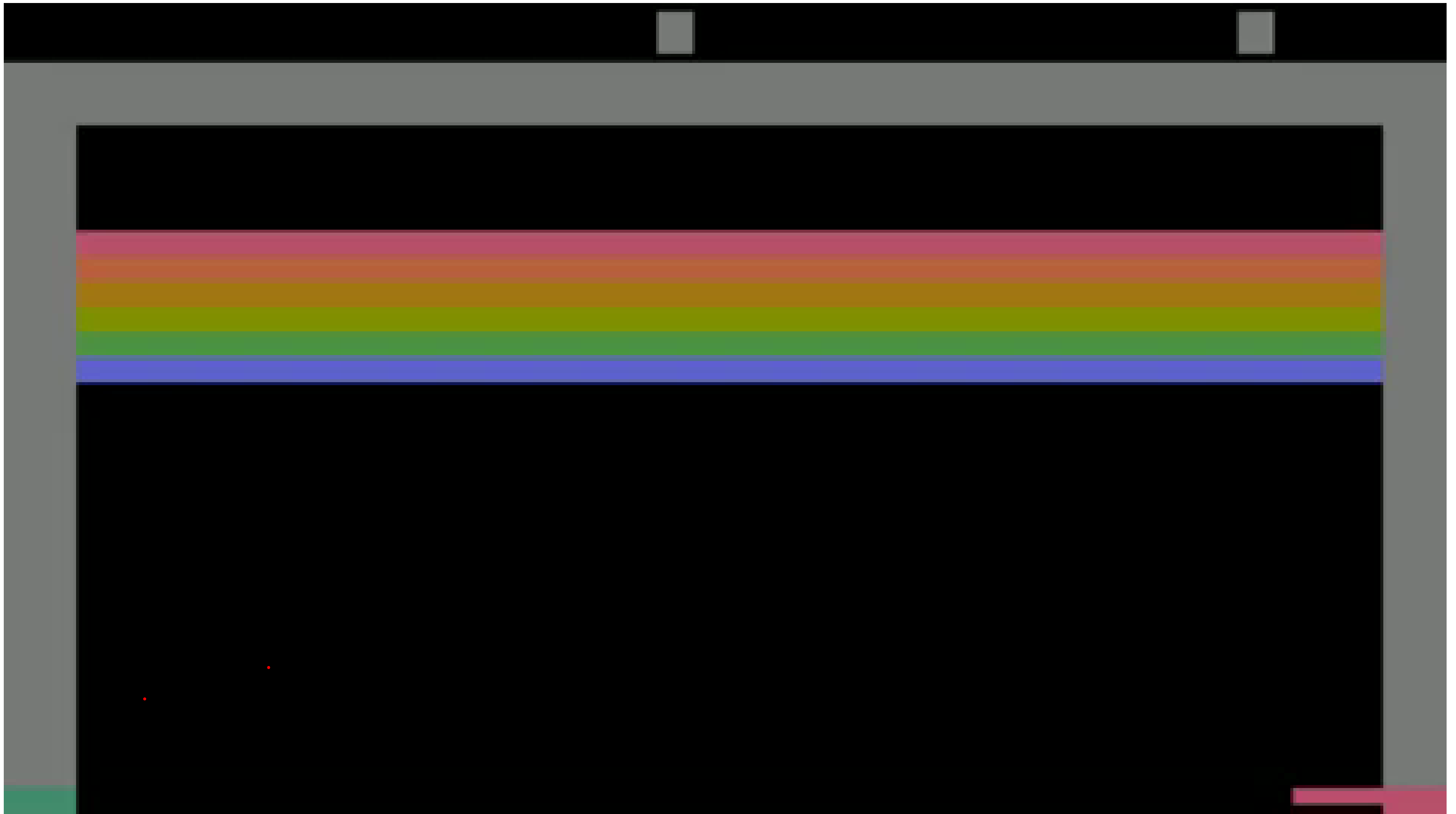
Computação II – Orientação a Objetos

Fabio Mascarenhas - 2016.1

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

Breakout



Componentes do Breakout

- Bola : posição, velocidade, cor, raio
- "Raquete" : posição, cor, tamanho
- Tijolos : posição, cor, tamanho
- Paredes
- Score e vidas
- Nem todos vão precisar de classes próprias para representa-los!

