

PROGRAMAÇÃO PARA JOGOS I

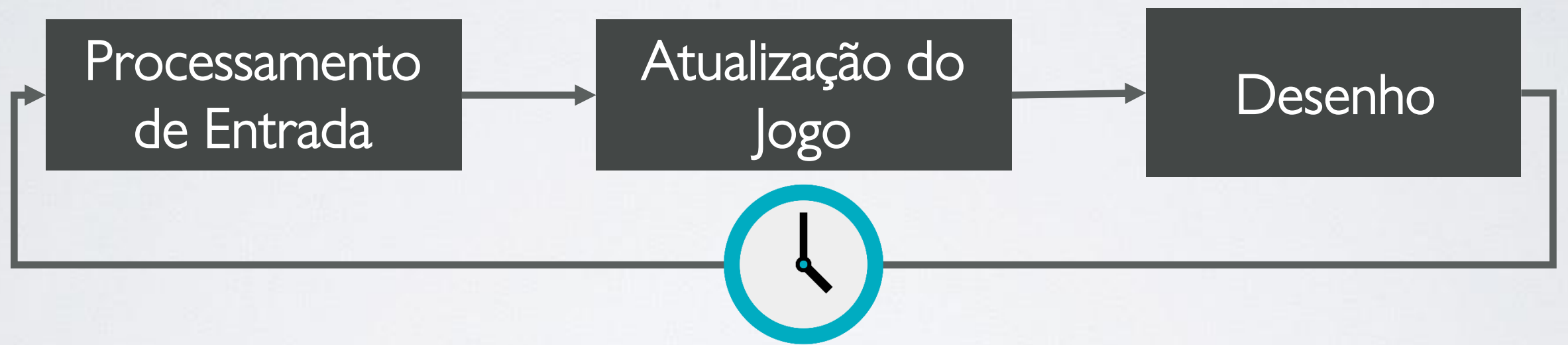
George Gomes

Vamos inserir animação no nosso Pong!

Mas antes vamos entender um pouco
sobre Game Loop, peça fundamental na
estrutura de um jogo

O que é um ciclo?

“ uma sequência de ações e tomadas de decisão programadas que se repetem dentro de um laço de repetição (ou loop)”, Humberto Rocha.



Processamento de Entrada

Entradas do usuário

Eventos gerados por dispositivos de entrada

Como um teclado e um mouse.

Atualização do Jogo

Resposta aos eventos processados e de outras mecânicas

Aplicação de gravidade, detecção de colisão, eventos programados dentre outros

Desenho

Retorno visual, o resultado, do processamento anterior

Renderização de sprites, efeitos, iluminação, etc.

Diferente de um software tradicional
que reage a uma interação, um jogo
executa continuamente ações
independente da entrada do usuário

No caso do nosso jogo Pong, a bolinha vai continuamente movimentar-se pelo cenário colidindo com outros elementos

Pra isso, vamos precisar de variáveis para
guardar a velocidade da bola

ballSpeedX

ballSpeedY

A cada iteração do *loop*, a posição do objeto *ball* é atualizada com a velocidade

```
# variáveis
```

```
ballSpeedX = 5
```

```
ballSpeedY = 5
```

```
while True:
```

```
    # Processando as entradas (eventos)
```

```
    for event in pygame.event.get():
```

```
        if event.type == pygame.QUIT:
```

```
            pygame.quit()
```

