Programação Funcional
12ª Aula — Jogos usando *Gloss*

Pedro Vasconcelos
DCC/FCUP

2014

Jogos usando *Gloss*

Na aula passada: vimos como usar a bibloteca *Gloss* para fazer gráficos e simulações.

Nesta aula: vamos usar *Gloss* para fazer um jogo de ação simples.

Simulações e jogos

Para uma simulação especificamos uma função de atualização do "estado do mundo" com a passagem de Δt segundos:

```
simulate :: ...
        -> model                                  — estado inicial
        -> (model -> Picture)                     – função de desenho
        -> (ViewPort -> Float -> model -> model)
                                                  — função de atualização
        -> IO ()
```

Simulações e jogos (cont.)

Num jogo, além de simular a passagem de tempo, necessitamos de reagir a eventos causados pelo jogador:

- pressionar/largar teclas;
- mover o cursor do rato;
- pressionar/largar botões do rato;
- etc.

A função `play` que permite tratar estes eventos.

Simulações e jogos (cont.)

```
play :: ...
     -> world                          – estado inicial
     -> (world -> Picture)             — desenhar
     -> (Event -> world -> world)      — reagir a eventos
     -> (Float -> world -> world)      — atualização
     -> IO ()
```

(Segue-se uma demonstração dos tipos de eventos.)

Asteroids

- Um dos primeiros jogos vídeo de arcada (1979)
- O jogador controla uma nave espacial num “mundo” 2D
- Deve disparar sobre os asteróides e evitar ser atingido pelos fragmentos

Vamos usar o *Gloss* para implementar um jogo deste género.

(Segue-se uma demonstração.)

Objetos em jogo

Três tipos:

1. a nave do jogador;
2. os asteróides (vários tamanhos);
3. os lasers (disparados pela nave).

Objetos em jogo (cont.)

Representamos o mundo do jogo por uma lista de *objetos*:

```
type World = [Object]
```

Cada objeto contém informação de forma e de movimento:

```
type Object = (Shape, Movement)
```

Formas

Representamos as três formas de objetos por um novo tipo com três construtores:

```
data Shape = Asteroid Float   -- asteróide (tamanho)
           | Laser Float      -- laser (tempo restante)
           | Ship             -- nave do jogador
```

Os asteroides têm um parâmetro que especifica o seu tamanho

Os *lasers* têm como parâmetro o tempo restante antes de "decairem"

(Na proxima aula: mais sobre declarações de tipos.)

Desenhar objetos

```
drawObj :: Object -> Picture
drawObj (shape, ((x,y), _, ang, _))
  = translate x y (rotate ang (drawShape shape))

drawShape :: Shape -> Picture
drawShape Ship            = color green ship
drawShape (Laser _)       = color yellow laser
drawShape (Asteroid size)
        = color red (scale size size asteroid)

ship, laser, asteroid :: Picture      — figuras básicas
...
```

Desenhar o mundo

Recorde que o “mundo” é uma lista de objetos:

```
type World = [Object]
```

Basta desenhar todos os objetos e combinar as figuras:

```
drawWorld :: World -> Picture
drawWorld objs = pictures (map drawObj objs)
```

Reagir a eventos

Pressionar teclas $\leftarrow$ ou $\rightarrow$: iniciar rotação da nave

Levantar teclas $\leftarrow$ ou $\rightarrow$: parar rotação da nave

Pressionar tecla $\uparrow$: acelerar a nave

Pressionar barra de espaços: disparar *laser*

Reagir a eventos (cont.)

A função que reage a eventos é:

```
react :: Event -> World -> World
```

Invariantes:

- O "mundo" é uma lista de objetos.
- Esta lista nunca é vazia.
- O primeiro objeto é sempre a nave do jogador.

Seguem-se alguns exemplos de tratamento de eventos.

Reagir a eventos (cont.)

— iniciar/terminar rotação à esquerda

```
react (EventKey (SpecialKey KeyLeft) keystate _ _)
      (ship:objs)
  = (ship':objs)
  where  (Ship, (pos, vel, ang, angV)) = ship
         angV' = if keystate==Down then (-180) else 0
         ship' = (Ship, (pos, vel, ang, angV'))
```

Reagir a eventos (cont.)

— disparar um “laser”

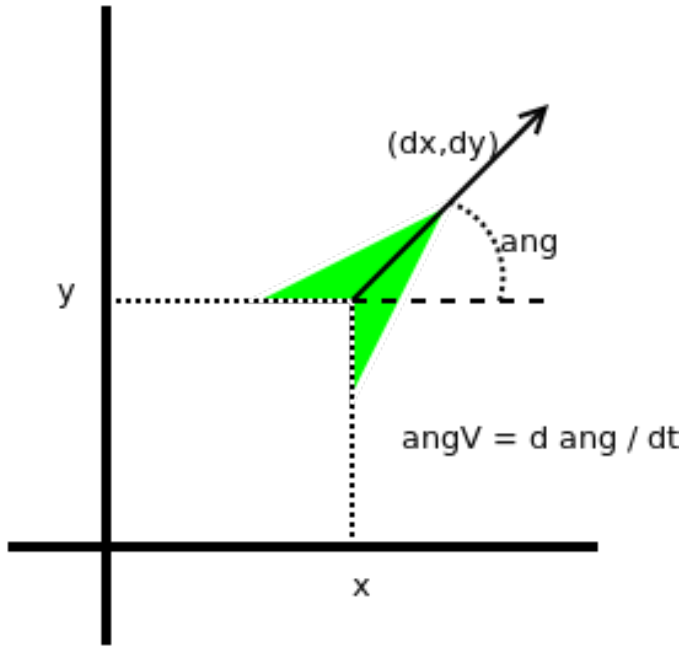
```
react (EventKey (SpecialKey KeySpace) Down _ _)
      (ship:objs)
  = (ship:proj:objs)
  where
    (Ship, (pos, _, ang, _)) = ship
    vel = (400*cos (-ang/180*pi), 400*sin (-ang/180*pi))
    proj = (Laser 1, (pos, vel, ang, 0))
```

— 1 seg.

