

ÁREA DEPARTAMENTAL DE ENGENHARIA DE ELETRÓNICA E
TELECOMUNICAÇÕES E DE COMPUTADORES

LICENCIATURA EM ENGENHARIA INFORMÁTICA E MULTIMÉDIA



ANO LETIVO 2019/2020 – SEMESTRE DE VERÃO

Gestão de Táticas de Basquetebol

RODRIGO ESTEVES

NÚMERO: 42372

TURMA: 61D

E-MAIL: A42372@ALUNOS.ISEL.PT

ORIENTADOR: PROFESSOR RUI JESUS

12 de Setembro de 2020

Resumo

O basquetebol é um desporto de equipa reconhecido mundialmente e praticado por milhões de pessoas, umas por diversão, outras profissionalmente, algumas de ambas as formas. Não existe uma forma correta de praticar este desporto, mas há técnicas e formatos que ajudam os jogadores a potencializar o seu desempenho. Organizar movimentos e criar uma tática é um desses formatos.

Nos tempos mais recentes, foi-se observando um grande crescimento na área da tecnologia, o que potenciou igualmente uma adaptação do desporto à mesma. Foi introduzido o vídeo, de forma a repetir acontecimentos passados, com o objetivo de estudá-los para mais tarde melhorá-los. Foram também criados sistemas capazes de simular as táticas criadas pelos treinadores, sendo este capítulo o foco deste projeto.

Pretende-se implementar um sistema que torna o utilizador capaz de criar e gerir a sua tática, tendo pela sua frente um ajuste automático da equipa defensiva. Consequentemente, o utilizador pode apurar os seus conhecimentos táticos, praticando e testando os seus movimentos até fortalecer ao máximo a eficiência e a eficácia da sua jogada.

Este projeto descreve todo o processo de desenvolvimento (incluindo a avaliação) de uma aplicação para dispositivos móveis Android, para gestão de táticas de basquetebol.

Abstract

Basketball is a team sport recognized worldwide and practiced by millions of people, some for fun, others professionally, some by both methods. There is no correct way to play this sport, but there are techniques and formats that help players enhance their performance. Organizing movements and creating a tactic is one of these formats.

In recent times, there has been a great growth in the area of technology, which has also fostered an adaptation of the sport to it. The video was introduced, in order to repeat past events, with the aim of studying them to improve them later. Systems capable of simulating the tactics created by the coaches were also created, being this chapter the focus of this project.

It is intended to implement a system that makes the user capable of creating and managing his tactics, having in front of him an automatic adjustment of the defensive team. Consequently, the user can refine his tactical knowledge, practicing and testing his moves until the maximum efficiency and effectiveness of his play is strengthened.

This project describes the entire development process (including evaluation) of an application for Android mobile devices, for managing basketball tactics.

Agradecimentos

A realização deste projeto final de curso contou com um apoio e incentivo muito importante, aos quais me sinto muito grato.

Ao Professor Rui Jesus, pela sua orientação, disponibilidade, opiniões, críticas e conversas sobre desporto.

Aos meus amigos mais próximos, semanalmente interessados em saber o ponto de situação da implementação, pelas palavras de incentivo.

E à minha família (pai, mãe e irmã), um agradecimento muito especial pelo constante apoio, preocupação e por toda a confiança transmitida ao longo do meu percurso académico, ao qual estiveram sempre lá para mim.

Índice

Resumo	3
Abstract	5
Agradecimentos	7
Lista de Figuras	11
1 Introdução	13
2 Trabalho Relacionado	15
3 Modelo Proposto	19
3.1 Requisitos	19
3.1.1 Caracterização Geral	19
3.1.2 Caracterização Pormenorizada	21
3.1.3 Casos de Utilização	23
3.2 Fundamentos	25
3.2.1 Conceito de Frame	25
3.2.2 Steering Behaviors	25
3.2.3 Cálculo da posição de ajuda	27
3.3 Abordagem	32
3.3.1 Aplicação dos comportamentos “Steering”	32
3.3.2 Posição de ajuda	33
4 Implementação do Modelo	35
4.1 Tecnologias usadas	35
4.2 Firebase + LibGDX	36
4.3 Ecrãs	38
4.3.1 Classes que controlam os ecrãs	38
4.3.2 Ecrã de início de sessão	41
4.3.3 Ecrã de registo de utilizador	43
4.3.4 Ecrã de atualização da palavra-passe	45
4.3.5 Ecrã principal (campo)	46
4.4 Agentes	59
4.4.1 Aplicação dos comportamentos (“Steering”)	60
4.5 Posicionamento defensivo	64
4.5.1 Defesa individual	64
4.5.2 Defesa zona	67
4.6 Tática	71
4.7 Sistema de ficheiros	72
4.7.1 Classe ‘TacticFileHandle’	72

4.7.2 Classe 'SaveLoad'	74
5 Validação e Testes	77
5.1 Contexto	77
5.2 Participantes	78
5.3 Análise dos Resultados	79
6 Conclusões e Trabalho Futuro	81
Referências	83
Bibliografia	85
A UML	87

Lista de Figuras

1	Diagrama de Blocos da Aplicação	14
2	Aplicação Relacionada [1] - Basket coach board	15
3	Aplicação Relacionada [2] - Coach Tactic Board: Basketball	16
4	Aplicação Relacionada [3] - Basketball Tactic Board	17
5	Casos de Utilização	23
6	Triângulo formado pelos elementos	27
7	Interseção entre os três círculos	28
8	Centro de um círculo	29
9	Pontos de interseção	31
10	Ecrã de início de sessão	41
11	Ecrã de registo do utilizador	43
12	Ecrã de atualização de palavra-passe	45
13	Ecrã principal (campo)	46
14	Exemplo de dialog - Registo com sucesso	53
15	Menu principal	55
16	Menu de ajuda	55
17	Exemplos de jogadores	60
18	Zonas do campo para a defesa zona 2-3	68
19	Exemplo da defesa zona 2-3 (bola na zona A)	68
20	Exemplo da defesa zona 2-3 (bola na zona C)	69
21	Afiliação dos participantes ao basquetebol	78
22	Género dos participantes	78
23	Intervalo de idades dos participantes	79
24	Diagrama UML geral da aplicação	87
25	Diagrama UML do package 'com.gdx.tdl.map.aggs'	88
26	Diagrama UML do package 'com.gdx.tdl.map.scr'	89
27	Diagrama UML do package 'com.gdx.tdl.map.dlg'	90
28	Diagrama UML do package 'com.gdx.tdl.map.tct'	90
29	Diagrama UML do package 'com.gdx.tdl.sgn.scr'	91
30	Diagrama UML do package 'com.gdx.tdl.util.sgn'	91
31	Diagrama UML do package 'com.gdx.tdl.util.map'	92

1 Introdução

Como se pode ter a certeza que as ideias e os planos irão correr como se pretende? Colocando-as em prática. Mas nem sempre isso é feito, seja por medo de arriscar e falhar ou por não se ter os recursos para tal. Felizmente, nos dias de hoje, existe a possibilidade de realizar simulações de forma mais precisa, o que ajuda na colocação daquilo que se tem no papel em prática.

Veja-se um exemplo. A meio de um treino de basquetebol, um treinador dirige-se aos seus atletas da seguinte maneira: "Fui eu que criei esta jogada. Se seguirem os meus conselhos, vão, com toda a certeza, pontuar.". A sua certeza faz questionar. Será que já a colocou em prática? Ou apenas tem o desenho muito bem feito? Este projeto visa encontrar uma solução capaz de responder a ambas as questões.

Com a evolução dos dispositivos móveis do ponto de vista tecnológico e uma maior aceitação das pessoas na sua utilização em diversos contextos, as maiores dificuldades estão centradas no desenvolvimento de uma solução que proporcione uma experiência ao utilizador adequada ao contexto.

Este projeto descreve a aplicação "Tactics Don't Lie" para ajudar treinadores de basquetebol a definirem e apresentarem aos seus jogadores, as movimentações táticas que pretendem. Contudo, não tem um público alvo limitado e pode igualmente ser usada por outros intervenientes, como jogadores ou simpatizantes da modalidade.

A aplicação é desenvolvida para dispositivos móveis *Android*, em particular, para *tablets* que são os dispositivos mais adequados ao contexto. É também desenvolvida utilizando metodologias centradas nas características do utilizador, de modo a proporcionar uma experiência de utilização adequada ao contexto de um jogo ou treino de basquetebol.

Assim, os objetivos principais deste projeto são:

- Desenvolver uma aplicação para *Android* para gerir táticas;
- Introduzir técnicas de inteligência artificial de modo a simular a movimentação defensiva do adversário;
- Garantir eficiência e maximizar eficácia das jogadas, adaptando vários estilos de jogo de acordo com as práticas do opositor;
- Melhorar a experiência do utilizador;

Na figura 1 é apresentado o diagrama de blocos geral da aplicação proposta para cumprir os objetivos. Existe um bloco para a autenticação guardar informação relativa ao utilizador (endereço de e-mail e palavra-passe). Depois, três blocos para criar a tática através dos movimentos da equipa ofensiva, da escolha do tipo de defesa e da adaptação posicional da equipa defensiva. Finalmente, um bloco para reproduzir a tática na sua totalidade e outro para gravá-las.

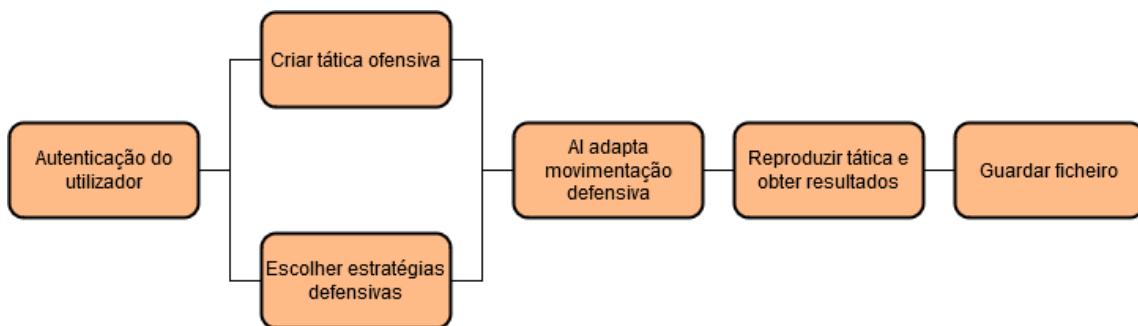


Figura 1: Diagrama de Blocos da Aplicação

Este relatório encontra-se dividido em várias secções, cada uma descriptiva de cada processo do desenvolvimento da aplicação. Primeiro, é apresentado o capítulo do trabalho relacionado. Depois, o modelo proposto para o desenvolvimento, seguido pelo capítulo da implementação do mesmo. Por fim, são apresentados os resultados dos testes de usabilidade realizados pelos utilizadores, assim como as conclusões retiradas acerca deste projeto e ideias para trabalho futuro.

2 Trabalho Relacionado

Existem no mercado algumas aplicações para gestão de táticas de basquetebol. Neste capítulo são apresentadas três dessas aplicações que mais se aproximam da aplicação proposta neste documento.

A *Basket coach board* [1] é uma aplicação com o objetivo de desenhar táticas manualmente. Contém um simples *design*, contendo o campo (que pode ser metade ou inteiro, conforme a preferência do utilizador) e uma barra de opções. Esta barra permite ao utilizador mudar a perspetiva do campo, a cor do marcador (entre azul e vermelho), apagar o esboço na totalidade ou apenas o último traçado realizado. Em geral, é dada muita liberdade ao utilizador, mas a aplicação acaba por não ser muito diferente de um desenho com caneta num papel. Por isso, esta aplicação é pouco versátil relativamente ao que um dispositivo móvel pode oferecer.

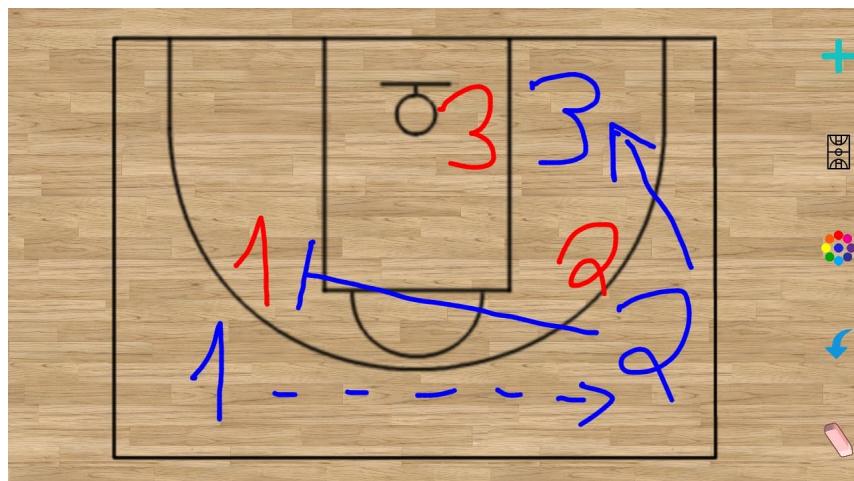


Figura 2: Aplicação Relacionada [1] - Basket coach board

A *Coach Tactic Board: Basketball* [2] contém características mais específicas e variadas que a aplicação anterior. Permite, através de agentes animados, gerir a sequência de uma tática, *frame a frame*. Em cada *frame*, é simulado o movimento de cada agente, partindo de uma posição inicial, terminando numa posição final. Para além do esboço de táticas de jogo, também podem ser feitos rascunhos de exercícios de treino, com vários objetos disponíveis. Também podem ser feitos esboços com linhas. É igualmente possível guardar as táticas e retomar as respetivas sequências mais tarde. Contudo, são apenas visíveis na própria aplicação.



Figura 3: Aplicação Relacionada [2] - Coach Tactic Board: Basketball

A *Basketball Tactic Board* [3] é uma aplicação semelhante à anterior, com a possibilidade de gerar uma táticas de forma sequencial, através de *frames*. Contém igualmente agentes animados, pode também realizar esboços com vários tipos de linhas, guardar as táticas no armazenamento interno e retomá-las mais tarde. Contudo, as táticas são apenas visíveis na aplicação e não tem opções próprias para planos de treino.

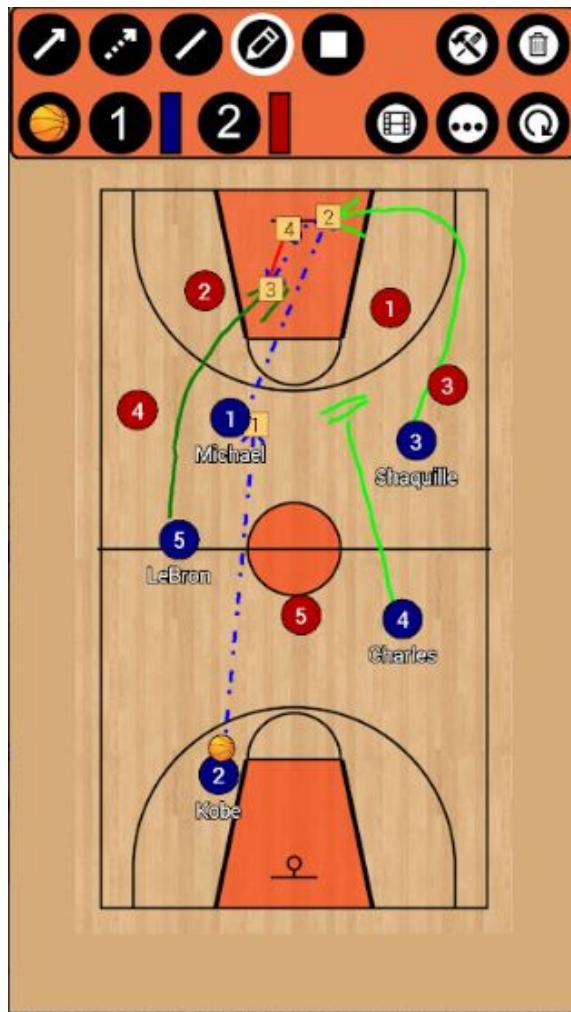


Figura 4: Aplicação Relacionada [3] - Basketball Tactic Board

Comparando o projeto desenvolvido com o trabalho relacionado apresentado acima, é possível destacar a melhoria na experiência do utilizador, tendo em conta a interface disponibilizada e os mecanismos de inteligência artificial, que ajudam no cumprimento dos objetivos traçados pelo próprio utilizador.

3 Modelo Proposto

O modelo proposto consiste num percurso descritor dos objetivos implementados no presente projeto. Este percurso inicia na análise dos requisitos (descrição das necessidades) e passa pelos fundamentos teóricos abordados e considerados fundamentais para a realização do trabalho, contendo depois uma explicação mais detalhada dos mesmos, de forma a que passe uma ideia clara e concisa dos conceitos teóricos aplicados.

3.1 Requisitos

Requisitos são descrições das necessidades ou propósitos de um produto. A sua análise é feita na fase inicial do projeto, de forma a ter uma melhor percepção do produto em si e do que se pretende implementar. São divididos em dois tipos de artefactos: uns de caracterização geral e outros de caracterização pormenorizada.

3.1.1 Caracterização Geral

Os requisitos de caracterização geral decompõem-se em três partes: os objetos principais do produto; os clientes aos quais o produto se destina; e as metas a cumprir pelo sistema.

a) Síntese de Objetivos

O principal objetivo do projeto é desenvolver uma aplicação Android que torna o utilizador capaz de criar e gerir táticas de basquetebol. O utilizador terá permissão para escolher os tipos de movimentos (correr ou passar a bola) que pretende para cada jogador da equipa atacante em cada parte da jogada. No que toca à equipa defensiva, esta terá um mecanismo que faz com que os jogadores se movimentem automaticamente, de acordo com o posicionamento dos jogadores da equipa atacante. Isto é, à medida que o utilizador modifica o comportamento dos jogadores da equipa atacante, a posição dos jogadores da equipa defensiva ajusta-se, de acordo com os princípios do tipo de defesa aplicada. Este tipo de defesa pode ser escolhido entre dois tipos: uma defesa do tipo individual (homem-a-homem) e uma defesa do tipo zona (ocupação do espaço).

Para além dos objetivos mencionados, está também planeada a existência de uma conta de utilizador que permite guardar as táticas criadas. Estas poderão, mais tarde, ser carregadas para a aplicação, de modo a que o utilizador as possa rever e reproduzir. Juntamente com as táticas existe um espaço dedicado a notas que o utilizador pretenda guardar acerca da tática.

b) Públco Alvo

Um primeiro raciocínio poderá levar a limitar o uso do produto apenas para profissionais da área, mais concretamente a equipa técnica (treinadores). E a verdade é que maioria dos utilizadores serão treinadores. Contudo, não são os únicos que estudam o jogo e que procuram melhorar certos aspetos táticos da sua equipa. Muitos jogadores não se cingem apenas a ouvir o que o treinador diz e muitos treinadores pretendem saber a opinião dos seus jogadores. Desta forma, o conhecimento do uso desta aplicação é aconselhado tanto aos treinadores, como aos jogadores.

