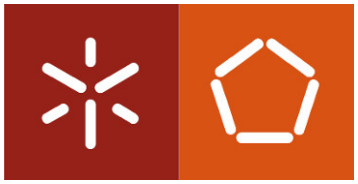

Universidade do Minho
Departamento de Informática
Licenciatura em Engenharia Informática

Projeto Laboratorial

Laboratórios de Informática I

Ano letivo 2024/2025

Data de lançamento: 17 de novembro de 2024
Data limite de entrega: 13 de janeiro de 2025

Conteúdo

1 Introdução

Neste enunciado apresentam-se as tarefas referentes ao Projeto de Laboratórios de Informática I para o ano letivo 2024/2025. O projeto consiste na implementação de um pequeno jogo utilizando a linguagem *Haskell* e a biblioteca gráfica *gloss*.

O jogo a implementar na presente edição é inspirado no conhecido jogo *Tower Defense*[1], onde o objetivo consiste em impedir que ondas de inimigos alcancem a base do jogador, através da colocação estratégica de torres que automaticamente disparam sobre estes. A versão adaptada deste jogo foi apelidada de *Immutable Towers*.

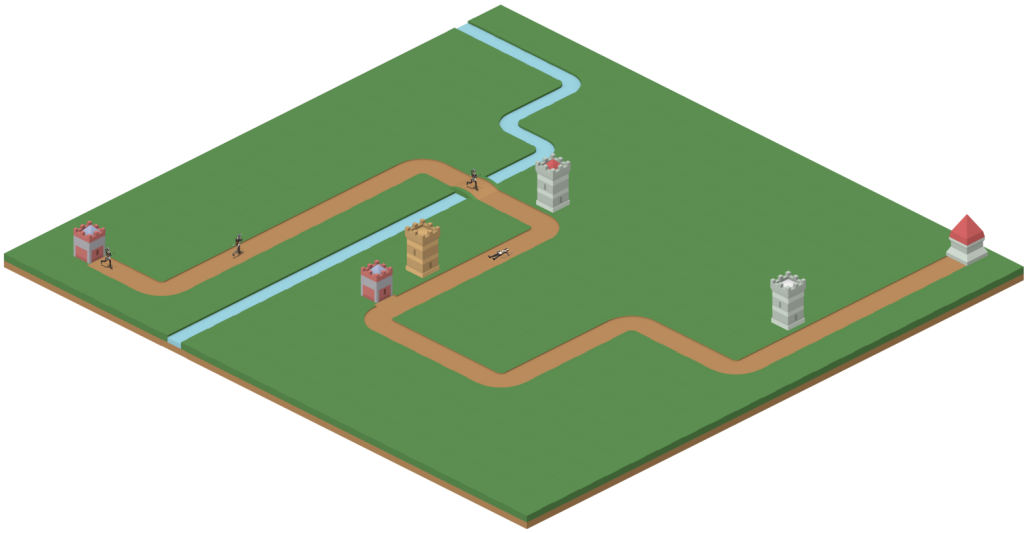

Figura 1: Exemplo representativo do Jogo *Immutable Towers*

O jogo tem início num mapa onde, para além dos vários tipos de terreno e obstáculos, existem:

1. a base do jogador; e
2. um ou mais portais, que são os sítios de onde os inimigos surgirão.

Com esta informação, o jogador deve agora comprar e colocar estrategicamente torres no mapa, que atacarão os inimigos sempre que estes estejam dentro do seu raio de alcance.

Sempre que um inimigo alcança a base do jogador, o nível de vida da base decresce devido ao dano infligido. Sempre que uma torre elimina completamente um inimigo, o jogador é premiado com créditos que pode usar para comprar mais torres.

O jogo termina com vitória do jogador quando não há mais inimigos por derrotar, ou com derrota do jogador quando o nível de vida da base do jogador chega a 0 (zero).

[1] `https://en.wikipedia.org/wiki/Tower_defense`

2 Descrição do jogo

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.

