



Escola de Engenharia  
Universidade do Minho

# Relatório do "Micro Machines"

Grupo 13  
João Sampaio e José Ferreira

31 de Dezembro de 2017



# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>4</b>
1.1	Contextualização . . . . .	4
1.2	Motivação . . . . .	4
1.3	Objetivos . . . . .	5
<b>2</b>	<b>Análise de Requisitos</b>	<b>6</b>
2.1	Fase 1 . . . . .	6
2.1.1	Tarefa 1 . . . . .	6
2.1.2	Tarefa 2 . . . . .	7
2.1.3	Tarefa 3 . . . . .	8
2.2	Fase 2 . . . . .	9
2.2.1	Tarefa 4 . . . . .	9
2.2.2	Tarefa 5 . . . . .	10
2.2.3	Tarefa 6 . . . . .	10
<b>3</b>	<b>A Nossa Solução</b>	<b>11</b>
3.1	Fase 1 . . . . .	11
3.1.1	Tarefa 1 . . . . .	11
3.1.2	Tarefa 2 . . . . .	13
3.1.3	Tarefa 3 . . . . .	15
3.2	Fase 2 . . . . .	16
3.2.1	Tarefa 4 . . . . .	16
3.2.2	Atualizar a quantidade de Nitro . . . . .	20
3.2.3	Tarefa 5 . . . . .	23
3.2.4	Tarefa 6 . . . . .	38
3.3	Documentação . . . . .	42
<b>4</b>	<b>Validação da Solução</b>	<b>43</b>
<b>5</b>	<b>Conclusão</b>	<b>45</b>
<b>6</b>	<b>Bibliografia</b>	<b>47</b>

# **Lista de Figuras**

1.1	Esquema dos Objetivos . . . . .	5
2.1	Peças do Mapa . . . . .	6
2.2	Mapas Válidos (esquerda) e Inválidos (direita) . . . . .	7
2.3	Movimentação do Carro . . . . .	8
2.4	Forças aplicadas no Carro . . . . .	9
3.1	Cenário de Ricochete . . . . .	16
3.2	Menu Inicial . . . . .	30
3.3	Teclados do Jogo . . . . .	34
3.4	Documentação Haddock . . . . .	42
4.1	Mapas visualizados no visualizador de Caminhos e Mapas. . . . .	43

# Listas de Tabelas

3.1	Fases da Resolução da Tarefa 1 . . . . .	11
3.2	Aceleração. . . . .	17
3.3	Travagem. . . . .	18
3.4	Atrito. . . . .	18
3.5	Força dos Pneus. . . . .	18
3.6	Gravidade. . . . .	19
3.7	Fases da Elaboração da <b>desenhaEstado</b> . . . . .	24
3.8	Fases de Elaboração da <i>reageTempo</i> . . . . .	31
3.9	Posições dos carros no inicio do <i>Jogo</i> . . . . .	35

# **Capítulo 1**

## **Introdução**

### **1.1 Contextualização**

Tem data de 1991, o lançamento da primeira versão do lendário jogo de miniaturas de carros, o intitulado "Micro Machines".

Capaz de relembrar a criança que gosta de brincar com pequenas miniaturas, a sua inovação na época de lançamento, fez com que este fosse uns dos primeiros grandes sucessos no mundo do videojogos.

No 1º semestre do curso de MIEI, do ano letivo 2017/2018, em "Laboratórios de Informática I" foi pedido ao aluno que faça uma réplica do jogo em questão. O resultado final será o único elemento de avaliação do aluno na disciplina.

### **1.2 Motivação**

Do cômputo geral, o estudante de qualquer Engenharia gosta mais do trabalho prático, pelo que não foi difícil fazer com que existisse bastante empenho e dedicação para a apresentação de um bom resultado. Para além disso, o trabalho solicitado permite o desenvolvimento das capacidades de qualquer informático.

### 1.3 Objetivos

Trabalhando em pares, a estes compete o cumprimento dos objetivos propostos pelos docentes, em 6 tarefas distintas. O projeto divide-se em 2 fases.

Na 1ºfase, pretende-se construir mapas e validá-los segundo um conjunto e critérios fornecidos, Tarefa 1 e 2 ,respectivamente. Dentro dela ainda se inclui a Tarefa 3, onde é implementada a movimentação do *Carro*.

Na 2ºfase, e ainda envolvendo a mecânica do jogo, com a Tarefa 4 obtém-se a atualização do estado do *Carro*. Com a ajuda do Gloss, na Tarefa 5 ocorre a criação executável do "Micro Machines". Por fim, na Tarefa 6 é programada uma estratégia de jogo para os *bots*.

Com este documento, espera-se realizar uma explicação sucinta de tudo aquilo que está por trás do executável do jogo acima referido. Pretende-se expor de forma clara quais foram as linhas orientadoras para a redação do seu código e quais as ferramentas utilizadas para a sua execução.

Mais a mais, será exposto quais as principais dificuldades, bem como quais os métodos encontrados para a sua resolução. O objetivo é claro : um leitor muito mais esclarecido sobre a temática abordada.

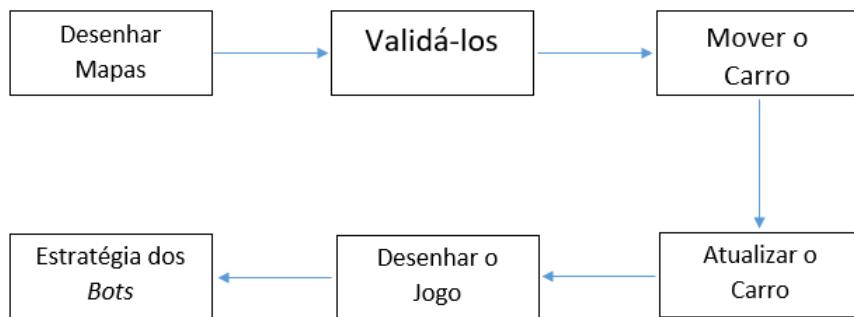


Figura 1.1: Esquema dos Objetivos

# Capítulo 2

## Análise de Requisitos

### 2.1 Fase 1

#### 2.1.1 Tarefa 1

Os mapas são a base de qualquer jogo de corrida. Dessa forma, faz sentido que a tarefa inicial seja a redação de um código que permita criar qualquer mapa a partir de um caminho dado.

O maior desafio desta Tarefa será fazer com que as diferentes peças (abaixo expostas) se situem no local correto.

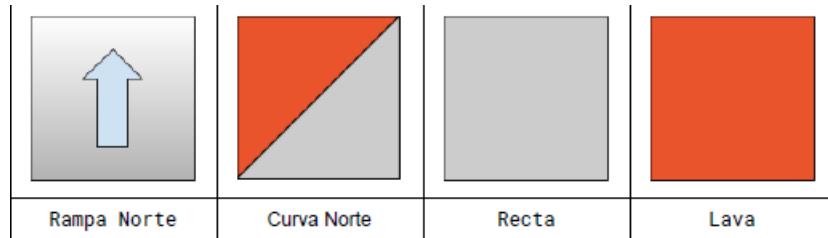


Figura 2.1: Peças do Mapa

Pretende-se definir a função **constroi**, que através de um *Caminho*, constroi o *Mapa* correspondente.

```
constroi :: Caminho -> Mapa
```

### 2.1.2 Tarefa 2

Para assegurar que isso acontece, na Tarefa 2 pretende-se a criação de um algoritmo capaz de averiguar se o mapa é válido.

Todas as hipotéticas situações devem ser tidas em conta. A condição mais relevante, será garantir que o ponto de partida da formação do mapa é também o seu ponto final (condição não assegurada nos mapas da direita da figura 2.2).

Pretende-se a elaboração da função **valida**, que recebe um *Mapa* e devolve um *Bool*.

```
valida :: Mapa -> Bool
```

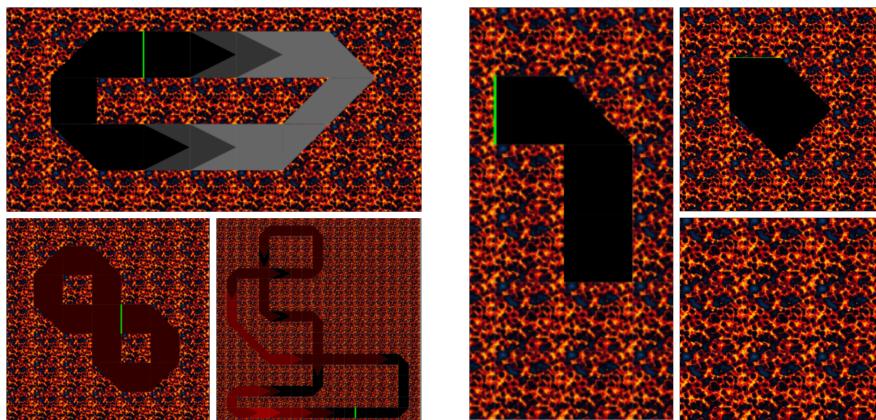


Figura 2.2: Mapas Válidos (esquerda) e Inválidos (direita)

### 2.1.3 Tarefa 3

A derradeira etapa desta 1ºfase é a movimentação de um carro. Perante a ação de um jogador, o *Carro* deve mover-se para a posição esperada. É necessário garantir que ocorre sua destruição quando estiver em contacto com a lava. Para além disso, deverá existir ricochete quando a diferença de alturas entre peças é superior a 1.

O objetivo desta Tarefa é definir a função **movimenta**, que dado um *Tabuleiro*, um *Tempo* e um *Carro*, devolve um *Carro* correspondente.

```
movimenta :: Tabuleiro -> Tempo -> Carro -> Maybe Carro
movimenta tab t c = isDestroyed tab (move t c)
```

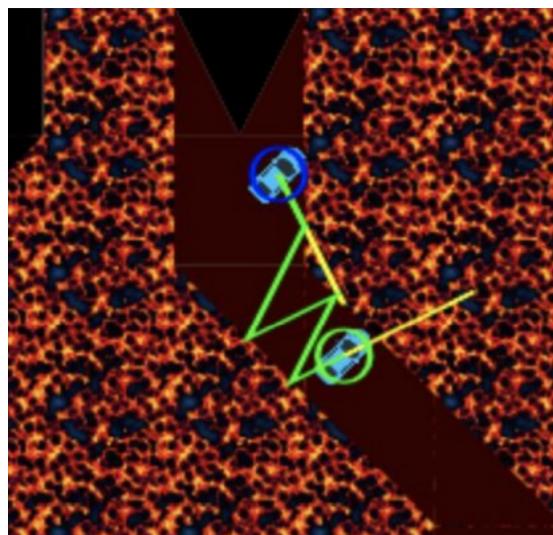


Figura 2.3: Movimentação do Carro

## 2.2 Fase 2

### 2.2.1 Tarefa 4

Para dar sucesão ao trabalho já realizado, é necessário implementar as consequências da movimentação do *Carro*. Na Tarefa 4, pretende-se a criação da função **atualiza** que a partir de um jogo inicial, de um tempo, de uma ação e de um inteiro, identificativo do jogador em questão, devolve um novo jogo.

