Jogos de Programação

aula 01: Fundamentos da programação

1 Introdução

Os métodos tradicionais de ensino dizem

→ *As coisas são assim.*
Você precisa fazer as coisas assim!

Mas nós queremos dizer

→ *Olha isso, veja que legal ...*
E agora, o que a gente pode fazer com isso?

Então nós temos o método "*Faça isso!*" e o método "*Olha isso ...*".

Quer dizer, quando alguém diz pra gente "*Faça isso!*", a gente sempre pode responder

→ *Mas, as coisas não precisam ser assim ...*
Olha isso, eu também posso fazer desse jeito.
E agora que eu vi isso, eu posso fazer outras coisas ...

O ponto importante não é como as coisas começam.

O ponto importante é como as coisas continuam ...

Ou melhor, as coisas sempre podem continuar de várias maneiras diferentes.

E daí, a gente sempre pode escolher como continuar — (*indo na direção daquilo que é legal ...*)

Ao ver isso, a gente entende que as coisas também podem começar de várias maneiras.

Escolhendo maneiras de começar e maneiras de continuar, a gente pode levar o pensamento para qualquer lugar.

E pensando assim, a gente até chega no lugar que o "*Faça isso!*" queria ...

Mas chega de papo furado.

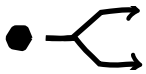
E vamos ver como essa coisa funciona na prática.

2 Fundamentos da programação

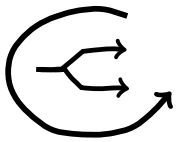
Só existem 3 coisas na programação — (*e mais uma quarta coisa ...*)

Quer dizer, a gente sempre pode pegar qualquer coisa e botar para repetir

E a gente sempre pode olhar para qualquer coisa, e depois ir para um lado ou para outro

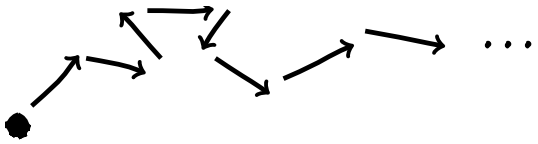


Mas, a coisa mais legal mesmo é botar a bifurcação dentro da repetição

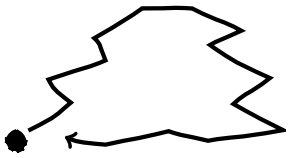


Porque agora, a gente pode fazer coisas diferentes a cada volta da repetição.

E fazendo isso, a gente constrói um caminho maluco

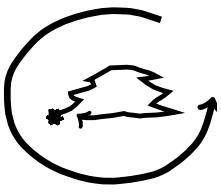


Daí, ajustando um pouquinho a nossa maluquice, a gente obtém uma coisa legal



Agora, a gente tem uma outra coisa.

Que a gente pode botar para repetir



Legal, não é?

Mas, você viu? — (ou será que não viu ainda ...)

3 Desenhando linhas

Mas chega de papo furado ...

E vamos ver como a coisa funciona na prática.

Bom, tudo começa quando a gente escolhe um qualquer coisa.

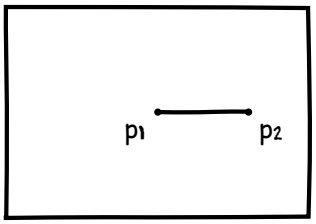
E a coisa que a gente vai escolher hoje é isso aqui

```
linha (p1, p2)
```

Quer dizer, isso é um comando que desenha uma linha na tela.

E daí, quando a gente coloca esse comando dentre de um programa (com algumas mutretagens), é isso o que a gente vê

```
...
linha (p1, p2)
...
```



Mas, e agora?

Bom, agora a gente bota esse troço para repetir.

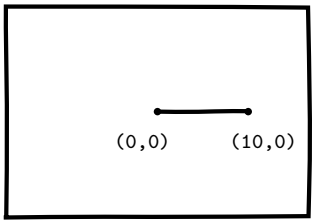
Mas antes disso, é preciso entender um pouquinho melhor as coisas.

