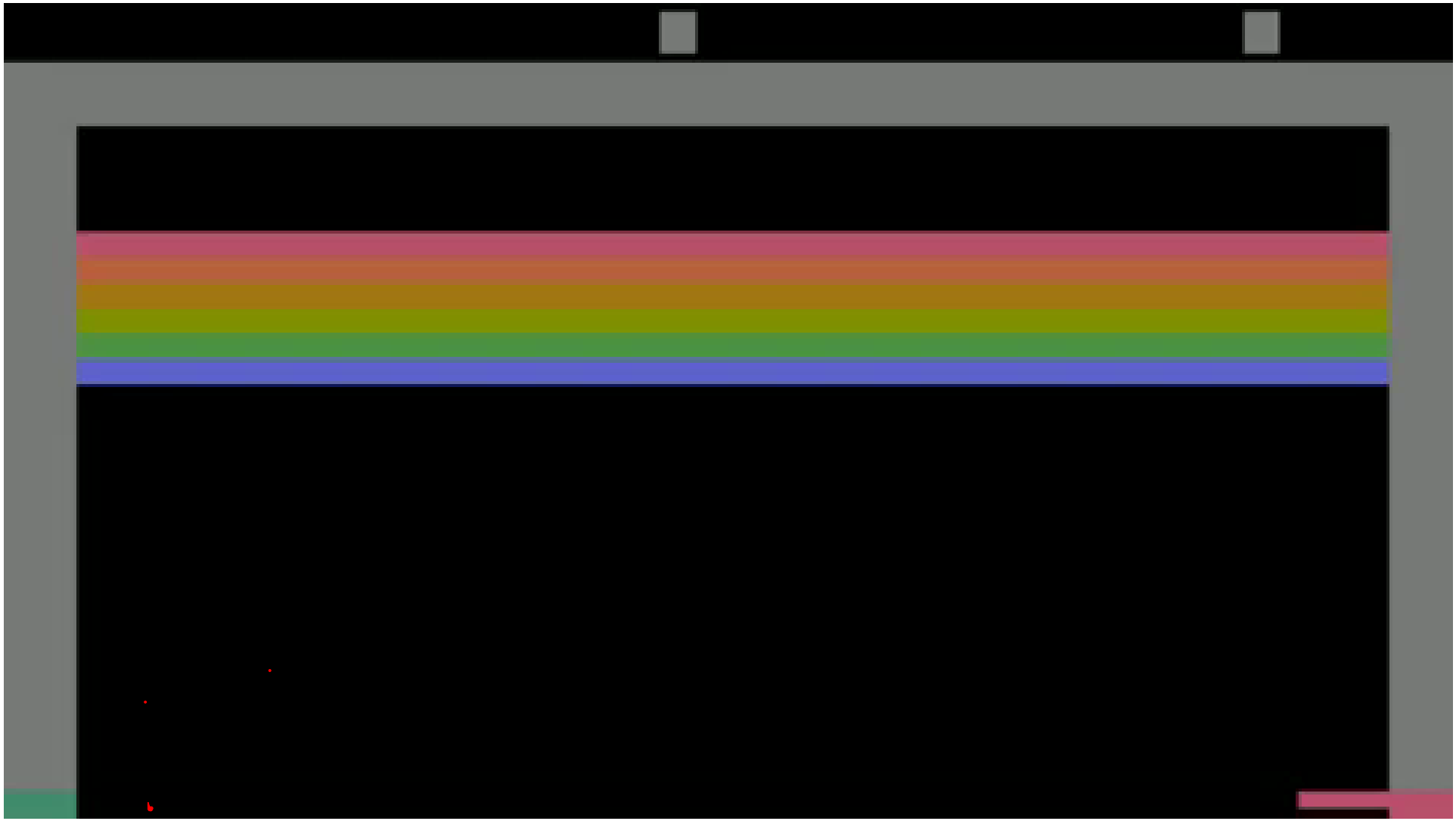
Computação II (MAB 225)

Fabio Mascarenhas - 2015.1

http://www.dcc.ufrj.br/~fabiom/pythonoo

Breakout



Coordenação

- Tanto no breakout como em outros jogos, precisamos verificar possíveis colisões entre os objetos do jogo, e tomar ações a depender de qual objeto colidiu com qual
    - Ex: *se a bola colide com um tijolo, o tijolo some e a bola é refletida*
- Verificar uma interação envolvendo campos de dois (ou mais) objetos diferentes, e disparar ações em todos eles, é um problema da modelagem OO
- De quem é a responsabilidade de *coordenar* essa interação?