2.1 Mapa e terrenos

O mapa do jogo é caracterizado por um plano composto por diferentes tipos de terreno. Existem inicialmente três tipos de terreno: *relva*, *terra* e *água*. Diferentes tipos de terreno têm diferentes características e restrições, entre estas:

1. Os inimigos apenas se podem deslocar sobre *terra*.
2. As torres apenas podem ser colocadas sobre *relva*.

```
data Terreno = Relva | Agua | Terra
```

(a) Terreno do tipo *Relva*

(b) Terreno do tipo *Terra*

(c) Terreno do tipo *Água*

Figura 2: Tipos de dados relativos a terrenos

O mapa do jogo será representado através de uma grelha retangular onde cada célula comporta um tipo de terreno. Nesta grelha, a coordenada $(0, 0)$ refere-se à célula no canto superior esquerdo.

```
type Mapa    = [[Terreno]]
```

Figura 3: Tipo de dados relativo ao mapa

Com este tipo de dados, podemos construir um mapa como definido na figura 4 e representado visualmente na figura 5.

```
mapa01 :: Mapa
mapa01 =
    [ [t, t, r, a, a, a],
      [r, t, r, a, r, r],
      [r, t, r, a, r, t],
      [r, t, r, a, r, t],
      [r, t, t, t, t, t],
      [a, a, a, a, r, r]
    ]
  where
    t = Terra
    r = Relva
    a = Agua
```

Figura 4: Exemplo de um mapa

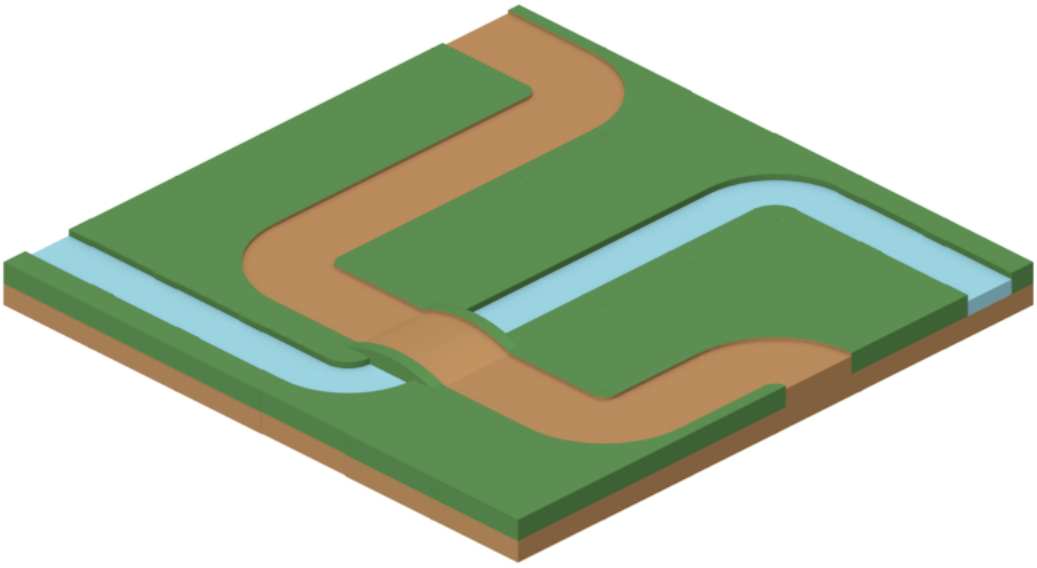

Figura 5: Representação visual do `mapa01` (definido na figura 4)

Uma `Posicao` denota o centro de uma qualquer entidade (*e.g.* inimigo, torre, base, portal) sobre o mapa.

```
type Posicao = (Float, Float)
```

Figura 6: Definição do tipo de dados `Posicao`

Considere o `mapa02` definido na figura 7 e as coordenadas das células.

```
mapa02 :: Mapa
mapa02 =
    [ [Terra, Terra],
      [Relva, Terra]
    ]
```

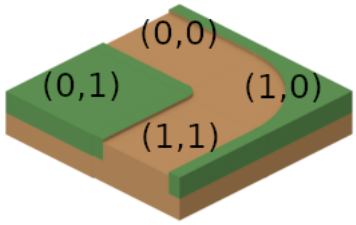

Figura 7: Definição de um mapa e sua representação visual com coordenadas

2.2 Base do jogador

Num mapa existe exclusivamente uma única base. O objetivo dos inimigos é alcançá-la. Sempre que um inimigo alcance a base do jogador, é retirado do mapa e a base do jogador sofrerá perda do nível de vida.