É importante que o aluno se aperceba da influência que as forças exercidas sobre o carro irão ter na alteração da sua velocidade. Pormenores como a direção e o sentido em que as forças são aplicadas no carro devem ser tidos em conta. Atenção também à presença da força do peso nas rampas.

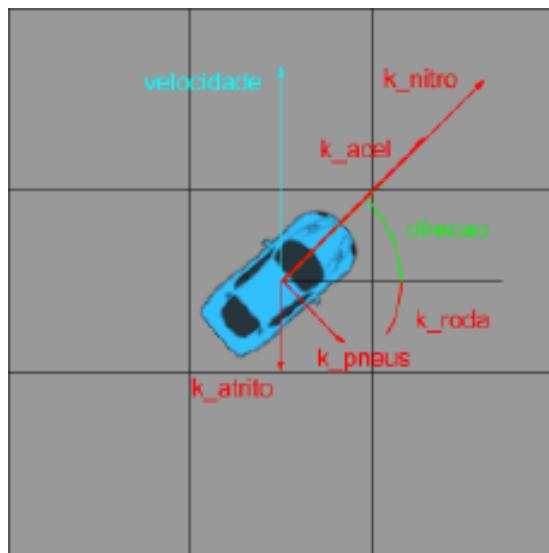


Figura 2.4: Forças aplicadas no Carro

O histórico (lista de listas de todas as posições em que os diferentes carros passaram) e a lista com os tempos de nitro de cada *Carro* devem ser devidamente atualizadas.

O cerne desta Tarefa é a definição da função **atualiza**. Quando recebe um *Tempo*, um *Jogo*, um *Int* e uma *Acao*, a função devolve o *Jogo* correspondente.

```
atualiza :: Tempo -> Jogo -> Int -> Acao -> Jogo
```

### 2.2.2 Tarefa 5

Na tarefa 5, a biblioteca *Gloss* adiciona a cada grupo, uma panóplia de funções que possibilitam a edição gráfica do jogo. O executável deve ser apelativo e cada grupo tem a liberdade de implementar funcionalidades originais na sua animação. Esperam-se jogos autênticos.

Nesta tarefa, a função **movimenta** e **atualiza** serão fundamentais para o bom funcionamento dos carros.

Pretende-se a definição da função **main** que devolve um *IO ()*

```
main :: IO ()
```

### 2.2.3 Tarefa 6

Por fim, na Tarefa 6 é esperado que cada grupo implemente uma estratégia de jogo para a movimentação dos bots.

Para tal é necessário ter em conta que :

- São permitidos atalhos , tendo em conta que estes não permitem a ultrapassagem de mais de 4 peças.
- A *Velocidade* dos *bots* deve ser inferior nas curvas relativamente a todas as outras peças.

Para que isso aconteça é necessária a definição da função **bot**, que dado um *Tempo*, um *Jogo* e um *Int*, devolve a *Acao* resultante.

```
bot :: Tempo -> Jogo -> Int -> Acao
```

# Capítulo 3

## A Nossa Solução

### 3.1 Fase 1

#### 3.1.1 Tarefa 1

Fases	Descrição	Função
1º	Construção de um Tabuleiro de Lava	<b>buildLava</b>
2º	Posição das Peças	<b>passosToPosition</b>
3º	Escolhas das Peças	<b>passosToPecas</b>
4º	Alteração Local das Peças	<b>editTabuleiro</b>
5º	Construção do <i>Tabuleiro</i>	<b>makeTabuleiro</b>
6º	Construção do <i>Mapa</i>	<b>constroi</b>

Tabela 3.1: Fases da Resolução da Tarefa 1.

A chave para a realização de um *Mapa* concordante com o *Caminho* fornecido foi a criação de um algoritmo capaz de colocar peças na posição adequada do *Tabuleiro*.

Na maior parte dos casos, a *Peca* que se repete mais vezes no *Tabuleiro* é a *Peca Lava 0*, pelo que foi intenção do grupo, criar uma função que a colocasse em todas as posições do *Tabuleiro*. Para tal, necessitou-se de considerar as dimensão do *Tabuleiro* correspondente ao *Caminho* dado.

Dessa forma, a primeira função a ser criada foi a função **buildLava**.

```
buildLava :: Caminho -> Tabuleiro
buildLava c = let x = fst (dimensao c)
              y = snd (dimensao c)
            in replicate y (replicate x (Peca Lava 0))
```

A inserção de *CurvaDir* e *CurvaEsq* no Caminho, faz com que se altere a *Orientacao* que é tida em conta na construção das peças.

Devido a isso, houve a necessidade de elaborar uma função que colocasse as peças no local adequado, intitulada de **passosToPosition**. Devolve uma lista de *Peca* a partir de um Caminho. Tem em conta *Orientacao* inicial Este.

```
passosToPosition:: Caminho -> [Posicao]
passosToPosition [] = []
passosToPosition c = positionEste (partida c) c
```

Esta função está dependente de outras 4 funções (**positionEste**, **positionOeste**, **positionNorte**, **positionSul**). Cada uma delas devolve a posição das peças segundo determinada *Orientacao*. Recursivamente, devolvem as posições pretendidas.

Sob a influência do *Caminho* está também a criação das peças que formam o Mapa. A *Orientacao* volta a ser de novo fundamental, uma vez que o mesmo *Passo* pode levar à execução de peças diferentes.

Para que a junção das peças seja a esperada comparativamente ao respectivo *Caminho*, é necessário seguir um raciocínio similar ao da colocação das peças na *Posicao* certa. Nas **pecasEste**, **pecasOeste**, **pecasNorte** e **pecasSul**) recursivamente irá ocorrer a formação de uma lista peças tendo em conta as diferentes orientações. A relação entre todas estas 4 funções irá ocorrer na função **passosToPecas**, que considera a *Orientacao* inicial Este.

```
passosToPecas :: Caminho -> [Peca]
passosToPecas [] = []
passosToPecas c = pecasEste 0 c
```

Para relacionar todas as funções descritas anteriormente, necessitou-se de ter uma função que faz a substituição local de peças no *Tabuleiro* de lava, apelidada de **editTabuleiro**. Para executar o primeiro passo recursivo, a função **insertPeca** faz a alteração da *Peca* numa *Posicao* específica do *Tabuleiro*.

Por fim, na função **makeTabuleiro** ocorre a construção do *Tabuleiro*. Com isto chegou-se à função objetivo (**controi**), que dá um *Mapa* em função de um *Caminho*.

```
makeTabuleiro :: Caminho -> Tabuleiro
makeTabuleiro c = editTabuleiro (buildLava c)
  (passosToPecas c)(passosToPosition c)

constroi :: Caminho -> Mapa
constroi c = Mapa (partida c, Este) (makeTabuleiro c)
```

### 3.1.2 Tarefa 2

Para a resolução desta Tarefa, inicialmente houve a necessidade de certificar que o *Mapa* em análise cumpria certos requisitos relacionados com a estrutura do seu *Tabuleiro*.

Um *Tabuleiro* é válido se o seu número de colunas for o mesmo em cada uma das linhas que o constitui. Para averiguar esta condição, o grupo criou a função **isValidTabuleiro**. Para além disso, esta função diz também que um *Tabuleiro* vazio impossibilita a existência de um *Mapa* válido.

A *Peca* de partida não pode ser *Peca Lava 0* e deve ter pelo menos uma *Peca* adjacente que também não o seja.

Para completar as restrições relacionadas com a estrutura do *Tabuleiro* a função **isBorderLava** certifica que as suas bordas são constituídas por *Peca Lava 0*.

Todo o *Mapa* é válido se for transitável, isto é, se for possível ir desde a *Peca* de partida até à *Peca* final sem encontrar nenhuma *Peca Lava 0*.

De maneira a descobrir se isto se verifica em qualquer *Mapa*, o grupo criou a função **isValidPercorreMapa**. Esta função verifica primeiramente se a *Peca* com a *Posicao* imediatamente antes da partida é uma *Peca* não lava. Se esta condição se verificar então calcula o registo da passagem do carro com recurso à função **percorreMapa**, elimina as posições repetidas e substitui no mapa original todas as peças que se encontram na lista por lava. Se o resultado for um tabuleiro só com lava e a última *Posicao* da lista for a da partida então a função devolve o valor lógico de verdade.

Para obter o registo por onde o Carro passa, foi desenvolvida a função **percorreMapa** que simula um *Carro* a percorrer o *Mapa*. Esta função é chamada pela função **isValidPercorreMapa** e recebe o Mapa, um tuplo que contém a *Posicao* imediatamente anterior à partida e a *Orientacao* de partida. Ao longo dos passos recursivos este tuplo é utilizado para manter a *Posicao* e *Orientacao* em que o *Carro* se encontra e dois inteiros.

O primeiro inteiro tem o valor do dobro do número de peças não lava do *Mapa*. Por cada passo de recursividade o valor diminui. Se este chegar a 0 é considerado como um caso de paragem e a função devolve no fim da Lista um Nothing. Tal impede que a função entre em ciclos infinitos sem conseguir voltar à partida com a *Orientacao* inicial.

O segundo inteiro indica se o *Carro* já passou uma vez pela partida ou não (Se ainda não passou o valor deste é 1, se já passou é 0). O valor que é passado pela função **isValidPercorreMapa** é sempre 1 pois, como o *Carro* começa sempre nas coordenadas imediatamente antes da partida e a primeira passagem pela partida é ignorada.

Para que o *Carro* avance para a próxima *Peca* essa transição tem de ser válida. Para verificar se é possível é utilizada a função **isValidNext**.

