Laboratórios de Informática I
2022/2023

Licenciatura em Engenharia Informática

Ficha 7
Programação Gráfica usando o Gloss

Para construir a interface gráfica do projecto far-se-á uso da biblioteca `Gloss`. O `Gloss` é uma biblioteca *Haskell* minimalista para a criação de gráficos e animações 2D. Como tal, é ideal para a prototipagem de pequenos jogos. A documentação da API da biblioteca encontra-se disponível no link `https://hackage.haskell.org/package/gloss`.

1 Instalar o `Gloss`

O `Gloss` está desponível no hackage como um pacote da linguagem *Haskell*. Como tal, e como foi feito anteriormente com a biblioteca `HUnit`, pode ser obtido de forma simples usando o gestor de pacotes cabal. Basta introduzir no terminal os comandos:

```
$ cabal update
$ cabal install --lib gloss
```

Uma vez instalada a biblioteca, pode-se utilizá-la carregando-a para um programa sob a forma da seguinte instrução, no início do programa:

```
import Graphics.Gloss
```

1.1 Criação de Gráficos 2D

O tipo central da biblioteca `Gloss` é o tipo `Picture`. Este permite criar uma figura 2D usando segmentos de recta, círculos, polígonos, ou até `bitmaps` lidos de um ficheiro. A cada um destes diferentes tipos de figura correspondem diferentes construtores do tipo `Picture` (e.g. o construtor `Circle` para um círculo - ver documentação para consultar listagem completa dos construtores). Por exemplo, o valor `circulo1` definido abaixo representa um círculo de raio 50 centrado na posição (0,0).

```
circulo1 :: Picture
circulo1 = Circle 50
```

Certos construtores do tipo `Picture` não representam propriamente figuras, mas antes transformações sobre sub-figuras. Por exemplo, o construtor

```
Translate :: Float -> Float -> Picture -> Picture
```

permite reposicionar uma figura efetuando uma translação das coordenadas. Assim, para posicionar o círculo atrás definido num outro ponto que não a origem bastaria fazer algo como:

```
circulo2 :: Picture
circulo2 = Translate (-40) 30 circulo1
```

Outras transformações possíveis são `Scale`, `Rotate` e `Color`. Por último, podemos ainda produzir uma figura agregando outras figuras usando o construtor

```
Pictures :: [Picture] -> Picture
```

, que recebe uma lista de figuras para serem desenhadas sequencialmente (note que essas figuras se podem sobrepôr entre si). Segue-se um exemplo onde se explora essa possibilidade juntamente com outras transformações:

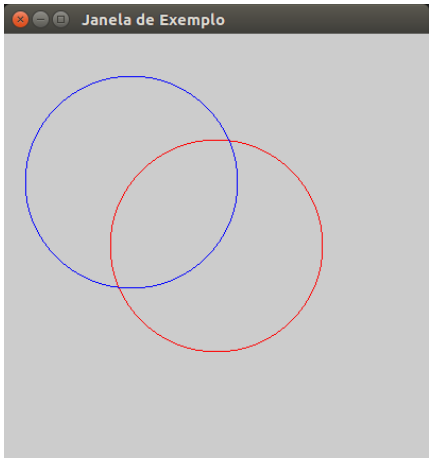
```
circuloVermelho = Color red circulo1
circuloAzul = Color blue circulo2
circulos = Pictures [circuloVermelho, circuloAzul]
```

Naturalmente que o objetivo de definir figuras como valores do tipo `Picture` é podermos visualizá-las no ecrã. Para tal temos de criar uma janela `Gloss` onde será desenhado o conteúdo da figura. O fragmento de código que se segue permite visualizar a figura `circulos` definida atrás, numa janela de fundo cinzento (definido em `background`):

```
window :: Display
window = InWindow
  "Janela de Exemplo" -- título da janela
  (200,200)           -- dimensão da janela
  (10,10)             -- posição no ecrã

background :: Color
background = greyN 0.8

main :: IO ()
main = display window background circulos
```



Note que poderá encontrar este programa (`Gloss_Exemplo0.hs`) em apêndice e na secção de conteúdos da plataforma BlackBoard.

Para correr o programa basta compilar o ficheiro *Haskell* usando o `ghc` e correr o executável. As instruções para tal serão:

```
$ ghc Gloss_Exemplo0.hs
$ ./Gloss_Exemplo0
```

Pode terminar a execução deste programa fechando a janela ou pressionando `Esc`. De notar que a convenção no `Gloss` é que a posição com coordenadas (0,0) é o centro da janela, e que, por defeito, as figuras são desenhadas no centro da janela, podendo isto ser alterado com o construtor `Translate`. Assim, o resultado obtido será a janela apresentada em cima.

Tarefas

1. Altere a figura `circulo2` escrevendo:

   ```
   circulo2 :: Picture
   circulo2 = rotate (-45) $ scale 0.5 1 $ Translate (-60) 30 circulo1
   ```

   Execute novamente o programa e analise o resultado obtido.