Figura 8: Imagem ilustrativa da *Base* do jogador

A base é representada pelo tipo de dados `Base` (ver figura 9), onde:

1. `vidaBase` denota o nível de vida da base.
2. `posicaoBase` denota a posição no mapa da base.
3. `creditosBase` denota os créditos da base, que podem ser usados para comprar torres na loja.

```
type Creditos = Int
data Base     = Base {
     vidaBase     :: Float,
     posicaoBase  :: Posicao,
     creditosBase :: Creditos
}
```

Figura 9: Tipos de dados relativos à base do jogador

2.3 Torres

As torres são responsáveis por automaticamente detetarem e atacarem inimigos ao seu alcance.

(a) *Projétil* do tipo *Fogo*

(b) *Projétil* do tipo *Gelo*

(c) *Projétil* do tipo *Resina*

Figura 10: Imagens ilustrativas para as *Torres* de acordo com o tipo de *Projétil*

Torres são representadas através do tipo de dados `Torre` (ver figura 11), onde:

1. `posicaoTorre` denota a posição atual da torre no mapa.
2. `danoTorre` dita quanto dano à vida de um inimigo atingido deve ser retirado.
3. `alcanceTorre` especifica a distância até onde consegue identificar inimigos.
4. `projetilTorre` indica o tipo de projétil que lança.
5. `rajadaTorre` indica quantos inimigos consegue atacar de uma só vez. Este valor é constante.
6. `cicloTorre` indica o tempo de espera entre disparos. Este valor é constante.
7. `tempoTorre` indica o tempo restante até que a torra possa voltar a disparar, *vulg.* *"cooldown"*. Este valor irá variar no decorrer do jogo, sendo atualizado (para `cicloTorre`) sempre que a torre disparar, ou decrementado com o passar do tempo. Enquanto este valor for positivo, a torre não pode disparar.

```
type Distancia = Float
type Tempo     = Float
data Duracao   = Finita Tempo | Infinita
data Torre = Torre {
     posicaoTorre   :: Posicao
     danoTorre      :: Float,
     alcanceTorre   :: Distancia,
     rajadaTorre    :: Int,
     cicloTorre     :: Tempo,
     tempoTorre     :: Tempo,
     projetilTorre  :: Projetil
}
```

Figura 11: Tipos de dados relativos às torres do jogo

2.4 Loja

A qualquer altura no decorrer do jogo, o jogador pode adquirir e colocar novas torres no mapa. O processo de adquirir uma torre passa por comprar (usando os créditos que a sua base permite) uma torre na loja. Uma vez adquirida, deve agora colocar a torre numa posição válida do mapa. Assim que seja colocada, a torre fica imediatamente ativa.

Uma loja será representada pelo tipo de dados `Loja` (ver figura 12), que mais não se trata que uma relação preço-torre.

```
type Loja = [(Creditos, Torre)]
```

Figura 12: Tipos de dados relativos à loja do jogo

2.5 Projéteis e sinergias

Projéteis são, essencialmente, as armas das torres. Projéteis distintos têm consequências distintas quando atingem um inimigo, para além do dano inicial que este sofre em virtude de ter sido atingido. Há, no mínimo, três tipos de projéteis:

1. *Fogo*, cujo efeito é aplicar dano por segundo ao inimigo durante um período de tempo.
2. *Gelo*, cujo efeito é proibir temporariamente os movimentos do inimigo.
3. *Resina*, cujo efeito com duração infinita, é reduzir a velocidade do inimigo por um fator a determinar.

Para além disso, projéteis podem ter efeitos compostos, aos quais damos o nome *sinergias*. A título de exemplo, atingir um inimigo com *Fogo* enquanto este estiver sob o efeito de um projétil *Resina*, exacerba o efeito do projétil *Fogo*; ou atingir um inimigo com *Gelo* enquanto este estiver sob o efeito de um projétil *Fogo*, cancela ambos os projéteis.

Um projétil é denotado pelo tipo de dados `Projetil` (ver figura 13), onde:

1. `tipoProjetil` denota o tipo de projétil.
2. `duracaoProjetil` denota a duração que o efeito do projétil tem no inimigo.

```
data TipoProjetil = Fogo | Gelo | Resina
data Projetil = Projetil {
     tipoProjetil    :: TipoProjetil,
     duracaoProjetil :: Duracao
}
```

Figura 13: Tipos de dados relativos aos projéteis

2.6 Inimigos

Um inimigo é caracterizado pelo tipo de dados `Inimigo` (ver figura 14), onde:

1. `direcaoInimigo` denota a direção atual do inimigo.
2. `posicaoInimigo` denota a posição atual do inimigo.
3. `vidaInimigo` denota o nível de vida do inimigo.
4. `velocidadeInimigo` denota a velocidade máxima do inimigo. Este valor é constante.
5. `ataqueInimigo` denota quanto dano este inimigo causará à base do jogador, caso a consiga alcançar.
6. `butimInimigo` denota quantos créditos o jogador obterá caso o consiga eliminar.
7. `projeteisInimigo` denota os projéteis atualmente em efeito no inimigo.

```
data Direcao = Norte | Sul | Este | Oeste
data Inimigo  = Inimigo {
     posicaoInimigo    :: Posicao,
     direcaoInimigo    :: Direcao,
     vidaInimigo       :: Float,
     velocidadeInimigo :: Float,
     ataqueInimigo     :: Float,
     butimInimigo      :: Creditos,
     projeteisInimigo  :: [Projetil]
}
```

(a) Direção *Norte*

(b) Direção *Sul*

(c) Direção *Este*

(d) Direção *Oeste*

Figura 14: Tipos de dados relativos ao inimigo

2.7 Portais

Num mapa existe, no mínimo, um portal de inimigos. Um portal é responsável por lançar inimigos no mapa que surgirão no mesmo local do portal. Mais ainda, os inimigos que um portal contém por lançar, encontram-se agrupados em ondas.

Quando um portal inicia o lançamento de uma onda, todos os inimigos dessa onda devem ser lançados um-a-um. Quando se esgotarem os inimigos dessa onda, haverá um compasso de espera até que possam ser lançados os inimigos da próxima onda. No decorrer de um jogo, é possível que um portal esgote todos os seus inimigos, momento a partir do qual fica inativo.

Figura 15: Imagem ilustrativa de um *Portal* de inimigos

Portais são representados através do tipo de dados `Portal` (ver figura 16), indicando a sua localização no mapa e ondas de inimigos. Numa `Onda` temos:

1. `inimigosOnda`, que lista por ordem de saída os inimigos.
2. `cicloOnda`, que indica o intervalo de tempo (mínimo) entre o lançamento de inimigos. Este valor é constante.
3. `tempoOnda`, que indica o tempo restante até ao lançamento (*vulg. "spawn"*) do próximo inimigo. Este valor é atualizado para o valor de `cicloOnda` sempre que um inimigo for lançado, ou decrementado com o passar do tempo. Enquanto este valor for positivo, a onda não pode lançar inimigos.
4. `entradaOnda`, que indica o tempo restante até a onda se tornar ativa. Este valor é decrementado com o passar do tempo, mas só quando a onda anterior tiver sido totalmente lançada. Enquanto este valor for positivo, a onda encontra-se inativa.

```
data Onda = Onda {
     inimigosOnda :: [Inimigo],
     cicloOnda    :: Tempo,
     tempoOnda    :: Tempo
     entradaOnda  :: Tempo,
}
data Portal = Portal {
     posicaoPortal :: Posicao,
     ondasPortal   :: [Onda]
}
```

Figura 16: Tipos de dados relativos aos portais

2.8 Jogo

Finalmente, o estado do jogo num qualquer instante é denotado pelo tipo de dados `Jogo` (ver figura 17), onde:

1. `baseJogo` denota a base do jogo.
2. `portaisJogo` denota os portais de inimigos do jogo.
3. `torresJogo` denota as torres atualmente ativas no jogo.
4. `mapaJogo` denota o mapa atual do jogo.
5. `inimigosJogo` denota os inimigos atualmente em jogo. Esta lista **não contém** os inimigos que ainda não foram lançados pelos portais.
6. `lojaJogo` denota a loja com as possíveis torres a comprar.

```
data Jogo = Jogo {
     baseJogo     :: Base,
     portaisJogo  :: [Portal],
     torresJogo   :: [Torre],
     mapaJogo     :: Mapa,
     inimigosJogo :: [Inimigo],
     lojaJogo     :: Loja
}
```