Se esta transição for válida, então a *Posicao* do *Carro* é adicionada à lista, e a função **percorreMapa** é chamada recursivamente, mas desta vez a *Posicao* é a da próxima *Peca* e a *Orientacao* é a que é dada pela função **dirChange**.

```

percorreMapa :: Mapa -> (Posicao, Orientacao) -> Int -> Int -> [Maybe Posicao]
percorreMapa m (p,o) 0 _ = [Nothing]
percorreMapa m@(Mapa (pi,oi) t) (p,o) v n
| isValidNext m p o && coorNextPeca == pi && o == oi && n == 1 = Just p : percorreMapa m (coorNextPeca, dirChange m p o)
(v - 1) (n - 1)
| isValidNext m p o && coorNextPeca == pi && o == oi && n == 0 = [Just p, Just coorNextPeca]
| isValidNext m p o = Just p : percorreMapa m (coorNextPeca, dirChange m p o) (v - 1) n
| otherwise = [Nothing]
where
    coorNextPeca = nudge p o

```

A **isValidNext** devolve um Booleano que indica se o movimento é possível atendendo à *Posicao* e à *Orientacao* que lhe é dada.

```

isValidNext :: Mapa -> Posicao -> Orientacao -> Bool
isValidNext m p o | tipoPeca == Recta      = isValidReta m p o
                  | tipoPeca == Curva Norte = isValidCurva m p o
                  | tipoPeca == Rampa Norte = isValidRampa m p o
                  | tipoPeca == Lava       = False
where
    tipoPeca = tipoFinder (pecaFinder m (nudge p o))

```

Dessa forma, temos tudo que é necessário para criar a função **valida**.

```

valida :: Mapa -> Bool
valida m = isValidTabuleiro m
          && isBorderLava m
          && isValidPartidaPeca m
          && isValidPercorreMapa m

```

### 3.1.3 Tarefa 3

Não pensando em possíveis colisões nem em destruições, o primeiro objetivo na Tarefa 3 foi fazer com que o carro se movesse. Em virtude disso mesmo, criou-se a função **move** que altera a *Posicao* de um *Carro* a partir do *Tempo* dado. A soma das coordenadas da *Posicao* do *Carro* com o produto da coordenadas da *Velocidade* pelo *Tempo* foram a chave para obter uma *Posicao* acertada.

As destruições ocorrem quando o *Carro* embate numa *Peca Lava 0*. Dessa forma, o grupo apercebeu-se que em peças do *Tipo Curva Orientacao* esse fenômeno poderia ocorrer.

Por conseguinte, a função **isDestroyedCurva** certifica que o Carro é destruído assim que embate na diagonal de peças do *Tipo Curva Orientacao*. Foi necessário calcular a distância entre o *Ponto* onde se situa o *Carro* e a borda da *Peca* onde este se encontra, através da função **coorInPeca**. Dependendo da *Orientacao* da curva, se o *Ponto* onde se situa o *Carro* estiver dentro da área englobada pela lava na *Peca*, então a função **isDestroyedCurva** devolve *Nothing*, acabando o *Carro* por ser destruído.

Foi necessário relacionar a função **isDestroyedCurva** com uma função genérica **isDestroyed**, que nos diga quando é que o *Carro* vai ser destruído. Para nos auxiliar tivemos a função **isCurva** que identifica peças onde existem curvas. Nesses casos tivemos também em conta se a *Altura* da *Peca* era superior a 0. Se não o fosse, iria ocorrer ricochete.

```
isDestroyed:: Tabuleiro -> Carro -> Maybe Carro
isDestroyed t c | peca == Peca Lava 0 = Nothing
                 | isCurva peca && altPeca >= 0 = isDestroyedCurva t c
                 | otherwise = Just c
where
    peca = pecaTabFinder t c
    altPeca = alturaPeca peca
```

Voltando ao raciocínio inicial, apenas faltava completar o algoritmo de forma a que as colisões ocorressem. Primeiramente criou-se a função **isRicochete** que identifica as situações onde ocorrem colisões, ou seja quando a diferença de alturas no *Mapa* é superior a 0. Como a *Direcao* do *Carro* não se altera ao longo do jogo, faltava saber qual era a *Posicao* para onde o Carro iria após o ricochete e qual a sua *Velocidade*.

Sabendo qual o ponto de colisão, a descoberta do *Ponto* para onde irá o *Carro* faz-se multiplicando o *Tempo* por cada uma das coordenadas desse ponto.

Por falta de tempo, o grupo não conseguiu desenvolver uma função que calculasse qual a *Velocidade* com que o *Carro* chegaria ao respetivo *Ponto* após a ocorrência de uma colisão. O grupo perpectivou que poderia calcular o vetor resultante da colisão, tendo que para isso considerar um vetor unitário no ponto onde ocorre o ricochete.

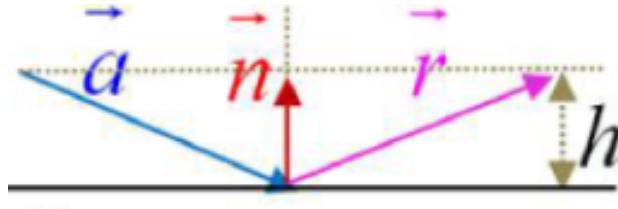


Figura 3.1: Cenário de Ricochete

## 3.2 Fase 2

### 3.2.1 Tarefa 4

Para a atualização dos estado do jogo, à que ter em conta três novos *datatypes*: o *Jogo*, as *Propriedades* e a *Acao*.

```

data Jogo = Jogo
{ mapa      :: Mapa      -- ^ o mapa do percurso
, pista     :: Propriedades -- ^ as propriedades do percurso
, carros    :: [Carro]   -- ^ o estado do carro de cada jogador
, nitros    :: [Tempo]   -- ^ a quantidade de nitro disponível para cada jogador
, historico :: [[Posicao]] -- ^ o histórico de posições de cada jogador
}

data Propriedades = Propriedades
{ k_atrito   :: Double -- ^ o atrito do piso
, k_pneus    :: Double -- ^ o atrito dos pneus
, k_acel     :: Double -- ^ a intensidade da aceleração
, k_peso     :: Double -- ^ o efeito da gravidade nas rampas
, k_nitro    :: Double -- ^ a intensidade do nitro
, k_roda     :: Double -- ^ a sensibilidade do guiador
}

data Acao = Acao
{ acelerar :: Bool      -- ^ Ativado quando o Carro acelera
, travar   :: Bool      -- ^ Ativado quando o Carro está a travar
, esquerda :: Bool     -- ^ Ativado quando o Carro curva à esquerda
, direita  :: Bool     -- ^ Ativado quando o Carro curva à direita
, nitro    :: Maybe Int -- ^ Ativado quando algum jogador dá Nitro
}

```

As variações das ações dos jogadores alteram o *Jogo*, nomeadamente os *carros*, os *nitros* e o *historico*. Devido a isso, o grande objetivo da Tarefa 4 é que a atualização desses parâmetros ocorra de maneira acertada.

### Atualização da Velocidade

No que diz respeito às alterações que ocorrem nos *carros*, o grupo devia criar funções capazes de atualizar a *Velocidade* de cada um deles, bem como a sua *Direcao*.

Considerem - se que os parâmetros do datatype *Propriedades* podem representar forças aplicadas nos *carros* ao longo do jogo. Cada um deles assume valores constantes.

O vetor velocidade do *Carro* calcula-se a partir do somatório de todas as forças envolvidas, à exceção do *k\_roda* e do *k\_nitro*. O *k\_nitro* apenas tem influência no valor do vetor velocidade se o nitro do jogador em questão for ativado. O *k\_peso* também só é considerado em rampas.

O grupo decidiu trabalhar com coordenadas cartesianas, ao invés de utilizar coordenadas polares. Considerando as coordenadas polares como um tuplo, o elemento da esquerda representa a direção da força, ao passo que o elemento da direita é a sua norma. A função **vetAngToCoor** é responsável pela conversão.

```
vetAngToCoor:: VetorAng -> Vetor
vetAngToCoor (a,n) = (x, y)
where
    x = n * cos (grauToRad a)
    y = - n * sin (grauToRad a)
```

### Forças Envolvidas

#### Aceleração

Força	Sentido	Direção
Aceleração	Igual ao do Carro	Igual à do Carro

Tabela 3.2: Aceleração.

A direção e o sentido da aceleração são iguais às do *Carro*. A norma é obtida multiplicando o *Tempo* pela constante correspondente das *Propriedades*, ou seja, o *k\_acel*. Quando não existe aceleração, o valor devolvido pela função **acel** é (0,0).

```
acel::Bool -> Tempo -> Carro -> Propriedades -> Vetor
acel b t car prop | b           = vetAngToCoor (direcao car, t * k_acel prop)
                   | otherwise = (0,0)
```

Travagem

<i>Força</i>	<i>Sentido</i>	<i>Direção</i>
Travagem	Oposto ao do Carro	Igual à do Carro

Tabela 3.3: Travagem.

Semelhante à Aceleração, com a particularidade de o seu sentido ser o oposto ao do *Carro*. A norma obtém-se multiplicando o *Tempo* pelo *k\_acel*. Quando não existe aceleração, o valor devolvido pela função **trav** é (0,0).

```
trav::Bool -> Tempo -> Carro -> Propriedades -> Vetor
trav b t car prop | b           = vetAngToCoor ((direcao car) + 180, t * k_acel prop)
                   | otherwise = (0,0)
```

Atrito

<i>Força</i>	<i>Sentido</i>	<i>Direção</i>
Atrito	Oposto ao da Velocidade	Igual à da Velocidade

Tabela 3.4: Atrito.

O ângulo da direção é o simétrico do ângulo da *Velocidade*. A sua norma obtém-se multiplicando o *Tempo* pelo *k\_atrito* e pela norma da velocidade.

Força dos Pneus

<i>Força</i>	<i>Sentido</i>	<i>Direção</i>
Pneus	Oposto ao da Velocidade	Perpendicular à do Carro

Tabela 3.5: Força dos Pneus.

A direção é perpendicular à do *Carro*, logo o ângulo da direção do Pneu pode ser superior ou inferior relativamente ao ângulo da direção do *Carro*. Considere-se as duas situações :

- Quando o ângulo da direção do vetor *Velocidade* é superior ou igual ao ângulo da direção do *Carro*, subtraí-se 90 graus a este último para obter o ângulo da direção da força do Pneu.
- Caso contrário, soma-se 90 graus ao ângulo da direção do *Carro*.

A sua norma obtém-se multiplicando o *Tempo* pelo *k\_pneus*.

Gravidade

Força	Sentido	Direção
Gravidade	Igual ao do declive da Rampa	Igual ao do declive da Rampa

Tabela 3.6: Gravidade.

Para descobrir o ângulo corresponde ao declive da Rampa em questão, o grupo utilizou a função **orToAng**. Consoante a *Orientacao* da Rampa, a função devolve um *Double* correspondente ao valor do ângulo.

```
orToAng:: Orientacao -> Double
orToAng Norte = 90
orToAng Sul   = 270
orToAng Este   = 0
orToAng Oeste  = 180
```

O valor da norma da gravidade correponde à multiplicacão do *Tempo* pelo *k\_peso*.