Para além disso, alguém que não seja nenhum dos tipos mencionados anteriormente e que pretenda ganhar alguma noção tática básica (mais propriamente a movimentação e o posicionamento defensivo), poderá igualmente aprender através desta aplicação.

c) Metas a Alcançar

Pretende-se que este sistema garanta o maior nível possível de eficiência, de modo a maximizar a eficácia das jogadas. Ou seja, que se torne capaz de adaptar vários estilos de jogo, de acordo com as práticas aplicadas, sem qualquer tipo de dificuldade.

Ao mesmo tempo, exige-se um certo grau de facilidade na visualização de todos os agentes existentes e nos movimentos, de forma a potencializar a experiência do utilizador.

3.1.2 Caracterização Pormenorizada

Os requisitos de caracterização mais pormenorizada do sistema expressam-se através daquilo que é funcional (funções do sistema) ou não-funcional (atributos do sistema).

a) Funções do Sistema

As funções do sistema (requisitos funcionais) simbolizam as funcionalidades que o sistema contém, ou seja, aquilo que o sistema consegue e tem de fazer. Tendo isto em conta, uma função de um sistema terá de fazer sentido numa frase iniciada por "*O sistema tem que fazer*". Exemplificando: "*O sistema tem que fazer o registo do utilizador.*" ou "*O sistema tem que fazer a criação de uma tática.*".

Todas as funções têm um grau de importância no sistema. Com isto, de forma a definir a prioridade de cada uma dessas funções, existem três categorias disponíveis, apresentadas na tabela 1.

<i>Categoria</i>	<i>Descrição</i>
<i>Evidente</i>	A função tem de ser realizada e o utilizador tem de ter conhecimento da sua realização
<i>Invisível</i>	A função tem de ser realizada, mas não é visível para o utilizador
<i>Adorno</i>	Função opcional, não afetando significativamente o custo ou outras funções

Tabela 1: Categorias das Funções do Sistema

Tendo isto em consideração, foram contabilizadas e classificadas as funções principais do sistema. Começando pela autenticação do utilizador, passando por todas as funcionalidades existentes que permitem criar, gerir e reproduzir uma tática, até tirar algumas notas e guardá-la no seu espaço no sistema de armazenamento. Estas funções são apresentadas na tabela 2.

<i>Referência</i>	<i>Função</i>	<i>Categoria</i>
R1.1	Registo do utilizador	Evidente
R1.2	Autenticação do utilizador	Invisível
R1.3	Menu principal e menu de ajuda	Evidente
R1.4	Escolher equipa cujas definições quer alterar	Evidente
R1.5	Escolher posição inicial dos atacantes	Evidente
R1.6	Definir ação do jogador atacante	Evidente
R1.7	Definir tipo de defesa	Evidente
R1.8	Posicionamento automático dos defesas	Invisível
R1.9	Guardar <i>frames</i> da tática	Evidente
R1.10	Reproduzir a tática	Evidente
R1.11	Limpar a tática atual	Evidente
R1.12	Guardar a tática	Evidente
R1.13	Carregar uma tática	Evidente
R1.14	Bloco de notas	Evidente

Tabela 2: Funções do Sistema

b) Atributos do Sistema

Os atributos do sistema (requisitos não-funcionais) são as suas características (e não funções). Em relação a este projeto, este é definido principalmente pelo desenvolvimento de uma aplicação Android para gerir táticas. Anexada a esta característica vem a distinta introdução a um mecanismo de inteligência artificial capaz de simular a movimentação e o posicionamento da equipa defensiva. Para além disto, outro requisito está na melhoria da experiência do utilizador, proporcionando um bom uso da aplicação. A tabela 3 apresenta este tipo de requisitos.

<i>Referência</i>	<i>Função</i>	<i>Categoria</i>
Plataforma	Android	Obrigatório
Simplicidade de Utilização	Várias opções simples disponíveis	Obrigatório
Interação Pessoa-Máquina	Melhorar a experiência do utilizador	Obrigatório
Tolerância a Falhas	Maximizar a eficácia dos movimentos defensivos	Desejável
Tempo de Resposta	Demonstração rápida e eficiente da jogada	Desejável

Tabela 3: Atributos do Sistema

3.1.3 Casos de Utilização

Os casos de utilização descrevem uma sequência de eventos e de acontecimentos desencadeados ao longo do uso do sistema. Aqui, são destacados os atores que são os estimuladores dessas sequências, sendo que são eles quem as inicia e/ou participam.

a) Atores

Os atores são aqueles que intervêm no sistema, ou seja, quando são necessários para que certa funcionalidade se realize. No caso deste sistema, existem três atores: o "Utilizador", a "Aplicação" e a "Base de Dados". Na figura seguinte, encontra-se o esquema que contém o ambiente do sistema, contendo todos os atores e responsabilidades existentes, estando incluídas as dependências entre o ambiente externo (atores) e interno (funcionalidades).

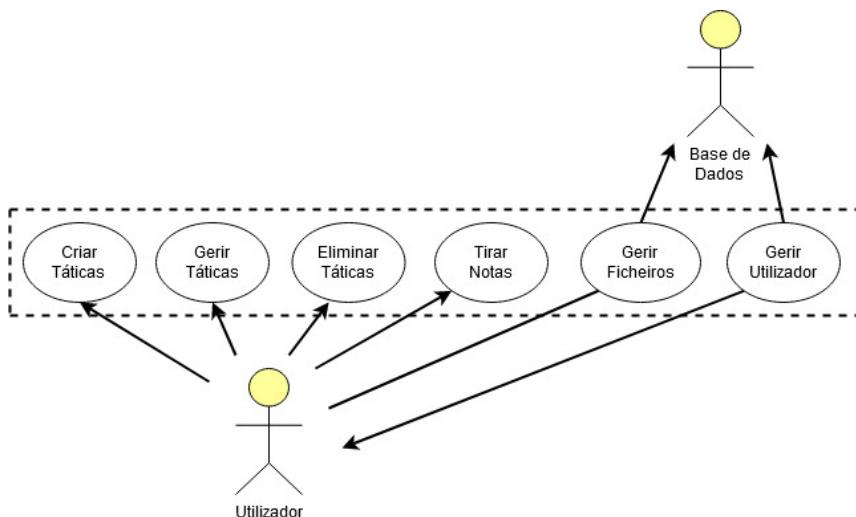


Figura 5: Casos de Utilização

Como se pode observar na figura acima, qualquer responsabilidade relacionada com a criação, gerência, reprodução ou eliminação da tática, assim como a escrita de notas, está apenas dependente das ações do utilizador. Este também terá influência na gerência de ficheiros e da sua conta de utilizador, mas estes eventos encontram-se também ligados ao sistema de armazenamento, que controla e guarda a informação relacionada com as funcionalidades referidas. No que toca à movimentação defensiva, a própria aplicação adapta-a ao longo do desenvolvimento da jogada.

b) Descrição dos Casos de Utilização

Tendo em conta que os casos de utilização descrevem uma sequência de eventos, foi possível fazê-lo através da tabela 4 apresentada abaixo.

Cenário Principal			
Ação do Ator		Resposta do Sistema	
1	Utilizador identifica-se	2	Sistema de autenticação valida credenciais do utilizador
3	Utilizador define a posição inicial dos jogadores		
4	Utilizador define tipo de defesa		
5	Utilizador define movimentos ofensivos		
6	Utilizador guarda frames		
7	Utilizador indica que pretende reproduzir a jogada	8	Aplicação reproduz a jogada no ecrã
9	Utilizador pretende tirar notas	10	Aplicação mostra no ecrã um menu com informação acerca das funcionalidades
11	Utilizador pretende obter ajuda	12	Aplicação mostra no ecrã um menu com informação acerca do uso da mesma
13	Utilizador pretende guardar a tática	14	Aplicação guarda um ficheiro com a tática

Tabela 4: Descrição dos Casos de Utilização

Ao executar a aplicação pela primeira vez, o utilizador depara-se com um ecrã destinado a realizar o início de uma sessão. Caso ainda não se tenha registado com uma conta de utilizador, contém uma opção nesse ecrã para tal. Após registo, pode realizar a autenticação, introduzindo as suas credenciais. O sistema irá aceder ao sistema de autenticação e verificar se as credenciais introduzidas são as corretas e, se for o caso, o utilizador é redirecionado para uma página principal. O utilizador pode escolher entre criar uma nova tática ou carregar uma já existente. Ao entrar no ecrã que contém maioria das funcionalidades, pode começar a criar a tática.

Inicialmente terá de definir as posições iniciais dos jogadores da equipa atacante (através do movimento da corrida). Assim que estiver pronto para dar início, os jogadores da equipa defensiva começam a adaptar a sua posição conforme o posicionamento dos jogadores da equipa atacante. De seguida, pode definir outros movimentos (corrida e passe) e o igualmente o tipo de defesa (individual ou zona). À medida que o utilizador adiciona movimentos, adiciona também as "frames" que dividem a tática. Por fim, quando o utilizador pretender reproduzir a tática, de forma a ver o resultado final, a aplicação responderá com a apresentação da jogada do início ao fim. Existe igualmente um espaço próprio para notas acerca da tática atual, caso o utilizador o queria fazer. Se tiver alguma dúvida, o utilizador pode igualmente obtê-la no menu dedicado a tal. Finalmente, assim que desejar guardar a tática, basta colocar o seu nome (ou deixar o pré-definido) e aplicação fá-lo no seu espaço interno e, se houver conexão com a *internet*, guarda igualmente no sistema de armazenamento.

3.2 Fundamentos

Nesta secção de fundamentos, são apresentados os conceitos teóricos usados neste projeto e fundamentais para o funcionamento do mesmo.

3.2.1 Conceito de Frame

No desenvolvimento de uma tática, os movimentos terão de ser organizados, pois um jogador pode realizar vários ao longo do tempo. De forma a dividir a tática, são criadas "frames". Uma frame contém no máximo um movimento iniciado pelo jogador, isto é, uma corrida ou um passe para um colega. Receber um passe não conta, visto que pode iniciar um movimento de corrida e, ao mesmo tempo, receber um passe. O conjunto formado por estas "frames" irá resultar numa tática sequencialmente organizada.

3.2.2 Steering Behaviors

Os comportamentos "Steering", como é explicado na documentação do "libgdx-ai"^[4], permitem que agentes naveguem no seu ambiente através de estratégias. Estas incluem a perseguição de um alvo, o escapar de um predador, vaguear, seguir um caminho pré-definido, entre muitos outros. No caso deste projeto, é necessária a utilização de dois tipos de comportamentos "Steering": o "Arrive" (chegada a um destino) e o "Collision Avoidance" (evitar colisões).

Teoricamente, todos os jogadores são considerados como “agentes” de duas dimensões. De modo a aplicar os comportamentos, estes terão de conter certas características que influenciam os resultados de cada movimento. Começando pela definição da sua forma, possuí uma área calculada através do raio, no caso de um círculo. Terá sempre uma posição, representada através de um vetor de duas dimensões (X e Y). Por fim, de forma a realizar certos movimentos, deverá conter valores de aceleração e velocidade, sejam estas lineares ou angulares. A velocidade linear é fruto da relação entre a variação da posição e a variação do tempo, enquanto que a velocidade angular expressa o valor da medida do arco de circunferência descreto pelo agente, dentro de um intervalo de tempo.

Os comportamentos aplicados nos agentes do presente projeto foram, como dito anteriormente, o “*Arrive*” (chegar a um certo destino) e o “*Collision Avoidance*” (evitar colisões entre agentes), sendo explicados nos parágrafos seguintes.

a) **Arrive**

A função deste comportamento está em mover o agente da sua posição atual até à posição do seu alvo (“*target*”). Contudo, com a condição de chegar ao objetivo com uma velocidade nula.

Para que isto seja possível, é necessário definir dois tipos de raio: um raio de desaceleração (mais largo) e um raio de tolerância à chegada (mais perto do objetivo). O primeiro especifica quando é que o agente deve começar a desacelerar, enquanto que o segundo permite que o agente se aproxime do seu alvo sem o entrave de pequenos erros. O algoritmo calcula a velocidade ideal do agente entre o raio de desaceleração e a posição do alvo, de modo a dar um ar mais realista à “travagem”. Isto, tendo em conta que a velocidade no ponto do raio de desaceleração é máxima e no objetivo é zero.

A direção do agente é calculada através da velocidade desejada, de modo a criar uma velocidade para o “*target*”. Para isto, o algoritmo verifica qual a velocidade atual do agente e adapta a aceleração de modo a transformar a velocidade na velocidade do “*target*”.

b) **Collision Avoidance**

Visto que existem muitos cruzamentos entre os agentes à medida que são feitos movimentos, a probabilidade de chocarem entre eles é muito alta. De forma a evitar essas colisões, é usada esta função que apenas é aplicada em agentes cujos obstáculos contêm uma forma que se aproxime de um círculo.

Esta implementação prevê a colisão através das distâncias e das velocidades dos agentes. Para evitar tal situação, verifica em que ponto é que a sua distância é menor que a soma dos seus raios. Assim, no momento em que verifica a aproximação, os agentes afastam-se ligeiramente um do outro, evitando a colisão.

3.2.3 Cálculo da posição de ajuda

Um jogador em posição de ajuda precisa de ter em atenção três elementos: o jogador que está a defender, o jogador com a bola e o cesto. Em todas as situações, o defesa terá de se encontrar entre os três. O cesto estará sempre nas costas do defesa. A sua posição dependerá das posições dos outros dois elementos.

a) Posição baseada nos ângulos entre três pontos

Assumindo que são conhecidas as coordenadas dos vetores correspondentes aos três pontos fundamentais (jogador com bola, jogador atribuído ao defesa e cesto), é possível formar um triângulo entre todos. A posição do defesa terá de se encontrar dentro desse triângulo, de forma a ter uma visão sobre todos.

Como se pode observar pela figura 3.2.3, retirada do website "math.stackexchange.com" [5], 'A', 'B' e 'C' exemplificam os três pontos conhecidos, sendo 'O' o ponto que queremos calcular. Este último cria três triângulos interiores, fazendo um ângulo θ dentro de cada um deles. Sabe-se que a soma de todos os ângulos θ resulta no valor de 180° ou 2π ($\theta_{AB} + \theta_{BC} + \theta_{CA} = 2\pi$, assumindo que todos são maiores que zero).

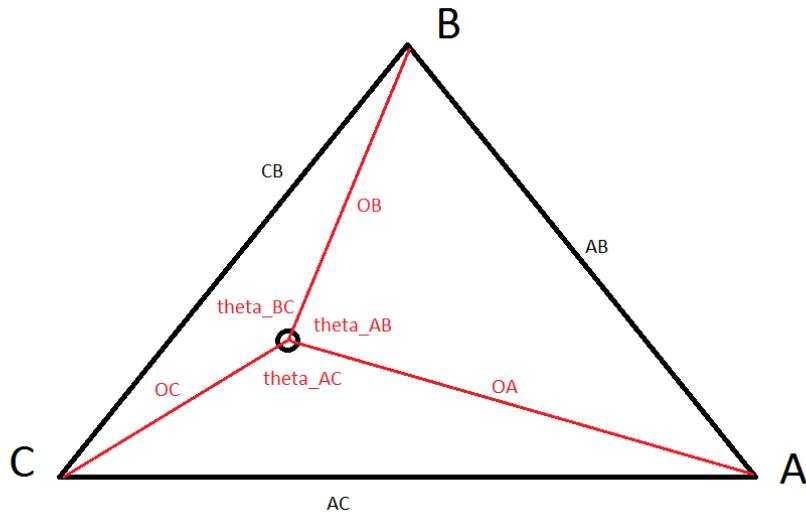


Figura 6: Triângulo formado pelos elementos

Após pesquisa sobre como descobrir as coordenadas do ponto ‘ O ’, escolheu-se a interseção de círculos que resultam num só ponto, como é explicado no website ”math.stackexchange.com”[5], sendo este ponto o nosso alvo. Cada um dos círculos contém dois dos três pontos conhecidos, totalizando três círculos. A figura 3.2.3 exemplifica um caso possível.

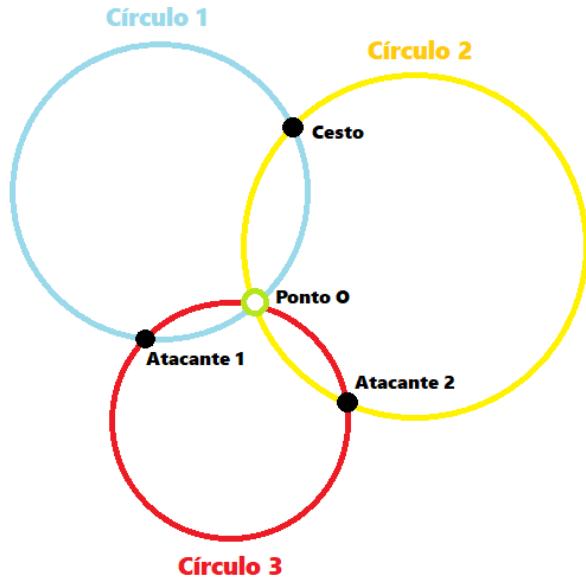


Figura 7: Interseção entre os três círculos

Com a informação apreendida, foram definidos os próximos passos. Primeiro, é necessário calcular as equações dos círculos. Com estas, pode-se calcular o centro do círculo, o seu raio e, consequentemente, a distância para qualquer ponto contido no círculo. Ao intersetar dois círculos, obtém-se dois pontos de interseção, um deles é um ponto já conhecido e o outro é o que se pretende conhecer. Conhecendo os raios e os centros de ambos os círculos, pode-se formar um triângulo com os raios e a reta formada pelos centros. Após cálculos (detalhados na secção (c) do presente capítulo), é possível calcular o ponto pretendido.

b) Desenho dos círculos que se intersetam no ponto pretendido

Tendo em conta que é necessário calcular coordenadas de pontos, raios e distâncias através de círculos, é necessário o conhecimento da equação que os determina. Por norma, um círculo com raio r e centro (X_0, Y_0) contém a seguinte equação,

$$(X - X_0)^2 + (Y - Y_0)^2 = r^2 \quad (1)$$

De forma a tornar a função completa para este caso, para se poder intersetar os círculos, terão de se descobrir os valores do raio e das coordenadas do centro, ambos possíveis graças ao conhecimento de dois pontos do círculo e dos ângulos formados entre o ponto a calcular e os pontos conhecidos.

