Bomberman

2ª fase do projeto de LI1 2016/17

Introdução

Neste enunciado apresentam-se as tarefas referentes à segunda fase do projecto da unidade curricular de Laboratórios de Informática I. O projecto será desenvolvido pelos mesmos grupos constituídos para a primeira fase, e consiste em pequenas aplicações Haskell que deverão responder a diferentes tarefas (apresentadas adiante).

O tema do projecto continua a basear-se no clássico jogo Bomberman em *battle mode*, sendo que nesta fase se pretende complementar as tarefas realizadas na fase anterior para produzir um programa Haskell com o jogo completo, incluindo uma interface gráfica. Será também implementado um *bot* para jogar Bomberman, que pode ser usado quando menos de 4 humanos pretendem jogar.

Será mantida uma FAQ com as perguntas mais frequentes que nos vão sendo colocadas sobre o enunciado. A FAQ pode ser encontrada em
https://goo.gl/9uwKQx

Tarefas

Tarefa 4 - Reagir à passagem do tempo

O objectivo desta tarefa é, dada uma descrição do estado do jogo, determinar o efeito da passagem de um instante de tempo nesse estado. O estado do jogo será representado no mesmo formato usado na Tarefa 2 da primeira fase do projecto.

O efeito principal da passagem do tempo é a explosão das bombas e a possível morte de jogadores em consequência dessa explosão. Cada bomba demora 10 instantes de tempo a explodir. Uma bomba tem um determinado raio de acção, determinado pelo número de *power ups* `Flame` que o jogador que a colocou possuía. Quando explode lança chamas com dimensão igual ao raio nas quatro direções principais (norte, sul, leste e oeste). As chamas eliminam todos os jogadores e *power ups* (já destapados) que atingem. As linhas referentes a jogadores e *power ups* eliminados devem ser removidas da representação do jogo. Se as chamas atingirem outra bomba o temporizador dessa bomba passa para 1, de forma a forçar a sua explosão no próximo instante de tempo. As chamas não destroem os blocos do tipo pedra, sendo a sua passagem bloqueada pelos mesmos. Quando atingem um bloco do tipo tijolo, destroem o mesmo revelando possíveis *power ups* que estejam escondidos atrás. Tal como a pedra, o tijolo e os *power ups* também bloqueiam a passagem das chamas.

Cada jogo tem uma duração fixa. Para forçar os jogadores a efectuar acções, num mapa de dimensão n, quando faltarem `(n-2)^2` instantes de tempo o mapa começa a fechar-se com blocos de pedra num efeito de espiral que começa na posição 1 1. Este efeito pode, por exemplo, ser visto no vídeo https://www.youtube.com/watch?v=I9-wQfkJCNM (começando sensivelmente no instante 1:57). Em cada instante de tempo cai um bloco de pedra que elimina tudo o que estiver na respectiva posição.

Por exemplo, considere o seguinte estado do jogo.

```
#########
#       #
# #?# # #
#     ? #
#?# # #?#
# ?  ?  #
# #?#?# #
#  ??   #
#########
+ 3 3
```

```
! 5 5
* 7 6 0 1 8
* 7 7 1 3 1
0 6 7 +
1 6 5 !
```

Se faltarem 49 instantes de tempo para terminar o torneio, o próximo estado do jogo deverá ser:

```
#########
##      #
# #?# # #
#     ? #
#?# # # #
# ?  ?  #
# #?#?# #
#  ?    #
#########
+ 3 3
! 5 5
* 7 6 0 1 1
1 6 5 !
```

No repositório SVN, dentro da directoria `src`, encontrará um ficheiro `Tarefa4.hs`, onde falta implementar a função `avanca :: [String] -> Int -> [String]` que, dado o estado actual do jogo no formato acima descrito e o número instantes de tempo que faltam para o jogo terminar, devolve o novo estado do jogo. Neste ficheiro encontra-se implementado um programa `main` que pode ser usado para testar esta função: após compilado, o programa aceita como parâmetro o número de instantes de tempo que faltam para o jogo terminar, fica à espera do estado do jogo no `stdin` e invoca a função `avanca`, imprimindo o resultado no `stdout`. O nome deste ficheiro e o respectivo `main` não podem ser alterados.