Conhecidas todas as particularidades das forças envolvidas para a atuação do vetor velocidade, o grupo decidiu organizar as coordenadas cartesianas de cada uma delas numa lista. Na função **vetorRes** é calculado o somatório de todas as coordenadas presentes dentro da lista anteriormente referida. Tal não seria possível sem a função **sumVet**, que soma os vetores existentes dentro de uma lista.

```
vetoresRes::Tempo -> Jogo -> Int -> Acao -> [Vetor]
vetoresRes t j i a = [(vx,vy),
                      acel (acelerar a) t car prop,
                      trav (travar a) t car prop,
                      peso isPeso t car prop anVetPeso,
                      pneu t car prop,
                      atrito]

where
      m          = mapa j
      prop       = pista j
      car        = (carros j) !! i
      (vx,vy)    = velocidade car
      (va, vn)   = vetCoorToAng (velocidade car)
      pecaLocal = pecaFinder m (coorPeca car)
      isPeso     = tipoFinder pecaLocal == Rampa Norte (*)
      atrito    = vetAngToCoor (va - 180, t * vn * (k_atrito prop))
      anVetPeso = orToAng (orFinder pecaLocal Norte) - 180
```

(\*) Rampa Norte é o resultado que a função **tipoFinder** devolve para peças do tipo Rampa.

```
vetorRes:: Tempo -> Jogo -> Int -> Acao -> Vetor
vetorRes t j i a = sumVet (vetoresRes t j i a)
```

### 3.2.2 Atualizar a quantidade de Nitro

A direção do vetor do nitro é a mesma que a direção do vetor do *Carro*. A sua norma é dada pela multiplicação do *Tempo* pelo *k\_nitro*.

```
nit::Bool -> Tempo -> Carro -> Propriedades -> Vetor
nit b t car prop | b           = (vetAngToCoor (direcao car, t * k_nitro prop))
| otherwise = (0,0)
```

A ativação do nitro leva a que o vetor velocidade seja alterado. A este será adicionado o vetor do nitro. A alteração da velocidade em cada um dos *carros* é feita de forma análoga à substituição de peças no Tabuleiro, na Tarefa 1. Primeiramente retira-se a parte da lista de *carros* que não vai ser alterada até à posição da lista onde irá ocorrer a substituição. De seguida, ocorre a mudança do vetor velocidade no *Carro* em que o nitro é ativado. Por fim devolve-se a lista de carros restante, que não irá sofrer nenhuma alteração. A função **atList** é a responsável pela realização deste processo na função **applyNit** que funciona quando o nitro é ativado.

```
applyNit::Tempo -> [Carro] -> [Tempo] -> Propriedades -> Int -> Acao -> [Carro]
applyNit t cars tNit prop i a | nitID == Nothing = cars
                                | otherwise = atList cars (car {velocidade = (vx + nx, vy + ny)}) (fromJust nitID)
where
    nitID      = nitro a
    car        = cars !! (fromJust nitID)
    (vx, vy)   = velocidade car
    (nx, ny)   = nit ((tNit !! i) > 0) t car prop
```

A função **tNitDiscount** ficou encarregue de descontar o tempo de nitro de cada jogador. Calcula a diferença entre o tempo dado e o valor de tempo de nitro final do jogador em questão. Com a ajuda de uma função auxiliar, sempre que a diferença for inferior a 0, a função **tNitDiscount** impossibilitava a ocorrência de mais nitro.

```
tNitDiscount:: Tempo -> Jogo -> Int -> Acao -> [Tempo]
tNitDiscount t j i a | isNit     = atList listTNits tNitFinal i
                     | otherwise = listTNits
where
    isNit      = (nitro a) /= Nothing
    tNitCar   = (listTNits) !! i
    listTNits = nitros j
    tNitFinal = aux (tNitCar - t)
    where
        aux:: Double -> Double
        aux t | t < 0      = 0
               | otherwise = t
```

### Atualizar Direção

Através da mudança do valor booleano do parâmetro esquerda ou direita da *Acao*, altera-se também a direção do *Carro*. Dessa forma, ao ângulo da direção irá ser somado um novo valor, que se obtém multiplicando o *Tempo* pelo *k\_roda*.

```
dirCarChange::Tempo -> Jogo -> Int -> Acao -> Double
dirCarChange t j i a | esq && dir = dirCar
| esq      = dirCar + deltaD
| dir      = dirCar - deltaD
| otherwise = dirCar
where
  dirCar = direcao ((carros j) !! i)
  esq    = esquerda a
  dir    = direita a
  prop   = pista j
  deltaD = t * k_roda prop
```

Tendo as funções necessárias para a atualização da *Direcao* e da *Velocidade*, tem-se tudo que é preciso para atualizar o *Carro*. A velocidade pode-se alterar em função da ativação do nitro. A função **atCarros** devolve o *Carro* atualizado, mesmo que este esteja sobre a influência do nitro.

```
atCarros::Tempo -> Jogo -> Int -> Acao -> Jogo
atCarros t j i a = j {carros = applyNit t carNotNit tNit prop i a}
where
  m      = mapa j
  prop   = pista j
  car    = (carros j) !! i
  v      = velocidade car
  dCha   = dirCarChange t j i a
  vRes   = vetorRes t j i a
  tNit   = nitros j
  carNotNit = atList (carros j) (car {direcao = dCha, velocidade = vRes}) i
```

**Atualizar o Histórico**

O histórico é uma lista das posições por onde o que *Carro* já passou e é atualizado em função da alteração da *Acao*. A nova *Posicao* é sempre colocada na cabeça da lista.

```
atHist:: Carro -> [Posicao] -> [Posicao]
atHist c log | head (log) == coorCarro = log
               | otherwise           = [coorCarro] ++ log
               where
                  coorCarro = coorPeca c
```

Todas as estas funções irão se relacionar na função **atualiza**.

```
atualiza t e j a = atCarros t atHistNit j a
                   where
                      carroHistNew = atHist carroTarget carroHistOld
                      carroTarget  = (carros e) !! j
                      carroHistOld = jogoHistOld !! j
                      atHistNit    = e {nitros = tNitNew, historico = jogoHistNew}
                      tNitNew      = tNitDiscount t e j a
                      jogoHistNew  = atList jogoHistOld carroHistNew j
                      jogoHistOld  = historico e
```

### 3.2.3 Tarefa 5

Após a análise da documentação referente à biblioteca *Gloss*, verificámos que para a realização desta Tarefa seria necessário utilizar a função **play**, pré-definida no package ”*Graphics.Gloss.Interface.Pure.Game*”.

```
play :: Display
-> Color
Cor do Fundo.

-> Int
Frame Rate.

-> world
O "mundo" inicial.

-> (world -> Picture)
Uma função para converter o world para uma Picture.

-> (Event -> world -> world)
Uma função que reage a eventos.

-> (Float -> world -> world)
Uma função que devolve um determinado world após a passagem do Tempo que lhe é dado.

-> IO ()
O resultado final.
```

Dessa forma, o grupo criou a função **joga**, onde se relacionam todas as funções que permitem a criação do executável do jogo. Esta função está dependente da função **play**, acima descrita.

Para a utilização da função **play** tivemos de criar um *datatype* que funcionasse como *world*. Apelidámo-lo de *Estado* e está definido sobre a forma de *record*.

```
data Estado = Estado
    { jogo      :: Jogo          -- ^ Principais parâmetros do jogo
    , actions   :: Actions       -- ^ Ações dos jogadores
    , images    :: Images         -- ^ Imagens do Jogo
    , nPlayer   :: Int           -- ^ Indica o número de jogadores não bot presentes
    , camera    :: Camera         -- ^ Tipo de câmera onde decorre a corrida
    , winSize   :: (Int, Int)     -- ^ Contém o tamanho da janela em que o jogo está a correr
    , pausa     :: Bool          -- ^ Indica se está em pausa ou não
    , inCountDown :: Bool        -- ^ Indica se está em contagem decrescente para começar ou não
    , tToStart   :: Float         -- ^ Indica quanto tempo falta para começar
    , tFromStart :: Float        -- ^ Indica quanto tempo é que passou desde que o jogo começou
    , inMenu    :: Bool          -- ^ Indica se está dentro de um dos Menus do jogo
    , whereMenu :: MenuPos       -- ^ Indica em qual dos Menus se encontra
    , pMapa     :: Caminho        -- ^ O caminho para a formação de mapas pelo jogador
    , nVoltas   :: Int           -- ^ Indica o número de Voltas a dar pelo utilizador
    , idPath    :: [Posicao]      -- ^ Indica o caminho ideal de uma volta à pista (utilizado para contar voltas)
    }
```

Nas páginas seguintes falaremos das etapas da elaboração desta fase do projeto, sendo especificado a utilidade dos diferentes parâmetros do *Estado*.

O jogo subdivide-se em dois diferentes momentos: o momento da corrida, que foi criado através da função **desenhaJogo**, e a parte em que o utilizador do executável se encontra num dos menus dos jogo. A função que ficou encarregue por desenhar o menus foi a **desenhaMenus**

Estas duas funções irão se relacionar na função **desenhaEstado** que irá construir tudo aquilo que aparece no ecrã.

```
desenhaEstado :: Estado -> Picture
desenhaEstado e | inMenu e = desenhaMenu e
                 | otherwise = desenhaJogo e
```

As diferentes fases que levaram o grupo a chegar à função pretendida estão sintetizadas no quadro seguinte.

<i>Etapa</i>	<i>Objetivo</i>
1º	Construção de mapas
2º	Colocação dos carros nos mapas
3ª	Extras no Modo Corrida
4ª	Desenho dos Menus
5º	Extras nos Menus

Tabela 3.7: Fases da Elaboração da **desenhaEstado**.

### Elaboração da *desenhaJogo*

#### Construção dos Mapas

O primeiro passo na resolução desta tarefa foi criar uma função que permitisse gerar qualquer mapa no ecrã a partir de um *Tabuleiro*. Para tal o grupo desenhou as diferentes peças que fazem parte do *Tabuleiro*.

Cada peça é um quadrado. Em virtude do *Tabuleiro* dado, o lado deste quadrado irá se alterar, pelo que não existia um valor fixo que o grupo lhe poderia atribuir.

A função **sizePeca** dá o valor do lado de cada peça. Dada uma janela do ecrã, e um *Tabuleiro* a função descobre qual o valor máximo que o lado pode tomar, atendendo ao números de peças que irão existir horizontalmente (comprimento da cabeça do *Tabuleiro*) e verticalmente (comprimento do *Tabuleiro*)

Depois disso, seleciona o menor dos dois valores encontrados, garantindo assim que todas as peças cabem no ecrã.

```
sizePeca:: Tabuleiro -> (Int,Int) -> Int
sizePeca a@(l:t) (x,y)= minimum [maxX, maxY]
  where
    maxX = floor ((fromIntegral y)/ fromIntegral (length a))
    maxY = floor ((fromIntegral x)/ fromIntegral (length l))
```

Para o desenho das peças à que considerar 4 grupos de peças : retas, curvas, rampas e lava.