Quer dizer, os pontos `p1`, `p2` são coordenadas `x`, `y` dos pontos na tela.

Aqui nós estamos imaginando que o $(0, 0)$ é o ponto que fica no centro da tela.

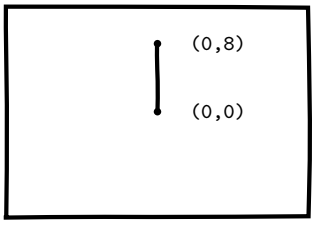
E daí, o que a gente fez foi desenhar a linha que vai do ponto $(0, 0)$ até o ponto $(10, 0)$ — (*um pouquinho para a direita*)

```
...
linha ((0,0),(10,0))
...
```



Mas, a gente também podia ter desenhado a linha para cima

```
...
linha ((0,0),(0,8))
...
```

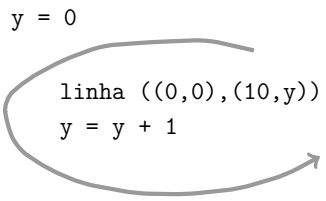


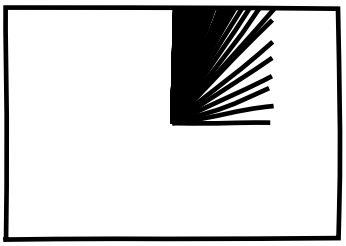
Agora que a gente entendeu isso, é fácil botar a coisa para repetir.

Quer dizer, a ideia é desenhar a linha até o ponto $(0, y)$

```
linha ((0,0),(10,y))
```

E daí, dentro da repetição a gente aumenta o valor do `y` — (*começando ele com 0*)





Para fazer isso na linguagem de programação, a gente precisa de um comando de repetição.

Na prática, existe vários comandos de repetição diferentes.

Mas nós vamos utilizar esse aqui

```
y = 0
while ( 1 == 1 ):
      linha ((0,0),(10,y))
      y = y + 1
```

Quer dizer, isso está dizendo o seguinte

→ *Enquanto o 1 for igual ao 1*
desenhe a linha, e aumente o valor do `y` *em 1*

Daí, como o 1 vai ser igual a 1 pra sempre, a coisa vai ficar rodando pra sempre.

Mas, quando a coisa roda pra sempre as linhas vão até o infinito.

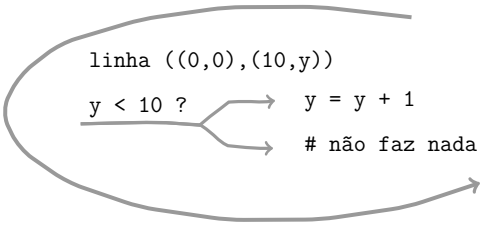
E assim, o desenho não fica muito bonito.

Só que isso é uma coisa bem fácil de consertar.

Quer dizer, é nessa hora que a gente usa a bifurcação.

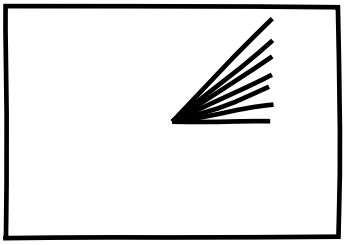
A ideia é que a gente vai olhar para o `y`.

E se o `y` ainda for menor do que 10, a gente aumenta ele de 1 — (*senão a gente não faz nada ...*)



Na linguagem de programação, a coisa fica assim

```
y = 0
while (1 == 1):
   linha ((0,0),(10,y))
   if (y < 10):
      y = y + 1
```



Legal, não é?

Exemplos

a. Subindo e descendo

E agora?

Bom, agora a gente pode fazer qualquer coisa ...