Observando a figura em baixo, também retirada do website "math.stackexchange.com" [5], verifica-se que o triângulo ABO é isóscele, que o ponto M_{AB} é a projeção ortogonal do centro do círculo na linha AB (sendo igualmente o ponto central da mesma) e o ângulo $\angle O_{AB}AB = \theta_{AB} - \frac{\pi}{2}$.

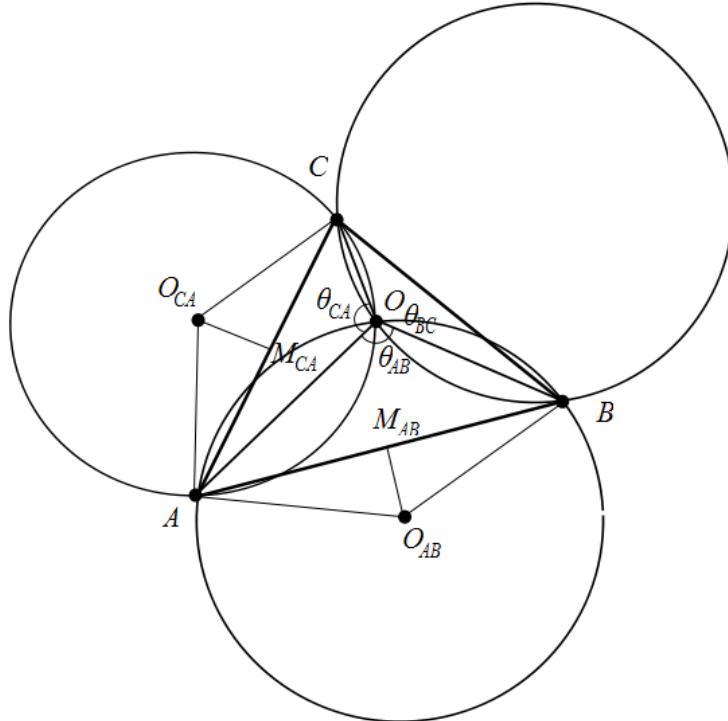


Figura 8: Centro de um círculo

Juntando a informação anterior e retirando a explicação feita no website [5], é possível construir e simplificar a seguinte expressão,

$$M_{AB} \cdot O_{AB} = \frac{1}{2} \cdot A \cdot B \cdot \tan(\theta AB - \frac{\pi}{2})$$

$$A \cdot O_{AB} = \frac{AB}{2 \cdot \cos(\theta AB - \frac{\pi}{2})}$$

Desta forma, consegue-se saber as coordenadas do centro do círculo e, consequentemente, através da equação 1, o seu raio.

Nota: Todos os ângulos entre jogadores são pré-definidos, conforme a sua posição no campo. Dividindo este último em várias áreas, onde os jogadores se encontram colocados, é possível definir os ângulos entre eles, visto o jogador já ter uma ideia pré-concebida da situação.

c) Interseção entre círculos

Tendo já obtido os raios e os centros dos círculos, é possível calcular a interseção entre eles. Estudando a informação obtida no website "paulbourke.net" [6], as intersecções terão de ser calculadas aos pares de círculos, para que depois sejam comparados os resultados e verificar o ponto em comum entre todos. Apenas é necessário a interseção de dois círculos, mas por uma questão de confirmação, são calculadas todas as intersecções. Como se pode observar pela figura 9, temos conhecimento do valor de $P0$, $P1$, $r0$ e $r1$. O objetivo será encontrar as coordenadas de ambos os pontos $P3$, visto que a interseção de dois círculos resulta em dois pontos.

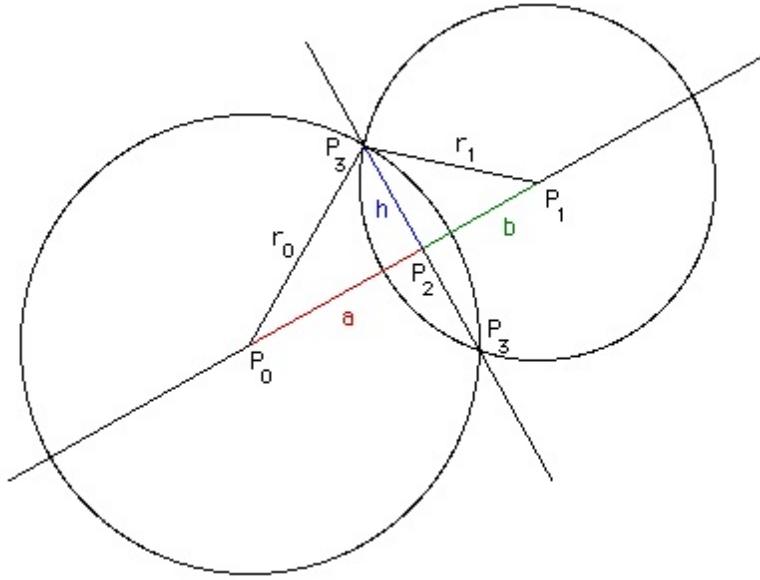


Figura 9: Pontos de interseção

Primeiro é calculada a distância entre os centros de ambos os círculos. Isto é possível através da seguinte equação,

$$a = \frac{r_1^2 - r_2^2 + distancia^2}{2 \cdot distancia}$$

Através do valor de ‘a’, são calculadas as coordenadas do ponto $P2$.

$$P2(x) = P0(x) + ((P1(x) - P0(x)) \cdot \frac{a}{distancia})$$

$$P2(y) = P0(y) + ((P1(y) - P0(y)) \cdot \frac{a}{distancia})$$

Com ambos estes valores e sabendo o raio $r0$, é possível saber a distância do ponto $P2$ para os pontos de interseção, através da regra de três simples. Da mesma forma que foi calculado o ponto $P2$, são agora calculadas as coordenadas de ambos os pontos de interseção:

$$(1) P3(x) = P2(x) - ((P1(y) - P0(y)) \cdot \frac{h}{distancia})$$

$$(1) P3(y) = P2(y) + ((P1(x) - P0(x)) \cdot \frac{h}{distancia})$$

$$(2) P3(x) = P2(x) + ((P1(y) - P0(y)) \cdot \frac{h}{distancia})$$

$$(2) P3(y) = P2(y) - ((P1(x) - P0(x)) \cdot \frac{h}{distancia})$$

3.3 Abordagem

Neste capítulo, são explicadas as aplicações feitas da teoria e a influência que esta tem na implementação deste projeto.

3.3.1 Aplicação dos comportamentos “Steering”

Tendo em conta que se pretende mover os jogadores entre duas posições, através das situações de corrida e de passe, e mover a bola entre jogadores nesta última, terão de ser aplicados os comportamentos denominados “*Steering*”, explicados na secção 3.2.2. No caso deste projeto, como foi dito, é necessária a utilização de dois tipos de comportamentos “*Steering*”: o “*Arrive*” e o “*Collision Avoidance*”.

O comportamento ”*Arrive*” é aplicado em duas situações: na corrida do jogador e no movimento da bola através de uma mudança na posse da bola (entre colegas de equipa).

No primeiro caso, são necessários dois toques no ecrã: o primeiro na posição de um jogador, definindo o agente a que será aplicado o movimento e o segundo na posição (espaço vazio) do campo que será definida como o seu ”*target*”. Este ”*target*” será também um agente invisível, de forma a que se tenha conhecimento da sua posição. As colisões no ambiente deste ”*target*” são inexistentes, de modo a não ter qualquer tipo de influência extra.

No segundo caso, tendo em conta que a bola segue sempre o seu próprio ”*target*” (jogador com a sua posse), não será necessário criar nenhum agente invisível. Apenas é preciso escolher um novo target, dentro dos existentes. Com isto, assim que é feita a escolha do passe, a bola realiza o comportamento na direção do seu novo ”dono”.

O comportamento ”*Collision Avoidance*” é aplicado nos dez jogadores, apenas entre eles (a bola não tem qualquer influência, pois assume-se que a defesa se encontra num modo passivo, ou seja, sem intenção de a roubar). Visto que cada um tem o seu ”círculo de proteção” à sua volta, assim que alguém se aproxima do mesmo, o agente alerta-se e tenta esquivar-se, tentando evitar uma colisão. Contudo, devido a certas velocidades e movimentos de alguns jogadores, a colisão por vezes acontece, algo que no jogo real acontece com frequência.

3.3.2 Posição de ajuda

No basquetebol, a defesa individual (chamada também “homem-a-homem”) é a defesa onde é atribuída a responsabilidade a cada elemento da equipa defender um certo adversário. Para quem está a defender o jogador com bola, é feita uma maior pressão sobre o jogador, enquanto que os restantes se encontram numa posição um pouco mais “descansada”, apesar de ser necessário um foco adicional. Isto porque é necessário tomar atenção a mais elementos. Enquanto que o jogador que pressiona o adversário com bola se preocupa apenas com este e com o cesto, os restantes terão de se preocupar com a sua marcação, com o cesto e com o jogador com bola. A vigilância em relação a este último serve para quando o colega é “batido” pelo adversário, passando a ser necessária uma ajuda. Daí a origem dos restantes jogadores se encontrarem em “posição de ajuda”.

Como dito anteriormente, um jogador em posição de ajuda precisa de ter em atenção três elementos: o seu jogador, o jogador com a bola e o cesto. De forma a realizar o cálculo desta posição, assume-se que o jogador terá de se encontrar entre os três elementos. Esse ponto será um agente invisível, tal como falado na secção anterior 3.3.1 e será um dos pontos de intersecção entre os pares de círculos criados entre os três elementos. O ponto escolhido será o ponto em comum entre todas as intersecções entre os círculos.

O movimento do defesa será igual ao da corrida de um atacante, sendo aplicado o comportamento ”*Arrive*”, com um ”*target*” definido como um agente invisível e sem qualquer tipo de influência extra.

4 Implementação do Modelo

Ao longo deste capítulo é detalhada a implementação do sistema proposto e quais as decisões tomadas de forma a obter um desempenho eficiente e eficaz. O capítulo está dividido nas secções que são consideradas mais importantes para o funcionamento principal do sistema, tais como o sistema de autenticação e armazenamento, os ecrãs, a criação dos jogadores e respetivos movimentos, o posicionamento da defesa e o sistema de ficheiros.

4.1 Tecnologias usadas

Neste projeto, de forma a implementar a aplicação proposta, é utilizado um conjunto de tecnologias. Do lado do cliente, foi escolhida a linguagem de programação *Java* [7] para o desenvolvimento da aplicação. Tendo em conta que será uma aplicação *Android* [8], recorreu-se ao uso da ferramenta *Android Studio* [9]. Visto que a aplicação contém partes que se podem relacionar com um jogo, como agentes que se movem, optou-se pela utilização da *framework* "LibGDX" [10], que se trata de uma *framework* de desenvolvimento de jogos em *Java*.

Uma vez que se irá desenvolver um sistema de ficheiros próprio para o utilizador, deve-se seleccionar uma ferramenta capaz de autenticar o utilizador e de armazenar os seus ficheiros. Para isto, é usado um serviço externo da *Google*, mais propriamente o *Firebase UI* [11]. Deste, são usadas as funcionalidades *Firebase Authentication* para a autenticação e *Firebase Storage* para o armazenamento. Contudo, não é possível usar diretamente esta ferramenta com o *framework* do *LibGDX*. Ou seja, teve de se usar uma *API* capaz de integrar a utilização de ambos, denominada de *gdx-firebase* [12].

4.2 Firebase + LibGDX

De forma a permitir a utilização do *Firebase* neste projeto, teve de se percorrer alguns passos. Inicialmente, regista-se a aplicação através do nome do pacote raiz do projeto que, neste caso, é ”*com.gdx.tdl*”. Após registo, é gerado um ficheiro de configuração do tipo *JSON* com o nome ”*google-services*”. Este ficheiro tem de se encontrar na pasta ”*android*” do projeto do *Android Studio*. Fica apenas a faltar a adição das dependências ao ficheiro ”*gradle*” (que especifica os recursos usados na criação da aplicação). Estas dependências fazem referência às funcionalidades usadas por parte do *Firebase*, sendo elas a autenticação e o armazenamento. Para além disto, é igualmente aplicado o ”*plug-in*” do ficheiro de configuração gerado. Isto tudo será feito no ficheiro ”*gradle*” contido na pasta ”*android*”, como se pode observar no código em baixo.

Listing 1: Dependências - autenticação e armazenamento

Ficheiro *build.gradle* (Android):

```
apply plugin: 'com.android.application'

dependencies {
    implementation 'com.google.firebaseio:firebase-auth:19.3.2'
    implementation
        'com.google.firebaseio:firebase-storage:19.1.1'
}

apply plugin: 'com.google.gms.google-services'
```

Se isto fosse uma aplicação *Android* comum, já seria possível utilizar os recursos disponibilizados pelo serviço externo. Visto que nos encontramos num projeto ”*LibGDX*”, terá ainda de ser adicionado uma ferramenta que conecta ambas as plataformas - *gdx-firebase* [12]. Para isto, basta adicionar uma nova dependência às já existentes, tanto no ficheiro *gradle* falado anteriormente, como no ficheiro *gradle* principal da aplicação. As adições foram feitas da forma como é apresentado o código em baixo.

Listing 2: Dependências - gdx-firebase

Ficheiro build.gradle (Core):

```
dependencies {  
    implementation  
        'pl.mk5.gdx-firebase:gdx-firebase-android:2.1.7'  
}
```

Ficheiro build.gradle (Android):

```
dependencies {  
    implementation 'pl.mk5.gdx-firebase:gdx-firebase-core:2.1.7'  
}
```

Assim, já se torna possível realizar um registo do utilizador, iniciar ou terminar a sua sessão, guardar ou fazer o download de ficheiros, entre outros. Estas ações são implementadas seguindo os exemplos [13] fornecidos pelo autor desta ferramenta. Mais à frente são explicadas estas implementações.

4.3 Ecrãs

Nesta secção é apresentada a implementação dos ecrãs resultantes da fase *design*. Nesta fase, os diversos *layouts* foram concebidos utilizando a ferramenta *Scene2D* e seguindo um conjunto de heurísticas de usabilidade.

4.3.1 Classes que controlam os ecrãs

Visto que as classes dos vários ecrãs contêm coisas em comum e estendem e implementam os mesmos objetos, foram criadas classes que as controlam. Estas são duas, ambas são abstratas e são implementações de ”*Screen*”. Visto que apenas a classe do campo principal não usa um ”*Stage*”, a classe que as restantes estendem, irá igualmente estender de ”*Stage*”.

No que toca à classe que controla os ecrãs desenhados através de um ”*Stage*”, foi criada a ”*AbstractStage*”. Como dito anteriormente, esta estende de ”*Stage*” e implementa a interface ”*Screen*”. Contém o método abstrato ”*buildStage()*” que será implementado por cada um dos ecrãs, sendo lá feito o desenho da mesma. ”*render()*” será o método da interface ”*Screen*” que renderiza o ecrã, enquanto que o método ”*show()*” é usado para definir o *input* do utilizador no ecrã. Em baixo encontra-se o excerto de código referido.

Listing 3: Excerto da classe ”*AbstractStage*”

```
public abstract class AbstractStage extends Stage implements
    Screen {
    public abstract void buildStage();

    @Override
    public void render(float delta) {
        super.act(delta);
        super.draw();
    }

    @Override
    public void show() {
        Gdx.input.setInputProcessor(this);
    }
}
```

Relativamente à segunda classe, ”*AbstractScreen*”, foi criada para ecrãs como o do campo principal que não é usado um ”*Stage*” para a sua criação. Contudo, apenas foi necessário para esse mesmo ecrã. Aqui, é criado uma instância de ”*World*” que serve para gerir as entidades físicas e *queries* assíncronas existentes. Contém também o método abstrato ”*buildScreen()*”, implementado pelo ecrã do campo principal para construí-lo e a função ”*render()*” com o mesmo objetivo da classe ”*AbstractStage*”. A função ”*resize()*” irá adaptar as dimensões do ecrã às dimensões do dispositivo físico. Em baixo encontra-se o excerto de código referido.

Listing 4: Excerto da classe ”*AbstractScreen*”

```
public abstract class AbstractScreen implements Screen {  
    public World world;  
  
    public AbstractScreen() {  
        this.world = new World(new Vector2(0, 0), true);  
    }  
  
    public abstract void buildScreen(float delta);  
  
    @Override  
    public void render(float delta) {  
        world.step(1/60f, 6, 2);  
        buildScreen(delta);  
    }  
  
    @Override  
    public void resize(int width, int height) {  
        this.orthCamera.viewportWidth = width;  
        this.orthCamera.viewportHeight = height;  
    }  
}
```

De forma a controlar a troca entre os ecrãs e não perder o estado das mesmas, foi criado um ”*singletons*”. Um ”*singleton*” garante a existência de apenas uma instância de uma classe, mantendo um acesso global ao seu objeto. Assim, todos os ecrãs podem aceder a esta classe e trocar para qualquer outro ecrã, mesmo que não seja do mesmo tipo.

Este ”*singleton*” é inicializado na classe *main* do projeto, através da inicialização do objeto ”*Game*” (este permite a utilização de vários ecrãs). Contém o método ”*showScreen()*”, usado para a referida troca de ecrãs. Para isto, existe um enumerado com uma instância de cada um dos ecrãs existentes. Visto que o ecrã do campo principal (”*AbstractScreen*”) é de um tipo diferente dos restantes ecrãs (”*AbstractStage*”), é verificado o ecrã para o qual se pretende alterar, de forma a saber se é necessário invocar, ou não, o método ”*buildStage()*” dos ecrãs que estendem de ”*AbstractStage*”. É feito um ”*setScreen()*” de forma a mudar o ecrã atual para o pretendido. Em baixo encontra-se o método referido.

