Programação Funcional
11ª Aula — Gráficos usando *Gloss*

Pedro Vasconcelos
DCC/FCUP

2014

A biblioteca *Gloss*

- Para fazer desenhos, animações, simulações e jogos 2D;
- Simples: pensada para ensino de programação;
- Implementada usando OpenGL e GLUT (mas não é preciso aprender nada disto)
- Sítio oficial: `http://gloss.ouroborus.net/`
- Documentação: `http://hackage.haskell.org/package/gloss`

Primeiro exemplo

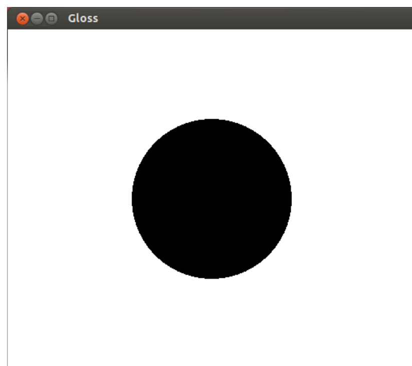
```
import Graphics.Gloss

main = display window white ex1
window = InWindow "Gloss" (800,600) (0,0)
ex1 = circleSolid 100
```

Primeiro exemplo (cont.)

Compilar e executar:

```
$ ghc exemplo.hs
$ ./exemplo
```

(*Esc* ou fechar a janela para sair.)



O exemplo em detalhe

```
import Graphics.Gloss

main :: IO ()
main = display window white ex1
```

— desenhar em fundo branco

```
window :: Display
window = InWindow "Gloss" (800,600) (0,0)
```

— numa janela com 800x600 pixels

```
ex1 :: Picture
ex1 = circleSolid 100
```

— um círculo cheio com 100 pixels de raio

Figuras

As figuras geométricas são valores de tipo *Picture*.

A biblioteca *gloss* exporta muitas funções para construir figuras; alguns exemplos:

```
circleSolid :: Float -> Picture
```
— círculo dado o raio

```
rectangleSolid :: Float -> Float -> Picture
```
— rectangulo dada largura, altura

```
line :: Path -> Picture
```
— linha poligonal

```
polygon :: Path -> Picture
```
— polígono cheio

```
type Path = [Point]
```
— percurso

```
type Point = (Float,Float)
```
— coordenada x,y

Cores

Podemos mudar a côr de uma figura:

```
color :: Color -> Picture -> Picture
```

As cores usuais estão pré-definidas na biblioteca:

```
red, green, blue, yellow, cyan, magenta, ...
```

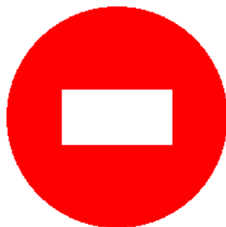
Sobrepor figuras

Podemos sobrepor várias figuras numa só:

```
pictures :: [Picture] -> Picture
```

Sobrepor figuras (cont.)

```
ex2 = pictures [color red (circleSolid 100),
                color white (rectangleSolid 100 50)]
```



Translações e rotações

Por omissão as figuras são desenhadas na *origem* (coordenadas $(0, 0)$).

Para desenhar noutro ponto basta fazer uma translação:

```
translate :: Float -> Float -> Picture -> Picture
```

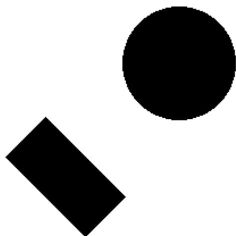
— translação por *dx*, *dy*

Também podemos fazer rotações por um ângulo (em graus):

```
rotate :: Float -> Picture -> Picture
```

Translações e rotações (cont.)

```
ex3 = pictures [translate 100 100 (circleSolid 50),
                rotate 45 (rectangleSolid 100 50)]
```



Ampliar ou reduzir

Podemos também ampliar ou reduzir figuras.

```
scale :: Float -> Float -> Picture -> Picture
```

— mudar a escala dados factores *x*, *y*

Ampliar ou reduzir (cont.)

```
ex4 = pictures [circleSolid 50,
                translate 0 100
                   (scale 1 0.5 (circleSolid 50))]
```

Simulações

Também podemos usar o *Gloss* para fazer animação de simulações discretas ao longo do tempo.

A função `simulate` da biblioteca faz a animação; precisamos de lhe passar:

1. um *modelo inicial*;
2. uma função para *converter um modelo numa figura*;
3. uma função para *avançar o tempo do modelo* por um intervalo Δt.

Exemplo

Simular o movimento de uma bola:

- movimento *linear* e *uniforme* (velocidade constante);
- sem atrito nem gravidade;
- colisões com os limites duma “caixa” virtual (janela).

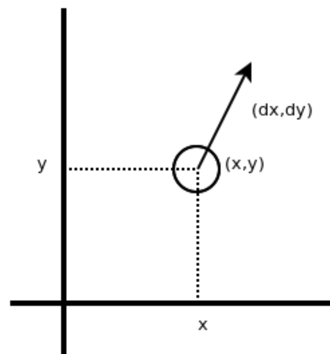
Modelo

— posição e velocidade

```
type Ball = (Point, Vector)
```

— definidos na biblioteca Gloss

```
type Point = (Float,Float)
type Vector = (Float,Float)
```



Função de desenho

```
drawBall :: Ball -> Picture
drawBall ((x,y),(dx,dy))
   = translate x y (color red (circleSolid ballRadius))

— raio da bola em pixels (constante)
ballRadius :: Float
ballRadius = 10
```

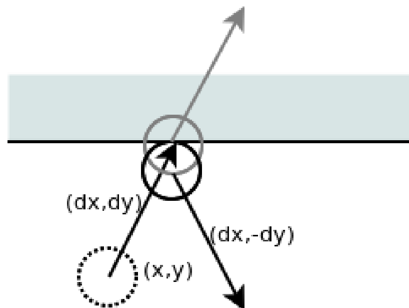
Atualizar posição e detetar colisões

Calculamos a nova posição:

$$
\begin{aligned}
x' &= x + \Delta t \times dx \\
y' &= y + \Delta t \times dy
\end{aligned}
$$

Se uma das coordenadas ultrapassar os limites:

- limitamos a coordenada;
- invertemos a componente correspondente do vector velocidade.



Atualizar posição e detetar colisões (cont.)

```
updateBall :: ViewPort -> Float -> Ball -> Ball
updateBall _ dt ((x,y),(dx,dy)) = ((x',y'), (dx',dy'))
  where (x',dx') = clip x dx (maxX-ballRadius)
        (y',dy') = clip y dy (maxY-ballRadius)
        clip h dh max
          | h' > max = (max, -dh)
          | h' < -max= (-max, -dh)
          | otherwise = (h', dh)
          where h' = h + dt*dh
```

— limites da “caixa” virtual

```
maxX, maxY :: Float
maxX = 300
maxY = 300
```

Programa principal

```
main = do
   ball <- randomBall
   simulate window black fps ball drawBall updateBall
```

— número de atualizações por segundo ("frames per second")

```
fps :: Int
fps = 60

window = InWindow "Gloss Ball" ...
```

— inicializar parâmetros aleatoriamente

```
randomBall :: IO Ball
randomBall = ...
```

Variações

- Simular múltiplas bolas independentes
  - sem colisões (fácil);
  - com colisões (mais difícil).
- Visualizar o vector velocidade
- Atribuir cores às bolas
- Desenhar os limites da caixa virtual