Por exemplo, seria legal a gente ver a linha subindo — (*como se alguém estivesse puxando ela por uma ponta*)

Mas, como é que a gente faz isso?

Bom, a ideia é simples.

Quer dizer, a ideia é a gente apagar a linha anterior antes de desenhar a próxima.

Sim, mas como é que a gente faz isso?

Bom, aqui é a hora para uma pequena esperteza.

Quer dizer, a tela do computador é preta, não é?

Então, se a gente desenha uma linha preta em cima da linha branca, vai parecer que a gente apagou a linha branca.

Para dizer qual é a com da linha, a gente usa o comando

```
linhac (p1,p2,white)     ou     linhac (p1,p2,black)
```

Agora é preciso entender como é que fica a lógica.

Quer dizer, a ideia é a seguinte

```
y = 0
desenha a linha

apaga a linha
aumenta o valor de y   (se y < 10)
desenha a linha
```

E a coisa fica assim

```
y = 0
linhac ((0,0),(10,y),white)
while ( 1 == 1 ):
      linhac ((0,0),(10,y),black)
      if (y < 10):
      y = y + 1
            linhac ((0,0),(10,y),white)
```

E então, você viu?

Sim! não dá pra ver nada.

Quer dizer, tudo acontece tão rápido no computador ...

E daí, a gente não consegue ver nada.

Só que isso é bem fácil de resolver.

Existe um comando que diz para o computador tirar uma sonequinha

```
sleep(2)              # dorme por 2 segundos
```

Então, a ideia é pedir para o computador tirar uma sonequinha antes de apagar a linha — (*para dar tempo da gente ver ...*)

```
y = 0
linhac ((0,0),(10,y),white)
while ( 1 == 1 ):
      sleep(1)
      linhac ((0,0),(10,y),black)
      if (y < 10):
            y = y + 1
      linhac ((0,0),(10,y),white)
```

Mas, isso ainda não está muito legal.

Porque a linha está subido aos pulinhos ...

Quer dizer, seria mais legal ver a linha deslizando para cima.

Só que isso é uma coisa bem fácil de fazer.

Quer dizer, basta diminuir o tamanho do pulo do `y`.

E diminuir o tempo da soneca também.

A coisa fica assim

```
y = 0
linhac ((0,0),(10,y),white)
while ( 1 == 1 ):
      sleep(1 * 0.1)
      linhac ((0,0),(10,y),black)
      if (y < 10):
            y = y + 1 * 0.1
      linhac ((0,0),(10,y),white)
```

E agora, é só ajustar o valor que está sendo multplicado até a coisa ficar legal.

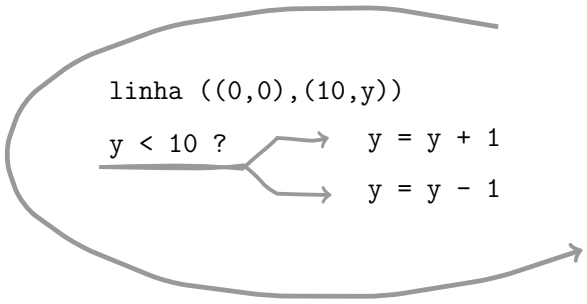
Mas, e agora?

Bom, agora seria legar ver a linha subindo e depois descendo — (*depois subindo, depois descendo*)

Só que isso é uma coisa bem fácil de fazer.

Quer dizer, basta a gente diminuir o valor do `y` para ver a linha descer.

E para isso, basta usar o outro lado da bifurcação



Só que isso não funciona muito bem ...

Porque?

Bom, porque a linha vai subir até o `y = 10`.

Daí, ela vai descer um pouquinho: `y = 9`.

E depois ela vai subir um pouquinho: `y = 10`.

E depois ela vai descer um pouquinho: `y = 9`.

E depois ela vai subir um pouquinho: `y = 10`.

Só que, não é isso o que a gente quer.

Quer dizer, a gente quer ver a linha subindo até lá em cima.