As retas fazem-se através da função **polygon**, que desenha um polígono em virtude das coordenadas dadas. Importante referir, que nas coordenadas dadas, a cabeça e o último elemento devem ser iguais.

```
reta :: Int -> Picture
reta l = Polygon [(-r,-r),(r,-r),(r,r),(-r,r),(-r,-r)]
    where
        r = fromIntegral l / 2
```

As curvas são feitas segundo o mesmo mecanismo. As coordenadas que constituem a lista dada à função **polygnon** é que é são diferentes.

```
curvaNorte :: Int -> Picture
curvaNorte l = Polygon [(-r,-r),(r,-r),(r,r),(-r,-r)]
    where
        r = fromIntegral l / 2
```

As rampas necessitam da função **pictures**, que forma uma imagem através da junção de várias figuras. Na lista de figuras dadas, foi necessário criar um identificativo das rampas, que neste caso foi a imagem que forma 2 triângulos paralelos. Houve a necessidade de aplicar a função **translate** que move um figura segundo as coordenadas dadas.

```
rampaNorte :: Int -> Picture
rampaNorte l = Pictures[reta 1, arrow1, arrow2]
    where
        r = fromIntegral 1/2
        arrow1 = color colorarrow (Polygon [(0,0),(r, -r/2), (r,-r), (0,-r/2), (-r,-r),(-r,-r/2), (0,0)])
        arrow2 = Translate 0 r arrow1
        colorarrow = greyN 0.5
```

A lava é uma figura em branco. Utiliza-se a função **blank**, pré-definida no *Gloss*.

```
lava :: Picture
lava = blank
```

O grupo optou por fazer a construção do mapa no ecrã de forma recursiva. Primeiro era necessário fazer a construção de uma linha de peças, para ocorrer a repetição deste processo nas restantes linhas.

A primeira linha é feita com a função **drawline**. Esta necessita da função **placePeca** para colocar as peças no local certo.

```
drawLine :: Int -> [Peca] -> Picture
drawLine s l = Pictures (placePeca s translationX (map (desenhaPeca s) 1))
    where
        translationX = -(fromIntegral(length l)/2)* (fromIntegral s)
```

Para construir o restante mapa, o raciocínio é análogo à construção de uma linha de peças. Neste caso teremos a função **drawlines**, que com o auxilio da **placelines**, coloca as linhas de peças no sitio certo do ecrã.

```
drawLines :: Int -> Tabuleiro -> Picture
drawLines s t = Pictures (placeLines s translationY (map (drawLine s) t))
    where
        translationY = (fromIntegral(length t)/2)* (fromIntegral s)
```

Construída em último lugar mas não menos importante, vem a colocação da linha de partida. A função **startLine** recebe o lado de cada peça, conseguindo desta forma fazer o desenho da partida.

Devido à sua extensão não a colocaremos neste relatório. De destacar que utilizámos uma função do *Gloss* ainda não mencionada, a função **color**. Ela aplica uma cor uma *picture*. Neste caso, foi necessária para que polígonos adjacentes ficassem com cores diferentes, de maneira a formar uma imagem semelhante a uma linha de partida.

Depois de tudo isto, basta fazer a função final que irá, efetivamente, desenhar o mapa no ecrã.

```
drawMapa:: Estado -> Picture
drawMapa e = Pictures [drawLines pSize tab, start]
    where
        (Mapa (p,o) tab) = (mapa (jogo e))
        (px,py) = p
        (sx,sy) = (fromIntegral px + 0.5, fromIntegral py + 0.5)
        pSize = sizePeca tab (winSize e)
        (x,y) = cornerToCenter pSize tab (sx, sy)
        start = Translate (realToFrac x) (realToFrac y) (startLine pSize)
```

Colocação dos carros nos mapas

As coordenadas da *Posicao* dos carros são dadas a partir de um referencial que tem a origem no canto superior esquerdo do ecrã.

O *Gloss* tem em conta que todas as coordenadas dados tem como ponto (0,0) o centro do ecrã.

Dessa forma, o grupo criou a função **cornerToCenter**.

Esta função funciona com base em vetores.

Primeiramente, multiplica-se as coordenadas dadas pelo tamanho da *Peca* em pixéis.

Em seguida, usando como referencial o canto superior esquerdo, calcula-se as coordenadas do centro do ecrã. Este será a origem do referencial para o qual queremos converter.

Por fim, calcula-se as coordenadas do vetor que vai da origem do referencial no centro do mapa, para as coordenadas do carro em pixéis.

No entanto, ocorre inversão do eixo das ordenadas. Para isso ser contabilizado, o valor devolvido do y é o simétrico do vetor calculado anteriormente.

```
cornerToCenter::Int -> Tabuleiro -> Ponto -> Ponto
cornerToCenter pSize tab (x,y) = ( cx + p * x, cy - y * p )
    where
        cy = ((fromIntegral(length tab)/2) + 0.5 )* p
        cx = (-(fromIntegral(length (head tab))/2) - 0.5) * p
        p = fromIntegral pSize
```

A função **drawCar** através da conjugação de um **translate**, em que os valores dos *floats* são as coordenadas onde o *Carro* se encontra, com um **rotate**, que permite que o *Carro* rode até ao simétrico da sua *Direcao*, desenha um carro no mapa do ecrã.

```
drawCar:: Estado -> [Picture] -> Int -> Picture
drawCar e picCars i = Translate (realToFrac x) (realToFrac y) (Rotate (realToFrac a) pic)
    where
        tab      = mapToTab (mapa (jogo e))
        listCar = carros(jogo e)
        (x,y)   = cornerToCenter pSize tab (posicao (listCar !! i))
        a       = -1 * direcao (listCar !! i)
        pic     = scale fs fs (picCars !! i)
        pSize   = sizePeca (mapToTab (mapa (jogo e))) (winSize e)
        fs      = fromIntegral pSize * 0.50 / 2400
```

Na função **drawCars** irá ser aplicada a função **drawCar** a cada um dos carros.

#### Extras no Modo de Corrida

É no modo de corrida onde se encontram a maior parte dos extras do jogo.

Começamos por explicar aquele que nos obrigou a criar um *datatype* novo, designado de *Camera*. Ele é o tipo que define o parâmetro *camera* do *Estado*.

No modo corrida, o jogador tem a possibilidade de alterar a câmera com que visualiza o jogo. Existem 3 opções: *full*, câmera pré-definida para o inicio da corrida, *close*, que efetua zoom sobre o jogador em questão e *birdEye*, na qual o jogador vê o jogo através de uma perspectiva superior à dos carros, ou por outras palavras, "vê os carros vistos de cima".

Para fazer a câmera em modo *close* basta aplicar um **translate** à imagem que desenha os carros e o mapas. Os dois *floats* que o *translate* irá receber serão o simétrico da coordenadas onde se encontra o carro.

Em modo *birdEye* aplica-se um **rotate** à imagem formada no modo *close*. Essa rotação irá ocorrer segundo o ângulo da direção inicial do carro, menos 90 graus.

É possível também colocar o jogo em Pausa. Basta que o parâmetro *pausa* do *Estado* seja verdadeiro. Quando isso acontece irá ser ativado o parâmetro *pauseMenu* (que faz parte do *datatype Images*) alterando as *images* do *Estado*. O *datatype Images* foi criado para armazenar todas as imagens que serão necessárias no jogo.

Cada jogador terá um identificativo no canto superior direito do ecrã, com o respectivo nitro. Estes identificadores aparecerão ordenados consoante a posição do carro do jogador na corrida. O nitro será renovado sempre que o jogador acabar de dar uma volta à pista.

Mais à frente será abordado de uma forma mais específica, mas em virtude deste tópico, refere-se já que existirão 3 níveis de dificuldade. No nível mais fácil, o jogo terminará após 3 voltas à pista. No nível intermédio, acabará com 2 voltas ao circuito e no nível mais avançada apenas será necessário completar uma volta.

No ecrã estará também o número de voltas percorridas pelo jogadores. Esse número corresponderá ao nº de voltas do jogador que se posiciona em primeiro lugar. A função responsável pelo desenho é a **drawLapCount**, que em virtude do *nVolts* que recebe do *Estado*, desenha no ecrã o número correspondente. Na última volta à pista, em vez de aparecer o número, aparece a expressão "*final lap*".

Para que a função **drawLapCount** saiba o que deve reproduzir no ecrã, auxiliou-a a função **lapCountList**. Esta função devolve o número de voltas dadas por cada jogador. Para saber o que desenhar, a função **drawLapCount** retira o maior número da lista de inteiros devolvida pela **lapCountList** e adiciona-lhe 1, uma vez que o inteiro a que corresponde o número de voltas dadas por cada jogador, só se altera quando o *Carro* termina a volta em que se encontra. No visor deve aparecer o número dessa volta.

Por fim, o jogador poderá também ver no ecrã um cronómetro com o tempo desde o inicio da partida. Mais uma vez, utilizá-mos para nos auxiliar uma função pré-definida no *Gloss*, a função **text**, que dada uma *string* a coloca no ecrã. A *string* que irá receber corresponderá ao valor do parâmetro *tFromStart* definido no *Estado*.

Desta forma, todas as funções que permitem o funcionamento da função *desenhaJogo* já foram explicadas. Em virtude do tamanho da sua definição não a iremos apresentar. Contudo, fica o sumário das situações em que esta função é ativada.

- Desenhar o jogo em *CountDown*;
- Desenhar o jogo em pausa;
- Desenhar o jogo nas diferentes câmeras.

**Elaboração da *desenhaMenus***

Para a construção dos menus será fundamental o parâmetro *whereMenu* presente no *Estado*. Na função **desenhaMenu** estarão programadas as imagens respectivas ao menu que o jogador selecionar.

O grupo utilizou a função **scale** para escalar as imagens dos menus. Foram utilizados dois pares de valores diferentes para os escalonamento das *pictures*.

À exceção do desenho do *help* no menu inicial, os valores de escalonamento foram o valor mínimo entre ,a divisão do coordenadas da janela dada a partir do *winSize*, e 2048.



Figura 3.2: Menu Inicial

Em relação ao caso que é exceção o que foi dito em cima, os valores utilizados foram na mesma o mínimo da divisão de cada uma das coordenadas da *winSize*, mas desta vez por valores mais elevados, em virtude do tamanho da palavra *help*.

Quando a corrida termina, um novo menu aparece no ecrã. A função responsável pela sua formação é a **desenhaMenuEndGame**. Os jogadores irão aparecer no ecrã conforme a sua posição no final da corrida. Para isso a auxiliar esta função vem a **listIDs** que dá uma lista ordenada dos carros consoante a sua posição no jogo.

Em baixo dessa lista aparecerá o simbolo do *enter*, obtido através do parâmetro *extraCreate* definido no datatype *Images*.

### Elaboração da *reageTempo*

<i>Etapa</i>	<i>Objetivo</i>
1º	<i>Ponto</i> onde se encontra <i>Carro</i> após as Destruíções
2º	Implementação da função <i>movimenta</i>
3ª	Implementação da função <i>atualiza</i>
5º	Tempos antes e depois a corrida

Tabela 3.8: Fases de Elaboração da *reageTempo*

Anteriormente no processo de elaboração da **desenhaEstado** falámos da função **drawCars** que desenha os carros no mapa do ecrã. Dessa forma seria necessário criar uma função que os fizesse movimentar.

Como já foi dito na análise da solução da tarefa 3, a função **movimenta** devolve uma lista de **Maybe Carro**, referente à movimentação de um *Carro*. Sempre que nessa lista aparece um *Nothing*, significa que o *Carro* foi destruído.