Figura 17: Tipo de dados relativo ao estado de jogo

```
jogo01 :: Jogo
jogo01 = Jogo {
     baseJogo     = Base {posicaoBase = (5.5,2.5), ...},
     portaisJogo  = [Portal {posicaoPortal = (0.5,0.5), ...}],
     torresJogo   = [
             Torre {
                 posicaoTorre = (4.5,3.5),
                 projetilTorre = Projectil{tipoProjetil = Resina, ...} ...},
             Torre {
                 posicaoTorre = (0.5,2.5),
                 projetilTorre = Projectil{tipoProjetil = Gelo} ...}
         ],
     mapaJogo     = mapa01,
     inimigosJogo = [
             Inimigo {
                 posicaoInimigo = (1.5,1.5),
                 direcaoInimigo = Sul, ...},
             Inimigo {
                 posicaoInimigo = (2.5,4.5),
                 direcaoInimigo = Este, ...}
         ],
     ...
}
```

(a) Definição parcial do `jogo01`

(b) Representação visual do `jogo01`

3 Tarefas

Cada uma das seguintes tarefas deverá ser implementada num módulo (e, por conseguinte, ficheiro) próprio.

3.1 Tarefa 1 — Invariantes de jogo

O objetivo desta tarefa é implementar a função

```
validaJogo :: Jogo -> Bool
```

que verifica se um estado de jogo é válido. Para isso, deverá ter em conta os seguintes critérios:

1. Relativamente a portais:
   (a) Existe pelo menos um.
   (b) Estão posicionados sobre *terra*.
   (c) Existe pelo menos um caminho (de *terra*) ligando um portal à base. De outra forma, não seria possível a base sofrer dano.
   (d) Não podem estar sobrepostos a torres ou à base.
   (e) Há, no máximo, uma onda ativa por portal.
2. Relativamente a inimigos:
   (a) Todos os inimigos por lançar têm a posição do respetivo portal, nível de vida positivo, e lista de projéteis ativos vazia.
   (b) Todos os inimigos em jogo encontram-se sobre *terra*.
   (c) Não podem estar sobrepostos a torres.
   (d) Velocidade não pode ser negativa.
   (e) A lista de projéteis ativos deve encontrar-se "normalizada", ou seja:
      i. Não pode conter mais do que um projétil do mesmo tipo.
      ii. Não pode conter, simultaneamente, projéteis do tipo `Fogo` e `Resina`, nem `Fogo` e `Gelo` (ver secção 3.2 para mais detalhes).
3. Relativamente a torres:
   (a) Estão posicionadas sobre *relva*.
   (b) O seu alcance é um valor positivo.
   (c) A rajada é um valor positivo.

(d) O ciclo é um valor não negativo.

(e) Não podem estar sobrepostas.

4. Relativamente à base:

   (a) Está colocada sobre *terra*.

   (b) Não tem crédito negativo.

   (c) Não pode estar sobreposta a uma torre ou portal.

3.2 Tarefa 2 — Auxiliares de jogo

Esta tarefa consiste na implementação de várias funções auxiliares que irá usar para o desenvolvimento da mecânica de jogo.

1. Defina uma função que calcula os inimigos ao alcance de uma dada torre:

   ```
   inimigosNoAlcance :: Torre -> [Inimigo] -> [Inimigo]
   ```

2. Defina uma função que atualiza o estado de um inimigo assumindo que este acaba de ser atingido por um projétil de uma torre:

   ```
   atingeInimigo :: Torre -> Inimigo -> Inimigo
   ```

   O inimigo resultante deve:

   (a) Perder tanto nível de vida quanto o dano da torre.

   (b) Consoante o tipo de projétil, atualizar a lista de projéteis ativos de acordo com as seguintes sinergias, ordenadas por precedência:

      i. `Fogo` e `Gelo` cancelam-se mutuamente, *i.e.* ambos projéteis devem ser removidos da lista de projéteis do inimigo.

      ii. `Fogo` e `Resina` dobra a duração do fogo, *i.e.* à lista de projéteis do inimigo deve somente dobrar a duração do projétil `Fogo` e, caso necessário, remover o projétil `Resina` da lista.

      iii. Projéteis iguais somam as suas durações. Por exemplo, `Fogo` e `Fogo`, simplesmente resulta num único `Fogo` com as durações somadas. O mesmo para `Gelo` e `Gelo` e `Resina` e `Resina`.

      iv. As restantes combinações de projéteis não resultam em sinergias e, portanto, a lista de projéteis apenas deve ser atualizada com a adição do projétil que atingiu o inimigo.

v. Quando o inimigo não contém projéteis ativos, o projétil que o atingiu deve ser incluído na lista.

