Universidade do Minho
Escola de Engenharia
Departamento de Informática

Projecto de Laboratórios de Informática I

Licenciatura em Engenharia Informática

1º Ano — 1º Semestre

Ano Lectivo 2023/2024

Data de Lançamento: 12 de Novembro de 2023

Data Limite de Entrega: 14 de Janeiro de 2024

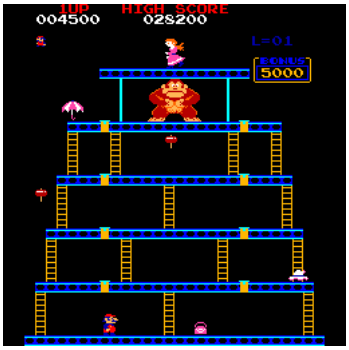
Novembro de 2023

1 Introdução

Neste enunciado apresentam-se as tarefas referentes ao Projecto de Laboratórios de Informática I 2023/2024, que consiste na implementação de um pequeno jogo utilizando a linguagem *Haskell* e a biblioteca gráfica *Gloss*.

1.1 Descrição do jogo

O jogo a implementar na presente edição é conhecido como *Donkey Kong*[1], onde o objectivo consiste em controlar um personagem até à saída de um mapa (representada por uma estrela, geralmente no topo), pelo meio enfrentando ou esquivando-se de inimigos.



O jogo termina assim que o jogador alcance a saída (caso de vitória) ou perca as vidas em confrontos com os inimigos (caso de derrota.)

1.1.1 Caracterização do mapa

Plataformas O mapa do jogo é composto por plataformas sobre as quais os personagens (*i.e.* o jogador e inimigos) se deslocam, e que se encontram ligadas por escadas. As escadas permitem que os personagens subam (resp. desçam) para (resp. de) uma plataforma.

[1] http://www.classicgaming.cc/classics/donkey-kong/play-guide

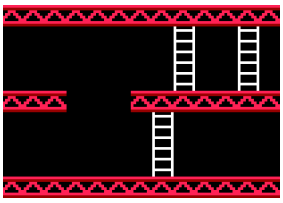


Figura 1: Plataforma e escadas

Alçapões Um alçapão é um bloco de plataforma que se auto-destrói no momento em que o jogador o pisa. O alçapão não se auto-destrói quando um inimigo o pisa.

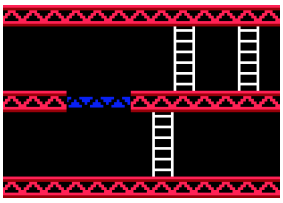


Figura 2: Plataforma com alçapão

Coleccionáveis Espalhados pelo mapa estão objectos que o jogador pode coleccionar, entre os quais destacamos *moedas* e *martelos* que, quando coleccionados, têm o seguinte efeito.

1. As moedas aumentam a *pontuação* do jogador.
2. Os martelos armam o jogador durante um período de 10 segundos. Enquanto activo, qualquer inimigo que se cruze (de frente) com o jogador sofre dano.

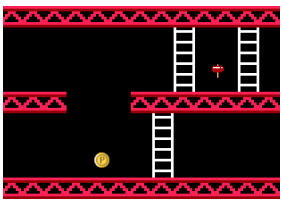


Figura 3: Martelo e moeda

Limites do mapa O mapa é fechado, *i.e.* não é possível aos personagens sair deste.

Estrela O objectivo do jogo é chegar à *estrela*. O jogo termina em situação de vitória assim que o jogador a tocar.

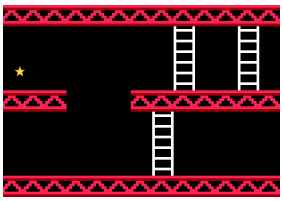


Figura 4: Estrela

1.1.2 Caracterização dos personagens

Movimentos Jogador e personagens movimentam-se de formas semelhantes: deslocam-se para esquerda ou para a direita sob plataformas, podem subir ou descer escadas, cair de várias plataformas, e saltar.

Inimigos Num mapa podem haver inimigos diferentes e cada um com as suas propriedades distintas. O objectivo destes é encurralar o jogador e colidir contra ele. Excepto quando na posse de um *martelo*, o jogador perde uma vida ao colidir (de frente) com um inimigo. Note que, mesmo quando armado, o jogador sofre dano se for atingido por outra direcção que não aquela para onde está voltado.