Tarefa 5 - Implementação do jogo em `Gloss`

O objectivo desta tarefa é implementar o jogo completo usando a biblioteca `Gloss`. Um breve tutorial de introdução a esta biblioteca do Haskell pode ser encontrado no final deste documento. Como ponto de partida deve começar por implementar uma versão com uma visualização gráfica simples e que use sempre uma dimensão e semente fixas para gerar o estado inicial. Note no entanto que esta tarefa se trata acima de tudo de uma “tarefa aberta”, onde se estimula que os alunos explorem diferentes possibilidades para melhorar o aspecto final e jogabilidade do jogo.

No repositório SVN, dentro da directoria `src`, encontrará um ficheiro `Tarefa5.hs` com um *template* de um jogo em `Gloss` (ver tutorial no final deste documento). Neste ficheiro encontra-se também implementado um programa `main` que pode ser usado para correr o jogo.

Tarefa 6 - Implementar estratégia de combate

O objectivo desta tarefa é implementar um *bot* que jogue Bomberman automaticamente. A estratégia de jogo a implementar fica ao critério de cada grupo, sendo que a avaliação automática será efectuada colocando o *bot* implementado a combater com diferentes *bots* de variados graus de "inteligência".

O *bot* deve estar preparado para jogar em qualquer posição (irá receber o identificador do jogador), sendo que nos torneios irá rodar por diferentes posições para evitar enviesamentos. O *bot* irá também receber o número de instantes que faltam para terminar o jogo, podendo assim alterar a sua estratégia na parte final em que o mapa se começa a fechar. Por forma a simular a situação real de jogo, a descrição do jogo a passar ao bot não irá conter os *power ups* que ainda estão escondidos atrás de tijolo.

No repositório SVN, dentro da directoria `src`, encontrará um ficheiro `Tarefa6_li1gXXX.hs` (onde XXX é o número do grupo), onde falta implementar a função `bot :: [String] -> Int -> Int -> Maybe Char` que, dado o estado actual do jogo, o identificador do jogador correspondente ao *bot* e o número instantes de tempo que faltam para o jogo terminar, devolve o comando a executar ou `Nothing` se não quiser efectuar nenhum comando. Se o comando devolvido for errado (ou seja, diferente de `'U'`, `'D'`, `'R'`, `'L'` ou `'B'`) o respectivo jogador será eliminado. Nem o nome deste ficheiro nem a declaração do módulo podem ser alterados. Também encontrará na directoria `src` um ficheiro `Tarefa6.hs` que implementa um programa `main` que pode ser usado para testar a função `bot`: após compilado, o programa aceita como parâmetros o identificador do jogador e o número de instantes de tempo que faltam para o jogo terminar, fica à espera do estado do jogo no `stdin` e invoca a função `bot`, imprimindo o resultado no `stdout`.

Relatório

Nesta fase devem também escrever um pequeno relatório sobre o desenvolvimento do projecto. Este relatório deverá ser escrito em LaTeX, uma ferramenta que será apresentada nas aulas práticas. Será disponibilizado um template para esse relatório na plataforma de *e-learning*.

Entrega e Avaliação

A data limite para conclusão de todas as tarefas desta segunda fase é **2 de Janeiro de 2017** e a respectiva avaliação terá um peso de 40% na nota final da UC. A submissão será feita automaticamente através do SVN: nesta data será feita uma cópia do repositório de cada grupo, sendo apenas consideradas para avaliação os programas e demais artefactos que se encontrem no repositório nesse momento. O conteúdo dos repositórios será processado por ferramentas de detecção de plágio e, na eventualidade de serem detectadas cópias, estas serão consideradas fraude dando-se-lhes tratamento consequente.

Para além dos programas Haskell relativos às 3 tarefas, será considerada parte integrante do projeto todo o material de suporte à sua realização armazenado no repositório SVN do respectivo grupo (código, documentação, relatório, ficheiros de teste, etc.). A utilização das diferentes ferramentas abordadas no curso (como `Haddock`, SVN, LaTeX, etc.) deve seguir as recomendações enunciadas nas respectivas sessões laboratoriais. A avaliação desta fase do projecto terá em linha de conta todo esse material, atribuindo-lhe os seguintes pesos relativos:

Componente	**Peso**
Avaliação automática da tarefa 4	15%
Avaliação automática da tarefa 6	15%
Avaliação qualitativa das tarefas	40%
Utilização do SVN, testes e documentação do código.	10%
Relatório e utilização do LaTeX.	20%

A nota final é atribuída independentemente a cada membro do grupo em função da respectiva prestação. A avaliação automática da tarefa 4 será feita através de um conjunto de testes que não serão revelados aos grupos. No caso da tarefa 6, a avaliação automática será feita colocando o *bot* implementado a combater com diferentes *bots* com um grau variado de "inteligência". A avaliação qualitativa incidirá sobre aspectos da implementação não passíveis de ser avaliados automaticamente (por exemplo, estrutura do código, elegância da solução implementada, etc). Será disponibilizado um sistema de *feedback* automático em http://li1.lsd.di.uminho.pt que incidirá sobre as tarefas 4 e 6. Este *feedback* tem por objectivo ajudar os grupos a validar as suas implementações mas, tal como na primeira fase do projecto, será distinto do sistema de avaliação automática usado para calcular as notas finais.

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, nomeadamente o Bomberman.

A documentação da API da biblioteca encontra-se disponíivel no em https://hackage.haskell.org/package/gloss.

Instalação

O `Gloss` pode ser instalado através do utilitário `cabal`, o gestor de bibliotecas Haskell que faz parte da distribuição Haskell Platform. Para instalar a biblioteca deve-se então utilizar os comandos:

```
cabal update
cabal install gloss
```

Uma vez instalada a biblioteca, os programas Haskell podem realizar o `import Graphics.Gloss` necessário para utilizar a biblioteca.

Criação de gráficos 2D

O tipo central da biblioteca `Gloss` é o tipo `Picture`. Ele 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 constructor `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 qualquer coisa 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. Para tal existe o constructor `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]
```

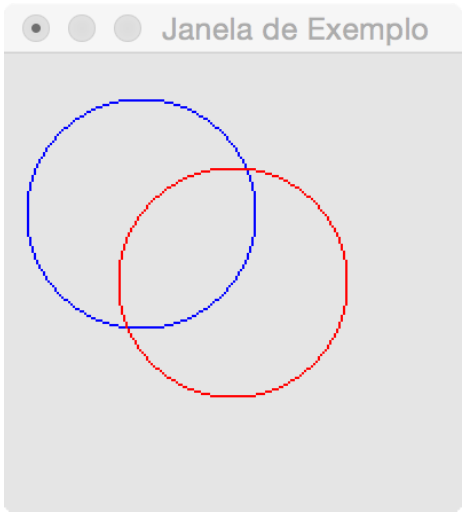
Naturalmente que o objetivo de definir figuras como valores do tipo Picture é podermos visualizá-las no ecrã. Para tal temos de criar uma janela Gloss para lá desenhar o conteúdo da figura. O fragmento de código que se segue permite visualizar a figura `circulos` definida atrás:

```
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
```

De notar que a convenção no Gloss é que a posição com coordenadas `(0,0)` é o centro da janela. Assim, o resultado obtido será a janela:



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 tipo que representa todo o estado do seu jogo.

```
type Estado = ...
```

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`:

```
estadoInicial :: Estado
estadoInicial = ...

desenhaEstado :: Estado -> Picture
desenhaEstado s = ...
```

Para implementar a reação a eventos, nomeadamente o pressionar das teclas, é necessário implementar uma função que, dado um valor do tipo `Event` (definido em `Graphics.Gloss.Interface.Pure.Game`) e um estado do jogo, gera o novo estado do jogo:

```
reageEvento :: Event -> Estado -> Estado
reageEvento e s = ...
```

Note que esta função é muito semelhante à que foi definida na tarefa 2 na primeira fase do projecto.

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`.

```
reageTempo :: Float -> Estado -> Estado
reageTempo n s = ...
```

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" (800, 600) (0, 0)

main :: IO ()
main = 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
```

Inclusão de *bitmaps* 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 Estado = (..., Picture)

estadoInicial :: Picture -> Estado
estadoInicial p = ...
```

```
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
               (estadoInicial p)   -- estado inicial
               desenhaEstado       -- desenha o estado do jogo
               reageEvento         -- reage a um evento
               reageTempo          -- reage ao passar do tempo
```

Uma dica adicional é que o `Gloss` 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
```