Laboratórios de Informática I
2022/2023

Licenciatura em Engenharia Informática

Ficha 7
Programação Gráfica usando o Gloss (Continuação)

Na Ficha 6 exploramos exemplos introdutórios de utilização da biblioteca `Gloss`. Vamos de seguida analisar alguns exemplos adicionais, úteis para a realização do projeto..

1 Coordenadas, Interacção e Estado

Posicionar imagens na janela de jogo

Na fase 1 do projeto, a origem das coordenadas correspondia ao canto superior esquerdo do mapa de jogo. O referencial do `Gloss` posiciona a origem no centro da janela de jogo. Na fase 2 do projeto, ao fazer a tradução do referencial utilizado na fase 1 para o referencial do `Gloss`, note que terá de ter em consideração a dimensão da janela.

Jogo interactivo utilizando teclas variadas

Relembre a função `play` definida no módulo `Graphics.Gloss.Interface.Pure.Game` usada nos exemplos da ficha 6 [1].

```
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 estad
   -> (Float -> world -> world)   -- função que altera o estado em função do tempo
   -> IO ()
```

Consulte os contrutores de `Event` na documentação do `Gloss` em `Graphics.Gloss.Interface.Pure.Game`. 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:

[1] A documentação da API da biblioteca encontra-se disponível no link `https://hackage.haskell.org/package/gloss`.

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

Estrutura do estado

Considere agora que pretende registar a pontuação de um jogador. Ou que pretende incluir menus para ser possível, por exemplo, incluir pausas no jogo sem terminar o programa (e.g. com opções para iniciar jogo, suspender jogo, retomar jogo, terminar jogo, ...). Em ambos os casos é necessário acrescentar novas componentes ao estado, para além das coordenadas do jogador.

Na fase 2 do projeto deverá ter em atenção os aspectos acima mencionados. Poderá encontar na plataforma BlackBoard alguns exemplos simples (e.g. Exemplo5.hs e Exemplo6.hs) que poderá usar como ponto de partida para o seu projeto.

2 Inclusão de imagens no jogo

É possível carregar ficheiros de imagens externos no formato `bitmap` (com extensão `bmp`). Para tal, pode usar a função `loadBMP :: FilePath -> IO Picture` disponibilizada pelo módulo `Graphics.Gloss.Data.Bitmap`. Como esta é uma função de I/O, deve ser executada diretamente na função `main` do jogo, devendo as imagens ser incluídos no estado do jogo, por exemplo:

```
type EstadoGloss = (Estado, (Picture, Picture))
...
main :: IO ()
main = do
    p1 <- loadBMP "pac_open.bmp"
    p2 <- loadBMP "pac_closed.bmp"
    play dm
           (greyN 0.5)
           (estadoGlossInicial p1 p2)
           desenhaEstadoGloss
           reageEventoGloss
           reageTempoGloss
```

Pode utilizar a ferramenta `convert` distribuída com o ImageMagick (`https://imagemagick.org/`) para converter imagens em formato PNG para o formato BMP:

```
convert in.png -depth 1 out.bmp
```

Note que a biblioteca `Gloss`, por definição, apenas suporta ficheiros `bitmap` não comprimidos. Pode utilizar `convert` para descomprimir um ficheiro `bitmap`:

```
convert compressed.bmp -compress None decompressed.bmp
```

Pode ainda utilizar um conversor online, por exemplo: `https://convertio.co/pt/png-bmp/`.

Em "Exemplo6.hs", na plataforma Blackboard, encontra um jogo que usa imagens no formato `bitmap`.

3 PlayIO

Para gravar em ficheiro o estado do jogo, uma possibilidade será importar "Graphics.Gloss.Interface.IO.Game" em vez de "Graphics.Gloss.Interface.Pure.Game" e usar a função `playIO` em vez de `play`:

```
playIO ::
             Display
             -> Color
             -> Int
             -> world
             -> (world -> IO Picture)
             -> (Event -> world -> IO world)
             -> (Float -> world -> IO world)
             -> IO ()
```

Os argumentos de `playIO` têm funções semelhantes às dos argumentos de `play`, mas agora são monádicos. Pode usar a notação "do" e funções de interacção como, por exemplo, `readFile` e `writeFile`. É necessário fazer o `return` do resultado nas componentes monádicas.