Jogador O objectivo principal do jogador é tocar na estrela. Enquanto este objectivo não for cumprido, permanecerá no mapa.

1.2 Tipos de dados

Apresentamos de seguida o modelo de jogo que deverá ter em conta na realização das tarefas propostas na secção seguinte. Será fornecido um módulo *Haskell* comum contendo estas definições preliminares, pelo que não necessita de as copiar.

1.2.1 Mapa

O mapa do jogo será representado discretamente por uma matriz de *blocos*, que podem ser: *escadas*, *plataforma*, *alçapão* e *vazio*. O índice $(0, 0)$ da matriz denota o bloco no canto superior esquerdo. O mapa indica também a posição e direcção inicial do jogador, e a posição da saída (*i.e.* a posição da estrela.)

```
data Bloco
  = Escada
  | Plataforma
  | Alcapao
  | Vazio

data Mapa = Mapa (Posicao, Direcao) Posicao [[Bloco]]
```

A `Posicao` é dada por um par ordenado (coluna, linha) de valores `Double`, seguindo o mesmo referencial, *cf.* figura 5.

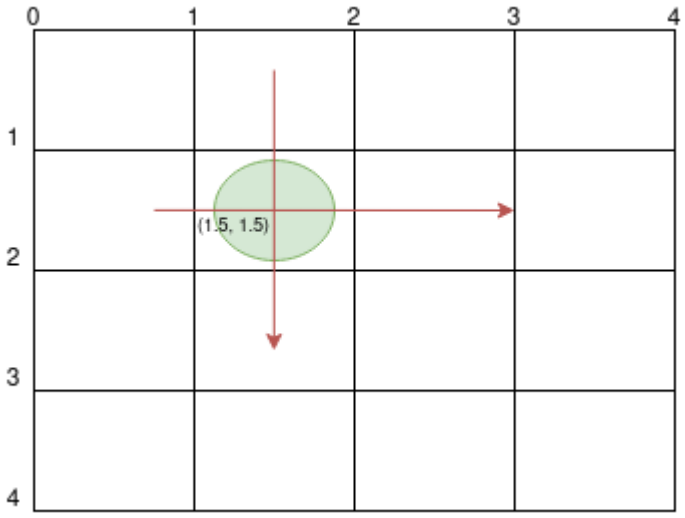
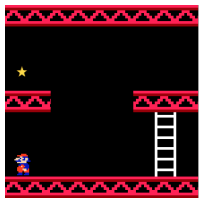


Figura 5: Exemplo da `Posicao` de um objecto.

Exemplo Ao mapa representado na figura 6b corresponderia a representação interna em *Haskell* representada na figura 6a, onde `P` denota `Plataforma`, `V` denota `Vazio`, `E` denota `Escada`, *etc.*

```
Mapa ((0.5, 5.5), Oeste) (0.5, 2.5)
 [[P, P, P, P, P, P, P, P, P, P]
 ,[V, V, V, V, V, V, V, V, V, V]
 ,[V, V, V, V, V, V, V, V, V, V]
 ,[P, P, V, V, V, V, P, P, P, P]
 ,[V, V, V, V, V, V, V, V, E, V]
 ,[V, V, V, V, V, V, V, V, E, V]
 ,[P, P, P, P, P, P, P, P, P, P]
 ]
```

(a) Representação interna



(b) Representação gráfica

Figura 6: Representação do mapa

1.2.2 Personagens

Tanto o jogador como os inimigos serão representados por um tipo de dados comum que contém toda a informação acerca do seu estado actual: posição, velocidade, vidas, *etc.*

```
data Personagem = Personagem
  { velocidade :: Velocidade
  , tipo       :: Entidade
  , posicao    :: Posicao
  , direcao    :: Direcao
  , tamanho    :: (Double, Double)
  , emEscada   :: Bool
  , ressalta   :: Bool
  , vida       :: Int
  , pontos     :: Int
  , aplicaDano :: (Bool, Double)
  }
```

Breve descrição de cada campo:

- `velocidade`. Vector velocidade actual.
- `tipo`. Indica se é um inimigo (e qual) ou o jogador principal. O tipo `Entidade` tem a seguinte definição:

  ```
  data Entidade = MacacoMalvado | Fantasma | Jogador
  ```

- `posicao`. Coordenadas actuais.
- `direcao`. Para qual das direcções (N, S, E, O) se encontra voltado.
- `tamanho`. Tamanho deste personagem, descrito através de um rectângulo largura × altura.
- `emEscada`. Indica se se encontra a subir/descer uma escada.
- `ressalta`. Indica se o personagem deve trocar de direcção ao colidir com uma parede ou ao chegar ao fim de uma plataforma. No caso do jogador, este valor deverá ser `False`.
- `vida`. Quantas vidas lhe restam.
- `pontos`. Pontuação acumulada.
- `aplicaDano`. Indica se tem o martelo armado e por quanto tempo ainda. Quando activo, cria uma *hitbox* à frente do jogador (com as dimensões do mesmo) e qualquer inimigo que a cruze sofre dano.

1.2.3 Jogo

Finalmente, um *jogo* é composto pela informação do mapa, personagens e coleccionáveis.

```
data Jogo = Jogo
  { mapa          :: Mapa
  , inimigos      :: [Personagem]
  , colecionaveis :: [(Colecionavel, Posicao)]
  , jogador       :: Personagem
  }
```

2 Tarefas

2.1 Tarefa 1 – Detecção de colisões

O objectivo desta tarefa é implementar o par de funções

```
colisoesParede      :: Mapa       -> Personagem -> Bool
colisoesPersonagens :: Personagem -> Personagem -> Bool
```

A função `colisoesParede` deve testar se um personagem se encontra em colisão com algum dos limites do mapa (laterais ou topo) ou com algum bloco de plataforma. Por seu turno, a função `colisoesPersonagens` deve testar se dois personagens se encontram em colisão.

Para determinarmos colisões recorremos ao conceito de *hitbox*: o menor rectângulo que contém um personagem ou objecto, e que interpretamos como sendo a área de colisão com esse personagem ou objecto, *cf.* figura 7. Assim, verificar *e.g.* se o jogador se encontra em colisão com um inimigo corresponde a verificar se ambas *hitboxes* se intersectam.

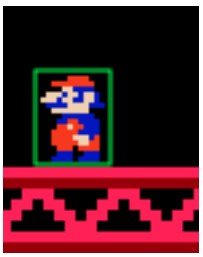


Figura 7: *Hitbox* do jogador: qualquer personagem ou objecto dentro da *hitbox* do jogador diz-se colidir com este.

2.2 Tarefa 2 – Validação do mapa

O objectivo desta tarefa é implementar a função

```
valida :: Jogo -> Bool
```

A função `valida` deve verificar que um dado jogo não viola nenhuma das seguintes restrições:

1. O mapa tem "chão", *i.e.* uma plataforma que impede que o jogador ou outro personagem caia fora do mapa.

2. Todos os inimigos têm a propriedade `ressalta` a `True`, enquanto que o jogador a tem a `False`.

3. A posição inicial de um jogador não pode colidir com a posição inicial de um outro personagem. Note que as posições iniciais de inimigos podem colidir entre estes.

4. Número mínimo de inimigos: 2 (dois.)

5. Inimigos `Fantasma` têm exactamente 1 (uma) vida.

6. Escadas não podem começar/terminar em alçapões, e pelo menos uma das suas extremidades tem que ser do tipo `Plataforma`.

7. Alçapões não podem ser menos largos que o jogador.

8. Não podem existir personagens nem coleccionáveis "dentro" de plataformas ou alçapões, *i.e.* o bloco (na matriz do mapa) correspondente à posição de um personagem ou objecto tem que ser `Vazio`.

2.3 Tarefa 3 – Movimentação de personagens

O objectivo desta tarefa é implementar a função

```
movimenta :: Semente -> Tempo -> Jogo -> Jogo
```

A função `movimenta` deve animar todos os personagens, *i.e.* calcular as suas novas posições e respectivas consequências. A seguinte lista **não é exaustiva**:

1. Um inimigo perde 1 (uma) vida se estiver dentro da *hitbox* de dano de um jogador armado. Por jogador armado entende-se um jogador cuja componente `aplicaDano` esteja activa e com tempo restante. Note que a *hitbox* de dano não é a mesma *hitbox* do jogador, mas antes uma *hitbox* com as dimensões do jogador posicionada exactamente à frente do jogador, *cf.* figura 8.