Listing 5: Método ”*showScreen()*”da classe ”*AbstractStage*”

```
public void showScreen(ScreenEnum screenEnum, Object... params) {  
    Screen currentScreen = game.getScreen();  
  
    if (screenEnum == ScreenEnum.COURT) {  
        AbstractScreen newScreen = screenEnum.getScreen(params);  
        game.setScreen(newScreen);  
    } else {  
        AbstractStage newStage = screenEnum.getStage(params);  
        newStage.buildStage();  
        game.setScreen(newStage);  
    }  
  
    if (currentScreen != null) currentScreen.dispose();  
}
```

4.3.2 Ecrã de início de sessão

Como dito anteriormente, o ecrã de início de sessão é um dos que estende de ”*AbstractStage*”, visto que usa o ”*Stage*” para a sua construção. Um ”*Stage*” é uma cena de duas dimensões que contém atores e distribui eventos (pode ser comparado a uma cena de um teatro, com os respetivos atores a desempenhar as suas funções). No caso deste ecrã, existe uma imagem de fundo, duas caixas de texto e dois botões, como se pode observar na figura

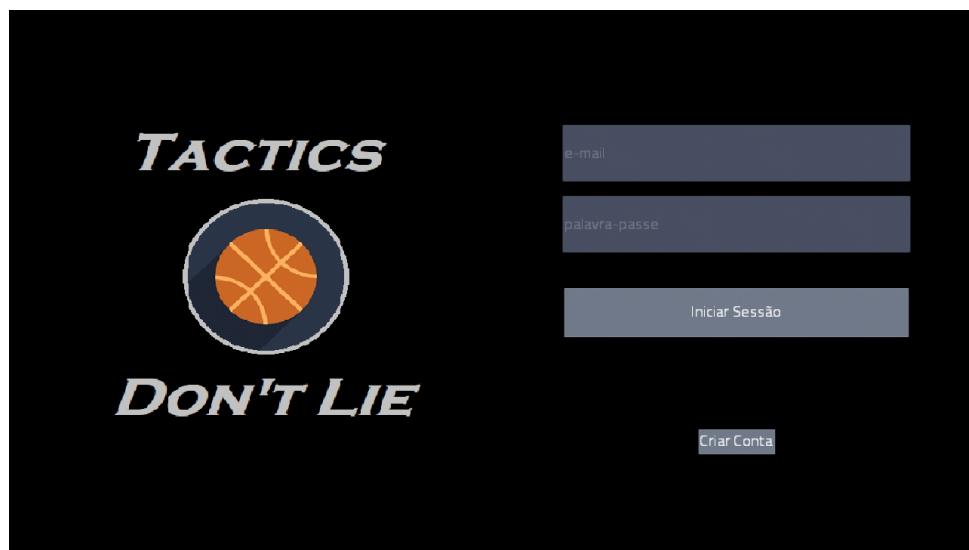


Figura 10: Ecrã de início de sessão

Toda a construção da página foi feita dentro do método ”*buildStage()*”. São criados os elementos e adicionados a uma tabela, sendo esta adicionada ao ”*stage*” como ator, tal como a imagem de fundo, como se pode observar nos excertos da função apresentados em baixo.

Listing 6: Excertos da função ”buildStage()”da classe ”SignInScreen”

```
@Override  
public void buildStage() {  
    Image bg = new Image(AssetLoader.I_login);  
    bg.setHeight(Gdx.graphics.getHeight());  
    bg.setWidth(Gdx.graphics.getWidth());  
    bg.setPosition(0, 0);  
    addActor(bg);  
  
    Table loginTable = new Table(AssetLoader.skin);  
    loginTable.setPosition(Gdx.graphics.getWidth()*4/7f,  
                           Gdx.graphics.getHeight()/8f);  
    loginTable.setSize(Gdx.graphics.getWidth()/2.75f,  
                      Gdx.graphics.getHeight()/1.5f);  
    addActor(loginTable);  
  
    final TextField mailTF = new TextField("", AssetLoader.skin,  
                                         "default");  
    mailTF.setMessageText("e-mail");  
    loginTable.add(mailTF).expand().fill()  
        .padBottom(Gdx.graphics.getHeight()/50f);  
    loginTable.row();  
  
    TextButton signUpTB = new TextButton("Iniciar Sessao",  
                                         AssetLoader.skin, "default");  
    loginTable.add(signUpTB).expand().fill()  
        .padBottom(Gdx.graphics.getHeight()/7.5f);  
    loginTable.row();  
}
```

No que toca à funcionalidade dos botões, foram adicionados ”*listeners*” a cada um. Ao botão de registo foi apenas invocado um método auxiliar que retorna um ”*InputListener*” que, ao toque do utilizador, altera o ecrã para o pretendido (neste caso, é o ecrã de registo).

Para o botão de iniciar a sessão, é necessária a utilização da instância (inicializada na classe *main* do projeto) do *gdx-firebase* [12]. Aqui, foi usada a função que permite conectar ao *firebase* e verificar se as credenciais introduzidas pelo utilizador são as corretas (”*signInWithEmailAndPassword()*”). Se assim for, sinaliza que pode iniciar a sessão e alternar para o ecrã do campo principal. Caso contrário, avisa-o do sucedido. O excerto de código descrito encontra-se em baixo.

4.3.3 Ecrã de registo de utilizador

Para o ecrã de registo do utilizador é utilizado o mesmo formato que o ecrã de início de sessão (4.3.2). Isto é, estende de ”*AbstractStage*” e implementa o método ”*buildStage()*”. Este contém os mesmos campos de texto para as credenciais e um botão que realiza o registo do utilizador, através da instância de ”*GdxFIRAuth*”, invocando a função ”*createUserWithEmailAndPassword()*”. O ecrã descrito encontra-se na figura 11.

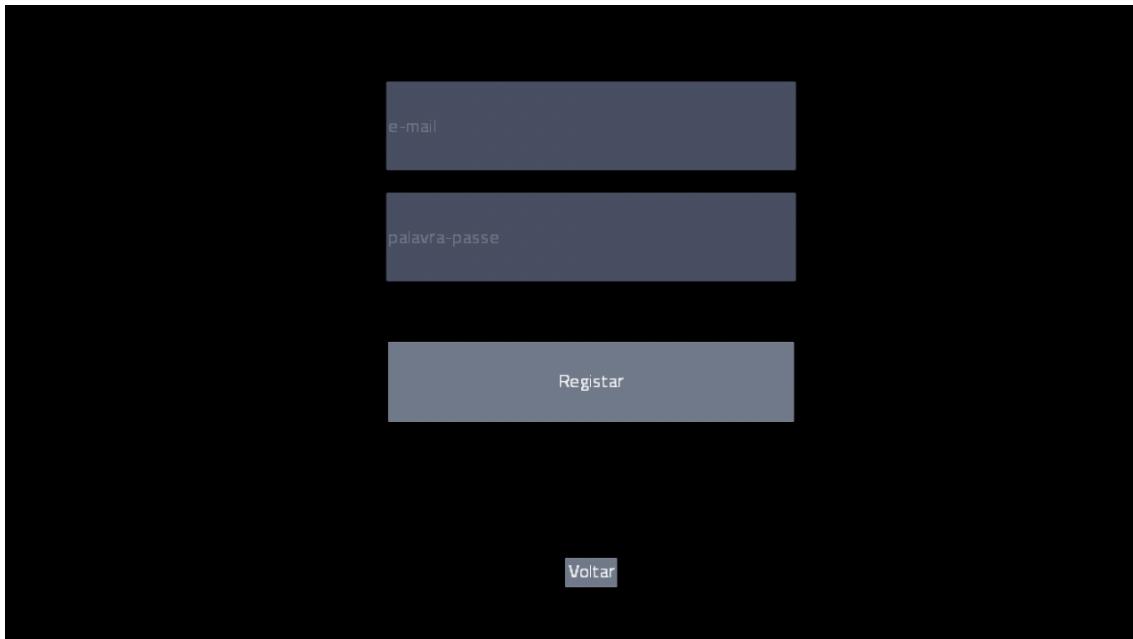


Figura 11: Ecrã de registo do utilizador

De forma a garantir que o endereço de *e-mail* tem um padrão correto, este é validado através de uma expressão regular, apresentada de seguida:

Listing 7: Expressão regular para o endereço de e-mail

```
public static final Pattern VALID_EMAIL_ADDRESS_REGEX =
    Pattern.compile("[A-Z0-9._%+-]+@[A-Z0-9.-]+\\.[A-Z]{2,6}$",
                    Pattern.CASE_INSENSITIVE);

public boolean validate(String emailStr) {
    Matcher matcher = VALID_EMAIL_ADDRESS_REGEX.matcher(emailStr);
    return matcher.find();
}
```

4.3.4 Ecrã de atualização da palavra-passe

Tal como em ambos os ecrãs anteriores, o sistema de implementação do ecrã da atualização da palavra-passe é exatamente o mesmo. Desta feita, é verificado o utilizador atual, através da inserção da palavra-passe atual, de modo a saber se é o próprio que está a tentar realizar a alteração. De seguida, caso a palavra-passe antiga se confirme, é feita a alteração com base na nova palavra-passe, com base na função "*updatePassword()*". Caso contrário, avisa o utilizador que a palavra-passe antiga introduzida não é a correta. O ecrã descrito encontra-se na figura 12.

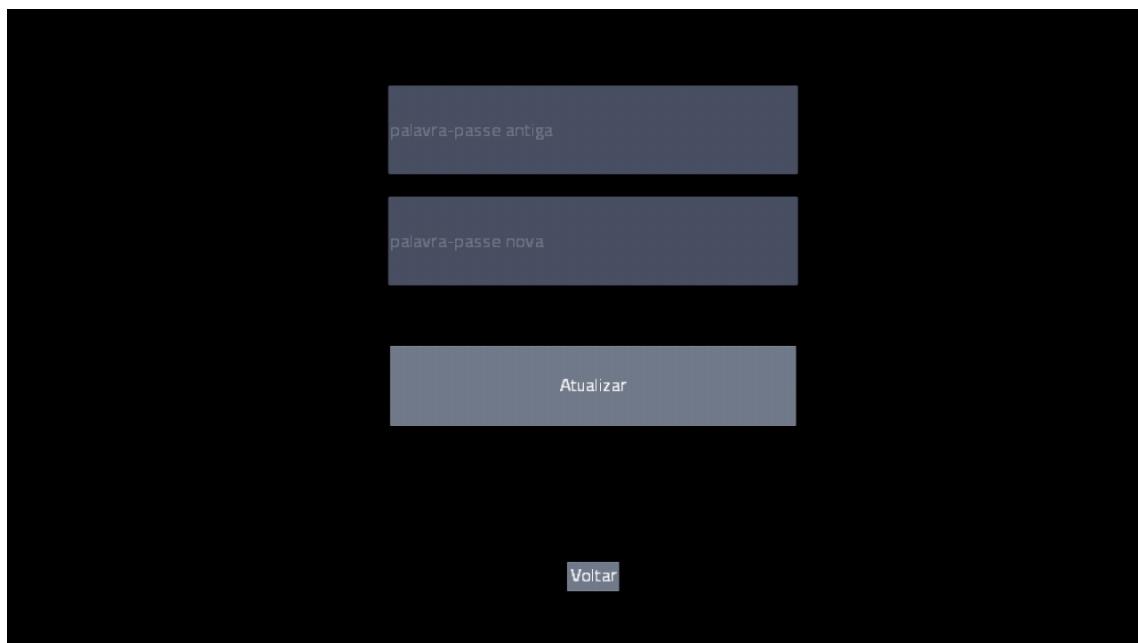


Figura 12: Ecrã de atualização de palavra-passe

4.3.5 Ecrã principal (campo)

A classe referente ao ecrã principal (onde se encontra o campo e os menus) é a classe mais abundante em termos de implementação, contendo grande parte da mesma relativa à aplicação. Aqui, encontram-se os objetos dos jogadores, da bola, do cesto, do ecrã de boas-vindas, dos menus, dos sub-menus, entre outros. São igualmente tratados os eventos provocados pelos utilizadores, assim como o tratamento dos botões e da reprodução da tática. Em baixo, é apresentada uma figura com o campo principal.

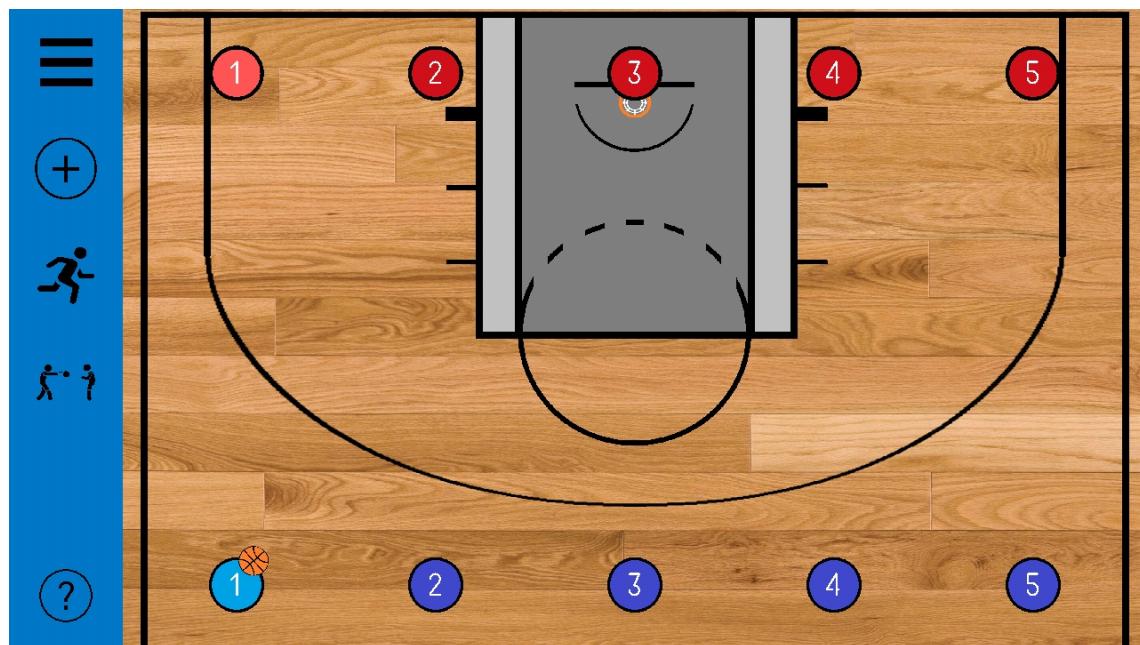


Figura 13: Ecrã principal (campo)

Botões

Ao longo da navegação pela aplicação, são apresentadas várias opções para o utilizador, através de botões. São vários os botões que permitem gerir a tática ou alternar entre menus. A intenção seria criar botões e adicionar a um *stage*, mas visto que pertencem ao mesmo mundo dos jogadores, torna-se impossível definir a parte do *input* do utilizador, pois apenas um poderia apoderar-se do *input processor*. Sendo assim, foram criados botões da mesma maneira que foram criados os agentes, criando um corpo, com uma forma retangular e com dois tipos de textura: uma normal e outra quando estiver selecionado (caso exista). De seguida apresentam-se dois excertos de código: o primeiro referente ao desenho do corpo e o segundo referente à atribuição da textura, conforme seleção do botão.

Listing 8: Criação do botão das opções

```
BodyDef bodyDef = new BodyDef();
bodyDef.type = BodyDef.BodyType.KinematicBody;
bodyDef.position.set(posX, posY);
PolygonShape rect = new PolygonShape();
rect.setAsBox(hx, hy, new Vector2(hx, hy), 0);
FixtureDef fixtureDef = new FixtureDef();
fixtureDef.shape = rect;
body = world.createBody(bodyDef)
    .createFixture(fixtureDef).getBody();

/* ----- */

sprite.setRegion(isSelected ? textureSelected : texture);
```

De modo a organizar todos os botões do ecrã principal, foi criada uma classe (“*OptionsTable*”) que cria vários *arrays* referentes a cada um dos sub-menus e com métodos que preenchem esses *arrays*, tal como um método que os renderiza.

'Input' do utilizador

De modo a controlar o *input* feito pelo utilizador, é feito com que a classe implemente a interface *GestureListener*. Das funções a implementar, foi apenas escolhido o uso da função ”*tap()*”. Esta é necessária, visto que contém um parâmetro que conta o número de toques seguidos que o utilizador deu. Isto, porque a alternância entre sub-menus é feito com um duplo toque, enquanto que os restantes eventos ativam-se com apenas um.

Inicialmente, é guardada a posição do(s) toque(s) e verifica-se que tipo de agente ou botão foi selecionado, através do método ”*whatDidITouch()*”(que será explicado mais à frente). De seguida, é verificado o número de toques. Caso tenha sido um duplo toque e estes tenham sido feitos no botão referente ao menu, altera o sub-menu atual, seguindo uma certa ordem. Caso tenha sido apenas um toque, é invocado o método ”*oneTap()*”, referente às possibilidades existentes com um toque no ecrã. Entre toques, é feita uma espera de 180 milissegundos, feito através da instância do objeto ”*ScheduledThreadPoolExecutor*”. Este realiza uma espera de uma certa funcionalidade ou tarefa, sem bloquear as restantes.

Listing 9: Tempo de espera entre toques

```
exec.schedule(() -> {
    if (trueCounter == 1) {
        oneTap(posHit);
        trueCounter = 0;
    }
}, 180, TimeUnit.MILLISECONDS);
```

Mencionado anteriormente, o método ”*whatDidITouch()*” verifica em que agente ou botão o utilizador realizou o toque. Foram criadas duas variáveis para o caso: ”*bodyHit*”, do tipo *Body*, que guarda o corpo do objeto tocado e ”*bodyHitId*”, do tipo *int*, que guarda o identificador desse objeto, entre os índices do seu *array*. Estes serão necessários para definir a ação a realizar, seja um movimento ou apenas a seleção de um botão ou de um agente.

Inicialmente, verifica qual o sub-menu atual. Depois, dentro dessas opções de botões, verifica se foi alguma delas, através do método ”*setBtn()*”. Aqui, são percorridas as opções de botões e, através do método auxiliar ”*touchedBody()*”, indica se foi realmente algum deles. Se for, guarda numa variável inteira *btn* o valor do índice, para mais tarde o botão ser selecionado e realizar a sua funcionalidade. Caso contrário, nada acontece e continua a verificação inicial, agora com os agentes. Como um toque na bola, no cesto ou num defesa não tem qualquer influência, este é ignorado. Apenas nos atacantes são considerados os toques, verificados através do método auxiliar ”*touchedBody()*”. Se foi realizado um toque num jogador atacante, são guardadas as suas informações na variáveis ”*bodyHit*”e ”*bodyHitId*”. De seguida é apresentado o excerto de código relativo à explicação anterior. Se a variável ”*bodyHitId*” já tiver o valor a ser verificado, significa que o jogador já se encontrava selecionado. Ou seja, indica para cancelar o toque inicial. Caso contrário, guarda a informação.

Listing 10: Verificações no toque num jogador atacante

```
if (touchedBody(offense[i].getBody(), posHit)) {
    if (bodyHitId == i)
        cancelSelect = true;
    else {
        bodyHit = offense[i].getBody();
        if (!isSomeoneSelected || option == PASS)
            bodyHitId = i;
    }
    return;
}
```

O método auxiliar ”*touchedBody()*” mencionado anteriormente verifica a lista de *fixtures* existentes num corpo de um botão ou de um agente. Retorna um valor verdadeiro se alguma das *fixture* se encontra na posição onde foi realizado o toque, analisado através da função ”*testPoint()*”do objeto *Fixture*.

