Laboratórios de Informática I
2024/2025

Licenciatura em Engenharia Informática

Ficha 6
Programação Gráfica usando o Gloss (continuação)

Na Ficha 5 exploramos exemplos introdutórios de utilização da biblioteca `Gloss`. Vamos de seguida analisar alguns exemplos adicionais.

1 Coordenadas, Interacção e Estado

Posicionar imagens na janela de jogo

É possível representar um mapa de um jogo como uma matriz. Quando trabalhamos com uma matriz definida como uma lista de listas (i.e. como uma lista de linhas), e a consideramos uma representação de um sistema de coordenadas cartesiano, é habitual assumir que um par de coordenadas (x,y) representa a linha y e a coluna x da matriz, e posicionar a origem dos eixos coordenados na linha de índice zero e na coluna de índice zero dessa linha (canto superior esquerdo do mapa). Desta forma, as coordenadas correspondem directamente aos índices das listas (e.g. y=0 representa a primeira linha da matriz, y=2 representa a terceira linha da matriz, x=1 a segunda coluna da matriz e (2,3) representa um elemento na quarta linha e terceira coluna da matriz).

O referencial do `Gloss` posiciona a origem no centro da janela de jogo. Ao fazer a tradução do referencial do mapa do jogo para o referencial do `Gloss`, note que terá de ter em consideração a dimensão da janela.

Tarefas

1. Relembre os exercícios da Ficha 1. Defina uma função que recebe uma lista de coordenadas relativas a um referencial com origem no canto superior esquerdo de uma janela e ajusta essas coordenadas ao referêncial do `Gloss`. Teste com exemplos de listas de figuras distribuidas numa janela.

2. Defina uma função que recebe uma lista de coordenadas de maçãs distribuidas num mapa (interpretado como uma matriz) e as desenha numa janela Gloss. Crie o jogo do "Pacman"em que um personagem se desloca num mapa sob a acção de teclas, e come as maçãs quando se encontra na mesma posição.

Jogo interactivo utilizando teclas variadas

Relembre a função `play` definida no módulo `Graphics.Gloss.Interface.Pure.Game` usada nos exemplos da Ficha 5 [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 e calcula 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` 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:

```
(EventKey (Char 'w') Down _ _ )
(EventKey (SpecialKey KeyLeft) Down _ _)
(EventKey (SpecialKey KeyUp) Down _ _)
(EventKey (MouseButton LeftButton) Down _ (px,py))   -- px e py coordenadas do rato
```

Tarefas

1. Altere os jogos anteriores de forma a que seja possível jogar com teclas associadas a letras.

2. Altere os jogos de forma a que seja possível jogar com o rato.

Estrutura do estado

Nos exemplos da Ficha 5 vimos como guardar no estado informação sobre as coordenadas de um jogador, como registar o tempo passado desde o inicio do jogo e como registar as teclas pressionadas num dado momento.

Considere agora que pretende registar a pontuação de um jogador. Ou que pretende incluir menus com opções para iniciar jogo, suspender jogo, retomar jogo ou terminar jogo, por exemplo. Em ambos os casos é necessário acrescentar novas componentes ao estado, para além das coordenadas do jogador.

[1] A documentação da API da biblioteca encontra-se disponível no link `https://hackage.haskell.org/package/gloss-1.13.2.2/docs/Graphics-Gloss-Interface-Pure-Game.html`

Tarefas

1. Altere os jogos anteriores de forma a que seja possível registar a pontuação de um jogador (e.g. em função do número de teclas pressionadas, em função do tempo ou em função das maçãs comidas).

2. Altere os jogos de forma a incluir um menu que permita iniciar o jogo, suspender o jogo, reiniciar o jogo e terminar dando a pontuação obtida.

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`. Note que esta função é monádica. As imagens devem ser incluídos no estado do jogo.
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 distribuída com o ImageMagick (`https://imagemagick.org/`) para converter imagens em formato PNG para o formato BMP:

```
magick source.png -depth 24 -compress None output.bmp
```

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

Tarefa

1. Inclua imagens em formato `bitmap` nos jogos anteriores.

3 PlayIO

Caso pretenda gravar em ficheiro o estado do jogo, uma possibilidade será importar o módulo `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. É necessário usar a "notação do"e fazer o `return` do resultado nas componentes monádicas.
Funções de interacção como, por exemplo, `readFile` e `writeFile` podem ser usadas.
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`.
Para verificar se um ficheiro existe, pode usar a "função"monádica `doesFileExist` tendo para isso de importar o módulo `System.Directory`.