Pelas instruções que recebemos do enunciado, sempre que tal situação ocorresse, o *Carro* returnaria para o centro da última *Peca* que percorreu sem ser destruído. Por esse motivo criou-se a função **colidePosFinal**. Dada a *Peca* e o ponto onde ocorre a destruição, a função redireciona o *Carro* para o centro dessa *Peca*.

```
colidePosFinal:: Peca -> Posicao -> Ponto
colidePosFinal (Peca (Curva Norte) _) (x,y) = (fromIntegral x + 0.7, fromIntegral y + 0.7)
colidePosFinal (Peca (Curva Sul) _) (x,y) = (fromIntegral x + 0.3, fromIntegral y + 0.3)
colidePosFinal (Peca (Curva Este) _) (x,y) = (fromIntegral x + 0.3, fromIntegral y + 0.7)
colidePosFinal (Peca (Curva Oeste) _) (x,y) = (fromIntegral x + 0.7, fromIntegral y + 0.3)
colidePosFinal _ (x,y) = (fromIntegral x + 0.5, fromIntegral y + 0.5)
```

Caso não haja nenhuma destruição, o *Ponto* onde se encontra o *Carro* após um *Evento* será a parte do resultado devolvido pela função *movimenta* sem o construtor *Just*. Assim, implementou-se a função **movimentar**, que movimenta um *Carro* após a passagem de um período de tempo.

```
movimentar:: Mapa -> Float -> Carro -> Posicao -> Carro
movimentar map tick car (x,y) | move == Nothing = car{velocidade = (0,0), posicao = pColide}
                                | otherwise      = fromJust move
where
    tab = mapToTab map
    move = movimenta tab (realToFrac tick) car
    pColide = colidePosFinal (pecaFinder map (x,y)) (x,y)
```

Esta função será aplicada aos 4 carros que constituem o *Jogo*, através da função **reageTempoMovimenta**.

```
reageTempoMovimenta:: Float -> Estado -> Estado
reageTempoMovimenta tick e = e{jogo = (jogo e) {carros = [moveCar1, moveCar2, moveCar3, moveCar4]}}
where
    map = mapa(jogo e)
    game = jogo e
    listCar = carros game
    moveCar1 = movimentar map tick (listCar !! 0) (head ((historico game) !! 0))
    moveCar2 = movimentar map tick (listCar !! 1) (head ((historico game) !! 1))
    moveCar3 = movimentar map tick (listCar !! 2) (head ((historico game) !! 2))
    moveCar4 = movimentar map tick (listCar !! 3) (head ((historico game) !! 3))
```

Existindo movimento, é necessária a sua atualização. De novo, necessitámos de evocar uma função construída numa tarefa anterior, neste caso a função **atualiza**. Esta será aplicada a cada um dos 4 carros.

```
reageTempoAtualiza::Float -> Estado -> Estado
reageTempoAtualiza tick e = act4
where
    act1 = e    {jogo = atualiza (realToFrac tick) (jogo e)    0 (a1 (actions e))}
    act2 = act1 {jogo = atualiza (realToFrac tick) (jogo act1) 1 (a2 (actions e))}
    act3 = act2 {jogo = atualiza (realToFrac tick) (jogo act2) 2 (a3 (actions e))}
    act4 = act3 {jogo = atualiza (realToFrac tick) (jogo act3) 3 (a4 (actions e))}
```

Ainda não falada, mas também necessária foi a função **bot**, que irá fazer com que ocorra a atualização do movimento de cada um dos bots a partir de um tempo dado.

Esta é sempre evocada, no entanto, o número de *carros* que são controlados por esta função dependem do número de jogadores. Caso seja 1 o carro 2, 3 e 4 são controlados. Se forem 2, apenas o carro 3 e 4 são controlados pela função.

```
reageTempoBots::Float -> Estado -> Estado
reageTempoBots tick e | nPlayer e == 1 = bot3
                      | nPlayer e == 2 = bot2
where
    bot1 = e    {actions = (actions e )}{a4 = (bot (realToFrac (tick)) (jogo e ) 3)} }
    bot2 = bot1 {actions = (actions bot1){a3 = (bot (realToFrac (tick)) (jogo bot1) 2)} }
    bot3 = bot2 {actions = (actions bot2){a2 = (bot (realToFrac (tick)) (jogo bot2) 1)} }
```

Uma outra função que também está relacionada com a função **reageTempo** é a **reageTempoCountDown**. Para a sua definição foram preponderantes dois

parâmetros existentes dentro do *record* que define o *Estado* : *inCountDown*, que nos diz se no jogo está a ocorrer a contagem decrescente para o inicio da corrida, e *tToStart*, que nos diz quanto tempo falta para a corrida começar. O objetivo da função **reageTempoCountDown** é atualizar o parâmetro *tToStart*.

Auxiliando esta função vem a função **ab** que altera o parâmetro *inCountDown* sempre que o jogo não esteja em contagem decrescente. Isso acontece sempre que a diferença entre o tempo para começar a corrida e o tempo de jogo passado for igual a 0. Quando este resultado é negativo, a função **ab** também dá como falso o resultado do *inCountDown*.

```
reageTempoCountDown:: Float -> Estado -> Estado
reageTempoCountDown tick e | ab dt == 0 = e {inCountDown = False, tToStart = 0}
                           | otherwise   = e {tToStart = dt}
where
  t = tToStart e
  dt = t - tick
  ab::: Float -> Float
  ab x | x < 0 = 0
        | otherwise = x
```

Quando a corrida termina, a função **reageTempoGame** devolve o *Estado* correspondente. Para o formar, esta função terá a influência da função **checkLaps**.

Através da função **lapCountList** forma-se uma lista com as voltas que cada *Carro* deu à pista. A partir do momento em que nessa lista existe algum número maior que o *nVoltas*, a função **checkLaps** altera o parâmetro *whereMenu* para *EndGame* e o *inMenu* torna-se verdadeiro.

O parâmetro *EndGame* faz parte de um *datatype* definido pelo grupo sobre a forma de *record*, apelidado de *MenuPos*. Nesse *record* estão definidos todos os menus em que o jogador se pode encontrar.

Na *reageTempoGame* o *Estado* que é passado à *checkLaps* ja tem em conta que os *nitros* são renovados a partir do momento em que o *Carro* passa na linha de meta.

Ainda de referir que, com o passar do tempo, o valor *tFromStart* presente no *Estado* aumenta de forma a manter um registo de quanto tempo já passou.

```
reageTempoGame::: Float -> Estado -> Estado
reageTempoGame tick e = checkLaps addTime
where
  botApplied      = reageTempoBots tick e
  actSemMovimenta = reageTempoAtualiza tick botApplied
  actualizado    = reageTempoMovimenta tick actSemMovimenta
  rNit           = restoreNits actualizado
  addTime        = rNit {tFromStart = tFromStart actualizado + tick}
```

Por fim, falta ainda referir como ocorreu a conjugação de todas as funções relacionadas com a atualização do *Tempo*. A função **reageTempo** para além de receber um *Tempo*, recebe também um *Estado*.

Quando no jogo, o jogador se encontra num dos menus ou em pausa, os parâmetros do *Estado* recebido não se alteram.

Em contagem decrescente para o inicio da corrida, o parâmetro *inCountDown* é verdadeiro pelo que o *tToStart* do *Estado* é alterado.

Em qualquer outra situação, a única função que influencia a **reageTempo** é a **reageTempoGame**.

### Elaboração da função reageEvento

Para a definição desta função, foi necessário a utilização do *data Event*, pertencente a umas das *packages* do *Gloss*, a *Graphics.Gloss.Interface.Pure.Game*.

Tendo como base o construtor *Event Key* é possível definir quais os elementos do teclado que permitirão a ocorrência alguma alteração no jogo feita pelo jogador.

Com o construtor *EventResize*, o grupo garante também que as figuras construídas no jogo irão se alterar proporcionalmente quando ocorrer a alteração das dimensões da janela onde este decorre.

Existem 3 partes distintas do jogo : quando o jogador se encontra num dos menus, no momento da contagem decrescente para o inicio da corrida e dentro dela. Dessa forma a auxiliar a função **reageEvento** vem as funções **reageEventoMenu**, **reageEventoCountDown** e **reageEventoJogo**.

```
reageEvento :: Event -> Estado -> Estado
reageEvento (EventResize size) e = e {winSize = size}
reageEvento (EventKey (SpecialKey KeyTab) Down _) e = e {inMenu = True, whereMenu = MenuInicialStart}
reageEvento ev e | inMenu e = reageEventoMenu ev e
| inCountDown e = e
| otherwise = reageEventoJogo ev e
```

Sempre que o jogador inicia uma corrida, existem 4 segundos em que este espera pelo seu começo. Nessa ocasião, atua a função **reageEventoCountDown**. Ela não irá alterar o *Estado* do jogo.

Na função **reageEventoJogo** são definidos os comandos a utilizar pelo jogador quando este se encontra no modo corrida.

De referir também que em qualquer momento do jogo, se o jogador premir o *tab*, então voltará ao menu inicial.

Em baixo encontra-se uma imagem que resume quais são as teclas a utilizar pelo jogador no "Micro Machines" feito pelo grupo.

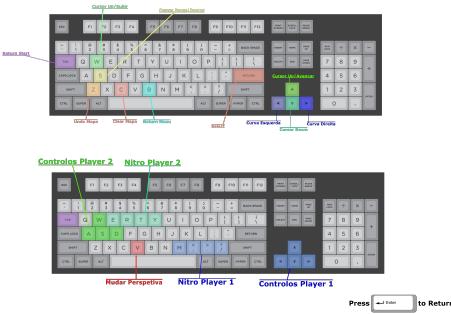


Figura 3.3: Teclados do Jogo

Na função **reageEventoMenu** são definidas as teclas que permitem a alteração do *Estado* do jogo quando o jogador se encontra em algum dos menus.

Ainda relacionada com a função **reageEventoMenu** vem um dos extras do jogo que o grupo implementou na parte dos menus, a construção de mapas pelo jogador. Se o mapa construído for válido, inicia-se automaticamente uma corrida *single player*.

Nada disto seria possível sem o parâmetro *pMapa* definido no *Estado*. Ele transmite o *Caminho* escolhido pelo jogador à função **specialMapCheck**. A esta cabe o papel de verificar se o *Caminho* dado forma um mapa válido. O mapa forma-se devido à função **constroi**, definida na Tarefa 1. Para verificar se o mapa formado é válido ocorre a intervenção da função **valida**, definida na Tarefa 2.

Se o mapa for válido, a **gameCreator** forma o *Jogo* para o inicio da corrida. Esta função recebe um *Jogo* e um *Mapa* e devolve um novo *Jogo*.

O *Mapa* do novo *Jogo* será o mapa dado à função.

As posições dos carros estarão próximas da posição de partida, dada pelo *Mapa*.