3. Defina uma função que coloca o próximo inimigo a ser lançado por um portal em jogo:

   ```
   ativaInimigo :: Portal -> [Inimigo] -> (Portal, [Inimigo])
   ```

   Dado um `Portal` e a lista de inimigos atualmente em jogo, deve **mover** o próximo inimigo a ser lançado pelo portal para a lista de inimigos ativos. Por mover, entende-se que o `Portal` resultante perde aquele inimigo, que deverá agora ser incluído na lista de inimigos ativos. Deve também garantir que move o inimigo **certo**, *i.e.* o primeiro inimigo da `Onda` atualmente ativa.

4. Defina uma função que decide se o jogo terminou:

   ```
   terminouJogo :: Jogo -> Bool
   ```

   que, por outras palavaras, verifica se o jogador ganhou ou perdeu o jogo. A vitória é caracterizada pela ausência de inimigos ativos ou inativos (*i.e.* por lançar) simultaneamente com um nível de vida da base positivo. A derrota é caracterizada simplesmente por um nível de vida da base inferior ou igual a 0 (zero.) Implemente, respetivamente, as funções

   ```
   ganhouJogo :: Jogo -> Bool
   perdeuJogo :: Jogo -> Bool
   ```

3.3 Tarefa 3 — Mecânica de jogo

O propósito desta tarefa é implementar integralmente a simulação do modelo de jogo, concentrando-se nos movimentos dos inimigos e nas interações com as torres e a base. Para isso, deverá definir a função:

```
atualizaJogo :: Tempo -> Jogo -> Jogo
```

devendo atender aos seguintes critérios relativos ao comportamento das torres, inimigos e portais.

3.3.1 Comportamento das torres

As torres devem:

1. Detetar inimigos dentro do seu alcance.
2. Escolher e disparar automaticamente projéteis contra os inimigos detetados.

A implementação destes comportamentos deve respeitar os parâmetros específicos de cada torre, nomeadamente:

1. Uma torre não poderá disparar mais projéteis do que o permitido pela sua rajada, nem disparar enquanto não estiver pronta, de acordo com o parâmetro `tempoTorre`.
2. Sempre que atingir inimigos, deverá atualizar adequadamente os níveis de vida dos mesmos, conforme o dano de ataque da torre. Caso existam sinergias, atualizar igualmente os projéteis associados a esses inimigos.
3. Sempre que disparar, deve atualizar o parâmetro `tempoTorre` da torre.

3.3.2 Comportamento dos inimigos

Os inimigos devem deslocar-se em direção à base do jogador mas, à medida que avançam, as torres de defesa estarão a atacá-los através de projéteis. Além do dano causado pelo seu impacto, os projéteis podem aplicar efeitos adicionais nos inimigos de acordo com o seu tipo, como de seguida se descreve:

1. `Gelo`: impede o movimento do inimigo.
2. `Resina`: reduz a velocidade do inimigo, por um fator a determinar, apenas uma vez no momento do impacto.
3. `Fogo`: aplica dano por segundo ao inimigo por uma taxa a definir.

Adicionalmente:

1. Quando o nível de vida de um inimigo atingir 0 (zero) ou valor inferior, este deverá ser removido da lista de inimigos ativos, e o seu butim deverá ser adicionado aos créditos da base do jogador.
2. A distância percorrida por cada inimigo, no intervalo de tempo fornecido, deverá ser ajustada em função destes efeitos e do parâmetro `velocidadeInimigo`. Por exemplo, um inimigo uma velocidade de 1.0 poderá percorrer, por segundo, uma distância de 1.0. Se estiver sob o efeito de um projétil `Resina` que reduz a sua velocidade, por exemplo, em 0.3 unidades, deverá no máximo percorrer 0.7 unidades de distância por segundo.

3. Quando um inimigo atinge a base, deve ser eliminado da lista de inimigos ativos, e a base deve perder uma quantidade de vida equivalente ao dano que o inimigo inflige

3.3.3 Comportamento dos portais

Os portais deverão lançar inimigos conforme as suas especificações:

1. Apenas ondas ativas — aquelas cujo parâmetro `entradaOnda` seja igual ou inferior a 0 (zero) — poderão lançar inimigos.
2. A ordem de saída dos inimigos numa onda será determinada pela ordem natural da lista de inimigos.
3. A capacidade de lançar inimigos estará sujeita aos parâmetros `cicloOnda` e `tempoOnda`.