E depois descer até lá embaixo.

E depois subir até lá em cima.

E depois descer até lá embaixo.

Mas, isso não é uma coisa difícil de fazer.

Quer dizer, a ideia é que a linha sempre está indo em uma direção

```
dir = 1              # subindo
dir = 2              # descendo
```

E quando a linha chega em cima ou embaixo, ela muda de direção

```
if (y == 10):
    dir = 2          # para começar a descer
if (y == 0):
    dir = 1          # para começar a subir
```

Daí, na hora em que a gente vai mudar o valor de `y`, é preciso olhar para a direção que a gente está indo

Na linguagem de programação, a gente escreve os dois lados da bifurcação assim

```
if (y == 10):
    y = y + 1 * 0.1          # para fazer a linha subir
else:
    y = y - 1 * 0.1          # para fazer a linha descer
```

E agora, o programa completo fica assim

```
# Inicialização
y = 0                                          # lá embaixo
dir = 1                                        # e subindo
linhac ((0,0),(10,y),white)

# Repetição
while ( 1 == 1 ):
     sleep(1 * 0.1)                            # tira uma sonequinha
     linhac ((0,0),(10,y),black)               # apaga a linha
     if (dir == 1):                            # atualiza o y
          y = y + 1 * 0.1
     else:
          y = y + 1 * 0.1
     linhac ((0,0),(10,y),white)               # desenha a linha
     if (y == 10):
          dir = 2
     else:
          dir = 1
```

Legal, não é? ◇

b. Dando a volta

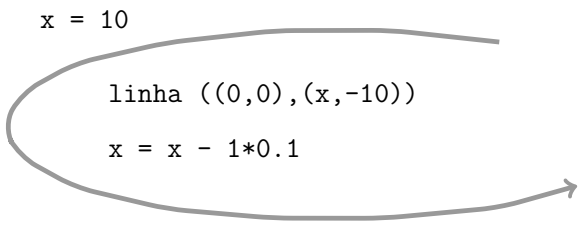
Mas, e agora?

Bom, agora a gente pode fazer a linha balançar para lá e para cá — (*como um pêndulo ...*)

Para isso, basta mexer na coordenada `x` — (*ao invés da coordenada* `y`)

Quer dizer, a linha vai começar no ponto $(10, -10)$.

Daí, a gente faz ela andar para a esquerda assim



E daí, quando a linha chega no ponto $(-10, 10)$, a gente faz ela voltar.

A coisa fica assim

```
# Inicialização
x = 10                                          # lá na direita
dir = 1                                         # indo para a esquerda
linhac ((0,0),(x,-10),white)

# Repetição
while ( 1 == 1 ):
     sleep(1 * 0.1)                             # tira uma sonequinha
     linhac ((0,0),(10,y),black)                # apaga a linha
     if (dir == 1):                             # atualiza o y
          x = x - 1 * 0.1
     else:
          x = x - 1 * 0.1
     linhac ((0,0),(10,y),white)                # desenha a linha
     if (x == -10):
          dir = 2
     else:
          dir = 1
```

Certo.

Agora nós queremos fazer a linha dar a volta completa.

Quer dizer, agora nós vamos mexer nas coordenadas `x` e `y`.

E nós vamos ter 4 direções de movimentação

- dir = 1: indo para a esquerda (diminuindo o x)
- dir = 2: subindo (aumentando o y)
- dir = 3: indo para a direita (aumentando o x)
- dir = 4: descendo (diminuindo o y)

