Pacman

Enunciado da 2ª fase do projeto de LI1 2020/21

Introdução

Neste enunciado apresentam-se as tarefas referentes à segunda fase do projecto da unidade curricular de Laboratórios de Informática I 2020/21.

Tarefas

Tarefa 4 - Reagir à passagem do tempo

O objectivo desta tarefa é **calcular o efeito da passagem de um instante de tempo** num estado do jogo. Os *inputs* são um número real positivo, que denota uma unidade temporal adicional que passou desde a última atualização e o estado do jogo, que contém a informação necessária para processar uma jogada. O resultado deverá ser um novo estado em que **todos** os jogadores e o mapa foram atualizados.

Esta tarefa deve ser dividida em duas componentes: 1) uma que integra os tipos e funções que advêm da 1ª fase numa biblioteca denominada Ncurses e 2) outra que usa o estado e informação vinda do primeiro componente para fazer avançar o estado em uma iteração, fazendo **"movimentar" todos os jogadores no mapa**.

Integração com NCurses

Neste ponto do projeto o que cada grupo possui é um estado estático que pode ser atualizado/jogado através da função play. Este estado precisa de ser explicitamente alterado por vocês e não está suscetível a ser modificado através de inputs de utilizador (e.g. cliques, pressionar teclas) ou simplesmente através da simples passagem de tempo. Como é óbvio, se queremos desenvolver um jogo próximo do jogo clássico Pacman que conhecemos, o jogo deve ser atualizado consoante a passagem do tempo e jogadas do utilizador.

Para esse fim, será usada a biblioteca NCurses. Esta biblioteca de Haskell foi desenvolvida para permitir aos programadores fazerem aplicações baseadas em interação de terminal com relativa facilidade, permitindo construir interfaces gráficas no terminal com gráficos ou texto e obter informação relativa a eventos efetuados pelo utilizador. Iremos usar esta biblioteca para controlar a passagem de tempo no jogo e reagir aos eventos de pressionamento de teclas por parte do utilizador.

De forma a integrar a biblioteca com o código que possuem após a conclusão da primeira fase, irá ser-vos fornecido um esboço de um ciclo que ajuda a controlar todos estes eventos. O tipo principal sobre o qual irão ter que trabalhar será o tipo Manager, conforme se apresenta de seguida:

```
data Manager = Manager
    {
         state :: State
    ,    pid    :: Int
    ,    step :: Int
    ,    before :: Integer
    ,    delta :: Integer
    ,    delay :: Integer
    }
```

Este tipo Manager contém o estado **state** do jogo (tipo cf definido no módulo Types da 1ª fase), o **pid** do jogador que o utilizador controla (só é possível controlar um jogador tal como no jogo original) e o **step** atual do jogo, que consiste num número que indica por quantas iterações o jogo já passou. Contém também 3 variáveis do tipo Integer, cuja unidade temporal é milissegundos e que servem para controlar a passagem de tempo no jogo:

- delay: representa o intervalo de tempo entre jogadas (em milissegundos);
- before: instante de tempo de tempo em que foi efetuada a última jogada;
- delta: tempo decorrido em milissegundos desde a última jogada.

Estas últimas variáveis servem para controlar a passagem de tempo no nosso estado, sendo que o estado deverá ser atualizado sempre que a condição `delta > delay` se verifique. Por exemplo, se definirmos o delay como 250 (ms), o programa deverá atualizar aproximadamente 4 jogadas por segundo, ou seja, sempre que o tempo entre a última jogada (before) e o tempo atual (delta) seja superior a 250 milissegundos.

Movimentar jogadores

Após a passagem do tempo definido para efetuar uma nova jogada, temos que movimentar os jogadores, tendo em conta também a sua velocidade e respetivo estado. Para este efeito, a função play da Tarefa 2 terá que ser estendida para suportar também jogadas de fantasmas (também têm que ser movimentados ao longo do tempo). As jogadas deverão agora ter também em consideração a velocidade do jogador. Em suma, a nova função play para além do comportamento esperado na tarefa 2, deverá também implementar o seguinte comportamento:

- O Pacman deve abrir e fechar a boca alternadamente entre cada jogada;
- O Pacman em modo Mega deve perder tempo mega em cada jogada;