Bola

- Representamos a bola com uma posição e uma velocidade

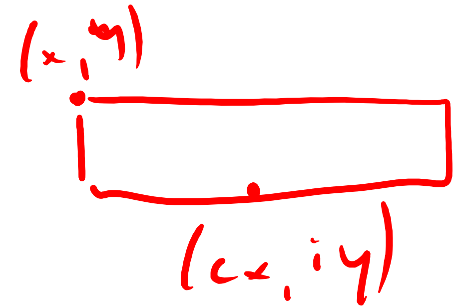
- Tanto posição quanto a velocidade têm um componente horizontal e um vertical

```
public class Bola {
    double x;
    double y;
    double vx;
    double vy;
    Cor cor;
    int raio = 8;
    ...
}
```

- O raio da bola é fixo, então poderíamos usar uma variável global também

Raquete

- A raquete tem uma posição, um tamanho e uma cor:



```
public class Raquete {
    double x;
    double y;
    Cor cor;
    int larg = 100;
    int alt = 20;
    public Raquete(double cx, double iy, Cor _cor) {
        x = cx - larg/2;
        y = iy - alt;
        cor = _cor;
    }
    ...
}
```

- As coordenadas da x e y são do canto superior esquerdo, mas criamos a raquete passando as coordenadas do ponto no centro da face de baixo

Tijolos

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

```
public class Tijolo {
    double x;
    double y;
    Cor cor = new Cor(Math.random(), Math.random(), Math.random());
    static int larg = 64;
    static int alt = 20;
    ...
}
```

- Todos os tijolos terão sempre o mesmo tamanho, e precisamos dessa informação na hora de posicionar os tijolos, então usamos variáveis globais

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

```
public class Jogo {
    Bola b;
    Raquete r;
    Tijolo[] ts;
    int score;
    int vidas;
    boolean gameOver;
    ...
}
```

```
public Jogo() {
   b = new Bola(getLargura()/2, getAltura()/2, 200, 200,
                                        Cor.AMARELO);
   r = new Raquete(getLargura()/2, getAltura(),
                                   Cor.BRANCO);
   int ts_por_lin = getLargura()/Tijolo.larg - 2;
   ts = new Tijolo[ts_por_lin * 6];
   for(int i = 0; i < ts.length; i++ ) {
       int lin = i / ts_por_lin;
       int col = i % ts_por_lin;
       ts[i] = new Tijolo(col * Tijolo.larg + Tijolo.larg,
                          lin * Tijolo.alt + 200);
   }
   vidas = 3;
}
```

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:

```
public void triangulo(double x1, double y1,
        double x2, double y2, double x3, double y3, Cor cor)
public void circulo(double cx, double cy, int raio, Cor cor)
public void quadrado(double x, double y, int lado, Cor cor)
public void retangulo(double x, double y, int largura, int altura, Cor cor)
public void texto(String texto, double x, double y, int tamanho, Cor cor)
```

- A posição para retângulos é a do canto superior esquerdo; para o círculo é a do centro, para triângulos dos vértices, e para o texto do canto inferior esquerdo

- Podemos criar uma cor com uma tripla de componentes vermelho, verde e azul inteiros, onde 0 é ausência e 255 a intensidade máxima, decimais, onde 0.0 é ausência e 1.0 a intensidade máxima, ou usar alguma das cores pré-definidas como variáveis globais de `Cor`

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

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`:

```
public void tique(HashSet<String> teclas, double dt) {
    if(teclas.contains("left") && !teclas.contains("right")) r.esquerda(dt);
    if(teclas.contains("right") && !teclas.contains("left")) r.direita(dt);
    ...
}
```

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

```
public void tecla(String t) {
    if(t.equals("escape")) 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:

```
r.esquerda(dt)
r.direita(dt)
b.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 `Jogo` 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 testar se ela colidiu com outra caixa de colisão (e em que lado dessa outra caixa):

```java
public class Hitbox
{
    public static int TOPO     = 1;
    public static int ESQUERDO = 2;
    public static int FUNDO    = 4;
    public static int DIREITO  = 8;

    // Canto superior esquerdo e
    // inferior direito
    double x0, y0, x1, y1;
    ...
}
```