2. Acrescente agora à lista `circulos` a figura:

   ```
   circulo3 :: Picture
   circulo3 = scale 1 0.5 $ color yellow $ circleSolid 20
   ```

3. Crie uma nova figura complexa acrescentando à figura anterior um quadrado verde:

   ```
   quadradoVerde :: Picture
   quadradoVerde = color green $ rectangleSolid 20 20

   figuras :: Picture
   figuras = Pictures [circulos, quadradoVerde]
   ```

4. Crie agora uma linha poligonal:

   ```
   linhaPoligonal :: Picture
   linhaPoligonal = Line  [(0,0), (-200,0), (200,200), (0,200), (0,0)]
   ```

   Acrescente-a à figura anterior.

2 Programação de jogos

Para além da visualização de gráficos 2D, a biblioteca `Gloss` permite criar facilmente jogos simples usando a função `play` da biblioteca `Graphics.Gloss.Interface.Pure.Game`. Para usar esta função é necessário começar por definir um novo tipo `Estado` que representa todo o estado do seu jogo. Imagine por exemplo que o estado apenas indica a posição actual de um objecto.

```
type Estado = (Float, Float)
```

Depois é necessário definir qual o estado inicial do jogo, e como é que um determinado estado do jogo será visualizado com gráficos 2D, ou seja, como se converte para um valor do tipo `Picture`. No nosso caso, vamos assumir que o nosso estado inicial é a posição (0,0) e que em cada instante de tempo apenas desenhamos um polígono na posição dada pelo estado atual.

```
estadoInicial :: Estado
estadoInicial = (0,0)

desenhaEstado :: Estado -> Picture
desenhaEstado (x,y) = Translate x y poligono
 where
```

```
poligono :: Picture
poligono = Polygon [(0,0),(10,0),(10,10),(0,10),(0,0)]
```

Para implementar a reação a eventos, nomeadamente o pressionar das teclas, é necessário implementar uma função que, dado um valor do tipo `Event`[1] e um estado do jogo, gera o novo estado do jogo. No nosso exemplo, vamos apenas alterar o estado conforme o utilizador carrega nas teclas "left", "right", "up", e "down". No código, isto reflecte-se como um `Event` em que a respetiva tecla passa a estar pressionada (i.e. `Down`).

```
reageEvento :: Event -> Estado -> Estado
reageEvento (EventKey (SpecialKey KeyUp) Down _ _)    (x,y) = (x,y+5)
reageEvento (EventKey (SpecialKey KeyDown) Down _ _)  (x,y) = (x,y-5)
reageEvento (EventKey (SpecialKey KeyLeft) Down _ _)  (x,y) = (x-5,y)
reageEvento (EventKey (SpecialKey KeyRight) Down _ _) (x,y) = (x+5,y)
reageEvento _ s = s -- ignora qualquer outro evento
```

Finalmente, é necessário definir a seguinte função que altera o estado do jogo em consequência da passagem do tempo. Se o jogo estiver a funcionar a uma frame rate `fr`, o parâmetro `n` será sempre `1/fromIntegral fr`. Vamos assumir para o nosso exemplo que a cada instante de tempo a posição actual é actualizada da seguinte forma:

```
reageTempo :: Float -> Estado -> Estado
reageTempo n (x,y) = (x,y-0.3)
```

Para colocar todas estas peças a funcionar em conjunto basta definir um programa como o que se segue:

```
fr :: Int
fr = 50

dm :: Display
dm = InWindow "Novo Jogo" (400, 400) (0, 0)

main :: IO ()
main = do play dm         -- janela onde irá correr o jogo
           (greyN 0.5)    -- cor do fundo da janela
           fr             -- frame rate
           estadoInicial  -- estado inicial
           desenhaEstado  -- desenha o estado do jogo
           reageEvento    -- reage a um evento
           reageTempo     -- reage ao passar do tempo
```

Pode encontrar o código completo no `Gloss_Exemplo2.hs` na BlackBoard.

[1] definido em `Graphics.Gloss.Interface.Pure.Game`

Tarefas

1. Experimente alterar os parâmetros do programa `Gloss_Exemplo2.hs` e analise o resultado.

2. Pretendemos agora incluir no jogo uma imagem que varie com o passar do tempo. Explore o código do `Gloss_Exemplo3.hs`. Altere as imagens ou outros parâmetros à sua escolha.

3. Actualize o código do programa de tal modo que a imagem se mantenha em movimento enquanto a tecla estiver a ser pressionada. Sugestão: é necessário mais informação no estado do que a disponível atualmente, pelo que terá de estender o estado.

Exemplo

```
module Main where

import Graphics.Gloss

circulo1 :: Picture
circulo1 = Circle 50

circulo2 :: Picture
circulo2 = Translate (-60) 30 circulo1

circuloVermelho = Color red circulo1
circuloAzul = Color blue circulo2
circulos = Pictures [circuloVermelho, circuloAzul]

window :: Display
window = InWindow
  "Janela de Exemplo" -- título da janela
  (200,200)           -- dimensão da janela
  (10,10)             -- posição no ecrã

background :: Color
background = greyN 0.8

main :: IO ()
main = display window background circulos
```