Coordenação, cont.

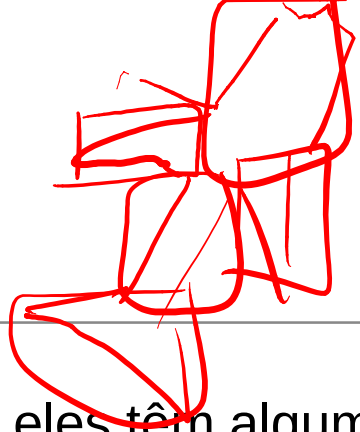
- Podemos eleger um dos objetos que estão participando da interação para ser o coordenador, mas isso aumenta o *acoplamento* entre os objetos que estão interagindo

- Ou podemos usar um mediador, um objeto que vai verificar se houve alguma interação, e mandar os objetos envolvidos tomarem uma ação

- O mediador precisa ter acesso ao estado dos objetos que estão interagindo, mas o acoplamento entre esses objetos diminui

- Vamos usar a instância de `Breakout` como mediador em nosso exemplo

Testando colisões



- Testamos se dois elementos do jogo colidiram testando se eles têm alguma interseção

- Jogos costumam simplificar esse problema com uma convenção, assumindo que todos os elementos são retângulos, independente da forma real deles: esses retângulos são as *caixas de colisão* ou *hitbox*

- Um elemento pode até ter mais de uma caixa de colisão, representando diferentes áreas dele, e elas vão acompanhando ele à medida que ele se move pela tela

- Podemos modelar caixas de colisão com uma classe própria, e nessa classe implementar a lógica para testar a interseção entre duas caixas de colisão

A classe `Hitbox`

- A caixa de colisão deve poder ser movida, e poder testar se ela colidiu com outra caixa de colisão (e em que lado dessa outra caixa):

```
class Hitbox:
    def __init__(self, x, y, l, a):
        self.x0 = x
        self.x1 = x + l
        self.y0 = y
        self.y1 = y + a

    def mover(self, dx, dy):
        self.x0 += dx
        self.y0 += dy
        self.x1 += dx
        self.y1 += dy
```

```
def intersecao(self, hb):
        w = ((self.x1 - self.x0) + (hb.x1 - hb.x0)) / 2.0
        h = ((self.y1 - self.y0) + (hb.y1 - hb.y0)) / 2.0
        dx = ((self.x1 + self.x0) - (hb.x1 + hb.x0)) / 2.0
        dy = ((self.y1 + self.y0) - (hb.y1 + hb.y0)) / 2.0
        if abs(dx) <= w and abs(dy) <= h:
            wy = w * dy
            hx = h * dx
            if wy > hx:
                if wy > -hx:
                    return "acima"
                else:
                    return "direita"
            else:
                if wy > -hx:
                    return "esquerda"
                else:
                    return "abaixo"
        else:
            return ""
```

Caixas de colisão no Breakout

- Cada tijolo tem uma caixa de colisão que ocupa o tijolo todo
- A bola tem uma caixa de colisão ligeiramente menor que ela
- A raquete tem três caixas de colisão diferentes, cada uma rebatendo a bola de um jeito
- Usamos também uma caixa de colisão para cada parede, e uma para o “chão”

Coordenando as colisões

- Para saber se as colisões aconteceram o coordenador usa as caixas de colisão e a velocidade da bola
- Se alguma colisão aconteceu o coordenador toma a ação apropriada
- Caso a bola precise mudar de direção o coordenador *delega* essa tarefa à instância de Bola, ao invés de mudar diretamente a velocidade
- Em um primeiro momento a lógica do método `tique` do coordenador vai ficar muito grande: isso é um sinal de que devemos *refatorar* esse método em diferentes métodos que cuidam de cada parte da lógica de atualização do jogo