Movimento dos objetos

O movimento de cada objeto é caracterizado por:

- posição (x, y)
- velocidade linear (dx, dy)
- orientação *ang*
- velocidade de rotação *angV*



Movimento dos objetos (cont.)

Representamos em Haskell por um tuplo:

```
type Movement = (Point,     — posição
                 Vector,    — velocidade linear
                 Float,     — orientação (graus)
                 Float)     — velocidade angular (graus/s)
```

Atualização da posição

- Calcular nova posição e orientação após Δt.
- Movimento limitado a uma "caixa":
  $-maxWidth \leq x \leq maxWidth$
  $-maxHeight \leq y \leq maxHeight$
- Se um objeto sair da janela deve re-entrar pelo lado oposto ("*wrap around*")
- Velocidade linear e de rotação são constantes (até o objeto ser destruido)

Atualização da posição (cont.)

```
move :: Float -> Movement -> Movement
move dt ((x,y), (dx,dy), ang, angV)
  = ((x',y'), (dx,dy), ang', angV)
    where x' = wrap (x+dt*dx) maxWidth
          y' = wrap (y+dt*dy) maxHeight
          ang' = ang + dt*angV
          wrap h max | h > max = h-2*max
                     | h < -max= h+2*max
                     | otherwise = h
```

Deteção de colisões

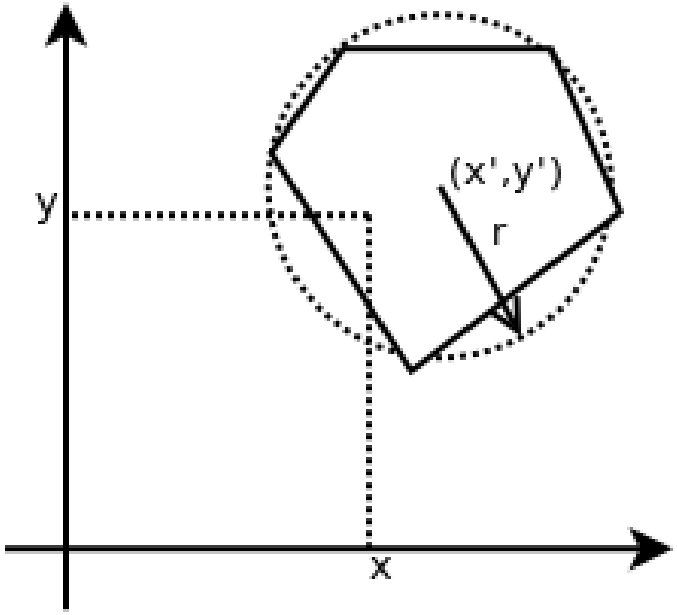
Num jogo de ação é útil detetar colisões entre objetos:

1. entre os *lasers* e asteróides;
2. entre a nave e os asteróides.

Para simplificar: vamos considerar apenas colisões do primeiro tipo.

Colisão de um ponto

Vamos aproximar o asteróide por uma "bola" de centro (x', y').



Um ponto (x, y) colidiu com o asteróide se está dentro da bola:

$$(x - x')^2 + (y - y')^2 \leq r^2$$

Colisão de um ponto (cont.)

— testar uma colisão entre um laser e um asteróide

```
hits :: Object -> Object -> Bool
hits (Laser _, ((x,y), _, _, _))
     (Asteroid sz, ((x',y'), _, _, _))
  = (x-x')**2 + (y-y')**2 <= (sz*10)**2
hits _ _ = False
```

O raio `sz*10` foi determinado experimentalmente.

Processar colisões

Para cada asteróide que é atingido por algum *laser*:

1. partir em fragmentos mais pequenos;
2. remover fragementos demasiados pequenos.

Sobrevivem após cada passo de simulação:

1. a nave do jogador;
2. os fragmentos resultantes de todas as colisões;
3. os asteróides e lasers não envolvidos em colisões.

Processar colisões (cont.)

```
collisions :: [Object] -> [Object]
collisions (ship:objs) = ship:(frags ++ objs' ++ objs'')
  where rocks  = filter isAsteroid objs
        lasers = filter isLaser objs
        frags = concat [fragment rock | rock<-rocks,
                                any (`hits`rock) lasers]
        objs' = [obj | obj<-rocks,
                  not (any (`hits`obj) lasers)]
        objs''= [obj | obj<-lasers,
                  not (any (obj`hits`) rocks)]

fragment :: Object -> [Object]    – fragmentar um asteróide
...
```

Função de atualização

Dado o intervalo de tempo Δt:

1. atualizar a posição cada objeto;
2. remover *lasers* que tenham "decaido";
3. processar colisões.

Função de atualização (cont.)

Exprimimos como a composição de três funções:

```
updateWorld :: Float -> World -> World
updateWorld dt = collisions . decay dt . map (moveObj dt)
```

```
decay :: Float -> World -> World
...
```

— ver código