3.4 Tarefa 4 — Interface gráfica

O objetivo desta tarefa consiste em aproveitar todas as funcionalidades já elaboradas nas outras *Tarefas* e construir uma aplicação gráfica que permita um utilizador jogar. Para a interface gráfica deverá usar a biblioteca *gloss* estudada nas aulas práticas.

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

1. Permitir que o jogador compre e coloque torres num mapa.
2. Simular corretamente todas as ações dos portais, base, inimigos e torres.
3. Reagir à vitória ou derrota do jogador (p.e. mostrar uma mensagem na tela)

3.5 Tarefa 5 — Extras

Esta tarefa destina-se a encorajar a criatividade e o pensamento crítico dos alunos. Podem ser implementadas funcionalidades adicionais ou melhorias no jogo que não estão explicitamente listadas nas tarefas anteriores. Alguns exemplos de extras incluem:

1. Adicionar novos tipos de projéteis com efeitos diferentes nos inimigos. Por exemplo, projéteis que causem medo no inimigo, resultando num movimento contrário à base do jogador.
2. Adicionar novos tipos de terrenos com efeitos especiais. Por exemplo, um pavimento do tipo asfalto que aumenta a velocidade dos inimigos que nele se deslocam.
3. Adicionar novos tipos de inimigos com comportamentos únicos (repare que na secção 2.5, não existe um campo para o tipo de inimigo). Por exemplo, inimigos que só andam pela água ou inimigos com invisibilidade temporária.

4. Adicionar outras características às torres. Por exemplo, um campo que descreve a precisão da torre, i.e., a probabilidade de acertar num inimigo.

5. Adicionar um sistema de melhorias a torres, onde o jogador pode gastar créditos para transitar uma torre para um estado mais poderoso, mantendo o mesmo tipo de projétil. Por exemplo, a torre passa a disparar mais projéteis por rajada e tem um alcance maior.

6. Adicionar sinergias novas entre projéteis. Por exemplo, um inimigo que esteja em chamas e está sobreposto com outro inimigo, transfere o fogo também para este.

7. Criar obstáculos dinâmicos ou modificáveis pelo jogador durante o jogo. Por exemplo, colocação de pedragulhos no caminho dos inimigos.

8. Possibilitar guardar e carregar jogos, permitindo ao jogador retomar um jogo anteriormente guardado.

9. Criar um sistema de progressão de jogo onde o jogador pode desbloquear novos mapas após vencer noutros.

10. Desenvolver modos de jogo alternativos, como um modo de sobrevivência onde as ondas de inimigos são infinitas.

11. Disponibilizar diferentes pacotes de texturas para o jogo, permitindo ao jogador personalizar o aspeto visual do jogo.

12. Implementar uma interface de configuração onde o jogador possa personalizar as regras do jogo, como o número de inimigos ou os atributos das torres.

13. Implementar uma interface gráfica de criação de mapas, onde o jogador pode desenhar o seu próprio mapa.

14. Implementar um *bot* que jogue o jogo automaticamente, i.e. que realize as várias ações do jogador (comprar, colocar, mover, destruir, etc. torres) de forma autónoma. Alternativamente, criar um botão que apenas sugere a próxima ação ao jogador.

15. Adicione outras funcionalidades gráficas, como animações, zoom, mudança de perspetiva (p.e. de 2D para 2.5D, e vice-versa), efeitos de partículas, projéteis disparados pelas torres, vários estados de movimento ou de representação dos inimigos e da base consoante a sua vida, etc.

Os extras serão avaliados de acordo com a sua complexidade, criatividade, impacto na jogabilidade e qualidade da implementação. Alunos são incentivados a documentar claramente as suas ideias e justificações para as funcionalidades implementadas nos extras, detalhando os passos seguidos e as decisões tomadas.

4 Entrega e avaliação

A data limite para conclusão de todas as tarefas é **13 de janeiro de 2025** e a respetiva avaliação terá um peso de **70%** 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 deteção de plágio e eventuais cópias serão consideradas fraude dando-se-lhes tratamento consequente.

Para além dos programas *Haskell* relativos às tarefas, será considerada parte integrante do projeto todo o material de suporte à sua realização armazenado no repositório do respetivo 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 respetivas sessões laboratoriais. A avaliação do projeto terá em linha de conta todo esse material, atribuindo-lhe os pesos indicados na tabela 1.

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%

Tabela 1: Componentes de avaliação do projeto

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 projeto, haverá uma discussão oral[2] 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 respetiva prestação.

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

[2] Prevista para a semana de 13 de janeiro