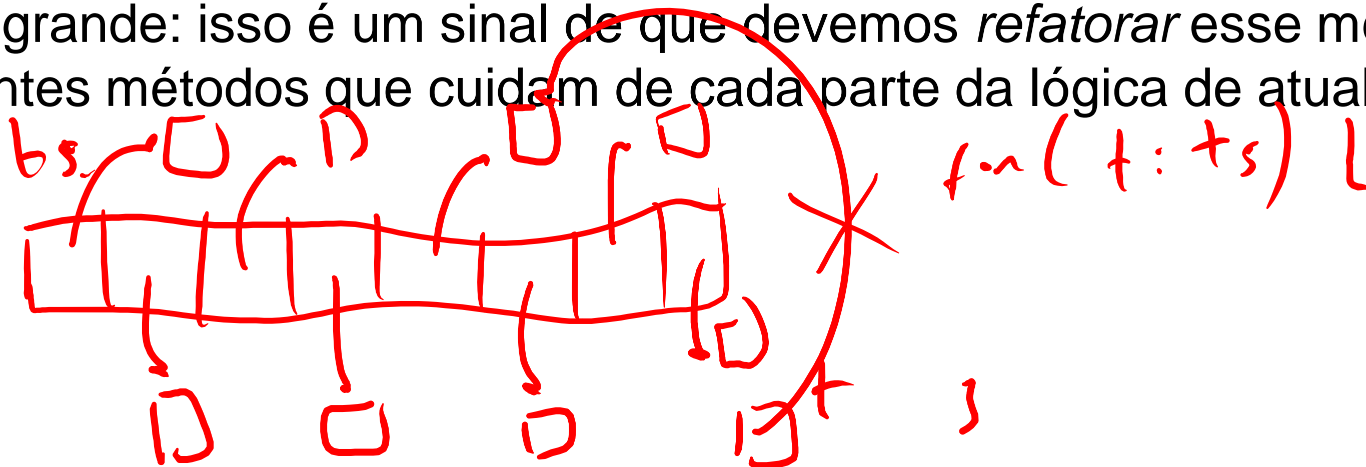
```java
// Esse retângulo colidiu com hb, e onde em hb?
public int intersecao(Hitbox hb) {
    double w = ((x1-x0) + (hb.x1 - hb.x0)) / 2;
    double h = ((y1-y0) + (hb.y1 - hb.y0)) / 2;
    double dx = ((x1 + x0) - (hb.x1 + hb.x0)) / 2;
    double dy = ((y1 + y0) - (hb.y1 + hb.y0)) / 2;
    if (Math.abs(dx) <= w && Math.abs(dy) <= h) {
        double wy = w * dy; double hx = h * dx;
        if (wy > hx) {
            if (wy > -hx) return FUNDO;
            else return ESQUERDO;
        } else {
            if (wy > -hx) return DIREITO;
            else return TOPO;
        }
    }
    return 0;
}
```

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
- Podemos usar também uma caixa de colisão para cada parede, e uma para o "chão", e assim simplificar os testes em `tique`

Coordenando as colisões

- Para saber se as colisões aconteceram o coordenador usa as caixas de colisão da bola e do outro objeto
- 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



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

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

```
public class Segmento {
  int valor;
  public Segmento(int _valor) {
    valor = _valor;
  }
  public boolean zerado() {
    return valor == 0;
  }
  public boolean tique() {
    valor = (valor - 1) % 60;
    return valor == 59
  }
  public String texto() {
    return String.format("%02d", valor);
  }
}
```

```
public class Timer {
    int x, y;
    Segmento min, seg;
    double tempo;
    int tamanho;
    Cor cor;
    ...
    public boolean tique(double _dt) {
        tempo = tempo + dt
        if(tempo >= 1.0) {
            tempo = tempo - 1.0
            if(seg.tique()) min.tique();
        }
        return min.zerado() && seg.zerado()
    }
    public void desenhar(Tela t):
        t.texto(x, y,
                min.texto()+":"+seg.texto(),
                tamanho, cor);
    }
}
```

Visibilidade

- Nem todos os campos e operações de um objeto são para consumo externo; várias delas podem ser apenas para uso pelo próprio objeto

- Em Java, podemos marcar qual a *visibilidade* de um campo ou um método:

  - `public` indica que o acesso é livre

  - `private` indica que o acesso é restrito apenas às instâncias da classe

- Quando não dizemos nada, temos um campo ou método que é público para quem estiver na mesma pasta, e privado para o resto