela direita. O programa continua a chamar a função `line`, mesmo depois de o X ter desaparecido, mas como o desenho cai fora da janela, essas instruções não têm efeito visível.

Em vez de deixar o xis desaparecer pela direita, podemos fazê-lo voltar para trás, quando bater no lado direito da janela, até chegar ao lado esquerdo e recomeçar, indefinidamente.

Temos, pois, de redefinir a função `x1`. Pretendemos que o resultado aumente linearmente até o argumento `t` valer `width - side`, altura em que o xis toca no lado direito da janela, e depois diminua linearmente até 0, altura em que o xis toca no lado esquerdo da janela, e depois tudo recomeça.

Portanto, a função `x1` será uma função periódica, em que o período é o dobro da largura da janela.

Consideremos, abstratamente, a seguinte função que tem o seguinte gráfico:



É uma onda em forma de triângulo, por assim dizer. Sobe e desce periodicamente, e as subidas e descidas são lineares. Podemos programá-la assim:

```
float triangleWave(float x)
{
  assert x >= 0.0;
  float result = x % 2;
  if (result > 1.0)
    result = 2.0 - result;
  return result;
}
```

Podemos programar um teste unitário:

```
void unitTestTriangleWave()
{
```

```
  assert triangleWave(2.5) == 0.5;
  assert triangleWave(2.0) == 0;
  assert triangleWave(10.25) == 0.25;
  assert triangleWave(11.25) == 0.75;
}
```

Tipicamente, será chamado na função `setup`, por meio da função `unitTests`:

```
void unitTests()
{
  unitTestTriangleWave();
}

void setup()
{
  // ...
  unitTests();
}
```

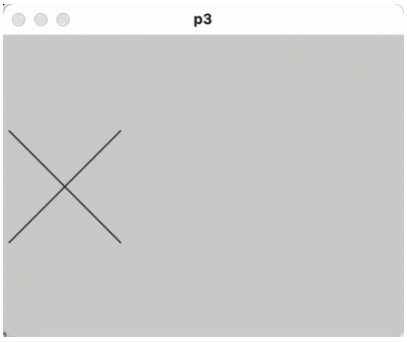
A função `triangleWave` serve-nos de base para a função `x1`:

```
float x1(float t)
{
  return (width - side) * triangleWave(t / (width - side));
}
```

A função `draw` fica assim:

```
void draw()
{
  background(200);
  float x1 = x1(frameCount);
  line(x1, y1, x1+side, y1+side);
  line(x1, y1+side, x1+side, y1);
}
```

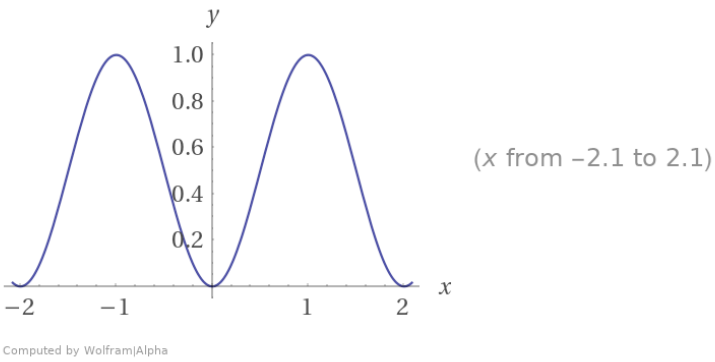
Temos agora a seguinte animação:



No fundo, a função `triangleWave` é uma função periódica, *f*, de período 2, com valores entre 0 e 1, e tal que *f*(0) é 0 e *f*(1) é 1. A função é contínua, mas a derivada não é contínua: o xis muda bruscamente de velocidade quando bate no lado da janela. Para tornar o movimento mais suave, convém-nos uma função cuja derivada seja também contínua. Buscando inspiração nas funções sinusoidais, experimentemos esta:

```
float cos1(float x)
{
  return (cos (x*PI - PI)+1.0)/2;
}
```

O gráfico, obtido com o Wolfram Alpha, confirma que é isso que queremos:

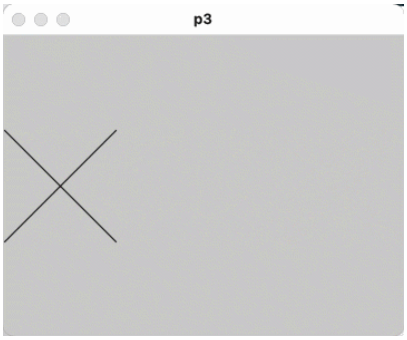


Ficamos com a seguinte função `x1` alternativa, `x1Alt`:

```
float x1Alt(float t)
{
  return (width - side) * cos1(t / (width - side));
}
```

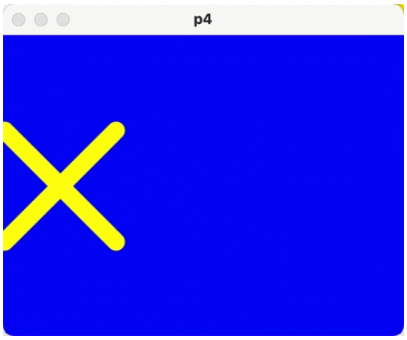
Usamos esta em vez da outra, na função `draw`:

```
void draw()
{
  background(200);
  float x1 = x1Alt(frameCount);
  //...
}
```



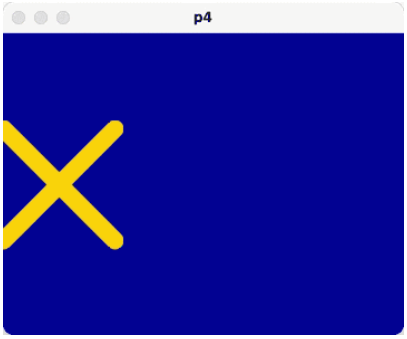
Agora que a dinâmica está resolvida, podemos tratar da estética. Para a animação ficar mais agradável, desenhemos um xis mais grosso, amarelo, sobre fundo azul:

```
color blue = color(0, 0, 255);
color yellow = color(255, 255, 0);

// ...

void draw()
{
  background(blue);
  stroke(yellow);
  strokeWeight(16);
  float x1 = x1Alt(frameCount);
  line(x1, y1, x1+side, y1+side);
  line(x1, y1+side, x1+side, y1);
}
```



Alternativamente, experimentemos com o azul e o amarelo da bandeira da União Europeia:

```
color euroAzure = color(0, 5, 153);
color euroGold = color(255, 204, 0);

void draw()
{
  background(euroAzure);
  stroke(euroGold);
  // ...
}
```



Multianimação

Inspirados por este último exemplo, desenhemos uma aproximação da bandeira da União Europeia, substituindo as estrelas por pequenos xis amarelos, dispostos em círculo no meio do fundo azul. A animação consistirá em fazer as estrelas passear horizontalmente na janela, para a frente e para trás. Usaremos as duas técnicas de mudança de sentido: a brusca e a suave.

Comecemos pela bandeira com as estrelas.

Convém-nos uma classe para representar estrelas. Incluímos membros de dados para as coordenadas do canto superior esquerdo do quadrado que contém a estrela e para a medida do lado do quadrado. Incluímos também um construtor e um método para desenhar a estrela, por agora na forma de um xis:

```
class Star {
  float x;
  float y;
```

```
  float side;

  Star(float x, float y, float side)
  {
    this.x = x;
    this.y = y;
    this.side = side;
  }

  void draw()
  {
    line(x, y, x+side, y+side);
    line(x, y+side, x+side, y);
  }
}
```

Teremos arrays de estrelas e vamos querer desenhá-los:

```
void starsDraw(Star a[])
{
  for (Star x : a)
    x.draw();
}
```

A bandeira da União Europeia tem 12 estrelas, mas deixemos o número de estrelas numa variável. As estrelas formam um círculo centrado no centro da janela. Admitamos que o raio do círculo é um terço da altura. Para construir o array das estrelas, calculamos as coordenadas dos pontos que onde as estrelas devem estar centradas, sobre o círculo, e para cada ponto construímos a estrela centrada nesse ponto. Precisamos de um construtor *ad hoc* para isso:

```
Star centered_at(float x, float y, float side)
{
  return new Star(x-side/2, y-side/2, side);
}
```