A coisa fica assim

```
# Inicialização
x = 10                                  # canto inferior direito
y = -10
dir = 1                                 # indo para a esquerda
linhac ((0,0),(x,-10),white)

# Repetição
while ( 1 == 1 ):
     sleep(1 * 0.1)                     # tira uma sonequinha
     linhac ((0,0),(10,y),black)
     if (dir == 1):                     # esquerda
          x = x - 1 * 0.1
     elif (dir == 2):                   # subindo
          y = y + 1 * 0.1
     elif (dir == 3):                   # direita
          x = x + 1 * 0.1
     else:                              # descendo
          y = y - 1 * 0.1
     linhac ((0,0),(10,y),white)

     if (dir == 1 and x == -10):        # mudança de direção
          dir = 2
     if (dir == 2 and y == 10):
          dir = 3
     if (dir == 3 and x == 10):
          dir = 4
     if (dir == 4 and y == -10):
          dir = 1
```

Legal, não é? ◇

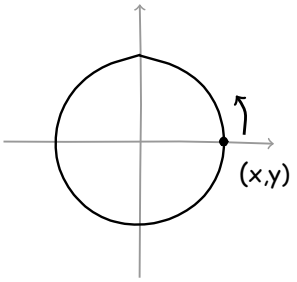
4 Desenhando bolinhas

Sim, mas esse negócio ainda está muito quadrado.

Quer dizer, seria mais legal se a linha desse a volta como o ponteiro do relógio — (*i.e., sempre com o mesmo tamanho*)

Mas agora a gente precisa pensar um pouquinho.

Para isso, a gente olha para um ponto rodando no círculo



Quer dizer, na hora em que o `x` anda para a esquerda, o `y` vai subindo — (*dir = 1*)

Até chegar lá em cima.

Porque daí, o `x` continua indo para a esquerda, mas o `y` vai descendo — (*dir = 2*)

Até chegar na esquerda.

Porque daí, o `x` começa a ir para a direito, e o `y` continua descendo — (*dir = 3*)

Até chegar lá embaixo.

Porque daí, o `x` continua a ir para a direito, mas o `y` vai subindo — (*dir = 4*)

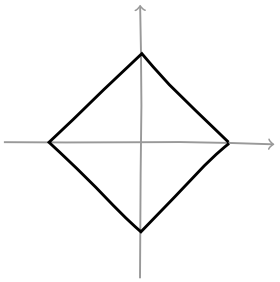
A coisa fica assim

```
# Inicialização
x = 10                                  # começa no lado direito
y = 0
dir = 1                                 # indo para a esquerda
linhac ((0,0),(x,-10),white)

# Repetição
while ( 1 == 1 ):
     sleep(1 * 0.1)                     # tira uma sonequinha
     linhac ((0,0),(10,y),black)
     if (dir == 1):                     # 1o quadrante
          x = x - 1 * 0.1
          y = y + 1 * 0.1
     elif (dir == 2):                   # 2o quadrante
          x = x - 1 * 0.1
          y = y - 1 * 0.1
     elif (dir == 3):                   # 3o quadrante
          x = x + 1 * 0.1
          y = y - 1 * 0.1
     else:                              # 4o quadrante
          x = x + 1 * 0.1
          y = y + 1 * 0.1
     linhac ((0,0),(10,y),white)
```

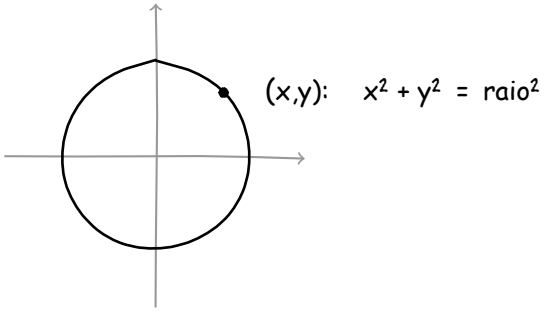
```
if (dir == 1 and x == -10):
     dir = 2
if (dir == 2 and y == 10):
     dir = 3
if (dir == 3 and x == 10):
     dir = 4
if (dir == 4 and y == -10):
     dir = 1
```

Mas a coisa não funcionou do jeito que a gente esperava



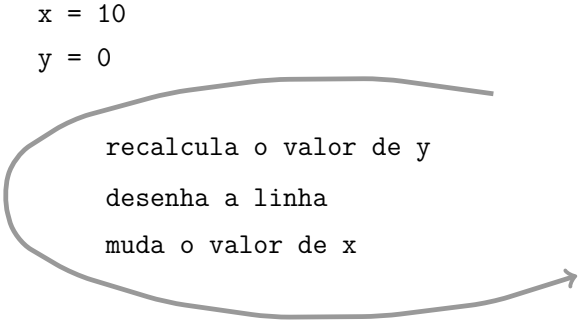
Porque?