<i>Carro</i>	<i>Posicao</i>
1	( <i>x</i> + 0.1, <i>y</i> + 0.2)
2	( <i>x</i> + 0.1, <i>y</i> + 0.4)
3	( <i>x</i> + 0.1, <i>y</i> + 0.6)
4	( <i>x</i> + 0.1, <i>y</i> + 0.8)

Tabela 3.9: Posições dos carros no inicio do *Jogo*

Na tabela acima representada, o *x* e o *y* correspondem, respectivamente, à abcissa e à ordenada dessa *Posicao* de partida.

A velocidade de todos será (0,0) e a direção será o ângulo correspondente à *Orientacao* inicial do *Mapa*.

Todos os carros terão 5 unidades de tempo em nitro.

O histórico de cada um contará apenas com a posição de partida definida no *Mapa*.

Ainda relacionado, com a função **reageEventoMenu** vem a função **idPathGenerator**.

O parâmetro *idPath* devolve a uma lista de *Posicao* correspondente às posições necessárias para dar uma volta completa à pista. A função **idPathGenerator** irá renovar esse parâmetro assim que o jogador completa uma volta à pista.

```
idPathGenerator :: Mapa -> [Maybe Posicao]
idPathGenerator m@(Mapa (p,o) t) = tail almostFinal ++ [head almostFinal] ++ [Just p]
  where
    nAvancos      = 2 * (nBlocosCaminho m + 1)
    coorAntesPartida = nudge p (orOposta o)
    rawListCoor   = percorreMapa m (coorAntesPartida, o) nAvancos 1
    almostFinal   = unique rawListCoor
```

No final da corrida a função *reageEventoMenu* tem de novo influência na *reageEvento*. O parâmetro *EndGame* será ativado pelo que o grupo definiu que quando isso se sucede-se, o *whereMenu* alteraria-se para *MenuInicialStart*. Isto significa que depois do final da corrida, o jogador voltará novamente para o menu inicial.

**Função joga**

Depois da explicação de todos os constituintes que fazem parte da função **joga**, eis que chegou a altura de apresentar a sua definição.

```
joga :: Estado -> IO()
joga inicio = play
    (InWindow ("Micro Machines") (1280,720) (0,0) )
    (greyN 0.8)
    60
    inicio
    desenhaEstado
    reageEvento
    reageTempo
```

As dimensões do jogo serão 1280 pixeis de comprimento por 720 pixeis de altura. A ideia por detrás desta escolha é muito simples. Nos dias de hoje a resolução mais comum de ecrãs é o fullHD, ou seja 1920\*1080 pixeis, no entanto ainda é normal a existência de ecrãs HD, ou seja, 1280\*720 páxeis. Assim, quando o jogo começa já está a ocupar o ecrã completo para estes casos e para os ecrãs fullHD basta expandir a janela visto que o jogo está preparado para se adaptar a qualquer janela de visualização. A janela do jogo começa na posição (0,0). A cor do fundo é cinzento claro, um tom muito utilizado no design gráfico e em jogos minimalistas. Por fim, o jogo funciona a 60 fps. Mais uma vez pois os ecrãs mais comuns são os que funcionam a 60 fps. O segmento denominado de inicio será discutido num tópico seguinte.

**Função main**

Esta função que fomos propostos a realizar nesta Tarefa 5.

```
main :: IO ()
main = do inicio <- estadoInicial
          joga inicio
```

O inicio será o estado primordial do jogo. Será o primeiro *Estado* que a função **joga** irá receber. A função é um *IO ()* pois realiza uma ação com um efeito colateral, ou seja, ela lê do dispositivo de entrada e faz com que algo seja imprimido na tela.

#### Elaboração do *estadoInicial*

O jogo inicia-se pelo Menu Inicial. Nele o jogador pode partir para os menus seguintes que o levarão à corrida, entrando na opção *start*, ou pode clicar em *help* e optar por esclarecer quais as funcionalidades das diferentes teclas do teclado no jogo.

A função **estadoInicial** é responsável por devolver o datatype *Estado* referente ao inicio do jogo. Com a **estadoInicial** serão carregados todos os *sprites* que farão parte do jogo. Desta forma justifica-se o facto de esta função devolver um *IO Estado*, uma vez que importa dados do exterior do jogo.

Para já, o mapa do jogo será vazio, por defeito serão consideradas inicialmente as propriedades relativas ao asfalto. Inicialmente, todos os parâmetros da *Acao* serão falsos. Os *nitros* serão constituidos por uma lista de *Nothing*. Os jogadores terão as suas imagens respectivas, carregadas no inicio da definição da função.

O número de jogadores será 1, a *camera* estará no modo *full*, haverá uma janela de dimensões (0,0) e como é óbvio, o jogo não estará em *Pausa*.

O *inCountDown* será falso, ao invés do *inMenu*. O *whereMenu* será *MenuInicialStart*

O *tToStart* será 4 (uma vez que serão necessários 3 segundos na contagem decrescente mais um outro para transmitir a mensagem "Go !"). Como a corrida ainda não começou, o *tFromStart* será 0.

Por fim, a lista com o *Caminho* que é dado para a geração do mapa pelo jogador será uma lista vazia, o número de voltas será igual a 0 e a lista com as posições a percorrer para dar uma volta completa à pista será também uma lista vazia.

### 3.2.4 Tarefa 6

Para jogar qualquer jogo de carros é necessário ter astúcia para controlar o acelerador, bem como saber quando é que é necessário virar. Pensando dessa forma, para programar a estratégia de corrida dos bots, o grupo optou por ter um conjunto de funções responsáveis pelo controlo das viragens e um outro responsável pelo controlo da aceleração.

#### Controlo das viragens

Qualquer movimentação de um carro irá provocar deslocamento. Este provoca sempre a formação de um ângulo, desde o ponto inicial até ao ponto final.

Dadas as coordenadas do deslocamento, com a função **desToAng** é saber qual o ângulo do deslocamento.

```
desToAng:: (Int,Int) -> Angulo
desToAng (0 , y) | y > 0 = 270
                  | y < 0 = 90
desToAng (x , 0) | x > 0 = 0
                  | x < 0 = 180
desToAng (1 , y) | y > 0 = 270
                  | y < 0 = 90
desToAng (x , 1) | x > 0 = 0
                  | x < 0 = 180
desToAng (-1 , y) | y > 0 = 270
                   | y < 0 = 90
desToAng (x , -1) | x > 0 = 0
                   | x < 0 = 180
```

Através do valor do *Angulo* pode-se deduzir qual foi a *Orientacao* do movimento. Mais à frente, perceberemos qual a importância deste pormenor.

Com esta função e através do histórico é possível saber qual o *Angulo* com que o carro circula. Isso obtém-se na função **angDes** em que ocorre o cálculo do *Angulo* do deslocamento entre a cabeça do *historico* e a *Posicao* que se encontra na cabeça da cauda. Salienta-se que a *Posicao* final do carro é a que se encontra no inicio do *historico*.

O *Carro* apenas necessita de virar nas curvas. Dependendo da curva em questão, a *Orientacao* que o *Carro* deve levar para não embater na lava difere. Sendo assim, foi necessário criar a função **orCurva** que nos diz qual é a *Orientacao* da curva. A partir dai, consegue-se prever qual o *Angulo* que o *Carro* deve levar para ultrapassar a *Peca* em questão sem ser destruido.

Em virtude disso mesmo, criou-se a função **angChangeCurva** que recebe o *Angulo* que o *Carro* leva e a *Peca* em que se encontra. A função **orCurva** analisa qual a *Orientacao* da curva. Dependendo do resultado que devolve, a **angChangeCurva** diz qual o *Angulo* que o *Carro* deve sair da *Peca*.

```

angChangeCurva:: Angulo -> Peca -> Angulo
angChangeCurva a p | a == 90 && orCurva == Norte = 0
| a == 90 && orCurva == Este = 180
| a == 270 && orCurva == Sul = 180
| a == 270 && orCurva == Oeste = 360
| a == 0 && orCurva == Sul = 90
| a == 0 && orCurva == Este = 270
| a == 180 && orCurva == Norte = 270
| a == 180 && orCurva == Oeste = 90
    where
        Peca (Curva orCurva) h = p

```

Assim sendo os **bots** apenas mudarão o seu *Angulo* de deslocamento nas curvas. A função **angChange** fica encarregue de fazer com que isso aconteça.

```

angChange:: Angulo -> Peca -> Angulo
angChange a p | tipoFinder p == Curva Norte = angChangeCurva a p
| otherwise = a

```

Desta feita, temos tudo que é necessário para que os **bots** possam virar. Definiu-se a função **cruiseAng** que devolve dois valores booleanos. O primeiro é verdadeiro quando o *bot* deve virar à direita. Por outro lado, quando o segundo elemento do tuplo é verdadeiro, então o carro vira à esquerda.

Utilizando a função **angChange** foi possível calcular o *Angulo* do possível trajeto do *bot*, atendendo à *Peca* em que este se encontra.

Considere-se **a** como sendo o valor do *Angulo* do deslocamento do *Carro* e **target** o valor do ângulo com que o carro deve abandonar a *Peca* em que se encontra. Por fim **trim** tem valor 3. Este é um valor que foi considerado como um desvio do valor de **target**. O grupo considerou este valor porque a nível visual, a movimentação dos carros tornou-se mais parecida com a realidade. Existem 3 situações hipotéticas.

Se :

**a > target + trim && a < target + 180 - trim**

Então o bot vira à direita

**a < target - trim && a > target - 180 + trim**

Então o bot vira à esquerda

Caso contrário, o bot não vira.

Controlo da Aceleração

Os *bots* não mantêm a *Velocidade* constante em todas as peças. No entanto, é possível calcular qual a *Velocidade* ideal a levar em qualquer das peças em que os *bots* estejam. Logicamente, nas curvas essa *Velocidade* será mais reduzida.

Dependo da *Posicao no Jogo*, os *bots* podem necessitar de ter uma *Velocidade* superior à *Velocidade* ideal, de maneira a chegar ao primeiro lugar.

Devido a isto, a função **speedCalc** devolve um par de *Double*, em que o primeiro elemento corresponde à *Velocidade* ideal a levar, ao passo que o segundo elemento é a *Velocidade* de emergência. A função faz também uma distinção entre os casos em que as peças são curvas e os restantes.

```
speedCalc:: Jogo -> Int -> (Double,Double)
speedCalc j i | isCurva currPeca = (1,2)
               | otherwise      = (2.1,3)
               where
                  currPeca = pecaFinder (mapa j) (cx, cy)
                  (cx, cy) = head carHist
                  carHist = historico j !! i
```

Devido a estas variações de *Velocidade*, haverão momentos em que os *bots* terão de acelerar ou de travar. A função **cruiseSpeed**, através de um *Jogo* e de um *Int*, identificativo do jogador, devolve um par de booleanos. Se o primeiro elemento de desse tuplo for verdadeiro, então os *bots* terão de acelerar. Quando é o segundo elemento verdadeiro, os *bots* tem de travar.