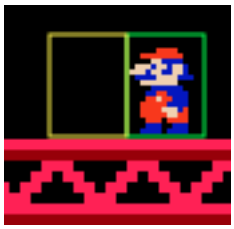


Figura 8: *Hitbox* de dano *vs.* *hitbox* do jogador.

2. Um inimigo morre quando as suas vidas chegam a 0 (zero.) Nessa altura, deixa de ser representado no mapa. Note que um inimigo (enquanto `Personagem`) não é removido da lista de inimigos de um `Jogo` mesmo quando morre.

3. Efeito de gravidade: qualquer personagem que não esteja sobre uma plataforma deverá "cair". A velocidade da queda será dada no código pela variável `gravidade`.

4. O jogador deverá perder 1 (uma) vida se for atingido por um inimigo. O inimigo não perde vidas nesta situação.

5. Ao recolher um coleccionável, este deverá desaparecer do mapa. No caso do martelo, este deve armar o jogador durante 10 (dez) segundos a contar do momento da sua recolha. No caso de uma moeda, esta deve aumentar a pontuação do jogador.

6. Um alçapão deverá desaparecer se o jogador o pisar. Por outro lado, não sofrerá qualquer alteração se for um inimigo a pisá-lo.

7. Colisões: personagens não podem sair do mapa nem atravessar blocos de plataforma. Mais ainda, deve também assumir que a *hitbox* da estrela ou de um objecto coleccionável tem tamanho 1×1, *i.e.* estrela/martelo/moeda ocupam um bloco da matriz na totalidade.

O parâmetro `Semente` deve ser usado para a geração de um número aleatório que deverá usar para introduzir alguma aleatoriedade ao movimento dos inimigos.

2.4 Tarefa 4 – Actualização das velocidades e direcções dos personagens

O objectivo desta tarefa é implementar a função

```
atualiza :: [Maybe Acao] -> Maybe Acao -> Jogo -> Jogo
```

A função `atualiza` deve validar e calcular as novas direcções e velocidades dos personagens (inimigos e jogador) de acordo com as acções dadas. O parâmetro `[Maybe Acao]` representa a lista de acções a aplicar a todos os inimigos, havendo tantas acções quantos inimigos no `Jogo`. O parâmetro `Maybe Acao` representa a acção a aplicar ao jogador.

Mais adiante no desenvolvimento do projecto esta função será invocada antes de um novo *frame* e sempre que o utilizador pressionar uma tecla, de acordo com o seguinte esquema:

1. Ao pressionar uma tecla deverá convertê-la (se possível) numa `Acao`, *e.g.* `KeySpace` fará o jogador saltar, e invocar a função `atualiza` com o argumento `Just Saltar` para a acção do jogador.

2. A cada novo *frame* deverá atribuir uma acção a todos os inimigos. A acção atribuída poderá ser tão sofisticada quanto quiser. Poderá ser (inclusivamente) `Nothing`, o que faria com que os inimigos não reagissem à passagem do tempo, ou seja, manteriam os seus percursos.

A lista de acções possíveis é dada pelo seguinte tipo de dados:

```
data Acao
  = Subir
  | Descer
  | AndarDireita
  | AndarEsquerda
  | Saltar
  | Parar
```

Breve descrição de cada acção:

- `Subir`. Somente válida quando o personagem já se encontra a usar uma escada ou na mesma posição desta, para que possa começar a usá-la. O seu efeito consiste em transformar o vector velocidade do personagem, decrementando a sua componente y e deixando a sua componente x a 0 (zero.)

- `Descer`. Análoga à acção `Subir`, desta feita incrementando a sua componente y.

- `AndarDireita`. Somente válida quando não está a subir ou a descer uma escada. O seu efeito consiste em incrementar o vector velocidade do personagem na componente x. Também deve ajustar a direcção do personagem, caso esta não seja já `Este`. Se o personagem tiver a propriedade `ressalta` a `True` e se a sua nova posição colidir com uma parede ou o fizer cair de uma plataforma, então a sua nova direcção deve ser oposta, assim permitindo que volte para trás em vez de ficar bloqueado em frente a uma parede ou caia de uma plataforma.