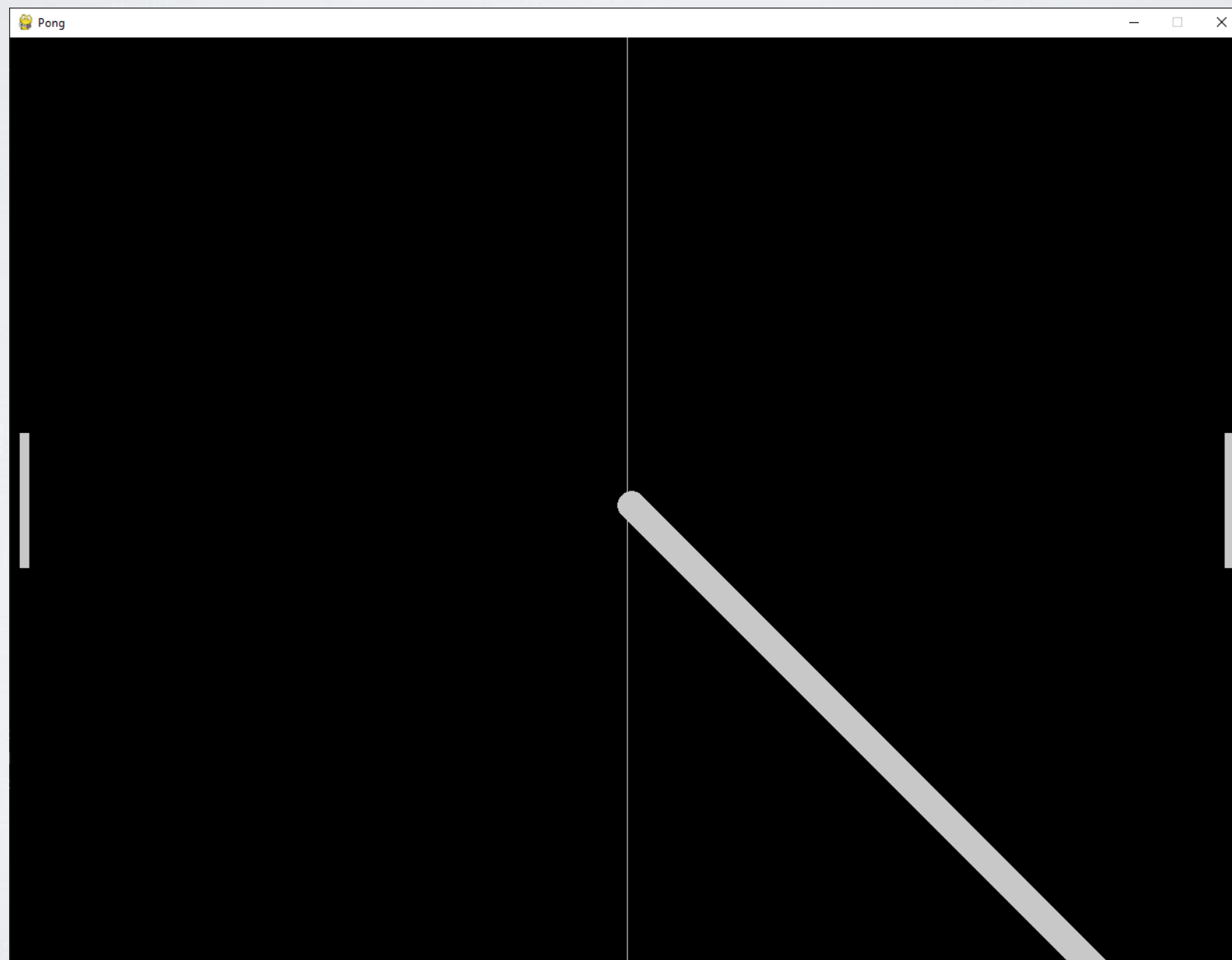
```
            sys.exit()
```

```
    # Atualização
```

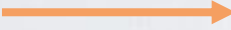
```
    ball.x = ball.x + ballSpeedX
```

```
    ball.y = ball.y + ballSpeedY
```