Para que ocorra o controlo das travagens e das acelerações de um *bot*, na função **cruiseSpeed** é calculado o módulo das sua *Velocidade*, atendendo ao *Jogo* que estava função recebe.

Quando a *Velocidade* é superior à velocidade de emergência, então os *bots* devem travar.

Sendo a velocidade ideal maior à *Velocidade* dos *bots*, então a aceleração é ativada.

Caso contrário, os *bots* não aceleram nem travam.

**Controlo do Nitro**

Relativamente à ativação do nitro, o grupo decidiu que os *bots* apenas o utilizariam quando o módulo da sua *Velocidade* fosse inferior a 2. Realça-se que em peças do *Tipo Curva Orientacao*, o nitro nunca é ativado.

A função responsável pela colocação de *nitros* nos *carros* chama-se **nitDecide**. Em função das circunstâncias do *Jogo*, a função devolve um *Maybe Int*, correspondendo o inteiro ao identificativo do *Carro* na qual o o tempo de nitro será descontado. Caso o nitro não seja ativado, a função devolve um *Nothing*.

O grupo optou também por criar *bots* que não ativam o nitro dos adversários.

**Elaboração da função bot**

Assim sendo, já foi explicado tudo aquilo que faz parte da função **bot**, responsável pela estratégia de corrida dos *bots*.

```
bot :: Tempo -> Jogo -> Int -> Acao
bot tick e j = Acao { acelerar = acel, travar = trav, esquerda = esq, direita = dir, nitro = nitDecide e j}
  where
    (acel, trav) = cruiseSpeed e j
    (esq , dir ) = cruiseAng   e j
```

### 3.3 Documentação

O que tem um papel preponderante para o mais fácil entendimento do código redigido pelo grupo é a documentação, gerada automaticamente em Haddock.

O grupo teve em conta diversos fatores para que fosse visualmente apelativa e de fácil compreensão, tais como o facto de gerar um menu no canto superior direito da página, com uma divisão estruturada das funções consoante a fase do método de execução.

Mais a mais todas as funções apresentam pelo menos um exemplo explicativo, acompanhado de uma definição clara da sua funcionalidade.

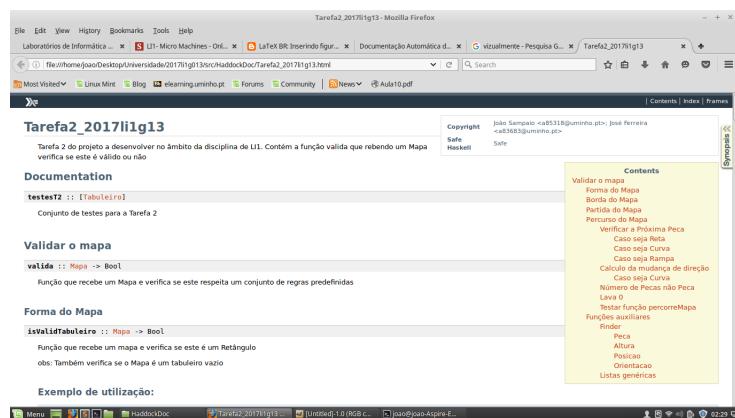


Figura 3.4: Documentação Haddock

# Capítulo 4

## Validação da Solução

Aos alunos foi também disponibilizado um Sistema de *Feedback* e um repositório de controlo de versões (intitulado SVN). Assim, cada grupo tem as ferramentas ideais para verificar qual o resultado daquilo que estão a produzir, com recurso a testes desenvolvidos pelo grupo e fornecer sugestões que visam a melhoria de erros de redundância no código.

Na 1ºfase do projeto, a validação da solução tomada, para além de ser em função do *feedback* devolvido pelo *ghci*, baseou-se essencialmente nos resultados obtidos após a colocação de testes no SVN.

De destacar, que na Tarefa 2, o visualizador de Caminhos e Mapas do SVN foi bastante importante para o grupo, uma vez que assim conseguiu-se uma projeção dos mapas a inserir nos testes referentes a essa mesma tarefa.

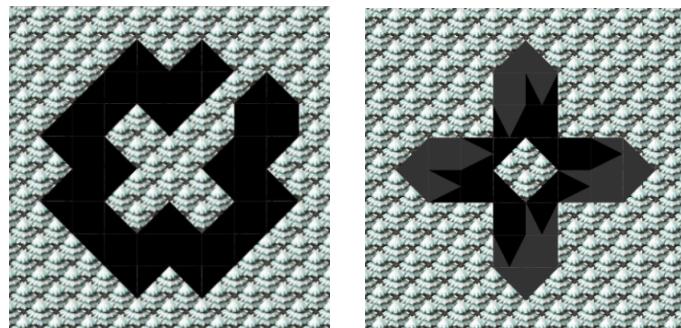


Figura 4.1: Mapas visualizados no visualizador de Caminhos e Mapas.

A partir do sistema de controlo de versões, o grupo conseguiu também perceber qual era o valor da solução encontrada para a Tarefa 3, relativamente ao oráculo elaborado pelos professores da disciplina.

No que diz respeito à 2ºfase do projeto, a Tarefa 5 foi a única em que o SVN não teve interferência.

Na Tarefa 4 ele permitiu que o grupo percebesse que o raciocínio inicial para obter a norma da *Velocidade do Carro* estava errado. O tempo deve ser multiplicado por todas as constantes das *Propriedades* que tem interferência no cálculo da norma da *Velocidade*, aspeto que o grupo não tinha considerado como

## CAPÍTULO 4. VALIDAÇÃO DA SOLUÇÃO

---

relevante.

O torneio dos *bots* permitiu perceber qual a influência do código do grupo na movimentação dos carros. Inicialmente não se tinha introduzido nenhum desvio no *Angulo* sobre o qual o *Carro* poderia sair das curvas. Após a visualização da sua movimentação no torneio, percebeu-se que o andamento do *Carro* era mais natural se se adicionasse um valor padrão ao valor considerado anteriormente.

O SVN foi também importante ao nível da documentação. Com ele, cada grupo verificava se todas as suas funções estavam documentadas.

A verificação dos resultados da Tarefa 5 realizou-se diretamente no executável criado. A "tentativa erro" foi importantíssima para o melhoramento da qualidade do jogo.

Para validar esta tarefa, não poderíamos deixar de falar da documentação do *Gloss*. Pela sua análise, o grupo percebeu se estaria a utilizar as funções disponibilizadas nas suas *packages* de forma adequada.

Para finalizar, o grupo através da instalação do pacote *hLint* conseguiu identificar erros redundantes no código sem recorrer ao SVN.

# Capítulo 5

## Conclusão

O grupo conseguiu concluir com validade os 6 objetivos pretendidos nas tarefas.

Começando pela Tarefa 1, reconhece-se com normalidade que esta foi aquela que apresentava um menor grau dificuldade. Necessitava-se de compreender bem como utilizar a recursividade, de maneira a chegar à função pretendida.

A tarefa 2 envolvia um pouco mais de imaginação relativamente à tarefa primordial. Foi necessário ser consciente da maior parte dos casos em que o mapa não poderia ser válido, para que a partir disso fosse possível criar um código capaz de identificar a sua invalidade.

A tarefa 3 não foi totalmente desenvolvida pelo grupo. Tal deve-se ao facto de que o grupo dedicou demasiado tempo à realização das duas primeiras tarefas desta parte. Por isso, a função **movimenta** não considera os casos em que ocorrem ricoschetes, nem as diferenças de altura. Na 2ºfase do projeto foi possibilitada a entrega de uma nova Tarefa 3. O grupo abdicou de poder fazer alguma alteração no código, preferindo dedicar mais tempo à Tarefa 5, podendo assim implementar funcionalidades novas.

Na tarefa 4, ocorreu reaproveitamento de funções criadas anteriormente, nomeadamente das funções **pecaFinder** e **orFinder**. Tal foi recorrente com outras Tarefas pois este trabalho funciona em torno de um raciocínio continuado. Em relação à tarefa propriamente dita, a sua resolução envolveu a consideração de pormenores fundamentais. Por exemplo, todas as constantes têm de ser multiplicadas pelo *Tempo*. No inicio isto não foi considerado pelo grupo levando ao aparecimento de erros inesperados.

A tarefa 5 permitiu que existisse mais liberdade na busca da solução. Na criação do executável do jogo, o grupo optou por criar um design mais minimalista, isto é, criar um jogo que fosse atrativo pela sua simplicidade gráfica. Procurou-se ter um jogo rico em mapas, sendo estes separados em três diferentes meios : Asfalto, Gelo e Terra . Para além disso, em cada meio, existem 3 diferentes níveis de dificuldade. Desta forma desperta-se mais facilmente o gosto pelo jogo no jogador, uma vez que este se sente desafiado. Pode mesmo estabelecer um objetivo a cumprir : terminar em primeiro lugar em todas as pistas dos diferentes meios.

O grupo investiu também nos vários extras que o jogo poderia ter.

Se o jogador não se sentir satisfeito com os mapas pré-definidos no executável, pode optar por criar um mapa. Para a implementação deste extra, ocorreu reaproveitamento de trabalho feito anteriormente, uma vez que a função **valida**

## CAPÍTULO 5. CONCLUSÃO

---

e **constroi** é que permitiram o desenvolvimento desta funcionalidade.

Entrando no modo de corrida, o jogador tem a seu dispor três angulos de câmeras distintas. Foi também colocada a renovação dos tempos de nitro no momento em que o *Carro* passa na linha de partida. Com isto o jogo torna-se mais dinâmico e apelativo.

O resultado da Tarefa 6, acaba por ser aplicado na Tarefa 5. Criou-se um grupo de *bots* ciente do estado da corrida, uma vez que sabiam como controlar a velocidade e qual a melhor altura para aplicar nitro em si mesmos. Não se programou que os nitros fossem aplicados pelos *bots* outros jogadores. Assim obtem-se um jogador muito mais satisfeito, uma vez o movimento do seu *Carro* depende exclusivamente de si ou de outro jogador com quem está a jogar. Picardias à parte, que sejam com alguém com que possam desfrutar um bom momento.

Em jeito de verdadeira conclusão, após o *terminus* das 6 tarefas, cada elemento do grupo apercebe-se que com este trabalho as suas qualidades como programadores foram bastante desenvolvidas. Aprendeu-se que se deve conhecer bem o problema que se tem em mão, que as ferramentas criadas podem ser posteriormente reutilizadas e que a chave para a criação de um código eficiente é perceber perfeitamente aquilo que se cria.

## **Capítulo 6**

## **Bibliografia**

- <http://www.wasd.pt/analises/analise-micro-machines-world-series>
- <http://aindasoudotempo.blogspot.pt/2013/03/do-micro-machines.html>
- <http://haskell.tailorfontela.com.br/input-and-output>