Bom, porque os pontos do círculo funcionam assim



Quer dizer, é a soma dos quadrados de `x` e de `y` que é sempre a mesma coisa — (*i.e., igual ao quadrado do raio do círculo*)

Então agora, a gente pode utilizar a seguinte lógica



Quer dizer, o valor de y é sempre calculado para que a soma dos quadrados seja igual a 10^2 — (*o quadrado do raio*)

$$y \quad = \quad \sqrt{10^2 \quad - \quad x^2}$$

E a boa noticia é que a linguagem de programação tem o comando da raiz quadrada

$$y \quad = \quad \texttt{sqrt}\,(\,10 * 10 \quad - \quad x * x\,)$$

Daí, a coisa fica assim

```
# Inicialização
x = 10                                  # começa no lado direito
y = 0
dir = 1                                 # indo para a esquerda
linhac ((0,0),(x,-10),white)

# Repetição
while ( 1 == 1 ):
     sleep(1 * 0.1)                     # tira uma sonequinha
     linhac ((0,0),(10,y),black)
     if (dir == 1):                     # 1o quadrante
          x = x - 1 * 0.1
          y = sqrt(10*10 - x*x)
     elif (dir == 2):                   # 2o quadrante
          x = x - 1 * 0.1
          y = sqrt(10*10 - x*x)
     elif (dir == 3):                   # 3o quadrante
          x = x + 1 * 0.1
          y = - sqrt(10*10 - x*x)
     else:                              # 4o quadrante
          x = x + 1 * 0.1
          y = - sqrt(10*10 - x*x)
     linhac ((0,0),(10,y),white)
                                        # mudança de direção
     if (dir == 1 and x == -10):
          dir = 2
     if (dir == 2 and y == 10):
          dir = 3
     if (dir == 3 and x == 10):
          dir = 4
     if (dir == 4 and y == -10):
          dir = 1
```

Mas, você percebeu uma coisa engraçada?

Quer dizer, os casos `dir = 1` e `dir = 2` agora ficaram iguais — (*e os casos* `dir = 3` *e* `dir = 4` *também ficaram iguais*)

Então agora, a gente pode simplificar o programa assim

```
# Inicialização
x = 10                                  # começa no lado direito
y = 0
dir = 1                                 # indo para a esquerda
linhac ((0,0),(x,-10),white)

# Repetição
while ( 1 == 1 ):
     sleep(1 * 0.1)                     # tira uma sonequinha
     linhac ((0,0),(10,y),black)
     if (dir == 1):                     # dando a volta por cima
          x = x - 1 * 0.1
          y = sqrt(10*10 - x*x)
     else:                              # dando a volta por baixo
          x = x + 1 * 0.1
          y = - sqrt(10*10 - x*x)
     linhac ((0,0),(10,y),white)

     if (x == -10):                     # mudança de direção
          dir = 2
     if (x == 10):
          dir = 1
```

Mas a coisa ainda não está perfeita.

Porque a linha anda bem rápido na esquerda e na direita.

E anda bem devagar em cima e embaixo.

Porque?

