Laboratórios de Informática I
2024/2025

Licenciatura em Engenharia Informática

Ficha 5
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 biblioteca encontra-se disponível em `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:

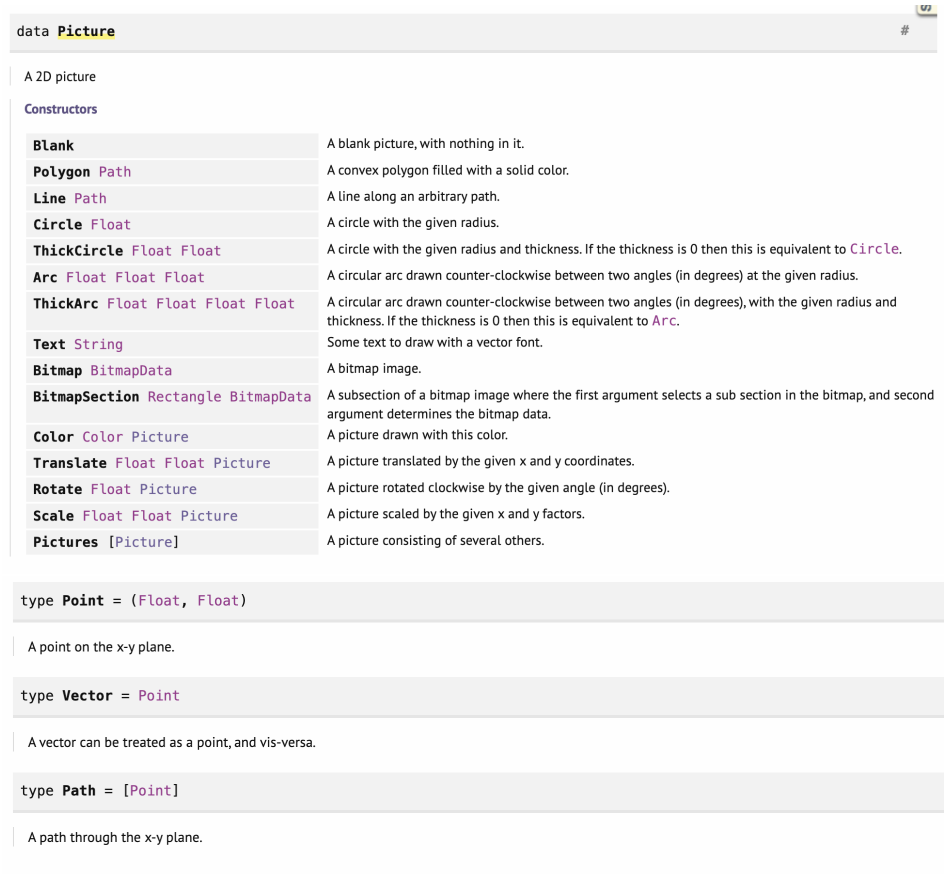
```
import Graphics.Gloss
```

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 - c.f. Figura 1 e ver documentação[1] 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
```

[1] `https://hackage.haskell.org/package/gloss-1.13.2.2/docs/Graphics-Gloss-Data-Picture.html`



data **Picture** #

A 2D picture

Constructors

Blank	A blank picture, with nothing in it.
Polygon Path	A convex polygon filled with a solid color.
Line Path	A line along an arbitrary path.
Circle Float	A circle with the given radius.
ThickCircle Float Float	A circle with the given radius and thickness. If the thickness is 0 then this is equivalent to Circle.
Arc Float Float Float	A circular arc drawn counter-clockwise between two angles (in degrees) at the given radius.
ThickArc Float Float Float Float	A circular arc drawn counter-clockwise between two angles (in degrees), with the given radius and thickness. If the thickness is 0 then this is equivalent to Arc.
Text String	Some text to draw with a vector font.
Bitmap BitmapData	A bitmap image.
BitmapSection Rectangle BitmapData	A subsection of a bitmap image where the first argument selects a sub section in the bitmap, and second argument determines the bitmap data.
Color Color Picture	A picture drawn with this color.
Translate Float Float Picture	A picture translated by the given x and y coordinates.
Rotate Float Picture	A picture rotated clockwise by the given angle (in degrees).
Scale Float Float Picture	A picture scaled by the given x and y factors.
Pictures [Picture]	A picture consisting of several others.

type **Point** = (Float, Float)

A point on the x-y plane.

type **Vector** = Point

A vector can be treated as a point, and vis-versa.

type **Path** = [Point]

A path through the x-y plane.

Figura 1: tipo `Picture`

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 sobrepor entre si). Segue-se um exemplo onde se explora essa possibilidade juntamente com outras transformações:

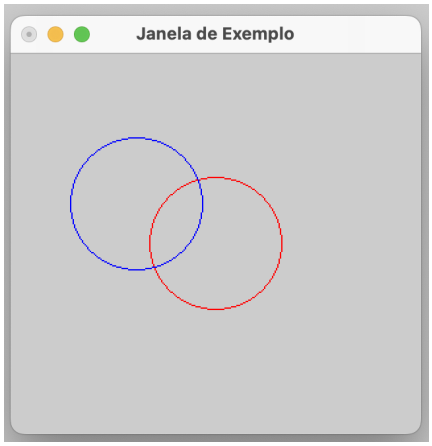
```
circuloVermelho = Color red circulo1
circuloAzul = Color blue circulo2
circulos = Pictures [circuloVermelho, circuloAzul]
```

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 [2].
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:

[2] Poderá obter informação acerca do tipo da função `display` em `https://hackage.haskell.org/package/gloss-1.13.2.2/docs/Graphics-Gloss-Interface-Pure-Display.html` e do tipo `Display` em `https://hackage.haskell.org/package/gloss-1.13.2.2/docs/Graphics-Gloss-Data-Display.html`