Método `tique` – antes

```
def tique(self, dt, teclas):
        if self.pausa or self.game_over: return
        self.bola.animar(dt)
        if self.raquete.hb_e.intersecao(self.bola.hb):
            self.bola.vx = +abs(self.bola.vx)
            self.bola.vy = -abs(self.bola.vy)
        if self.raquete.hb_c.intersecao(self.bola.hb):
            self.bola.vy = -abs(self.bola.vy)
        if self.raquete.hb_d.intersecao(self.bola.hb):
            self.bola.vx = abs(self.bola.vx)
            self.bola.vy = -abs(self.bola.vy)
        if self.hb_par_e.intersecao(self.bola.hb):
            self.bola.vx = abs(self.bola.vx)
        if self.hb_par_c.intersecao(self.bola.hb):
            self.bola.vy = abs(self.bola.vy)
        if self.hb_par_d.intersecao(self.bola.hb):
            self.bola.vx = -abs(self.bola.vx)
```

```
remover = []
for tijolo in self.tijolos:
    lado = tijolo.hb.intersecao(self.bola.hb)
    if lado:
        remover.append(tijolo)
        self.score += 10
        if lado == "acima":
            self.bola.vy = -abs(self.bola.vy)
        elif lado == "abaixo":
            self.bola.vy = abs(self.bola.vy)
        elif lado == "esquerda":
            self.bola.vx = -abs(self.bola.vx)
        else:
            self.bola.vx = abs(self.bola.vx)
for tijolo in remover:
    self.tijolos.remove(tijolo)
if self.hb_chao.intersecao(self.bola.hb):
    self.game_over = True
```

Princípios de projeto OO

- Estamos procurando seguir dois princípios básicos do projeto de programas OO
- O primeiro diz que métodos de um objeto não devem modificar diretamente campos de outro objeto
- O segundo diz que métodos devem ser curtos e terem uma função bem clara
- Numa primeira implementação podemos violar esses princípios, mas depois sempre devemos voltar e procurar resolver essas violações criando novos métodos e delegando para eles

Método `tique` – depois

```
def tique(self, dt, teclas):
        if self.pausa or self.game_over: return
        self.bola.animar(dt)
        self.colisao_bola_raquete()
        self.colisao_bola_paredes()
        self.colisao_bola_tijolos()
        self.colisao_bola_chao()


def colisao_bola_raquete(self):
        if self.raquete.hb_e.intersecao(self.bola.hb):
            self.bola.para_cima()
            self.bola.para_esquerda()
        if self.raquete.hb_c.intersecao(self.bola.hb):
            self.bola.para_cima()
        if self.raquete.hb_d.intersecao(self.bola.hb):
            self.bola.para_cima()
            self.bola.para_direita()
```

Composição

- *Composição* é a ferramenta principal da modelagem OO: objetos são compostos por outros objetos

- A composição anda de mãos dadas com a *delegação*: um objeto deve sempre delegar parte da implementação de suas operações para suas partes

  - Em geral, se estamos usando apenas os campos de um objeto, está faltando delegação na modelagem

- Estamos usando composição e delegação desde o início em nossos exemplos

Um timer para o Breakout

- Como mais um exemplo de composição e delegação, vamos adicionar um timer de minutos e segundos ao Breakout

- O timer começa em 05:00, e se chegar a 00:00 o jogo termina

- O timer será uma instância de `Timer`, que por sua vez será uma composição de duas instâncias de `Segmento`, uma para os minutos e uma para os segundos

Segmento e Timer

```
class Segmento:
    def __init__(self, valor):
        self.valor = valor

    def zerado(self):
        return self.valor == 0

    def tique(self):
        self.valor = (self.valor - 1) % 60
        return self.valor == 59

    def texto(self):
        return "%02d" % (self.valor)
```

```
class Timer:
    def __init__(self, x, y, minutos, segundos):
        self.x = x
        self.y = y
        self.segundos = Segmento(segundos)
        self.minutos = Segmento(minutos)
        self.tempo = 0.0

    def tique(self, dt):
        self.tempo = self.tempo + dt
        if self.tempo >= 1.0:
            self.tempo = self.tempo - 1.0
            if self.segundos.tique():
                self.minutos.tique()
        return (self.minutos.zerado() and
                self.segundos.zerado())

    def desenhar(self, tela):
        tela.texto(self.x, self.y,
                   ("%s:%s" %
                    (self.minutos.texto(),
                     self.segundos.texto())))
```