Listing 11: Método auxiliar ”touchedBody()”

```
private boolean touchedBody(Body body, Vector2 pos) {  
    for (Fixture fixture : body.getFixtureList())  
        if (fixture.testPoint(pos.x, pos.y))  
            return true;  
    return false;  
}
```

No que toca ao método ”*setBtn()*”, como dito anteriormente, guarda o índice do botão, dentro dos índices do *array* a que este pertence. Este é depois usado no método ”*buttonHit()*” que verifica em que sub-menu se encontra e qual dos botões se trata, retornando uma das variáveis estáticas que indicam o índice do botão. Por exemplo, se o sub-menu atual for das opções atacantes, a análise é feita da seguinte maneira:

Listing 12: Verificação do botão selecionado

```
int opt = optionsTable.getCurrentOption();  
if (opt == OptionsTable.OFFE) {  
    if (btn == 1) return FRAME;  
    if (btn == 2) return RUN;  
    if (btn == 3) return PASS;  
    if (btn == 4) return HELP;  
}
```

O método ”*oneTap()*” é o que indica a cada funcionalidade o início da sua execução, tratando da organização de todas as variáveis e necessidades até se realizar. Por exemplo, no caso do utilizador indicar que pretende reproduzir a tática, este método coloca todos os jogadores na posições iniciais, a bola no primeiro jogador e dá permissão para a reprodução da jogada. No caso de escolher adicionar uma nova *frame*, é invocado o método que realiza a funcionalidade. Caso pretenda abrir algum dos menus ou algum dos *dialogs*, dá autorização para a mudança de ecrã e altera o *input processor* do ecrã para a nova janela. No caso do toque ter sido num jogador, verifica se é para o selecionar ou se é para cancelar a seleção já existente nesse jogador. No fim, se houver algum movimento a realizar por algum jogador, também é aqui feito, invocando o método referente à realização do mesmo.

Listing 13: Seleção de um jogador no método ”*oneTap()*”

```
if (bodyHitId != -1) {
    if (!isSomeoneSelected) {
        tagPlayer();
        return;
    }

    if (cancelSelect) {
        cancelSelect = false;
        isSomeoneSelected = false;
        offense[bodyHitId].setTagged(false);
        bodyHitId = -1;
        return;
    }
}
```

Dialogs

Assim que o utilizador realize o início de sessão, depara-se com um ecrã de boas-vindas que lhe permite escolher entre duas opções: começar a gerir uma tática (nova ou já existente) ou alterar a sua palavra-passe. Este ecrã é na verdade um *Dialog*, considerado como uma janela de "pop-up", com o propósito de avisar o utilizador de algum acontecimento ou para o utilizador introduzir algum tipo de informação (como o nome de uma tática ou notas). A janela existente já dentro da aplicação serve para avisar o utilizador quando pretende reproduzir uma tática inexistente. As restantes janelas de aviso encontram-se na parte respeitante à autenticação e registo (como avisar, por exemplo, a introdução de um endereço de e-mail inexistente no sistema de autenticação). No que toca aos *dialogs* relativos à introdução de conteúdo por parte do utilizador, temos o que guarda a tática, outro que carrega a tática e outro que guarda as notas.

Foi criada uma classe abstrata que relaciona as partes em comum deste tipo de janelas: "*AbstractDialog*". Esta contém um *Stage* onde serão colocados todos os atores referentes aos componentes da janela. Estes serão implementados no método abstrato "*dialogDraw()*", sendo depois renderizados na classe principal.

Esta classe tem uma variável booleana "*showing*" que indica se foi pedido pelo utilizador que o *dialog* fosse mostrado. Quando for o caso, realiza a renderização da respetiva janela e altera o *input processor* da aplicação para o seu *Stage*, de modo a que possa receber o *input* do utilizador. Assim que sair da janela, volta a alterar o *input processor* para o do ecrã principal. De seguida apresenta-se dois excertos de código (em diferentes localizações na classe) que indicam o explicado: a primeira parte refere-se à indicação do utilizador para mostrar a janela para guardar uma tática, realizando a criação do seu *stage*, sinalizando que se pretende mostrar e alterando o *input processor*. O segundo excerto mostra a renderização do *stage*, no caso da sinalização ser positiva (ver código em baixo). É igualmente apresentado um exemplo na figura 14.

Listing 14: Mostrar de um Dialog (ex: guardar ficheiro)

```
if (option == SVFILE) {  
    currentDialog = D_FILE;  
    dialogSaveFile.dialogDraw();  
    dialogSaveFile.setShowing(true);  
  
    Gdx.input.setInputProcessor(dialogSaveFile.getStage());  
}  
  
/* ----- */  
  
if (currentDialog == D_FILE) {  
    if (dialogSaveFile.isShowing()) {  
        dialogSaveFile.dialogStageDraw();  
    } else {  
        Gdx.input.setInputProcessor(gestureDetector);  
    }  
}
```

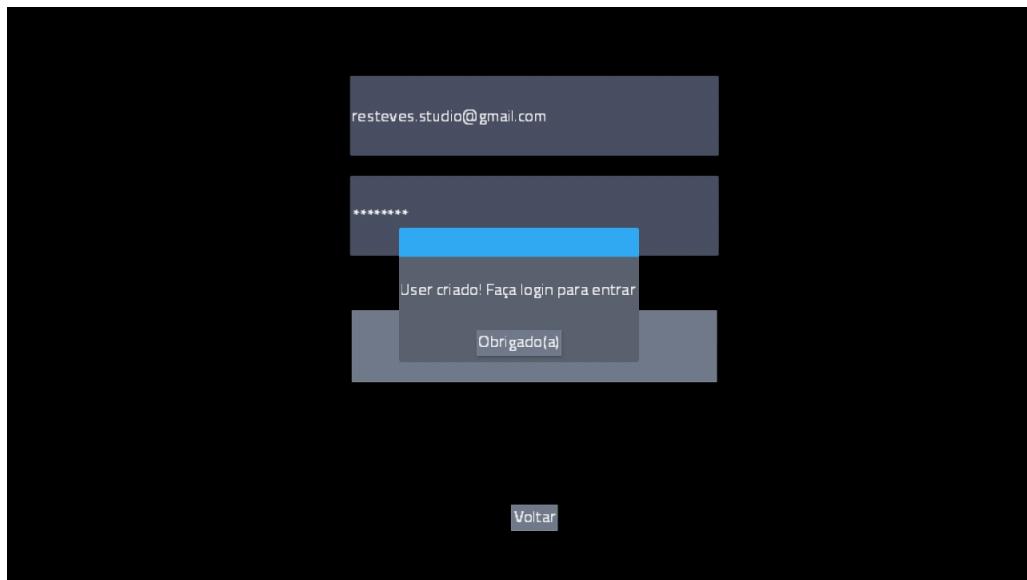


Figura 14: Exemplo de dialog - Registo com sucesso

Menus

Existem dois menus: o principal e o de ajuda. O menu principal tem dois separadores, um que permite alterar as cores da interface (jogadores, bola e campo), enquanto que o outro permite gerir a conta do utilizador, seja através da alteração da palavra-passe, seja através do término da sessão atual. No que toca ao menu de ajuda, este contém quatro separadores, cada um com informações acerca de cada um dos sub-menus existentes e com explicações acerca da criação de uma tática, escolha do tipo de defesa, reproduzir ou limpar a tática, guardar e carregar ficheiros e tirar notas. Ambos estes ecrãs contêm um *Stage* com todo o seu conteúdo. Para distribuir melhor a exposição de cada um aquando dessa escolha por parte do utilizador e de forma a controlar melhor o *input* do utilizador, foi criada uma classe "ScreenHelper" que contém instâncias dos objetos de cada um dos menus. Assim que este é ativo por parte do utilizador, verifica qual dos menus foi escolhido e realiza o desenho do *stage* pretendido, ao mesmo tempo que é trocado o *input processor*. De seguida, é apresentado o excerto de código que verifica a activação do *screen helper* e a actualização dos valores referentes às cores da interface, assim como do *input processor*.

Listing 15: Ativação de um dos menus

```
if (screenHelper.isActivated()) {  
    screenHelper.screenDraw();  
    setPlayersColor(screenHelper.getMenuScreen().getOffenseColor(),  
        screenHelper.getMenuScreen().getDefenseColor());  
    ball.setBallColor(screenHelper.getMenuScreen().getBallColor());  
    setCourtColor(screenHelper.getMenuScreen().getCourtColor());  
    Gdx.input.setInputProcessor(screenHelper.getStage());  
}
```

Unidade Curricular de Projeto



Figura 15: Menu principal



Figura 16: Menu de ajuda

Reprodução da tática

Para reproduzir uma tática, é necessário que esta tenha movimentos dos jogadores. Para isto, é necessário adicionar *frames*, de modo a que os jogadores tenham uma posição inicial e alguns movimentos posteriores. De modo a adicionar *frames*, o utilizador carrega no respetivo botão e é invocado o método ”*addNewFrame()*”. Este método começa por verificar se a tática já tem as posições iniciais definidos. Se não for o caso, percorre todos os atacantes e guarda uma cópia das respetivas posições (visto que os valores das posições do *body* são dinâmicos) no *array* relativo às posições iniciais compreendido na instância da tática. São igualmente definidas as posições iniciais dos *targets* dos defesas, com indicação de quem é o primeiro com a posse da bola.

Listing 16: Adição das posições iniciais

```
for (int d = 0; d < offense.length; d++) {  
    Vector2 posInit = offense[d].getBody().getPosition().cpy();  
    tactic.addInitialPos(d, posInit);  
    if (offense[d].hasBall()) firstPlayerWithBall = d;  
}  
  
for (int d = 0; d < defense.length; d++) {  
    defense[d].setPlayerWithBall(offense[firstPlayerWithBall]);  
    defense[d].setInitMainTargetPosition();  
    Vector2 posInit = defense[d].getMainTargetPosition();  
    tactic.addInitialPos(d+5, posInit);  
}
```

Após a adição desta primeira *frame*, o utilizador adiciona as *frames* restantes, relativas já aos movimentos dos jogadores. Como será explicado no capítulo 4.6, o dicionário contém um múltiplo de cinco do índice do jogador em questão e uma entrada que inclui o movimento, quem dos colegas foi afetado nesse movimento (recetor de um passe, numa corrida mais ninguém participa senão o jogador em questão) e uma cópia das posição do *target* nesse movimento. São adicionados os valores em questão ao dicionário pertencente à tática.

Listing 17: Adição de um movimento

```
int size = tactic.getSize();

for (int d = size; d < offense.length + size; d++) {
    Integer[] moves = new Integer[] {
        offense[d % 5].getLastMove(), offense[d % 5].getReceiver()
    };

    tactic.addToMovements(d, moves,
        offense[d % 5].getTarget().getBody().getPosition().cpy());

    offense[d % 5].setLastMove(-1);
    offense[d % 5].setReceiver(-1);
}
```

Após a conclusão da construção da tática, o utilizador pretende reproduzi-la. Ao premir o botão respetivo, irá invocar o método ”*playTactic()*”. Aqui, é usado de novo o ”*ScheduledThreadPoolExecutor*” para o intervalo de tempo entre a reprodução de cada *frame*. Contudo, foi necessário recorrer a um *Thread.sleep()* de modo a bloquear a reprodução das frames que se encontram em fila de espera. Apenas quando uma *frame* terminar a sua reprodução é que a seguinte entra em ação.

De modo a reproduzir a *frame*, é invocado o método ”*applyMove()*” que, através dos valores do dicionário, respetivos ao movimento do jogador que está a ser dada atenção, preenche os valores de ”*bodyHit*” e de ”*bodyHitId*”, assim como indica o movimento a realizar. Depois, é invocado o método que realiza o movimento do jogador.

Listing 18: Reprodução da tática

```
exec.schedule(() -> {
    stillPlayingMove = true;

    for (int f = flag - 5; f < flag; f++) {
        applyMove(f, tactic.getEntryKey(f)[0],
                  tactic.getEntryKey(f)[1]);
    }

    waitTime(6500);
    flag += 5;
    startFrameOver();

}, 2, TimeUnit.SECONDS);
```

Também é possível para o utilizador limpar o quadro da tática atual. Para isso, existe o método ”*resetTactic()*” que limpa todo o conteúdo existente na classe atual e na instância da tática, começando pelo dicionário do movimento e pelo *array* das posições iniciais. Após esta seleção, não existirá nenhum valor guardado e todos os jogadores encontrar-se-ão nas suas posições de partida, prontos a ser colocados nas suas posições iniciais. Se o utilizador pretender reproduzir a tática que tinha, não será possível e será avisado do sucedido.

4.4 Agentes

Um agente, no contexto deste projeto, é caracterizado, principalmente, pelos jogadores e pela bola. Contudo, existem outros tipos de agente que passam por despercebidos, como os agentes invisíveis (cesto ou *target* dos jogadores), um "check" de verificação por cada movimento de um jogador e o número do jogador. Todos estes são agentes e estendem da mesma classe abstrata: "*SteeringAgent*".

Um *SteeringAgent* comporta-se tal como um corpo na vida real: tem uma forma, um tamanho, uma posição, uma aceleração e uma velocidade. No caso deste projeto, foi possível fazer o desenho de todos os agentes da mesma maneira. Definiu-se o seu tipo de corpo, sendo que apenas os agentes invisíveis foram definidos como estático, visto que não têm qualquer tipo de movimento ou reação a forças, enquanto que os restantes definiu-se como dinâmico, pois têm uma massa e movem-se com a aplicação de forças sobre ele. Tem a forma de um círculo, pois é a forma de uma bola e é como normalmente são feitos os desenhos dos jogadores com papel e caneta. Foram igualmente definidos os corpos com que cada colide, sendo que cada um tem a sua implementação dos métodos abstratos que definem a sua categoria e a(s) categoria(s) com que se embate. A construção do agente descrita encontra-se no excerto de código em baixo. É igualmente apresentado na figura 17 dois exemplos das imagens dos jogadores.

Listing 19: Construção do corpo dos agentes

```
private void buildAgent(World world, float posX, float posY,
    BodyDef.BodyType bodyType) {
    BodyDef bodyDef = new BodyDef();
    bodyDef.type = bodyType;
    bodyDef.position.set(posX, posY);
    circle = new CircleShape();
    circle.setRadius(boundingRadius);
    fixtureDef = new FixtureDef();
    fixtureDef.shape = circle;
    fixtureDef.filter.categoryBits = isCattegory();
    fixtureDef.filter.maskBits = collidesWith();
    body = world.createBody(bodyDef)
        .createFixture(fixtureDef).getBody();
}
```

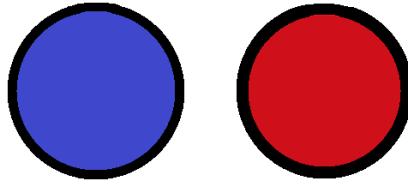


Figura 17: Exemplos de jogadores

No que toca às colisões existentes entre os agentes, apenas os jogadores colidem entre si. A bola não colide com ninguém, pois assume-se que esta apenas passa pelas mãos dos atacantes, sem qualquer interseção pelo meio. O cesto, apesar de ser um agente, é invisível e não tem qualquer tipo de contacto, tal como o ”check” dos jogadores e o *target* dos mesmos.

4.4.1 Aplicação dos comportamentos (“*Steering*”)

Após análise da explicação do vídeo [16] apresentada por *Conner Anderson*, para que se possa aplicar os comportamentos *Steering*, a classe ”*SteeringAgent*” tem de estender a classe ”*Steerable*”, do tipo *Vector2*. Dentro das funções obrigatórias a implementar, destacam-se as que referem os tipos de velocidades e de acelerações, sendo elas a lineares e as angulares (explicadas na secção 3.2.2). Estas serão necessárias para o cálculo dos movimentos. São igualmente necessárias instâncias do tipo de comportamento que será aplicado ao agente (*SteeringBehavior*, do tipo *Vector2*) e um *output* que irá receber todos os cálculos feitos pelo AI, sendo depois aplicado ao corpo do agente, fazendo-o mover.

De modo a aplicar o comportamento no agente, foi criado o método ”*applySteering()*”. Recebe como argumento uma variável ”*delta*” do tipo *float*, com origem na função que renderiza o mundo. Primeiro, verifica-se se existe alguma força para aplicar. No caso de existir uma força linear, esta é aplicada ao corpo através da função ”*applyForceToCenter()*” e confirma-se a existência de acelerações. Caso contrário, define-se tanto a velocidade linear como angular como nulas.

Listing 20: Aplicação de uma força linear ao corpo do agente

```
if (!steeringAcceleration.linear.isZero()) {
    Vector2 force = steeringAcceleration.linear.scl(delta);
    body.applyForceToCenter(force, true);
    accelerationApplied = true;
} else {
    body.setLinearVelocity(0f, 0f);
    body.setAngularVelocity(0);
}
```

No caso de existir uma força que provoque uma aceleração angular, aplica-se o "torque", através da função "*applyTorque()*", pois esta afeta a velocidade angular do corpo. Contudo, com apenas esta aplicação, o agente não tem a rotação que se pretende. Isto porque é necessário verificar se o corpo tem uma velocidade linear maior que zero. Se for o caso, é calculada a nova orientação do corpo do agente e aplicada a transformação.

Listing 21: Aplicação de uma força angular ao corpo do agente

```
if (steeringAcceleration.angular != 0) {
    body.applyTorque(steeringAcceleration.angular * delta, true);
    accelerationApplied = true;
} else {
    Vector2 linVel = getLinearVelocity();
    if (!linVel.isZero()) {
        float newOrientation = vectorToAngle(linVel);
        body.setAngularVelocity((newOrientation -
            getAngularVelocity()) * delta);
        body.setTransform(body.getPosition(), newOrientation);
    }
}
```

Por fim, verificada a existência de algum tipo de aceleração através do *booleano accelerationApplied*, tem de ser aplicado um limite no mesmo, visto que, sem ele, o valor não parava de incrementar. Para isto, foram criadas variáveis com valores que limitam as velocidades e acelerações lineares e angulares. Assim que estes valores se deparam com esse limite, este é aplicado de forma a que não seja ultrapassado.

Listing 22: Limitação das velocidades do agente

```
if (accelerationApplied) {  
    Vector2 velocity = body.getLinearVelocity();  
    float currentSpeedSquare = velocity.len2();  
    if (currentSpeedSquare > maxLinearSpeed * maxLinearSpeed) {  
        body.setLinearVelocity(velocity.scl(maxLinearSpeed /  
            (float) Math.sqrt(currentSpeedSquare)));  
    }  
  
    if (body.getAngularVelocity() > maxAngularSpeed) {  
        body.setAngularVelocity(maxAngularSpeed);  
    }  
}
```

Depois, este método é invocado no método ”*update()*”. Existindo um comportamento, é feito primeiro o cálculo do *steering* do mesmo, através da função ”*calculateSteering()*” (passando o *output* mencionado anteriormente) e, de seguida, feita a aplicação do comportamento no agente.