A seguinte função cria um array de `n` estrelas, dispostas equidistantemente sobre uma circunferência de raio `r` centrada no ponto de coordenadas `x`, `y`, cada estrela ocupando um quadrado de lado `side`:

```
Star[] stars(int n, float x, float y, float r, float side)
{
  Star result[] = new Star[n];
```

```
  for (int i = 0; i < n; i++)
  {
    float x1 = x + r * cos(i * TWO_PI / n);
    float y1 = y + r * sin(i * TWO_PI / n);
    result[i] = centered_at(x1, y1, side);
  }
  return result;
}
```

Tipicamente, as constantes do problema ficam em destaque no início do programa:

```
int numberOfStars = 12;
int sideOfStarSquare = 18;
```

Precisamos de uma variável para o array das estrelas:

```
Star stars[];
```

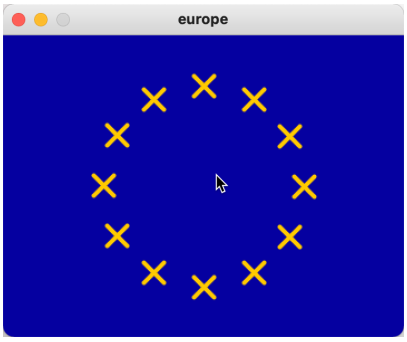
A função `setup` trata das inicializações:

```
void setup()
{
  size(360, 270);
  stars = stars(numberOfStars, width/2, height/2, height/3,
sideOfStarSquare);
}
```

A função `draw` desenha a bandeira:

```
void draw()
{
  background(euroAzure);
  stroke(euroGold);
  strokeWeight(4);
  starsDraw(stars);
}
```

Obtemos o seguinte:



Resta pôr as estrelas a mexer.

Para fazer isso, precisamos de calcular a posição de uma estrela ao fim de um dado tempo, conhecida a posição inicial. Calculada essa posição, construímos uma estrela aí, e desenhá-la-emos, como se fosse a estrela original, que se tivesse deslocado. Na verdade, a estrela original não muda.

Para calcular a posição da estrela ao fim de um dado tempo, usamos a seguinte função, que recorre à técnica da função `triangleWave`:

```
float starX(float t)
{
  return (width - sideOfStarSquare) * triangleWave(t / (width
- sideOfStarSquare));
}
```

Finalmente, para construir a nova estrela, usamos o método `after`, na classe `Star`:

```
class Star {
  //...
  Star after(float t)
  {
    return new Star(starX(t+x), y, side);
  }
}
```

Assim, sendo `s` e `t` um número representado a medida de um intervalo de tempo, a expressão `s.after(t)` é *outra* estrela que representa a estrela `s` após ter viajado durante `t` unidades de tempo.

Para desenhar um array de estrelas após um tempo dado, calculamos cada estrela após esse tempo e desenhamo-la:

```
void starsDraw(Star a[], float t)
{
  for (Star x : a)
    x.after(t).draw();
}
```

Para ter a animação, substituímos na função `draw` a instrução `starsDraw(stars)` pela instrução `starsDraw(stars, framecount)`:

```
void draw()
{
  background(euroAzure);
  stroke(euroGold);
  strokeWeight(4);
  // starsDraw(stars);
  starsDraw(stars, frameCount);
}
```

Constatamos que as estrelas mudam de sentido bruscamente. Para fazê-las mudar de sentido suavemente, os cálculos envolvem alguma trigonometria, tal como antes.

Partimos da função que calcula posição da estrela ao fim de algum tempo, supondo que estrela estava encostada ao lado esquerdo da janela. Esta função substitui a função `starX` anterior:

```
float starX(float t)
{
  return (width - sideOfStarSquare) * cos1(t / (width -
sideOfStarSquare));
}
```

Precisamos da função inversa desta, para calcular o deslocamento correspondente à posição inicial das estrelas que não estão encostadas ao lado esquerdo da janela:

```
float starXInverse(float y)
{
  float a = width - sideOfStarSquare;
  float z = acos(1 - (2*y)/a);
  float result = z*a/PI;
  return result;
}
```

Para as estrelas mudarem de sentido suavemente, basta agora substituir o método `after`, que devolve a estrela ao fim do tempo dado, pelo seguinte:

```
class Star {
  // ...

  Star after(float t)
  {
    float alpha = starXInverse(x);
    return new Star(starX(t+alpha), y, side);
  }
}
```