Precisamos limpar o desenho anterior para
desenhar um novo quadro

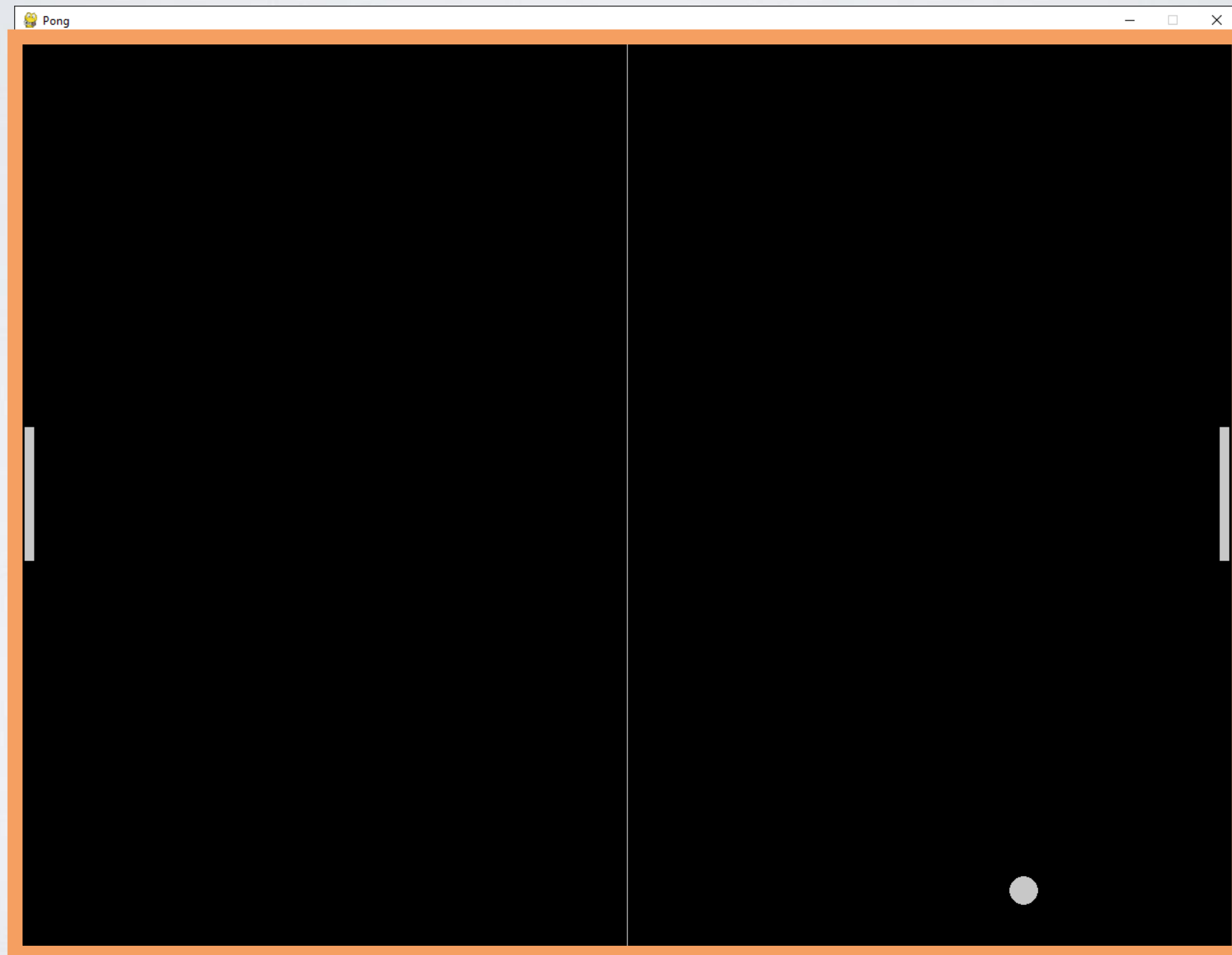


Apagar toda a tela (*screen*) antes de desenhar os elementos



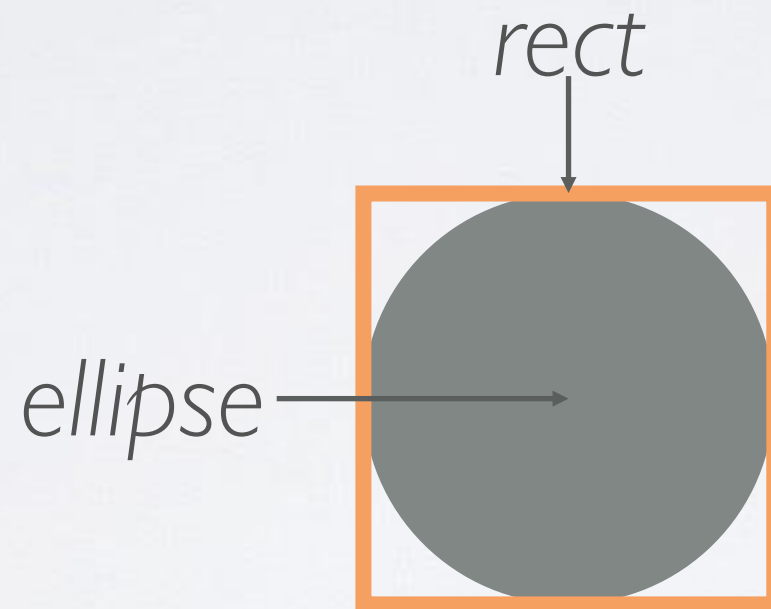
```
# Desenho
screen.fill((0, 0, 0))
pygame.draw.ellipse(screen, (200, 200, 200), ball)
pygame.draw.rect(screen, (200, 200, 200), player)
pygame.draw.rect(screen, (200, 200, 200), opponent)
pygame.draw.aaline(screen, (200, 200, 200), (1280/2, 0), (1280/2, 960))
```

