Computação II (MAB 225)

Fabio Mascarenhas - 2015.1

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

Orientação a Objetos

- Orientação a objetos é um paradigma de programação no qual estruturamos um programa como uma teia de *objetos* que se comunicam entre si através de *mensagens*

- As mensagens às quais um objeto responde formam a sua *interface*

- Dizemos quais mensagens um objeto responde definindo *métodos* na sua classe

- Mandamos uma mensagem para um objeto chamando um de seus métodos

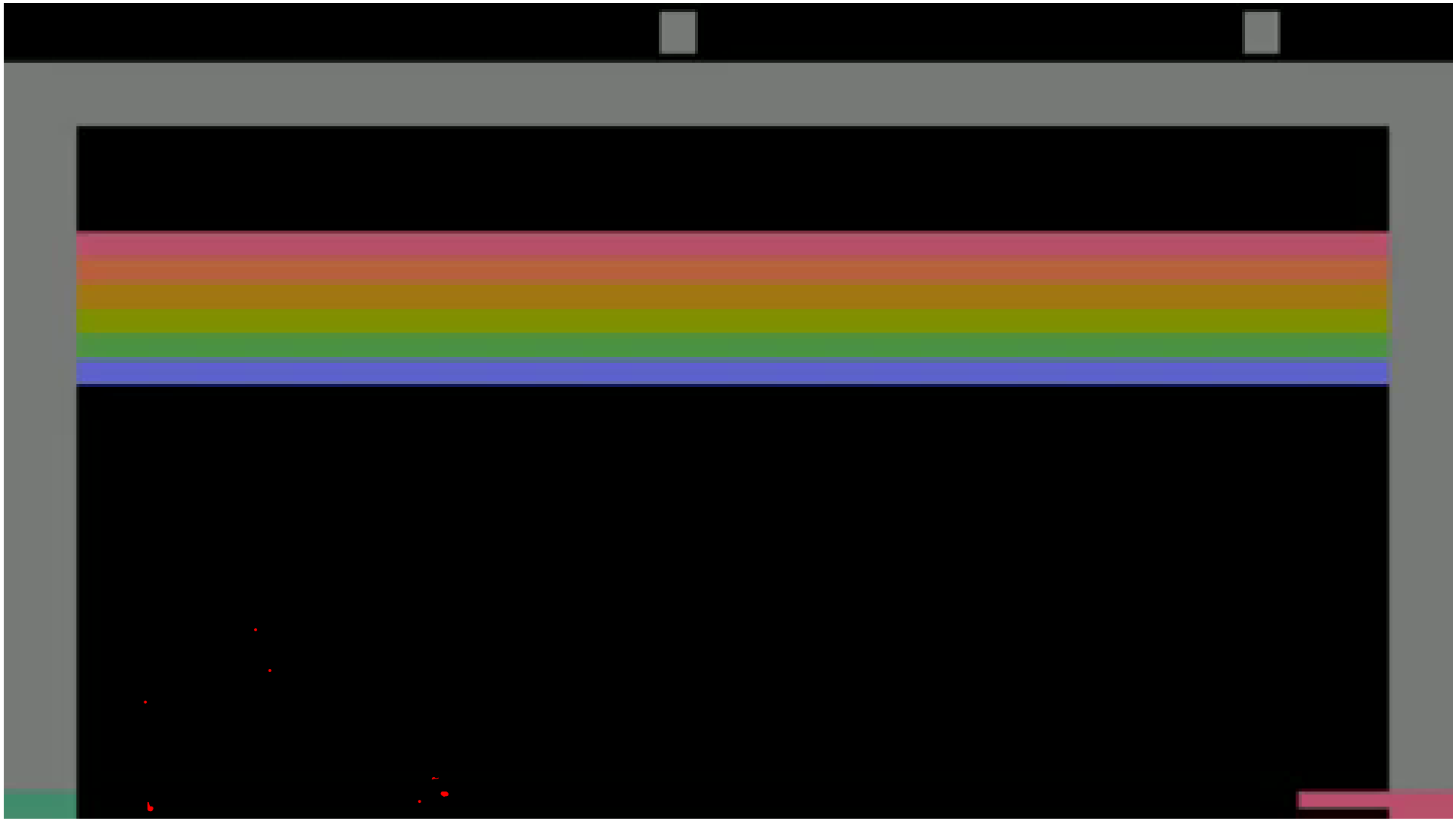
Frameworks

- É bastante comum que uma aplicação OO seja construída para usar um *framework* (arcabouço)
- Um framework é como uma máquina com algumas peças faltando, que serão fornecidas pela aplicação
- Essas “peças faltando” são instâncias de classes cuja estrutura é ditada pelas necessidades do framework
- Os objetos do framework interagem com os objetos da aplicação, que por sua vez interagem de volta com objetos do framework, para requisitar serviços

