

Laboratórios de Informática I

2021/2022

Licenciatura em Engenharia Informática

Sessão Laboratorial 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 disponível 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 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 **circulo** definido abaixo representa um círculo de raio 50 centrado na posição (0,0).

```
circulo :: Picture
circulo = 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:

```
outroCirculo :: Picture
outroCirculo = Translate (-40) 30 circulo
```

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:

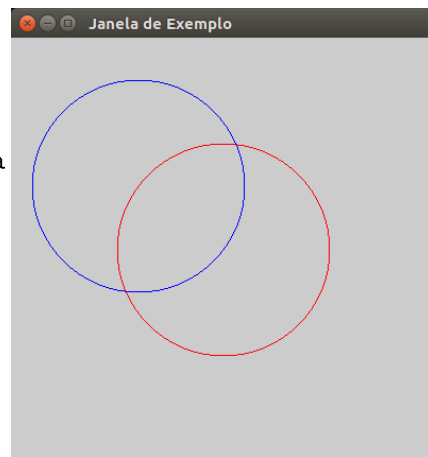
```
circuloVermelho = Color red circulo
circuloAzul = Color blue outroCirculo
circulos = Pictures [circuloVermelho, circuloAzul]
```

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, assim como o resto do material usado nesta ficha, na secção de conteúdos da disciplina na BlackBoard. Para correr o programa basta compilar o ficheiro *Haskell* usando o **ghc** e correr o executável. As instruções para tal serão:

```
$ ghc Exemplo2.hs
$ ./Exemplo2
```

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.

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 actual.

```
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 reacção 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:

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

```

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)       -- côr 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

```

3 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 os gráficos ser incluídos no estado do jogo:

```

type EstadoGloss = (Estado, Picture)

...

main :: IO ()
main = do p <- loadBMP "imagem.bmp"
        play dm          -- janela onde irá correr o jogo
        (greyN 0.5)       -- côr do fundo da janela
        fr                -- frame rate
        estadoGlossInicial p -- estado inicial
        desenhaEstadoGloss -- desenha o estado do jogo
        reageEventoGloss   -- reage a um evento
        reageTempoGloss    -- reage ao passar do tempo

```

Uma dica adicional é que a biblioteca `Gloss`, por definição, apenas suporta ficheiros `bitmap` não comprimidos. Em sistemas Unix, pode-se utilizar a ferramenta `convert` distribuída com o `ImageMagick` para descomprimir um ficheiro `bitmap`:

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

Alternativamente, pode-se também utilizar o pacote `gloss-juicy` que suporta ficheiros de imagens genéricos (incluindo formatos populares como `png` e `jpg`). Para instalar esta extensão deve executar o comando:

```
cabal install gloss-juicy
```

Imagens de formato genérico podem então ser alternativamente carregadas da seguinte forma:

```
import Graphics.Gloss.Juicy (loadJuicy)

...

do
  Just img1 <- loadJuicy "imagem.jpg"
  Just img2 <- loadJuicy "imagem.png"
...
```

Está disponível um exemplo mais complexo de utilização desta biblioteca na BlackBoard. Neste exemplo, usa-se a biblioteca **gloss-juicy** para carregar duas imagens do **Pacman** de boca aberta e fechada, e a imagem que é desenhada vai alternando a cada 200 milissegundos. Para se fazer esse controlo, é preciso expandir o estado para incluir as duas imagens do **Pacman**, assim como um valor de número de segundos passados desde o início do jogo. Note que ainda se poderia fazer várias melhorias neste protótipo, como por exemplo rodar o **Pacman** na direção em que este se desloca.

4 Exercícios

1. Actualize o código do módulo Exemplo3 de tal modo que o **Pacman** 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.
2. Altere o código para começar a desenhar elementos do projecto da disciplina. Por exemplo: desenhe o jogador, e desenhe os diferentes tipos de peças presentes no projeto.
3. Pense como fazer a tradução do referencial usado na 1a fase do projecto (origem no canto superior esquerdo) para o referencial do **Gloss** (origem no centro da janela). Sugestão: tem que ter em consideração a dimensão da janela.