Para terminar o jogo usando uma tecla eventualmente diferente de `Esc`, pode usar a "função" `exitSuccess` de tipo `IO a`, por isso compatível com o `playIO`. Mas nesse caso é necessário importar também o módulo `System.Exit`.

Excerto de exemplo com playIO

(Pode encontrar este exemplo completo na plataforma BlackBoard em "Exemplo7.hs".)

```
module Main where

import Graphics.Gloss
import Graphics.Gloss.Interface.IO.Game
import System.Exit

data Menu = Save | ModoJogo

type Estado = (Menu, (Float, Float))

-- estado do Gloss com menu, coordenadas do jogador,
-- duas imagens e um valor de segundos passados desde o início do programa
type EstadoGloss = (Estado, (Picture, Picture, Float))

-- Inicialização do estado com os valores anteriormente guardados em ficheiro
estadoInicial :: (Float,Float) -> Estado
estadoInicial c = (ModoJogo, c)

estadoGlossInicial :: Picture -> Picture -> ((Float,Float),Float) -> EstadoGloss
estadoGlossInicial p1 p2 (c,t)= (estadoInicial c, (p1, p2, t))

-- função que reage a eventos  (Note o "return")
reageEventoGloss :: Event -> EstadoGloss -> IO EstadoGloss
reageEventoGloss (EventKey (SpecialKey KeyUp) Down _ _) ((ModoJogo, (x,y)), e) =
       return $ ((ModoJogo, (x,y+5)), e)
       ....
-- ao pressionar tecla "Space" suspende jogo e guarda estado
```

```
reageEventoGloss  (EventKey (SpecialKey KeySpace) Down _ _) ((ModoJogo, c), (p1,p2,t)) =
      do writeFile "save.txt" (show (c,t)) -- grava coordenadas e tempo em ficheiro
         return ((Save, c),(p1,p2,t)) -- suspende jogo

-- retoma jogo  ao pressionar tecla "Space" a partir do estado anterior
reageEventoGloss  (EventKey (SpecialKey KeySpace) Down _ _) ((Save, c),e) =
      return ((ModoJogo,c), e)

-- termina jogo e guarda estado ao pressionar a tecla "q"
reageEventoGloss (EventKey (Char 'q') Down _ _) ((_, c), (_,_,t)) =
      do writeFile "save.txt" (show (c,t))
         putStrLn "FIM"
         exitSuccess
    ...

-- Note o "return" em cada um dos casos
reageTempoGloss :: Float -> EstadoGloss -> IO EstadoGloss
reageTempoGloss n ((ModoJogo, (x,y)), (p1, p2, b)) = return $ ((ModoJogo, (x,y)), (p1, p2, b+n))
reageTempoGloss n ((Save, (x,y)), e) =  return $ ((Save, (x,y)), e) -- efeito do tempo suspenso

fr :: Int
fr = 50

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

-- alternar a imagem a cada 100 milissegundos (Note o "return")
desenhaEstadoGloss :: EstadoGloss -> IO Picture
desenhaEstadoGloss ((_, (x,y)), (p1, p2, b))
 | (mod (round (b*1000)) 200) < 100 = return $ Translate x y p1
 | otherwise = return $ Translate x y p2

--- usa playIO e inicializa estado com informação do estado anterior guardada em ficheiro
main :: IO ()
main = do
--  writeFile "save.txt" "((0,0),0)"
    p1 <- loadBMP "pac_open.bmp"
    p2 <- loadBMP "pac_closed.bmp"
    saved <- readFile "save.txt"
    let (coord, time) = (read saved)
    playIO dm
           (greyN 0.5)
           fr
           (estadoGlossInicial p1 p2 (coord, time))
           desenhaEstadoGloss
           reageEventoGloss
           reageTempoGloss
```