Um framework simples para jogos

- Vamos usar um framework bem simples para construir jogos 2D
- O framework fornece uma *tela* para desenhar figuras geométricas simples, além de texto
- A cada “tique” do relógio interno do framework, ele fornece uma tela vazia com fundo preto
- Ele também avisa a aplicação de *eventos* que acontecem: teclas pressionadas, e a própria passagem do tempo

Breakout



Componentes do Breakout

- Bola
- “Raquete”
- Tijolos
- Paredes
- Score
- Nem todos vão precisar de classes próprias para representa-los!

Bola

- Representamos a bola com uma posição e uma velocidade

- Tanto posição quanto a velocidade têm um componente horizontal e um vertical

```
class Bola:
    def __init__(self, x, y, vx, vy):
        self.x = x
        self.y = y
        self.vx = vx
        self.vy = vy
        self.raio = 5
        self.cor = (1.0, 1.0, 1.0)
```

- Tanto o raio quanto a cor da bola são fixos; a posição será do centro da bola

Raquete

- A raquete tem uma posição, um tamanho, uma cor, e uma velocidade horizontal que será 0 se ela estiver em repouso, ou algum valor quando ela estiver se movendo:

```
class Raquete:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.cor = (1.0, 1.0, 1.0)
        self.larg = 100
        self.alt = 30
        self.vx = 0
```

Tijolos

- Cada tijolo tem uma cor, uma posição e um tamanho

- Quando criamos um tijolo dizemos qual a cor dele, já que cada tijolo pode ter uma cor

```
class Tijolo:
    def __init__(self, x, y, cor):
        self.x = x
        self.y = y
        self.cor = cor
        self.larg = 50
        self.alt = 30
```

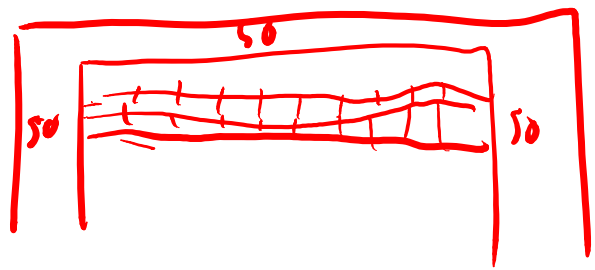
Paredes e Score

- As paredes são fixas, enquanto o score é só um valor escalar, então podemos manter os dados necessários para ambos na própria classe que representa o estado do jogo

- Nessa classe também instanciamos os objetos iniciais: a bola, a raquete, e os tijolos

```
class Breakout:
    def __init__(self):
        self.TITULO = "Breakout"
        self.LARGURA = 800
        self.ALTURA = 600
        self.larg_parede = 50
        self.score = 0
        self.bola = Bola(400, 565, 50, 50)
        self.raquete = Raquete(350, 570)
        self.tijolos = []
        for i in range(0, 14):
            for j in range(0, 6):
                self.tijolos.append(Tijolo(i * 50 + 50, j * 30 + 100, cor_aleat()))
```

} exigidos pelo framework



Métodos

- Um *método* é uma operação de um objeto
- Sintaticamente, um método se parece com uma função: é declarado dentro de uma classe possui uma lista de parâmetros e um bloco de comandos, e pode retornar valores
- A única diferença sintática que o primeiro parâmetro de um método sempre é `self`
- Para chamar um método, precisamos dizer qual objeto vai *receber* a chamada: `<obj_rec>.metodo(<arg1>, ..., <argn>)`
- Dentro de um método, o parâmetro `self` é o objeto que está recebendo a chamada, e podemos ter acesso aos campos e métodos desse objeto

Interação com o motor de jogo

- O motor interage com nosso jogo mandando mensagens para o objeto principal do jogo, ou seja, chamando seus métodos