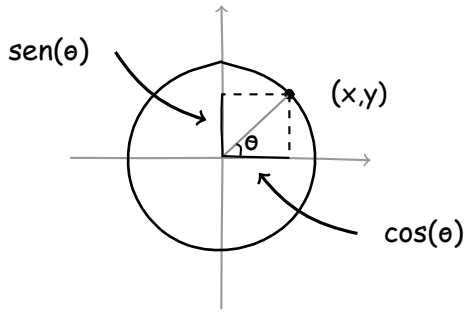
Bom, porque na esquerda e na direita quando o `x` anda um pouquinho, o `y` anda um montão.

E em cima e embaixo, quando o `x` anda um pouquinho, o `y` também anda só um pouquinho.

Certo.

Mas, como é que a gente conserta isso?

Bom, para isso é preciso olhar para o círculo outra vez.



E daí a gente olha para o ângulo θ que a nossa linha faz com o eixo X.

E vê que as coordenadas $(\mathtt{x}, \mathtt{y})$ correspondem a $\mathtt{cos}(\theta)$ e $\mathtt{sen}(\theta)$.

Então agora, ao invés de fazer o `x` andar um pouquinho, a gente vai fazer o ângulo θ andar um pouquinho.

E daí, a gente calcula as cordenadas $(\mathtt{x}, \mathtt{y})$ como $\mathtt{cos}(\theta)$ e $\mathtt{sen}(\theta)$.

Mais uma vez, a boa notícia é que a linguagem de programação tem os comandos para calcular o seno e o coseno

$$\mathtt{x} \;=\; \mathtt{cos}\,(\,\mathtt{teta}\,) \qquad \text{e} \qquad \mathtt{y} \;=\; \mathtt{sin}\,(\,\mathtt{teta}\,)$$

E a outra boa notícia, é que essa modificação deixa o programa ainda mais simples — (*com apenas um caso ...*)

A coisa fica assim

```
# Inicialização
teta = 0                                    # começa no lado direito
x = 10 * cos(teta)
y = 10 * sin(teta)
linhac ((0,0),(x,-10),white)

# Repetição
while ( 1 == 1 ):
      sleep(1 * 0.1)                        # tira uma sonequinha
      linhac ((0,0),(10,y),black)
      teta = teta + 1/90                    # aumenta o teta um pouquinho
      x = 10 * cos(teta)
      y = 10 * sin(teta)
      linhac ((0,0),(10,y),white)
```

Agora está perfeito.

E agora nós vamos ver mais alguns exemplos.

Exemplos

a. Vendo o círculo (e outras coisas)

A ideia agora é que a gente quer ver o círculo — (*e não apenas a linha rodando*)

Quer dizer, imagine que a gente coloca um lápis na ponta da linha.

Daí, a medida que a linha roda ela vai desenhando a linha no chão — (*ou na tela*)

Mas, como é que a gente faz isso?

Bom, aqui é preciso uma esperteza.

A ideia é a gente lembrar onde o ponto $(\texttt{x}, \texttt{y})$ estava na repetição anterior: $(\texttt{xant}, \texttt{yant})$.

Porque daí, na hora que o ponto anda, a gente pode fazer o risquinho

```
linhac ((xant,yant),(x,y))
```

E daí, quando a linha anda os risquinhos vão formando o círculo.

Mas não é só isso.

Quer dizer, agora que a gente vai ver o círculo, a gente não precisa mais ver a linha.

A coisa fica assim

```
# Inicialização
teta = 0
x = 10 * cos(teta)
y = 10 * sin(teta)
xant = x
yant = y

# Repetição
while ( 1 == 1 ):
      sleep(1 * 0.1)                    # tira uma sonequinha
      teta = teta + 1/90                # aumenta o teta um pouquinho
      x = 10 * cos(teta)
      y = 10 * sin(teta)
      linhac ((xant,yant),(x,y),white)
```

Legal.

Mas, e agora?

Bom, agora a gente pode fazer o raio aumentar um pouquinho, a medida que o círculo vai sendo desenhado.

Porque daí, ao invés de ver um círculo a gente vai ver uma espiral

A coisa fica assim

(. . . C O N T I N U A . . .)

◇