Vamos fazer a bola “quicar” nas bordas da tela



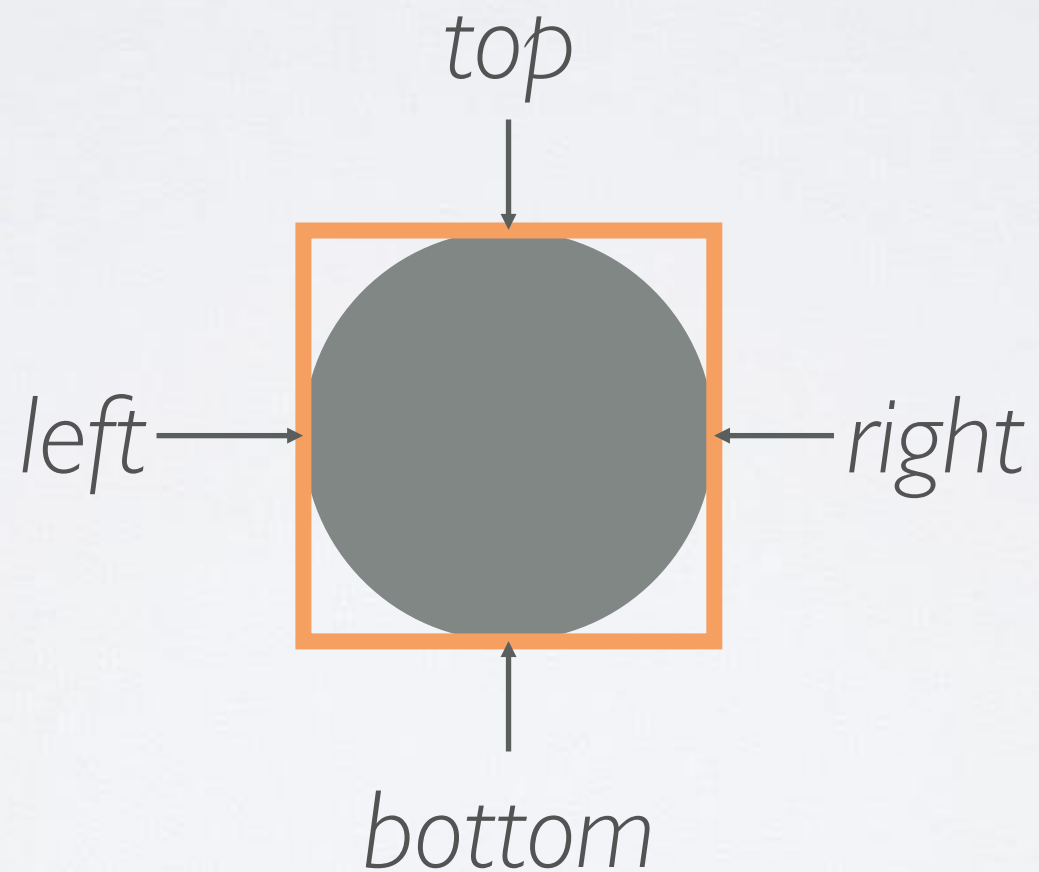
Antes vamos entender um pouco sobre o
objeto *ball*

ball é um objeto do tipo *rect* mas é desenhado como uma *ellipse*

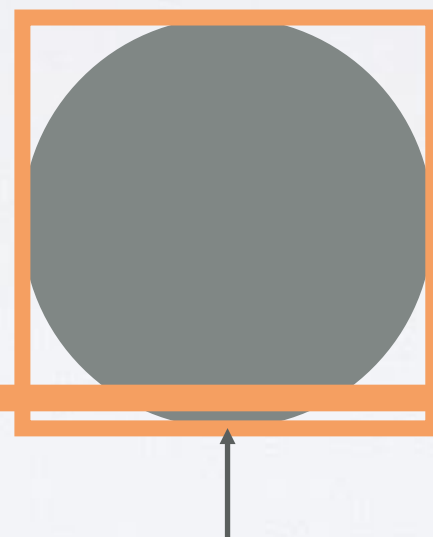


```
ball = pygame.Rect(1280 / 2 - 15, 960 / 2 - 15, 30, 30)  
pygame.draw.ellipse(screen, (200, 200, 200), ball)
```