- O método `desenhar` avisa ao jogo que ele deve desenhar um quadro da sua interface; como parâmetro, o jogo recebe um objeto que representa a tela de desenho

- O método `tique` avisa ao jogo da passagem de tempo, para ele atualizar seu estado interno; como parâmetros, o jogo recebe quantos segundos se passaram, e quais teclas o jogador está pressionando no momento

- O método tecla avisa do jogo que uma tecla foi solta, informando qual foi essa tecla

O objeto `tela`

- O objeto tela responde a cinco métodos de desenho:

```
def linha(self, x1, y1, x2, y2, larg = 5, cor = (1.0, 1.0, 1.0))
def retangulo(self, x, y, larg, alt, cor = (1.0, 1.0, 1.0), borda = 0)
def elipse(self, x, y, larg, alt, cor = (1.0, 1.0, 1.0), borda = 0)
def triangulo(self, x1, y1, x2, y2, x3, y3, cor = (1.0, 1.0, 1.0), borda = 0)
def texto(self, x, y, s, cor = (1.0, 1.0, 1.0))
```

- A posição para retângulos é a do canto superior esquerdo; para elipses e texto, a posição é a do canto superior esquerdo do retângulo que os circunscreve

- O parâmetro borda é 0 para indicar uma figura preenchida, ou o número de pixels da borda para uma figura vazada

- A cor é uma tripla com os componentes vermelho, verde e azul, onde 1.0 é a intensidade máxima

Desenhando o jogo

- Começamos pelo método `desenhar`
- Poderíamos desenhar tudo acessando os campos dos nossos objetos e chamando os métodos apropriados no objeto `tela`, mas isso não é um bom projeto
- Saber se desenhar deve ser responsabilidade de cada um dos objetos do jogo
- Fazemos isso definindo métodos `desenhar` em cada uma das classes do jogo: `Bola`, `Raquete` e `Tijolo`, e *delegando* a tarefa de desenhar para as instâncias dessas classes

Conectando o motor ao jogo

- Para conectar o motor ao jogo, preciamos importar a motor:

```
import motor
```

- Uma vez que tenhamos uma instância da classe principal do nosso jogo, passamos essa instância para a função rodar do motor:

```
jogo = Breakout()
motor.rodar(jogo)
```

- Se o jogo tiver algum bug a janela da sua interface pode ficar congelada; não se preocupe, basta editar e rodar de novo o programa do jogo que essa janela será fechada a uma nova aberta

Campos da classe

- E se queremos que **todos** os objetos de determinada classe tenham um campo que sempre vai ter o **mesmo** valor?

- No nosso jogo, temos como exemplo disso a largura e altura dos tijolos e da raquete

- Nesse caso, usamos um *campo da classe*, declarado ele no corpo da classe:

```
class Tijolo:
    larg = 50
    alt = 30
```

- Acessamos um campo da classe diretamente pelo nome da classe:

  - `Tijolo.larg, Raquete.alt`

Interagindo com o usuário

- Vamos ter dois tipos de interação com o usuário: segurar a seta esquerda move a raquete para a esquerda, segurar a seta direita move a raquete para a direita, apertar a tecla de escape reinicia o jogo

- Verificamos o estado das setas respondendo ao método `tique`:

```
def tique(self, dt, teclas):
        if "left" in teclas and "right" not in teclas:
            self.raquete.esquerda()
        if "left" not in teclas and "right" in teclas:
            self.raquete.direita()
```

- Verificamos a tecla de escape respondendo ao método `tecla`:

```
def tecla(self, t):
        if t == "escape":
            self.reset()
```

- Note que em ambos os casos delegamos o efeito no estado do jogo a outros métodos

Animando a bola e raquete

- Podemos completar o método tique para fazer o movimento da raquete e da bola, novamente delegando isso para esses objetos:

```
self.raquete.mover(dt)
self.bola.mover(dt)
```

- O movimento em si é uma linha de código no caso da raquete, e duas linhas para a bola, mas é um bom projeto sempre delegar para o objeto qualquer mudança em seu estado interno

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

Finalizando

- A versão inicial do Breakout está quase pronta, faltando apenas implementar as colisões

- Na próxima aula veremos como fazer isso, e também faremos algumas extensões ao funcionamento do jogo