- O Pacman deve voltar ao modo Normal se o tempo mega for <=0 e estiver em Mega;
- Os Fantasmas devem voltar ao modo Normal se não houver nenhum Pacman em modo Mega;
- Os jogadores devem progredir N jogadas em cada iteração da função, em que N é um número inteiro inferido a partir da velocidade do jogador e paridade da iteração (step) da jogada. Ex.:
    - Se um jogador tiver velocidade 0.5, só deve progredir de 2 em duas iterações;
    - Se um jogador tiver velocidade 1.5, deve progredir 3 jogadas a cada 2 iterações;
    - Se um jogador tiver velocidade 2, deve progredir 2 jogadas em cada iteração.

Neste momento o jogo contempla apenas 2 valores diferentes de velocidade. Contudo, poderá ser possível contemplar mais no futuro, através de novas funcionalidades como subida de nível.

O objectivo desta tarefa é definir a seguinte função no módulo Tarefa4.hs:

```
passTime :: Int -> State -> State
```

que, para um dado *step* e estado, altera o estado do jogo em uma iteração. Para implementar o comportamento esperado desta função, a invocação desta deve resultar na execução de jogadas por parte de todos os jogadores em jogo (presentes na lista do tipo State).

Além disso, devem também complementar as definições das seguintes funções no módulo Main.hs:

```
nextFrame  :: Integer -> Manager -> Manager
updateTime :: Integer -> Manager -> Manager
updateControlledPlayer :: Key -> Manager -> Manager
```

Estas funções permitem controlar o estado do jogo (através da manipulação do tipo Manager e respetivas variáveis) e definir a reação aos eventos do utilizador e passagem do tempo.

Tarefa 5 - Movimentação dos fantasmas

O objetivo desta tarefa é implementar um comportamento para os fantasmas. Para tal, deve começar por complementar as funções da Tarefa 2 para que contemplem jogadas para os

fantasmas. Ou seja, deve rever a Tarefa 2 e considerar que a função `play` pode receber jogadas para os fantasmas.

Em cada instante, é necessário tomar em consideração o estado de cada fantasma e determinar uma jogada, analisando o `State` do momento anterior e devolvendo uma jogada `Play` que depois será processada pela função `play` anteriormente definida. Como tal, requerem-se pelo menos dois métodos para o fazer, um para quando o fantasma está em modo Alive e outro para quando este está em modo Dead.

```
chaseMode :: State -> Int -> Play
scatterMode :: State -> Int -> Play
```

No comportamento normal no qual o fantasma se encontra em modo Alive, pretende-se que este persiga o Pacman. Para tal, considera-se que, sempre que o fantasma se depara com uma parede, ele determine a posição do Pacman e tente tomar a direção mais direta ao mesmo tanto quanto possível, repetindo este comportamento até que atinja outra parede ou o Pacman, ou entre em modo Dead. Outra alternativa a este comportamento é fazer com que o fantasma faça uma perseguição rudimentar do movimento do Pacman, tentando movimentar-se para o lugar quatro espaços à frente dele, baseado na orientação do Pacman.

Caso este esteja em modo Dead, pretende-se que, sempre que o fantasma se depare com uma parede, este se mova para a sua direita, circulando a área onde se encontra no sentido dos ponteiros do relógio como uma forma básica de fuga do Pacman em modo Mega até que se volte a encontrar em modo Alive. Adicionalmente, pretende-se que no momento em que um fantasma se torna Dead, este obtenha a orientação oposta à que tinha no momento anterior.

O objetivo desta tarefa é definir a função no módulo Tarefa5.hs:

```
ghostPlay :: State -> [Play]
```

no qual, ao receber o estado do jogo, devolve um conjunto de jogadas, uma de cada fantasma, com a melhor alternativa que cada consegue para reagir ao Pacman.

Esta função, quando terminada, deve ser integrada na Tarefa 4 de forma a atribuir jogadas aos fantasmas.

Tarefa 6 - Implementar um Robô

O objetivo desta tarefa é implementar um robô que jogue *Pacman* automaticamente.