Listing 23: Aplicação do steering

```
if (behaviour != null) {  
    behaviour.calculateSteering(steeringAcceleration);  
    applySteering(delta);  
}
```

Exemplificando a adição de um comportamento *Steering* a um agente, como é o caso da bola, é criada uma instância do comportamento *Arrive*, passando como argumentos o agente e o *target* do agente. A função ”*setTimeToTarget()*” indica o tempo desejado até atingir a velocidade desejada. A função ”*setArrivalTolerance()*” indica o raio de aproximação que o agente tem do seu *target*, ou seja, o momento em que ele para de se aproximar, evitando a existência de pequenos erros. A função ”*setDecelerationRadius()*” define o raio em que o agente inicia a redução da sua velocidade, diminuindo os valores de aceleração. A função ”*setBehaviour()*” aplica o comportamento criado no agente.

Listing 24: Aplicação do steering

```
Arrive<Vector2> ballBehaviour = new Arrive<>(ball,  
    ball.getTarget())  
    .setTimeToTarget(0.01f)  
    .setArrivalTolerance(0.1f)  
    .setDecelerationRadius(playerBoundingRadius * 2)  
    .setEnabled(true);  
ball.setBehaviour(ballBehaviour);
```

4.5 Posicionamento defensivo

Como se sabe, o defesa contém um *target* invisível, cuja posição será a que o primeiro irá sempre seguir. De forma a saber a posição desse *target*, são feitos cálculos (detalhados no tópico 3.2.3) para o tipo de defesa individual, de modo a descobrir as coordenadas, enquanto que no tipo de defesa zona é verificada a posição do jogador com a bola, de forma a saber em que zona do campo se encontra.

4.5.1 Defesa individual

Existem duas formas de estar no tipo de defesa individual: a pressionar o jogador com bola, ou em posição de ajuda. Quando é feita pressão no jogador com bola, o defesa encontra-se, normalmente, na vida real, com um braço de distância. Neste sistema, esta distância traduz-se na multiplicação do raio do agente. Para calcular a sua posição, é criado um vetor, normalizado, que traduz a distância entre o jogador com bola e o cesto (pois o defesa estará sempre entre os dois). A partir deste vetor, é adicionada uma certa percentagem do mesmo à posição do jogador com bola (teoricamente, um braço de distância), o que faz com que a posição do defesa se mantenha sempre entre ele e o cesto com a distância atribuída. Esta explicação especifica o excerto de código apresentado em baixo.

Listing 25: Cálculo da posição do defesa do jogador com bola

```
if (getPlayerTarget().hasBall()) {  
    Vector2 u = getBasketTarget().getPosition().cpy()  
        .sub(getPlayerTarget().getPosition().cpy()).nor();  
    Vector2 v = getPlayerTarget().getPosition().cpy()  
        .add(u.scl(4f*getBoundingRadius(),  
                  4f*getBoundingRadius()));  
    return new Vector2(v.x, v.y);  
}
```

Passando para a posição de ajuda e tendo em conta que é necessário o cálculo de círculos e respetivas interseções, foi utilizado o código disponibilizado pelo sítio rosettacode.org [15], criando-se as classes ”*Point*” e ”*Circle*”. A classe ”*Point*” refere-se a um elemento cuja posição se traduz por um *Vector2*, contendo igualmente as coordenada *x* e *y*. Contém também uma função que calcula a distância entre dois pontos, uma função que converte um *Vector2* num *Point* e uma função que verifica se dois pontos são iguais. A ”*Circle*” contém um centro (do tipo *Point*) e um raio. Como funções, tem uma que calcula o centro de um círculo e outra que calcula os pontos de interseção entre dois círculos, sendo o conteúdo de ambas explicado no tópico 3.2.3.

São criadas três instâncias de *Point*: ’*p1*’ para o jogador que está a defender, ’*p2*’ para a bola e ’*p3*’ para o cesto. Entre estes três pontos, é calculado o ângulo formado entre eles, de forma a criar uma noção da distância e do posicionamento entre o defesa e a bola. No caso do ângulo formado entre eles ser obtuso, ou o jogador que defende se encontra num dos cantos, significa que estão longe um do outro. Aqui, o defesa entrará numa posição de ajuda mais interior, visto que, se for necessária a ajuda, esta será dada mais perto do cesto. Isto significa que a posição do defesa encontra-se entre o seu jogador e o cesto, tendo uma variação dependente do ângulo existente com a bola. No caso da bola se encontrar muito perto (ângulo entre 0° e 45° centígrados), significa que o defesa terá de se concentrar em ”fechar a linha de passe” do seu jogador, ou seja, impedir que lhe passem a bola. Para isto, será feita uma pressão parecida à pressão feita com o jogador da bola, aproximando-se mais do seu jogador. O cálculo da posição do defesa será o mesmo que o jogador que defende a bola. Esta explicação detalha o excerto de código apresentado em baixo.

Listing 26: Cálculo da posição do defesa que se encontra perto ou longe da bola

```

if (thetaInv >= 90 || (p1.getY() > Gdx.graphics.getHeight()*2/3f
    && p2.getY() > Gdx.graphics.getHeight()*2/3f)) {
    float per = 4.5f * (-180 / thetaInv);
    Vector2 u = getBasketTarget().getPosition().cpy()
        .sub(getPlayerTarget().getPosition().cpy()).nor();
    Vector2 v = getBasketTarget().getPosition().cpy()
        .add(u.scl(per*getBoundingRadius(),
            per*getBoundingRadius()));
    return new Vector2(v.x, v.y);
} else if (thetaInv < 45) {
    Vector2 u = getBasketTarget().getPosition().cpy()
        .sub(getPlayerTarget().getPosition().cpy()).nor();
    Vector2 v = getPlayerTarget().getPosition().cpy()
        .add(u.scl(4f * getBoundingRadius(), 4f *
            getBoundingRadius()));
    return new Vector2(v.x, v.y);
}

```

Caso o defesa se encontre numa posição dentro do intervalo de ângulos entre os dois intervalos referidos anteriormente, terá de tomar mais atenção na sua posição e na variação da mesma. Aqui, será feito o cálculo da interseção dos círculos formados entre os três elementos principais da posição defensiva.

São inicialmente criadas instâncias de *Circle* para cada um dos três círculos necessários e calculado o centro de cada um, através da função ”*findCircleCenter()*” de *Circle*. Como argumentos, tem os pontos pertencentes ao círculo e o ângulo formado entre eles e o defesa. Contém igualmente o terceiro elemento, apenas para efeitos de comparação. De seguida, é invocada a função ”*findCircleIntersection()*” de *Circle*, que retorna ambos os pontos de interseção entre cada par de círculos. Tendo os seis pontos de interseção à mão, é feita uma comparação entre todos (com um pequeno erro, pois nem todos os pontos podem ser iguais em todas as casas decimais), de modo a saber qual o ponto em comum. É, por fim, retornado o ponto de interseção comum entre os círculos, ponto esse que será a posição do *target* do defesa.

No que toca ao valor dos ângulos, foi feita uma definição prévia dos mesmos, tendo sido estudados e testados, de forma a maximizar a aproximação à realidade. A definição dos ângulos apresenta-se de seguida:

Listing 27: Pré-definição dos ângulos

```
if (theta >= 70 && theta < 90) {  
    angles[0] = 120;  
    angles[1] = 120;  
    angles[2] = 120;  
} else if (theta >= 50 && theta < 70) {  
    angles[0] = 130;  
    angles[1] = 130;  
    angles[2] = 100;  
} else if (theta >= 30 && theta < 50) {  
    angles[0] = 135;  
    angles[1] = 130;  
    angles[2] = 95;  
}
```

4.5.2 Defesa zona

O tipo de defesa zona, como diz o nome, serve para defender um certo espaço no campo, em vez da usual e mais conhecida defesa individual, onde se dá prioridade a um certo jogador. Por enquanto, apenas foi adotada uma das variações da defesa zona, a chamada *”zona 2-3”*, onde se encontram dois jogadores a defender as zonas mais centrais e perto da linha de três pontos, enquanto que os restantes três jogadores defendem os espaços mais perto do cesto e próximo dos cantos. Normalmente são os jogadores das posições de base (posições 1 e 2) que defendem os espaços frontais, enquanto que os extremos e o poste (posições 3, 4 e 5) defendem mais atrás. As posições que este ocupam são pré-definidas, pois não há uma grande dependência nos movimentos dos atacantes (apenas saber onde se encontra a bola). O campo foi dividido da maneira apresentada na figura 18.

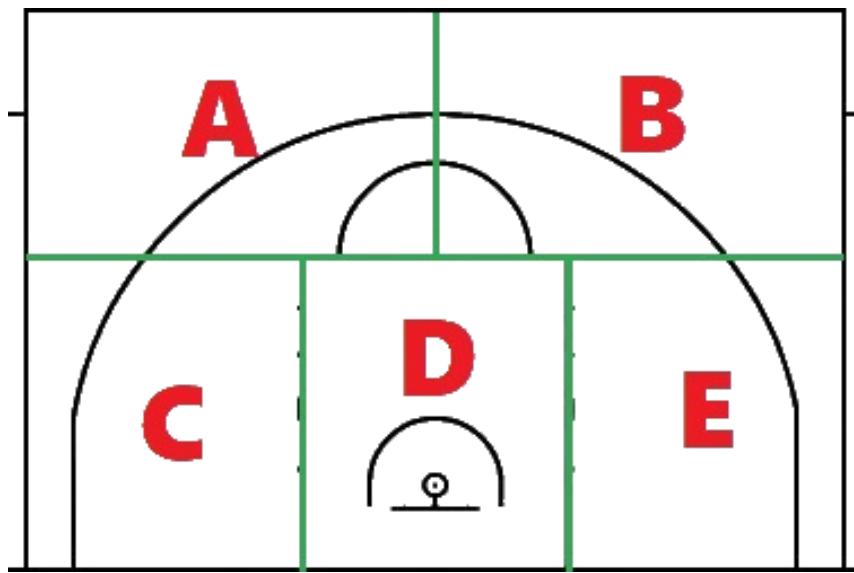


Figura 18: Zonas do campo para a defesa zona 2-3

Os jogadores com a posição de 1 a 5 irão ocupar os espaços de A a E do campo, respetivamente, por ordem alfabetica. A mecânica é simples: se a bola estiver na sua zona, o defesa pressiona o atacante, enquanto que os restantes ajustam-se. É apresentado um exemplo do posicionamento em campo na figura 19.

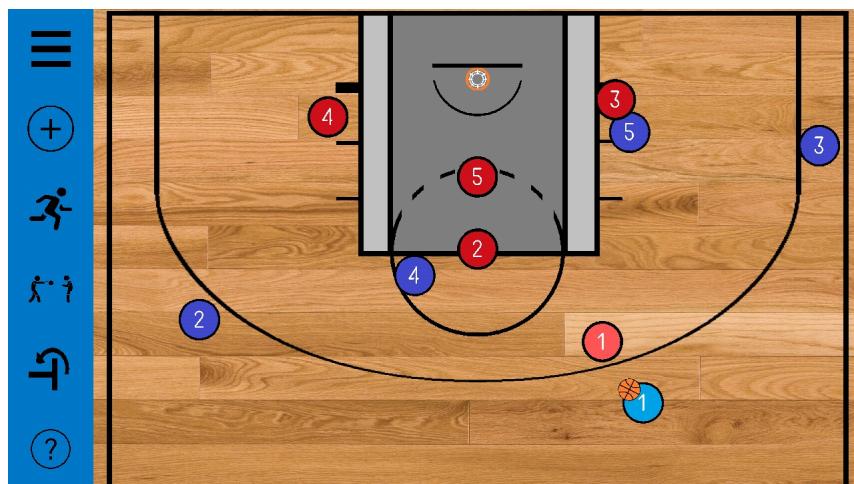


Figura 19: Exemplo da defesa zona 2-3 (bola na zona A)

Exemplificando: se a bola estiver na zona C, o respetivo defesa (número 3) irá defender individualmente o atacante, os jogadores 1 e 2 baixam ligeiramente a sua posição, o jogador 5 (da zona D) toma a posição da zona C e o jogador 4 (da zona E) toma a posição da zona D. É apresentado um exemplo do posicionamento em campo na figura 20.

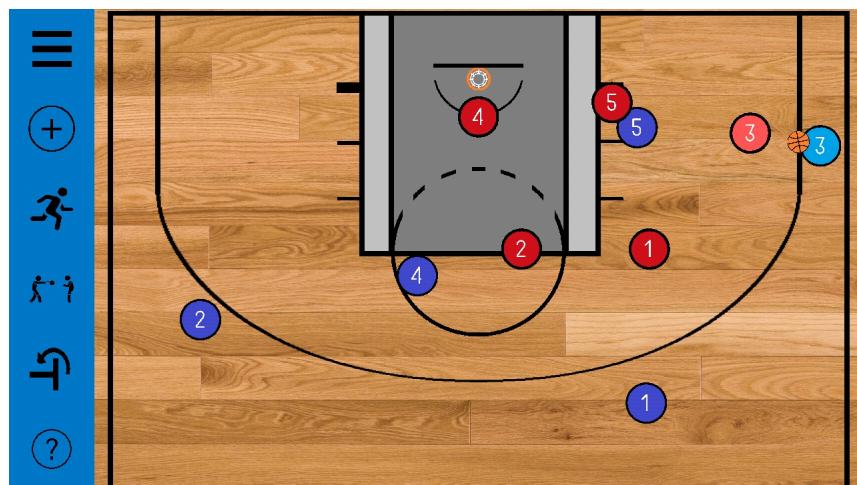


Figura 20: Exemplo da defesa zona 2-3 (bola na zona C)

Foi criado um dicionário por espaço, cuja chave se refere ao jogador e o valor à respetiva posição, caso a bola se encontre no espaço do respetivo dicionário. De referir que o identificador do defesa será a letra do seu espaço e que o jogador próprio dessa zona não tem uma entrada visto que se encontra a pressionar a bola individualmente. De seguida, é apresentado um excerto de código exemplo do dicionário referente ao espaço A e as respetivas posições por jogador quando a bola se encontra nesta zona.

Listing 28: Dicionário referente às posições na zona A

```
A = new HashMap<>();
A.put("B", new Vector2(Gdx.graphics.getWidth()*5.50f/10f,
    Gdx.graphics.getHeight()/2f));
A.put("C", new Vector2(Gdx.graphics.getWidth()*7.25f/10f,
    Gdx.graphics.getHeight()*7.75f/10f));
A.put("D", new Vector2(Gdx.graphics.getWidth()*5.50f/10f,
    Gdx.graphics.getHeight()*6.50f/10f));
A.put("E", new Vector2(Gdx.graphics.getWidth()*3.75f/10f,
    Gdx.graphics.getHeight()*7.75f/10f));
```

No método que retorna a posição em que o jogador tem de se encontrar, verifica em que zona do campo a bola se encontra e, tendo em conta a zona do defesa atual, vai buscar ao dicionário a respetiva posição. De seguida, é apresentado um excerto de código exemplo da verificação referente ao espaço A e o retorno da posição do defesa, caso pertença a um espaço diferente.

Listing 29: Verificação da bola na zona A

```
float x = getPlayerWithBall().getPosition().x;
float y = getPlayerWithBall().getPosition().y;

if (x > Gdx.graphics.getWidth()/2f && y <
    Gdx.graphics.getHeight()/2f) {
    getPlayerWithBall().setZoneSpace("A");
    if (!this.zoneSpace.equals("A")) return A.get(this.zoneSpace);
}
```

4.6 Tática

A tática é composta pelos movimentos dos jogadores, divididos em várias fases (*frames*). Estas *frames* encontram-se guardadas num dicionário (*HashMap*), com a chave contendo um múltiplo de cinco do índice do jogador (sendo eles cinco jogadores, é uma forma de cada um ter vários movimentos, através do resto da divisão inteira do índice) e o valor que contém uma entrada de um mapa (*Map.Entry*). Esta entrada contém um *array* de inteiros que refere o tipo de movimento feito pelo jogador e, se afetar mais alguém (por exemplo, o recetor da bola através de um passe), o índice deste último. Caso não tenha realizado nenhum movimento, ou nenhum colega tenha participado, esse valor será igual a -1. O segundo argumento da entrada será um *Vector2* contendo a posição do *target* do jogador que realizou o movimento.

Para além deste dicionário, existe igualmente um *array* de *Vector2*, contendo as posições iniciais de cada um dos dez jogadores.

Contém igualmente duas *string*, uma referente ao nome da tática e outra referente às notas. O nome da tática, caso não seja alterado, será pré-definido com a seguinte estrutura: *"tacticddMMyyHHmm"*. *dd* corresponde ao dia atual, *MM* corresponde ao mês, *yy* corresponde ao ano, *HH* corresponde às horas e, por fim, *mm* corresponde aos minutos. As notas iniciam-se sem qualquer corpo de texto.

4.7 Sistema de ficheiros

Como se sabe, o utilizador, depois de criar a sua tática, pode guardá-la de modo a reproduzi-la mais tarde. Para que isto seja possível, é necessário haver um sistema que gere os ficheiros que guardam as táticas.

Tendo em conta os elementos existentes numa tática (detalhados na secção 4.6), foram escolhidos aqueles que são considerados fundamentais. Entre eles, está o seu nome, estão as posições iniciais dos jogadores atacantes, todos os movimentos feitos e, por fim, as notas. Foi também adicionado o valor da variável "*nFrames*" de forma a se saber em que ponto da jogada se ficou. Todos os valores destes elementos são guardados num ficheiro *JSON*. Desta forma, obtém-se uma boa organização dentro do ficheiro, o que ajuda na escrita e no acesso desses elementos.

4.7.1 Classe 'TacticFileHandle'

Esta classe serve para escrever e aceder aos elementos dentro de um ficheiro pertencente à tática. É utilizada uma biblioteca [14] capaz de manipular os ficheiros *JSON* de forma simples e eficaz. Foi necessária a criação de uma pasta "*lib*" dentro do projeto e a adição de uma dependência no ficheiro "*gradle*" pertencente ao pacote "*core*", com referência ao caminho da pasta criada.

Através desta biblioteca, são criados objetos cujo conteúdo se refere a cada um dos elementos referidos anteriormente. Cada um desses objetos pode conter um valor, um ou mais *arrays* ou até mesmo outros objetos. Por exemplo, no caso do nome da tática, é apenas necessário a atribuição do seu valor (*string* do nome), enquanto que no caso das posições iniciais dos jogadores existe um array dentro do respetivo objeto, referente a essas mesmas posições. O excerto de código apresentado de seguida exemplifica a criação de instâncias de objetos e de arrays, para que os valores dos elementos sejam escritos no ficheiro.