```
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`[3].

```
play ::
   Display    -- definição da janela
   -> Color    -- cor do fundo da janela
   -> Int    -- número de passos de simulação por segundo (frame rate)
   -> world   -- estado inicial
   -> (world ->Picture)    -- função que converte um estado num valor do tipo Picture
   -> (Event -> world -> world)  -- função que reage a um evento calculando o próximo estado
   -> (Float -> world -> world)  -- função que altera o estado em função do tempo
   -> IO ()
```

Consulte os construtores de `Event` na documentação do `Gloss`. Note em particular o construtor `EventKey` que permite representar teclas pressionadas. Repare que o estado de uma tecla (`KeyState`) pode ter o valor `Up` e `Down`. Exemplos:

```
(EventKey (Char 'w') Down _ _ )
(EventKey (SpecialKey KeyLeft) Down _ _)
(EventKey (SpecialKey KeyUp) Down _ _)
```

Para usar a função `play` 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)
```

[3] `https://hackage.haskell.org/package/gloss-1.13.2.2/docs/Graphics-Gloss-Interface-Pure-Game.html`

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` 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 (`SpecialKey KeyLeft`, `SpecialKey KeyRight`, `SpecialKey KeyUp` ou `SpecialKey KeyDown`) 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_Exemplo1.hs` em apêndice e na BlackBoard.

Tarefas

1. Experimente alterar os parâmetros do programa `Gloss_Exemplo1.hs` e analise o resultado.

2. Altere o programa de forma a registar no estado o tempo passado desde o início do jogo. Desenhe o valor do tempo na janela de jogo.

3. Altere o jogo de forma a não permitir que a figura saia da janela. Quando for executada uma acção (resultante de pressionar uma tecla ou da passagem do tempo) que deslocaria a figura para fora da janela, a figura deverá ficar na mesma posição.

4. Pretende-se associar à figura do jogo um vector velocidade que caracterize o seu movimento. Redefina o tipo do estado de forma a incluir informação sobre as coordenadas da figura e sobre as componentes do vector velocidade em ambos os eixos. Altere os argumentos da função `play` de forma a representar o movimento da figura ao longo do tempo, sob acção do vector de velocidade. Considere que o movimento da figura não é alterado por acção de teclas.

5. Altere o exercício anterior de forma a que a figura inverta o sentido do movimento ao atingir os limites da janela.

6. Altere o jogo de forma a que a figura ao ultrapassar um limite da janela (por acção de teclas ou do tempo), entre pelo lado oposto.

7. Altere o jogo de forma a incluir no jogo uma figura que varie com o passar do tempo (e.g. alterando a cor ou mudando de forma, ...).

8. Actualize o código do programa de tal modo que a imagem se mantenha em movimento enquanto uma dada tecla estiver a ser pressionada e que pare logo que a tecla deixe de ser pressionada.

9. Relembre as funções predefinidas: `concat`, `take`, `drop`, `splitAt`, `zip`, `unzip`, `words`, `unwords`, `lines`, `unlines`, `map` e `filter`.

   (a) Reformule algumas das funções definidas em exercícios anteriores (desta ou de outras fichas) usando estas funções.

   (b) Use o HUnit para testar as funções redefinidas.

   (c) Use o Haddock para documentar as funções redefinidas.

A Exemplos Gloss

`Gloss_Exemplo0.hs`:

```
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
```

Gloss_Exemplo1.hs:

```
module Main where

import Graphics.Gloss
import Graphics.Gloss.Interface.Pure.Game

type Estado = (Float, Float)

estadoInicial :: Estado
estadoInicial = (0,0)

desenhaEstado :: Estado -> Picture
desenhaEstado (x,y) = translate x y poligono
   where poligono :: Picture
         poligono = color red $ polygon [(0,0), (10,0), (10,10), (0,10), (0,0)]

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

reageTempo :: Float -> Estado -> Estado
reageTempo n (x,y) = (x, y-0.3)   -- diminui o valor do y


fr:: Int
fr = 50

dm :: Display
dm = InWindow
       "Novo Jogo"  -- título da janela
       (400, 400)   -- dimensão da janela
       (200,200)    -- posição no ecran

corFundo = (greyN 0.5)

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