Por exemplo, se tivermos um estado em que registamos as coordenadas de uma figura e o tempo decorrido desde o inicio do jogo, e quisermos iniciar um jogo a partir de um estado anteriormente guardado no ficheiro "save.txt", poderemos escrever:

```
leEstado ::  IO ((Float,Float),Float)
leEstado = do
               fileExist <- doesFileExist "save.txt"
               saved <- if fileExist then readFile "save.txt"
               else return "((0,0),0)"
               return (read saved)
```

Se o ficheiro não existir o estado é inicializado com ((0,0),0).

Se quisermos guardar o estado em ficheiro (para suspender o jogo e retomá-lo mais tarde, por exemplo), podemos escrever:

```
...
reageEventoGloss (EventKey (Char 'q') Down _ _) ((x,y),t) =
      do writeFile "save.txt" (show ((x,y),t))
         exitSuccess
```

Para além de guardar os dados no ficheiro "save.txt", estamos a permitir que o jogo termine quando é pressionada a tecla com o caractere 'q'.

Tarefas

1. Altere os jogos anteriores de forma a poder terminar o jogo com uma tecla diferente de `Esc`.
2. Altere os jogos anteriores de forma a poder gravar o estado do jogo em qualquer momento, interromper o jogo e retomar a partir de estado guardado.

4 Records em Haskell

Os `records` em Haskell são uma extensão dos `data types` que permitem associar nomes às várias componentes do tipo de dados.

```
data Aluno = Aluno
    { numero :: Int
    , nome :: String
    , nota :: Int
    }
```

Esses nomes poderão ser usados para seleccionar directamente as respectivas componentes. Se no interpretador verificar o tipo destes selectores, obterá:

```
ghci>:t numero
numero :: Aluno -> Int
ghci>:t nome
nome :: Aluno -> String
ghci>:t nota
nota :: Int -> Aluno
```

Se agora construir um valor `a1` do tipo `Aluno`:

```
ghci> a1 = Aluno 99999 "Joana Vasconcelos" 16
ghci>:t a1
a1 :: Aluno
ghci> a1
Aluno {numero = 99999, nome = "Joana Vasconcelos", nota = 16}
ghci> numero a1
99999
ghci> nome a1
"Joana Vasconcelos"
```

Podemos alterar apenas algumas componentes do `record`, e deixar inalteradas as restantes.

```
ghci> let a2 = a1 {nota=18}
ghci> a2
Aluno {numero = 99999, nome = "Joana Vasconcelos", nota = 18}
```

Note que no exemplo acima apenas foi alterado o campo `nota`.

Podemos usar `pattern matching` com records.

```
case  a  of
  Aluno {numero = 99999} -> ...
```

Note que podemos não mencionar todos os campos.

Tarefas

1. Reestruture os exemplos das secções e fichas anteriores usando records.

5 Mini-projeto: Snake

Sugerem-se agora um pequeno projeto em que poderá aplicar tudo o que aprendeu até ao momento.

1. Crie o jogo "Snake"em que uma cobra se desloca de forma autónoma num tabuleiro, come maçãs que estão distribuidas no tabuleiro, aumenta de tamanho ao comer maçãs e pode mudar a direcção do movimento por acção de teclas (ou do rato). A cobra não deve sair dos limites do tabuleiro (e.g. pode inverter o sentido da marcha ao atingir os limites). O jogo deve terminar quando for pressionada a tecla com o caractere 'q'.

2. Altere o jogo anterior de forma a que só exista uma maçã de cada vez no tabuleiro - uma maçã só é desenhada quando a anterior é comida.

3. Altere o jogo anterior de forma a que as posições das maçãs sejam geradas aleatóriamente. Pode gerar números aleatórios num dado intervalo usando a função monádica `randomRIO`. Para tal deve importar o módulo `System.Random`. Por exemplo, `randomRIO (0,50)` permite gerar um valor aleatório entre 0 e 50.
   No programa:

   ```
   do
     x <- randomRIO (0,50)
     y <- randomRIO ('a','z')
   ```

   `x` terá um valor entre 0 e 50, enquanto `y` terá um valor entre 'a' e 'z'.

4. Altere o jogo anterior de forma a que a cobra se desloque de forma autónoma e mude de direcção de forma aleatória.

5. Experimente outras funcionalidades que considere interessantes.