Listing 30: Adição das posições iniciais a um ficheiro JSON

```
JSONObject initialPositionsObject = new JSONObject();
for (int i = 0; i < tactic.getInitialPosLength(); i++) {
    JSONArray initPos = new JSONArray();
    initPos.add(tactic.getInitialPos(i).x);
    initPos.add(tactic.getInitialPos(i).y);
    initialPositionsObject.put(i, initPos);
}
JSONObject initialPositions = new JSONObject();
initialPositions.put("initialPositions", initialPositionsObject);
```

Para escrever no ficheiro, cujo nome é igual ao nome da tática, basta executar a seguinte linha de código:

```
fileHandle.writeString(tacticList.toJSONString(), false);
```

O objeto ”*fileHandle*” refere-se à diretoria do ficheiro (que terá o formato local da aplicação - *Gdx.files.getLocalStoragePath()*). O primeiro argumento converte o *array* que contém todos os elementos numa *string* com o formato *JSON*, de forma a ser escrito corretamente no ficheiro. O segundo argumento questiona se é pretendida a adição (*true*) ou substituição (*false*) do conteúdo existente do ficheiro (caso exista). Tudo isto encontra-se dentro do método ”*writeTacticToJSON()*”, cuja invocação será mencionada na secção seguinte (4.7.2).

Para ler a partir de um ficheiro *JSON*, existe o método ”*readTacticFromJSON()*” (cuja invocação será igualmente mencionada na secção seguinte - 4.7.2), que retorna um objeto da classe ”*Tactic*”. Aqui, é necessária a utilização de um *parser* (divide informação em bocados mais pequenos), mais propriamente uma instância de *JSONParser*, de maneira a que se possa converter e manipular a *string* existente dentro do ficheiro *JSON*. Após conversão, torna-se possível obter os valores dos elementos e guardá-los numa nova instância de ”*Tactic*”. O excerto de código apresentado de seguida exemplifica a obtenção dos valores das posições iniciais a partir do ficheiro e a inserção numa nova tática (objeto ”*loadedTactic*”), tendo já o seu formato inicial.

Listing 31: Obtenção das posições iniciais de um ficheiro JSON

```

JSONObject initialPositions = (JSONObject) tacticList.get(0);
JSONObject initialPositionsObject = (JSONObject)
    initialPositions.get("initialPositions");
for (int i = 0; i < initialPositionsObject.size(); i++) {
    JSONArray pos = (JSONArray)
        initialPositionsObject.get(String.valueOf(i));
    float x = Float.parseFloat(String.valueOf(pos.get(0)));
    float y = Float.parseFloat(String.valueOf(pos.get(1)));
    loadedTactic.addInitialPos(i, new Vector2(x, y));
}

```

4.7.2 Classe 'SaveLoad'

Esta classe contém métodos que realizam tanto o *upload* como o *download* dos ficheiros referentes às táticas, tanto localmente como na *cloud*. Contém igualmente uma instância do objeto ”*TacticFileHandle*” e uma instância do objeto ”*Tactic*”, referente à tática atual.

Para que uma tática possa ser guardada localmente, esta tem de cumprir alguns requisitos. Isto é, não pode ser vazia, tendo obrigatoriamente de conter as posições iniciais dos jogadores e, no mínimo, uma *frame* de movimentos. Sem isto, não pode ser considerada uma tática. Se for o caso, é definido o nome do ficheiro, definido o seu caminho e invocado o método ”*writeTacticToJson()*”, da classe ”*TacticFileHandle*”, explicado anteriormente.

Quando for para guardar na *cloud*, os requisitos são exatamente os mesmos. Neste caso, terá de ser feita uma comunicação com o *firebase* e feito o *upload* do caminho para o ficheiro, após ter sido verificado o utilizador atual.

No que toca ao carregamento (*download*) do ficheiro, o processo passa pela verificação da existência do mesmo, de forma a saber se é possível carregar o ficheiro desejado. Se for o caso, falando localmente, é invocado o método ”*readTacticFromJSON()*”, da classe ”*TacticFileHandle*”, explicado anteriormente, modificado o valor do objeto da tática presente na classe para o valor do retorno do método, assinalando ao mesmo tempo o sucedido. No que toca ao *download* a partir da *cloud*, o processo de verificação é o mesmo e a troca de mensagens com o *firebase* é feita através da função ”*download()*”, passando como parâmetro o caminho para o ficheiro.

a) Dialog para guardar ficheiro

Assim que o utilizador pretende guardar a tática, seleciona o botão correspondente que irá mostrar uma caixa (*dialog*) com uma caixa de texto e dois botões. Essa caixa de texto serve para introduzir o nome que pretende dar à tática, enquanto que os botões servem para submeter ou cancelar. Assim que o utilizador submete, o sistema verifica se é necessário alterar o nome da tática ou se permanece com o valor *default*. De seguida, invoca os métodos que guardam o ficheiro, tanto localmente, como na *cloud*. Em baixo é apresentado um excerto de código que mostra o sucedido.

Listing 32: Métodos da classe ”DialogSaveFile”

```
private void saveLocal(String name) {
    if (!name.isEmpty()) saveLoad.setTacticName(name);
    saveLoad.saveLocalData();
    setShowing(false);
}

private void saveCloud(String name) {
    if (!name.isEmpty()) saveLoad.setTacticName(name);
    saveLoad.saveCloudData();
    setShowing(false);
}
```

b) Dialog para carregar ficheiro

No que toca ao carregar de um ficheiro, o processo será exatamente o mesmo, com exceção do sítio onde ele é feito. Assim que o utilizador submete o nome da tática que pretende carregar, o sistema verifica se consegue comunicar com o *firebase*, de modo a procurar o ficheiro no armazenamento da *cloud* (pois será sempre o mais atualizado). Caso contrário, tenta carregar o ficheiro localmente. Em baixo é apresentado um excerto de código que mostra o sucedido.

Listing 33: Métodos da classe "DialogSaveFile"

```
if (loadCloud(name)) {
    Gdx.app.log("Loaded", "Cloud");
} else {
    loadLocal(name);
    Gdx.app.log("Loaded", "Local");
}

private void loadLocal(String name) {
    saveLoad.loadLocalData(name);
}

private boolean loadCloud(String name) {
    return saveLoad.loadCloudData(name);
}
```

5 Validação e Testes

De forma a testar a aplicação e visto que o objetivo da mesma é cumprir com as necessidades do utilizador, foi criado um questionário com o intuito deste arbitrar acerca da usabilidade da aplicação. Este questionário contém um *link* referente ao *APK* gerado, fornecido aos participantes, de maneira a que estes tenham acesso à aplicação.

5.1 Contexto

Este questionário contém um tutorial que guia o utilizador pelas funcionalidades existentes na aplicação. Assim, o utilizador pode experimentar cada uma delas ao pormenor, enquanto responde às questões.

Foram definidos alguns objetivos principais de forma a saber que tipo de *feedback* se pretende obter do utilizador. Dividiu-se o questionário em várias secções, sendo cada uma delas relativa a cada tipo de funcionalidade da aplicação. As secções são as seguintes: Caracterização (do utilizador); Menus; Criar tática; Reproduzir tática e guardá-la; Carregar tática e reproduzi-la; Gerir conta. Foi igualmente adicionado uma secção relativa à avaliação geral, designada por *SUS* (*System Usability Scale*), ou escala de usabilidade do sistema.

No que toca ao tipo de questões colocadas nas secções mencionadas acima, focou-se maioritariamente na experiência do utilizador. Isto é, pretendeu-se obter a sua opinião acerca da interface gráfica, da estética dos vários menus, dos agentes, dos movimentos, da tática e do realismo que cada um representa. Inquiriu-se igualmente o utilizador de algum tipo de dificuldade que possa ter tido, se conseguiu adaptar as suas táticas na aplicação e de sugestões que possa ter para melhorar o seu sistema.

5.2 Participantes

Para este questionário, obteve-se vinte respostas, sendo dez delas vindas de treinadores(as) de basquetebol, outras três de jogadores(as), quatro simpatizantes e três sem qualquer afiliação a este desporto. Sendo que é para treinadores o propósito desta aplicação, é importante obter uma boa afluência por partes dos mesmos. Contudo, a opinião de qualquer utilizador que pretenda utilizar este sistema é relevante e é mais do que bem-vinda.

Participaram doze indivíduos do sexo masculino, sendo as restantes oito do sexo feminino. Metade dos inquiridos encontra-se no intervalo de idades dos 18 aos 25 anos, oito encontram-se entre os 26 e 40 anos e os restantes dois entre os 41 e 65 anos.



Figura 21: Afiliação dos participantes ao basquetebol

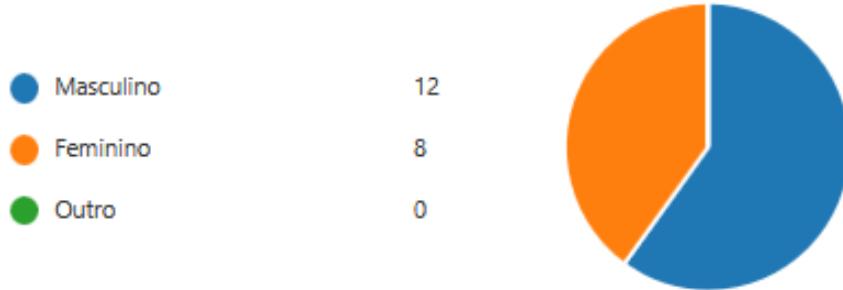


Figura 22: Género dos participantes

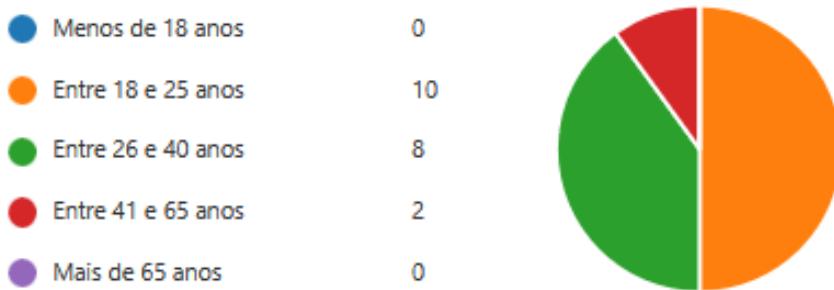


Figura 23: Intervalo de idades dos participantes

5.3 Análise dos Resultados

Todas as questões encontram-se numa escala de 1 a 5, sendo que, da esquerda para a direita, equivale o nível de concordância do utilizador com a questão em concreto (1 - Discordo Totalmente; 5 - Concordo Totalmente). Para efeitos de análise geral e de cálculos, foram consideradas as médias de todas as respostas.

Relativamente ao SUS, este serve para medir a usabilidade de aplicação, avaliando a sua eficiência, a sua eficácia e a satisfação do utilizador. São feitos cálculos de modo a obter um resultado, onde terá de se encontrar acima da média geral de 68 pontos, de modo a ter uma taxa de aceitação positiva. Esta aplicação, após cálculos, obteve um resultado de **70.375** num máximo de 100 pontos. Apesar de ser positivo, apresenta alguns problemas de usabilidade que terão de ser resolvidos no futuro, tal como a escassez de algumas funcionalidades que podem ser úteis. Contudo, apesar dos problemas de usabilidade, o utilizador comum tem a opinião de que é útil a existência desta aplicação (pontuação de 4.45), sendo que maioria considera utilizá-la com frequência (pontuação de 3.7).

No que toca às restantes secções, foram analisadas as médias dos resultados de cada questão. Começando pela secção referente aos menus, o conteúdo dos mesmos foi considerado percepável (pontuação de 4.3). Contudo, podiam ser mais intuitivos (pontuação de 3.5) e esteticamente apelativos (pontuação de 3.65). Na parte das sugestões, foi proposta a alternativa para mudar de sub-menu, como uma opção dentro do próprio menu em vez do atual duplo clique.

Na secção relativa à criação da tática, poucos tiveram dificuldades (pontuação de 1.9), considerando o seu desenvolvimento intuitivo (pontuação de 3.95), mas com melhorias por fazer. Foram feitas com sucesso as adaptações das táticas reais dos treinadores à aplicação (pontuação de 4.1). Obteve-se ainda uma boa pontuação quando se fala da estética e do realismo dos movimentos: pontuação de 4.0 na estética da corrida e do passe, pontuação de 3.7 para o realismo da corrida e pontuação de 3.85 para o realismo do passe. Foi igualmente aceite a adição de "frames" como uma boa prática para a divisão da tática (pontuação de 4.15). No que toca à defesa, maioria considerou ambos os tipos de defesa realistas (pontuação de 4.05 em ambos os tipos).

Contudo, foram feitas algumas sugestões. Primeiro acerca da velocidade dos movimentos, pedindo que fossem aumentadas as suas velocidades. Foram pedidas adições de novos movimentos, como o de um "bloqueio direto" ou de uma pressão de dois defesas contra um atacante e também a adição de uma ferramenta de desenho. Outra sugestão está em fazer "*undo*" do movimento, de modo a ter mais controlo ao longo da criação da tática.

Passando para a parte da reprodução da tática, obteve um "*feedback*" positivo, sendo considerada intuitiva, esteticamente apelativa e realista (pontuações de 4.2, 4.2 e 4.25, respetivamente). Contudo, foram sugeridas a adição da escolha de pontos específicos da tática para visualizar e de várias velocidades (como na reprodução de um vídeo - 0.5x, 1x, 2x, etc.).

No que toca aos separadores relativos a guardar e carregar a tática e das notas, teve uma pontuação mediana referente ao seu aspeto visual (pontuação de 3.3 para as notas, 3.5 para guardar a tática e 3.4 para carregar a tática). Outra nota mediana está no carregamento da tática e na eficiência do sistema de ficheiros (pontuações de 3.6 e 3.65, respetivamente), sendo pouco intuitivo, visto que para carregar uma tática é necessária saber de cor o nome do ficheiro. Foi sugerida a adição de uma lista com todas as táticas guardadas pelo utilizador. Contudo, os separadores para guardar a tática e das notas foram considerados simples e intuitivos (pontuações de 4.15 e 4.05, respetivamente).

No geral, tendo em conta o "*feedback*" por parte do utilizador, é considerada uma aplicação com potencial e, sendo feitas as melhorias propostas, pode tornar-se numa aplicação muito útil para o contexto em questão.

6 Conclusões e Trabalho Futuro

Finalizado este projeto, pode-se concluir que foram cumpridos os objetivos traçados no seu início. Entre eles está o desenvolvimento de uma aplicação Android que permite a criação de uma tática de basquetebol com uma equipa defensiva, cujos movimentos se adaptam automaticamente e a inclusão de um sistema de autenticação e de armazenamento que permite guardar as táticas criadas num espaço próprio para o utilizador.

Como trabalho futuro, serão estendidas e adicionadas funcionalidades ao projeto, tais como a adição de uma forma de *login* anónimo, de forma a que o utilizador possa experimentar a aplicação, outros tipos de defesa zona, mais tipos de movimentos atacantes (como um bloqueio direto), a possibilidade de modificar a tática, *frame* a *frame*, todas as sugestões feitas pelos utilizadores que responderam ao teste de usabilidade e outras futuras. Do ponto de vista adaptativo, seria apropriado existir a aplicação para ambientes iOS, visto que ampliaria o grupo de utilizadores.

Referências

- [1] <https://play.google.com/store/apps/details?id=com.coachboard>
- [2] <https://play.google.com/store/apps/details?id=com.bluelinden.coachboardbasket>
- [3] <https://play.google.com/store/apps/details?id=com.jenda.basketballboard>
- [4] <https://github.com/libgdx.gdx-ai/wiki/Steering-Behaviors>
- [5] <https://math.stackexchange.com/questions/1965002/calculate-position-based-on-angles-between-three-known-points>
- [6] <http://paulbourke.net/geometry/circlesphere/>
- [7] https://www.w3schools.com/java/java_intro.asp
- [8] <https://www.android.com/>
- [9] <https://developer.android.com/studio>
- [10] <https://libgdx.badlogicgames.com/>
- [11] <https://firebase.google.com/>
- [12] <https://github.com/mk-5/gdx-firebase>
- [13] <https://github.com/mk-5/gdx-firebase/wiki/Examples>
- [14] <https://github.com/fangyidong/json-simple>
- [15] https://rosettacode.org/wiki/Circles_of_given_radius_through_two_points#Java
- [16] <https://www.youtube.com/watch?v=pnKcuJQT31A>

Bibliografia

- [1] GDX-FireApp:
<https://github.com/mk-5/gdx-fireapp>
- [2] Screens:
<http://www.pixnbgames.com/blog/libgdx/how-to-manage-screens-in-libgdx/>
- [3] Skins:
<https://ray3k.wordpress.com/software/skin-composer-for-libgdx/>
- [4] Comportamentos Steering:
<https://github.com/libgdx.gdx-ai/wiki/Steering-Behaviors>
- [5] Escrever e ler em JSON:
<https://github.com/libgdx/libgdx/wiki/Reading-and-writing-JSON>
- [6] Colisões:
<https://badlogicgames.com/forum/viewtopic.php?f=11&t=1352>
- [7] Firebase:
<https://firebase.google.com/>
- [8] SUS:
[https://brasil.uxdesign.cc/guia-como-medir-a-usabilidade-de-produtos\\-com-system-usability-scale-sus-e08f4361d9db](https://brasil.uxdesign.cc/guia-como-medir-a-usabilidade-de-produtos-com-system-usability-scale-sus-e08f4361d9db)
- [9] Documentação disponibilizada pelos docentes da UC de MCG
- [10] Documentação disponibilizada pelos docentes da UC de MSSN
- [11] Documentação disponibilizada pelos docentes da UC de FSO
- [12] Documentação disponibilizada pelos docentes da UC de IASA
- [13] Documentação disponibilizada pelos docentes da UC de DAM
- [14] Documentação disponibilizada pelos docentes da UC de PRJ