Pygame disponibiliza formas de verificar
colisão pelos limites do retângulo



Para saber se a bola colidiu com a borda inferior é só verificar se *ball.bottom* \geq 960 (altura da tela)



bottom

Mas antes vamos organizar nosso código,
substituindo 1280 e 960 por variáveis
screenWidth e *screenHeight*

```
# Configurando a janela  
screenWidth = 1280  
screenHeight = 960  
screen = pygame.display.set_mode((screenWidth, screenHeight))
```

E nas outras linhas que usam a largura e altura

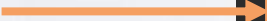
```
# Objetos
ball = pygame.Rect(screenWidth/2-15, screenHeight/2-15, 30, 30)
player = pygame.Rect(screenWidth-20, screenHeight/2-70, 10, 140)
opponent = pygame.Rect(10, screenHeight/2-70, 10, 140)
```

```
pygame.draw.aaline(screen, (200, 200, 200), (screenWidth/2, 0), (screenWidth/2, screenHeight))
```

Assim, nosso código fica mais 'elegante' quando testamos se a bola colidiu com a borda inferior

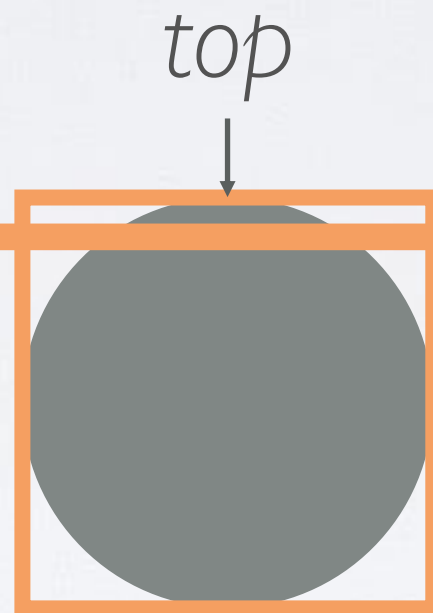
```
# Atualização
ball.x = ball.x + ballSpeedX
ball.y = ball.y + ballSpeedY

if ball.bottom >= screenHeight:
    ballSpeedY = ballSpeedY * -1
```



Precisamos testar também para a borda superior

Para saber se a bola colidiu com a borda superior
é só verificar de $ball.top \leq 0$



Assim, nosso código fica mais 'elegante' quando testamos se a bola colidiu com a borda inferior

```
# Atualização
```

```
ball.x = ball.x + ballSpeedX
```

```
ball.y = ball.y + ballSpeedY
```



```
if ball.top <= 0 or ball.bottom >= screenHeight:
```

```
    ballSpeedY = ballSpeedY * -1
```

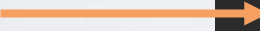
ballSpeed * -1 inverte a velocidade fazendo a bola quicar. Exemplo, se *ballSpeedX* = 5, ele tocar a borda direita, ele vai ficar com -5, fazendo ele mover para à direita

E também para o eixo X usando o *left*, *right*, *screenWidth* e *ballSpeedX*

```
# Atualização
ball.x = ball.x + ballSpeedX
ball.y = ball.y + ballSpeedY

if ball.top <= 0 or ball.bottom >= screenHeight:
    ballSpeedY = ballSpeedY * -1

if ball.left <= 0 or ball.right >= screenWidth:
    ballSpeedX = ballSpeedX * -1
```



Agora eu tenho a bola quicando em
todas as bordas



Agora vamos analisar a velocidade de movimento da bola

Comente a linha que controla o FPS

```
# Atualizando a janela 60fps  
pygame.display.flip()  
→ # clock.tick(60)
```

Agora a bola está se movimentando na velocidade que seu computador suporta

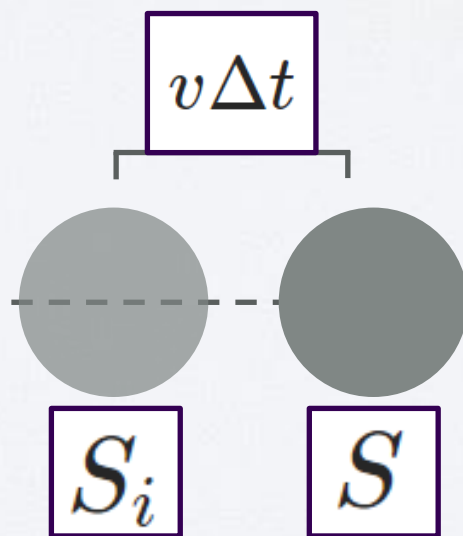
Não é o indicado para o desenvolvimento de
jogos

Como garantir que a bola terá a mesma velocidade independente do *hardware*?