Em cada instante, o robô tem apenas conhecimento do estado atual do jogo, e pode tomar uma única decisão usando o tipo `Play` definido anteriormente. Para definir a estratégia, deve assumir que o robô recebe sempre um `State` do jogo válido e executa uma jogada a cada iteração do *loop* principal.

O objetivo desta tarefa é definir a função

```
bot :: Int -> State -> Maybe Play
```

que dado o identificador de um jogador e um estado do jogo, devolve uma possível jogada a realizar pelo robô. O tipo `Maybe` modela a possibilidade de o robô não querer efetuar nenhuma jogada (ou seja, `Nothing`).

Tarefa Extra - Gloss

Esta última tarefa é extra e não obrigatória. O objetivo desta tarefa é implementar o jogo completo usando a biblioteca `Gloss`. Na página da UC podem encontrar um breve tutorial sobre esta biblioteca. Como ponto de partida, deve começar por implementar uma versão com uma visualização gráfica simples. Apesar de dever ser construída inicialmente sobre as tarefas anteriores, esta tarefa trata-se acima de tudo de uma "tarefa aberta", onde se estimula que os alunos explorem diferentes possibilidades para melhorar o aspecto final e a jogabilidade do jogo. Sugestões de extras incluem, por exemplo:

- Gráficos visualmente apelativos;
- Suportar diferentes câmaras ou perspectivas;
- Menus de início e fim do jogo;
- Mostrar informação sobre o estado do jogo (como a pontuação, tempo restante de modo mega, etc.);
- Permitir jogar diferentes mapas e/ou carregar mapas definidos pelo utilizador (Tarefa 1);
- Permitir jogar contra outros jogadores e contra *bots* (robôs da Tarefa 6);
- Suportar novas funcionalidades (e.g. permitir jogar com um fantasma).

No cerne desta tarefa estão as funções da Tarefa 2, que permite aos jogadores efetuarem jogadas, e da Tarefa 4, que faz evoluir o jogo ao longo do tempo. Note que apesar do tipo interno do estado do jogo ser usado por estas funções, é provável que cada grupo necessite de criar um novo tipo que contenha informação adicional relevante para a sua implementação do jogo (denominado pelo nome `EstadoGloss` no guião `Gloss` da disciplina). O tipo `State` definido deve no entanto permanecer inalterado.

Relatório

Nesta fase deve ser escrito um relatório, dividido em 3 partes, com maior ênfase sobre a realização das Tarefas 3, 5 e 6. O relatório deverá ser escrito com recurso ao Haddock, sob a forma de comentários por cada tarefa. A documentação gerada para cada tarefa deverá ser organizada em seções, incluindo *pelo menos* as seguintes seções:

- Introdução - Descrever sumariamente o desafio e os resultados;
- Objetivos - Indicar claramente, e por palavras vossas, as estratégias utilizadas e objetivos finais da tarefa;
- Discussão e conclusão - Sumariar os principais resultados obtidos.

Para mais detalhes sobre como deve elaborar um relatório técnico de um trabalho prático consulte o seguinte guião.

Entrega e Avaliação

A data limite para conclusão de todas as tarefas desta primeira fase é **23 de janeiro de 2021 às 23h59m59s (Portugal Continental)** e a respectiva avaliação terá um peso de 50% na nota final do projeto. A submissão será feita automaticamente através do GitHub: 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 três tarefas, será considerada parte integrante do projeto todo o material de suporte à sua realização armazenado no repositório GitHub do respectivo grupo (código, documentação, ficheiros de teste, relatório, etc.). A utilização das diferentes ferramentas abordadas na disciplina (como Haddock, git, 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	20%
Avaliação qualitativa da Tarefa 5	20%
Avaliação automática e qualitativa da Tarefa 6	20%
Qualidade do código	10%
Utilização do git e testes	10%

Relatório sob a forma de documentação Haddock do código	20%

A avaliação automática será feita através de um conjunto de testes que não serão revelados aos grupos. A avaliação qualitativa incidirá sobre aspectos de qualidade de código (por exemplo, estrutura do código, elegância da solução implementada, etc.), qualidade dos testes (quantidade, diversidade e cobertura dos mesmos), documentação (estrutura e riqueza dos comentários) e bom uso do git como sistema de controle de versões. A avaliação do relatório incidirá sobre a sua estrutura e organização de ideias.