- `AndarEsquerda`. Análoga à acção `AndarDireita`, desta feita decrementando a sua componente x.

- `Saltar`. Somente válida fora de uma escada. O seu efeito consiste em incrementar o vector velocidade do personagem na componente y.

- `Parar`. Impede que o jogador avance mais, efectivamente definindo a sua velocidade a 0. Note que se o personagem se encontrar em queda esta acção não tem qualquer efeito.

A ausência de acção, *i.e.* `Nothing` deve seguir o princípio da inércia do movimento, quer para o jogador quer para os inimigos.

2.5 Tarefa 5 – Parte gráfica

O objectivo desta tarefa consiste em aproveitar todas as funcionalidades já elaboradas nas outras *Tarefas* e construir uma aplicação gráfica que permita um utilizador jogar. Para o interface gráfico deverá utilizar a biblioteca *Gloss*.

O grafismo e as funcionalidades disponibilizadas ficam à criatividade dos alunos, sendo que, **no mínimo**, a aplicação deverá:

1. Mostrar um menu inicial, no qual deve incluir a opção para jogar.
2. Uma vez escolhida a opção de jogar, deverá carregar um puzzle e iniciar o jogo propriamente dito, *i.e.* animar os inimigos e permitir que o utilizador controle o jogador através do teclado. Finalmente, em situação de vitória ou derrota o jogo deverá terminar, sendo mostrada uma mensagem ou, quiçá, avançando para outro nível.

2.6 Tarefa 6 – Extras

Nesta tarefa os alunos poderão escolher como melhorar o seu projecto com funcionalidades extra. Sugestões:

1. Carregar um nível do sistema de ficheiros – por exemplo, através de um menu de navegação.
2. Guardar um jogo em progresso (vulgo *save game*) e continuá-lo mais tarde.
3. Manter uma pontuação (vulgo *highscore*) para cada nível jogado e jogador.
4. Disponibilizar um editor de mapas.
5. Adicionar novos tipos de inimigos ou blocos de mapa.
6. *Bot* de jogador.
7. Escolha de temas gráficos.
8. Modo de jogo campanha.
9. Permitir um mapa maior que o tamanho da janela.
10. *Cheatcodes*.

11. *Etc.*

Poderá alterar o código (*e.g.* tipos de dados) dado e/ou desenvolvido nas primeiras tarefas, desde que não quebre as funcionalidades implementadas nem os respectivos testes.

3 Entrega e Avaliação

A data limite para conclusão de todas as tarefas é **14 de Janeiro de 2024** e a respectiva avaliação terá um peso de **60%** na nota final da UC. A submissão será feita automaticamente através do GitLab onde, nessa 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 tarefas será considerada parte integrante do projecto todo o material de suporte à sua realização armazenado no repositório do respectivo grupo (código, documentação, ficheiros de teste, *etc.*). A utilização das diferentes ferramentas abordadas no curso (como *Haddock* ou *git*) deve seguir as recomendações enunciadas nas respectivas sessões laboratoriais. A avaliação do projecto terá em linha de conta todo esse material, atribuindo-lhe os seguintes pesos relativos:

Componente	**Peso**
Avaliação automática e qualitativa das tarefas	85%
Quantidade e qualidade dos testes	7%
Documentação do código	5%
Utilização do sistema de revisões (Git)	3%

Os grupos de trabalho devem ser compostos por **dois elementos** pertencentes ao **mesmo turno PL**. Cada grupo gere a distribuição de tarefas como achar conveniente, mas ambos os elementos do grupo são responsáveis pelo trabalho submetido e ambos assumem a autoria global do trabalho entregue.

Após a submissão do projecto, haverá uma discussão oral do trabalho, onde poderá ser pedida alteração de alguma componente do código.

A avaliação é individual: a nota final será atribuída **independentemente** a cada membro do grupo em função da respectiva prestação.

A avaliação automática será feita através de um conjunto de testes que não serão revelados aos grupos. A avaliação qualitativa incidirá sobre aspectos da implementação não passíveis de serem avaliados automaticamente (como a estrutura do código ou elegância da solução implementada).