Vamos uma breve lembrança de conceitos
de física básica

MRU (Movimiento Retilíneo Uniforme)
garante uma velocidade constante através da
fórmula:

$$S = S_i + v\Delta t$$



Vamos criar um novo projeto para usar um
código bem simples para entendermos
melhor

○ código abaixo mostra uma bola deslocando 100 pixels por segundo

```
1 import time
2 import pygame
3 pygame.init()
4 screen = pygame.display.set_mode((640, 480))
5 positionX = 0
6 speedX = 100 # 100 pixels por segundo
7 ti = time.time() # captura o tempo inicial
8 while True:
9     if pygame.event.poll().type == pygame.QUIT:
10         break
11     tf = time.time() # captura o tempo deste ciclo
12     dt = (tf - ti) # calcula o delta
13     ti = tf # atribui o tempo final como tempo inicial
14     positionX += speedX * dt # move o quadrado na velocidade média definida
15     screen.fill((0, 0, 0))
16     pygame.draw.ellipse(screen, (255, 255, 255), [positionX, 230, 20, 20])
17     pygame.display.flip()
```

Este código não depende do poder de processamento da máquina, pois ele mede quanto tempo demorou de um ciclo do *loop* para o próximo através do *delta* (*dt*)

No entanto, apesar de manter a movimentação consistente independente do poder computacional da máquina, esse código apresenta um problema

O *loop* está sendo executado sem controle, sem sincronização. Ele pode está sendo executado mais vezes do que o necessário

Para isso, incluimos o conceito de quadros por segundo (*Frames Per Second* - PFS). Assim, teremos sincronização e controle do tempo

Benefícios

Reduz o uso desnecessário de recursos

Facilita a sincronização de jogos multiplayer

Diminui a propagação de erro em operação de ponto flutuante

Aumenta a previsibilidade e facilita o planejamento do seu jogo

Vamos alterar nosso código para ter controle do *fps* pelo *clock.tick()*, uma função que calcula o intervalo de tempo para *dt*, no caso 1/60 (60fps)

```
1  import time
2  import pygame
3  pygame.init()
4  screen = pygame.display.set_mode((640, 480))
5  positionX = 0
6  speedX = 0.1 # 100px/1000ms
7  clock = pygame.time.Clock()
8  while True:
9      if pygame.event.poll().type == pygame.QUIT:
10         break
11     dt = clock.tick(60) # 60fps
12     positionX += speedX * dt # move o quadrado na velocidade média definida
13     screen.fill((0, 0, 0))
14     pygame.draw.ellipse(screen, (255, 255, 255), [positionX, 230, 20, 20])
15     pygame.display.flip()
```

Quantos segundos a bola demora para atravessar a tela com a velocidade de 0.1 (100 pixels / 1000 ms)? E com 0.2?

Agora vamos mudar o *fps* de 60 para 10. Com a velocidade 0.1, o tempo para a bola atravessar a tela aumentou?

Curiosidade, experimente ver a relação fps x
velocidade em

<https://www.testufo.com/>

Vamos voltar para nosso projeto do Pong, o que precisamos modificar no código para ter controle de fps e velocidade consistente?

Basicamente é incluir a variação de tempo (delta - dt) no cálculo do deslocamento

Vamos alterar nosso código para ter controle do *fps* pelo *clock.tick()*, uma função que calcula o intervalo de tempo para *dt*, no caso 1/60 (60fps)

```
# variáveis
ballSpeedX = 0.5 # 0.1 => 100 pixels/ 1000ms
ballSpeedY = 0.5 # 0.1 => 100 pixels/ 1000ms

while True:
    # Processando as entradas (eventos)
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

    # Atualização
    dt = clock.tick(60)
    ball.x = ball.x + ballSpeedX * dt
    ball.y = ball.y + ballSpeedY * dt
```

Tudo funcionando direitinho, mas nosso *Game Loop* ainda não está pronto. Ele ainda não garante consistência do movimento para máquinas muito lentas. Trabalharemos nele nas próximas aulas.

Qual o próximo passo? Trabalhar a Colisão e pontuação (próximas aulas).

Mas antes de acabarmos nossa aula. Gostaria que vocês pesquisasassem e compartilhassem no Discord, como é possível verificar a colisão entre dois *rects* no Pygame? Existe mais de uma forma?

PROGRAMAÇÃO PARA JOGOS I

George Gomes