A UML

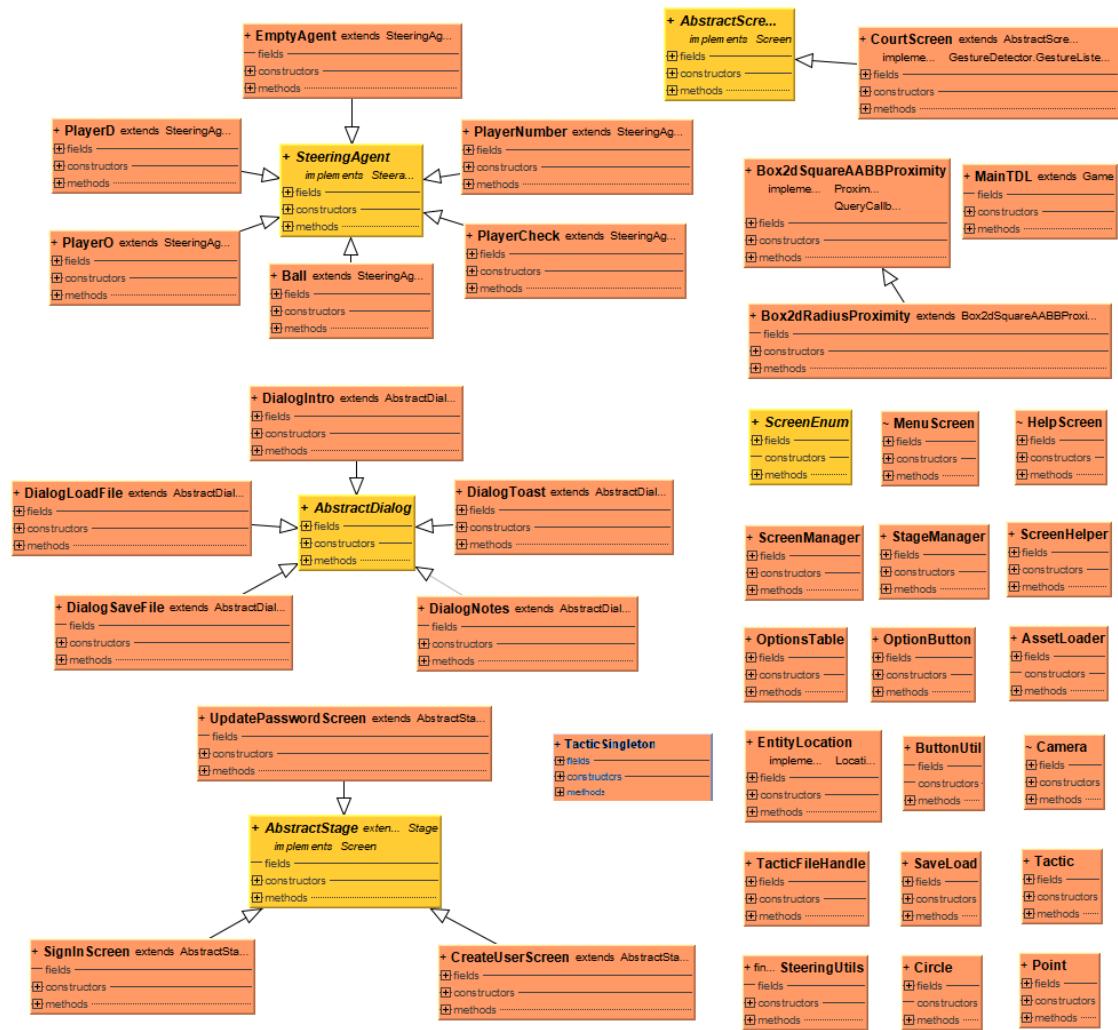


Figura 24: Diagrama UML geral da aplicação

Unidade Curricular de Projeto

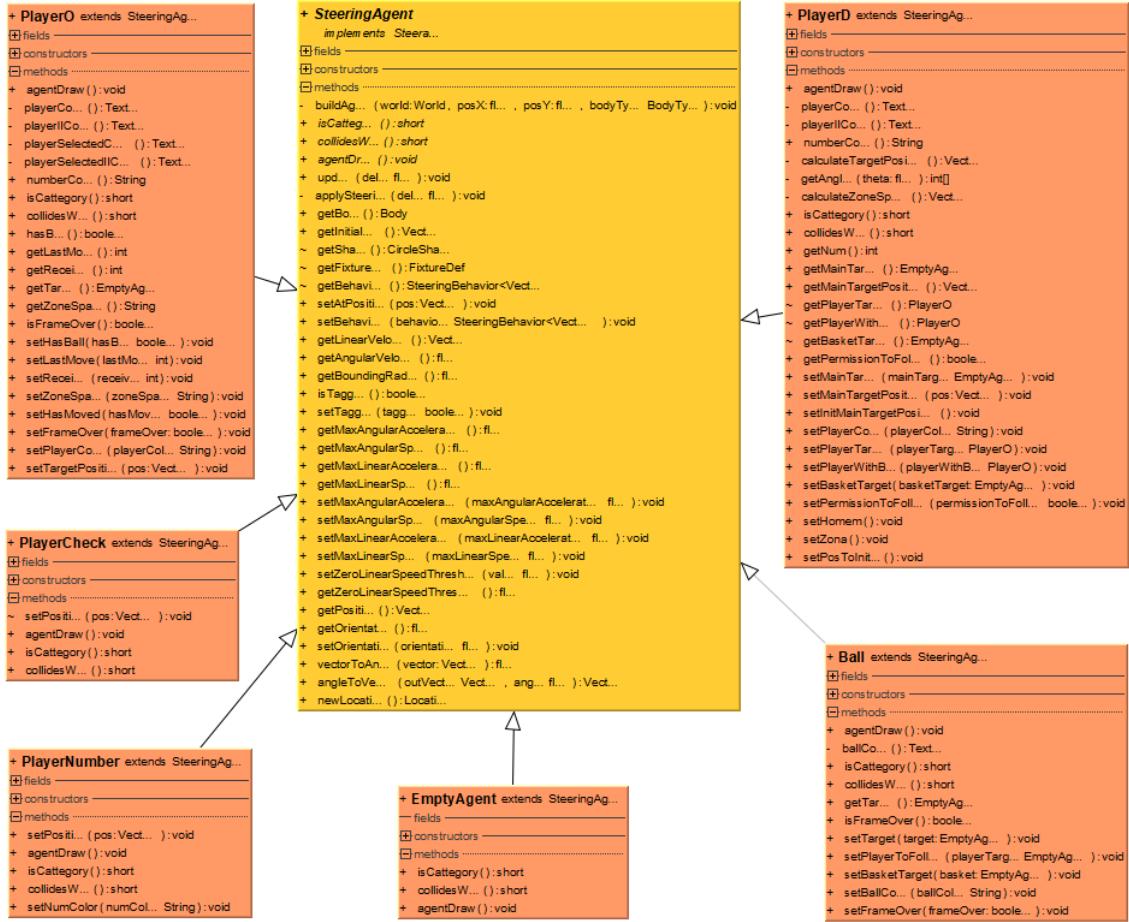


Figura 25: Diagrama UML do package 'com.gdx.tdl.map.ags'

Unidade Curricular de Projeto

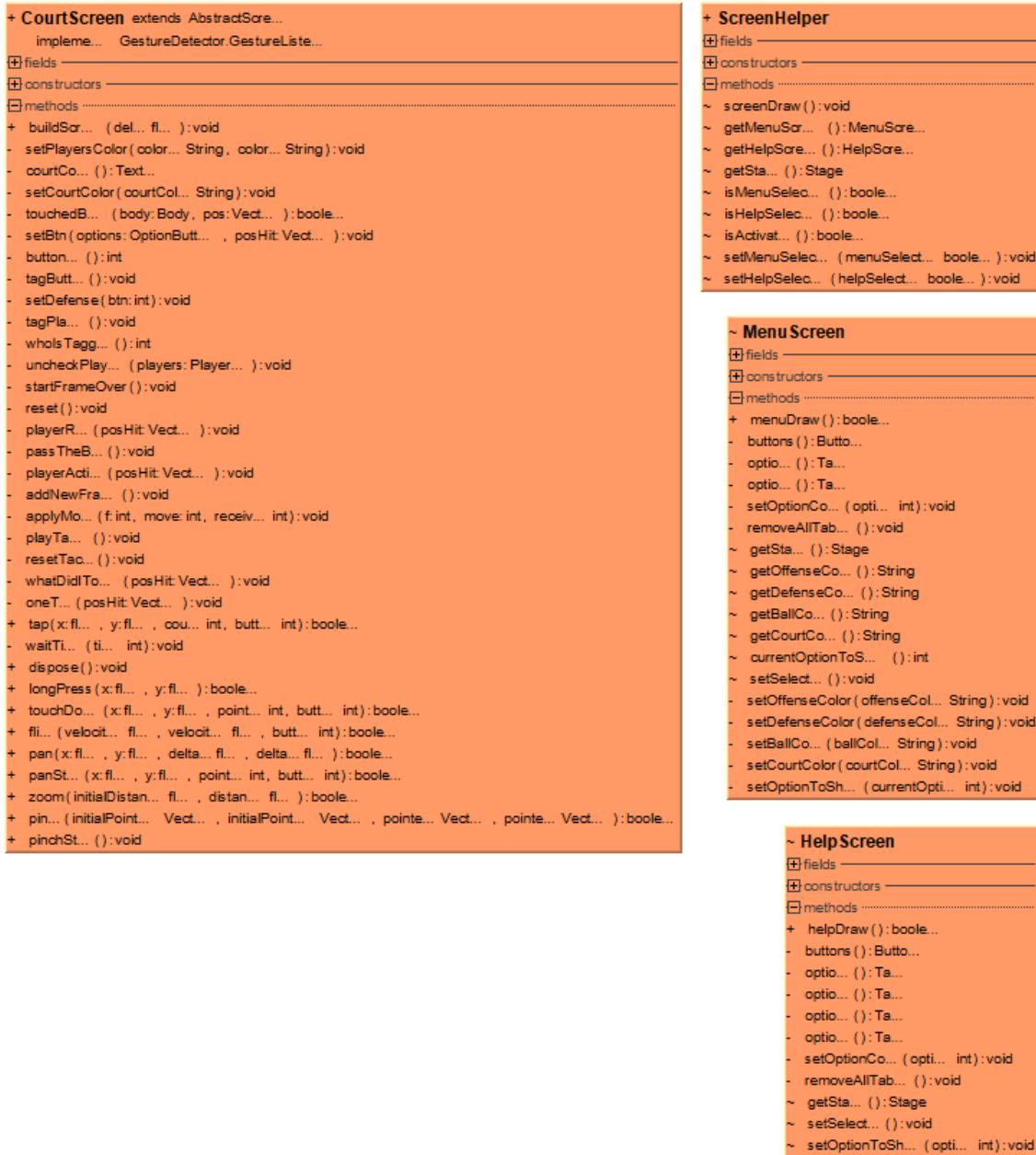


Figura 26: Diagrama UML do package 'com.gdx.tdl.map.scr'

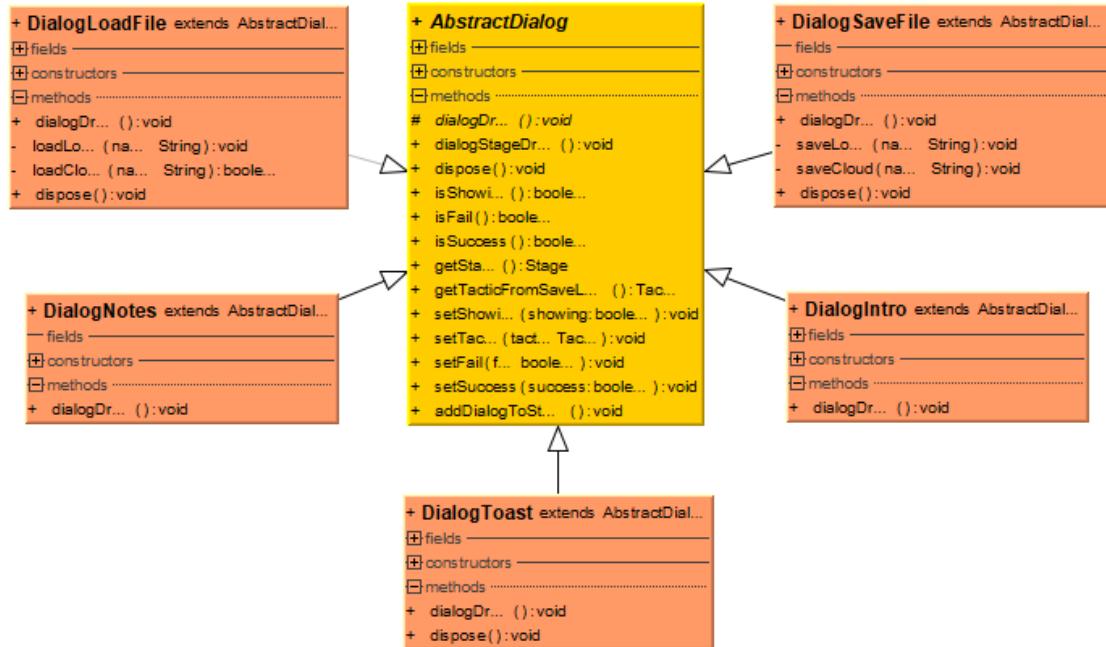


Figura 27: Diagrama UML do package 'com.gdx.tdl.map.dlg'



Figura 28: Diagrama UML do package 'com.gdx.tdl.map.tct'

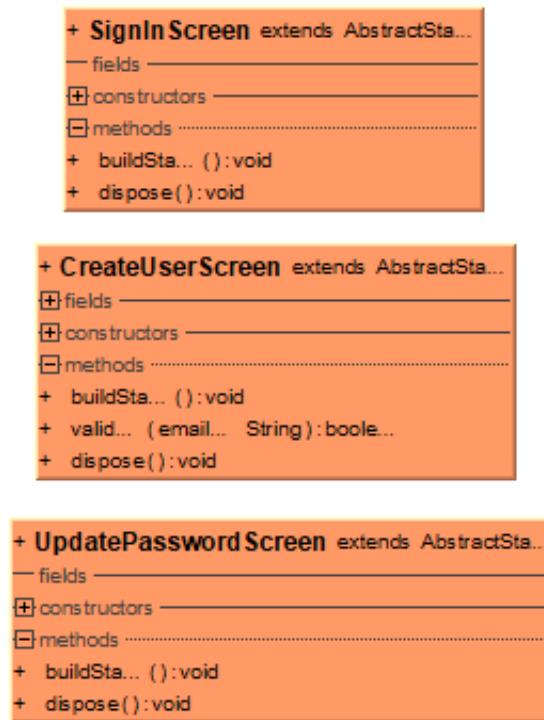


Figura 29: Diagrama UML do package 'com.gdx.tdl.sgn.scr'

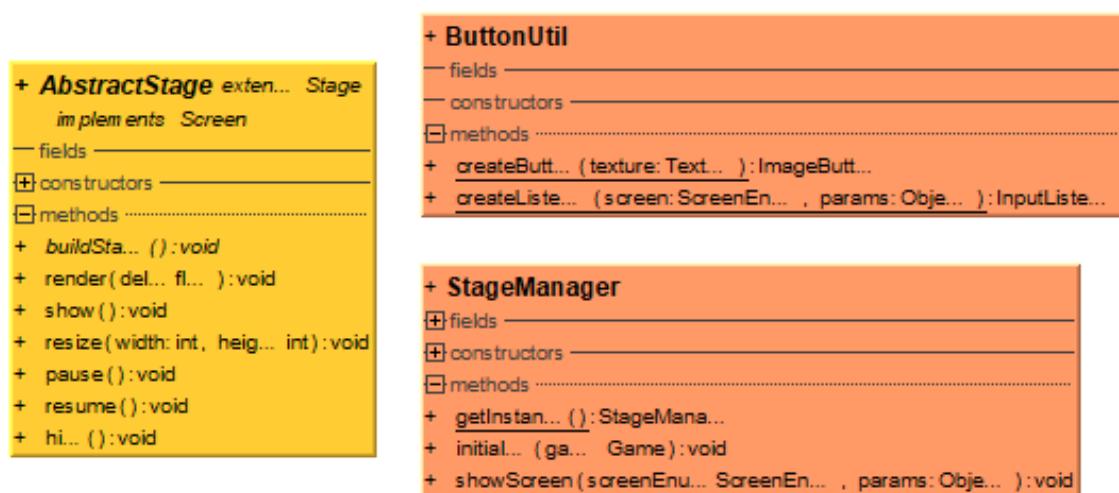


Figura 30: Diagrama UML do package 'com.gdx.tdl.util.sgn'

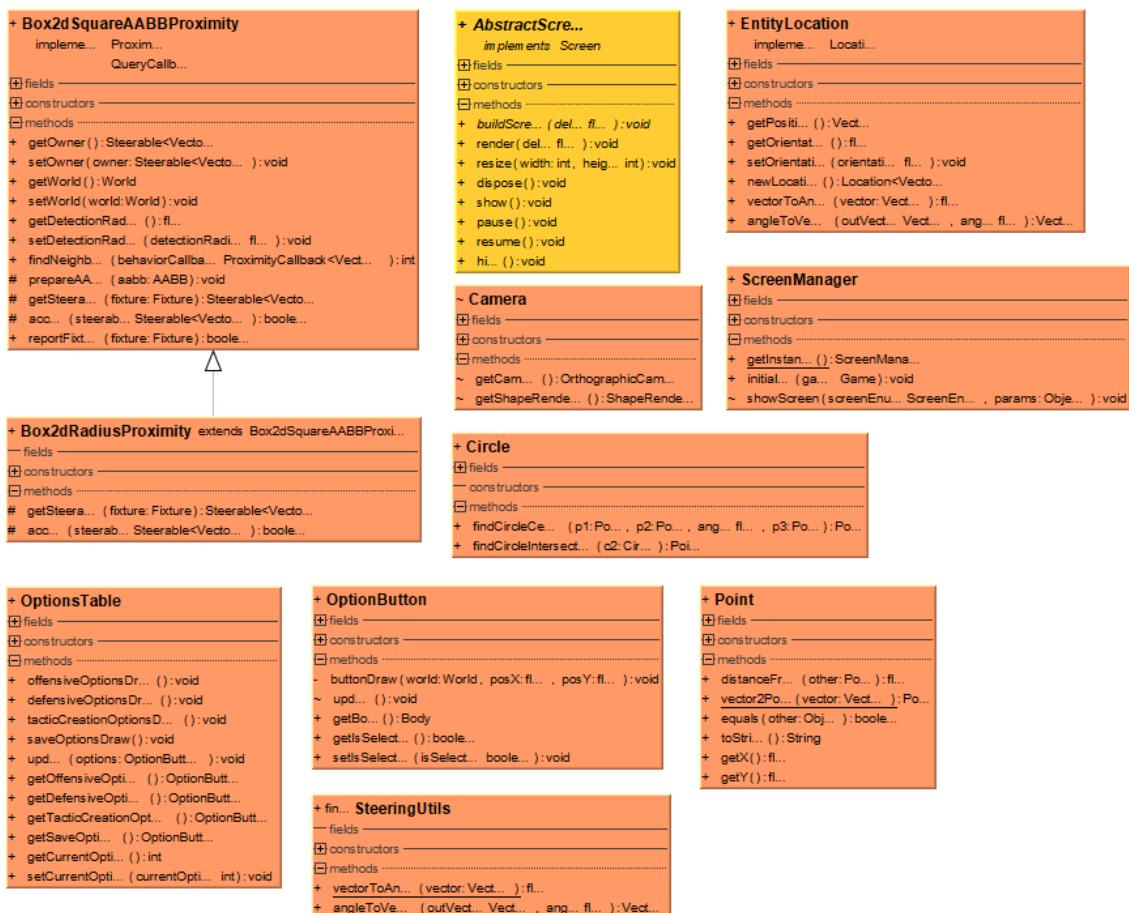


Figura 31: Diagrama UML do package 'com.gdx.tdl.util.map'