
Introdução ao Desenvolvimento de Jogos em Android

Introdução ao Desenvolvimento de Jogos em Android

Ricardo Queirós e Alberto Simões



FCA – Editora de Informática, Lda
R. D. Estefânia, 183-1.º Esq.º – 1000-154 Lisboa
Tel.: 213 173 252 (Departamento Editorial)
E-mail: fca@fca.pt url: www.fca.pt

DISTRIBUIÇÃO



Lidel – edições técnicas, lda

SEDE:

R. D. Estefânia, 183, R/C Dto., 1049-057 LISBOA

Internet: 21 354 14 18 – livrarialx@lidel.pt / Revenda: 21 351 14 43 – revenda@lidel.pt

Formação/Marketing: 21 351 14 48 – formacao@lidel.pt / marketing@lidel.pt

Ensino Línguas/Exportação: 21 351 14 42 – depinternacional@lidel.pt

Fax: 21 352 26 84

LIVRARIA:

Av. Praia da Vitória, 14A – 1000-247 LISBOA

Tel.: 21 354 14 18 – e-mail: livrarialx@lidel.pt

Copyright © junho 2014

FCA – Editora de Informática, Lda.

ISBN: 978-972-722-763-1

Capa: Emília Calçada

Impressão e acabamento:

Depósito Legal N.º

Livro segundo o Novo Acordo Ortográfico

Os nomes comerciais referenciados neste livro têm patente registada

Marcas Registadas de FCA – Editora de Informática, Lda. –



Depressa & Bem

Este pictograma merece uma explicação. O seu propósito é alertar o leitor para a ameaça que representa para o futuro da escrita, nomeadamente na área da edição técnica e universitária, o desenvolvimento massivo da fotocópia. O Código do Direito de Autor estabelece que é crime punido por lei, a fotocópia sem autorização dos proprietários do *copyright*. No entanto, esta prática generalizou-se sobretudo no ensino superior, provocando uma queda substancial na compra de livros técnicos. Assim, num país em que a literatura técnica é tão escassa, os autores não sentem motivação para criar obras inéditas e fazê-las publicar, ficando os leitores desabilitados de ter bibliografia em português. Lembramos, portanto, que é expressamente proibida a reprodução, no todo ou em parte, da presente obra sem autorização da editora.



Aos meus eternos amores: Márcia e Gabriela,

Ricardo

À minha Mãe,

Alberto

OS AUTORES

Ricardo Queirós (ricardo.queiros@gmail.com) – é Doutorado em Ciências de Computadores pela Faculdade de Ciências da Universidade do Porto (FCUP). Exerce a sua atividade como docente na Escola Superior de Estudos Industriais e de Gestão (ESEIG), onde é responsável por disciplinas na área das linguagens e técnicas de programação e bases de dados. Paralelamente, desenvolve atividade científica na área da interoperabilidade entre sistemas de e-learning. É membro fundador do grupo de investigação Knowledge Management, Interactive and Learning Technologies (KMILT) na ESEIG e membro efetivo do Center for Research in Advanced Computing Systems (CRACS), uma unidade de investigação do laboratório Associado INESC-TEC Porto.

Alberto Simões (albertovski@gmail.com) – é doutorado em Inteligência Artificial, ramo de Processamento de Linguagem Natural, pela Universidade do Minho. Tem vindo a exercer a sua atividade como docente no Instituto Politécnico do Cávado e do Ave (IPCA), onde é responsável por disciplinas na área do desenvolvimento de jogos digitais, bem como no Departamento de Informática da Universidade do Minho (DIUM), onde tem colaborado na lecionação de diversas disciplinas na área da programação e processamento de linguagens. A sua atividade científica tem focado o processamento de linguagem natural, sendo membro efetivo do Centro de Estudos Humanísticos da Universidade do Minho (CEHUM) e membro colaborador do Centro de Ciências e Tecnologias da Computação (CCTC) da Universidade do Minho.

AGRADECIMENTOS.....	VII
1 INTRODUÇÃO AO DESENVOLVIMENTO EM ANDROID.....	1
1.1 Introdução	1
1.2 Android 5 (Lollipop)	2
1.3 Android Studio	6
1.4 A minha primeira aplicação	9
141 Criação de projeto.....	10
142 Estrutura de ficheiros.....	15
143 Interface Gráfica.....	17
144 Criação de um <i>Android Virtual Device</i>	18
145 Execução da aplicação.....	25
1.5 Componentes principais.....	27
151 Activity.....	28
152 Service	31
153 BroadcastReceiver	34
154 ContentProvider	39
2 API DA GOOGLE E MOTORES DE JOGOS.....	43
2.1 Introdução	43
2.2 API da Google.....	44
221 Gráficos	44
222 <i>Input</i>	64
223 Áudio.....	80
224 <i>Input/Output</i>	84
2.3 Motores de Jogo	100
231 Componentes	100
232 Exemplos.....	102
233 Reflexão final.....	115
3 API 2D PARA ANDROID.....	117
3.1 Introdução	117
3.2 Tabuleiro.....	118
321 Classe <i>Posicao</i>	118
322 Classe <i>Tabuleiro</i> : representação base.....	119
323 Classe <i>Tabuleiro</i> : Cálculo de Jogadas.....	121
3.3 Inteligência Artificial.....	125
331 <i>Minimax</i> : Intuição	126
332 <i>Minimax</i> : Algoritmo.....	127
333 Função de Avaliação	129
3.4 Interface de Jogo	132

3.5	Interface Auxiliar.....	143
351	Ecrã Inicial	143
352	Ecrã de Opções.....	148
4	LIBGDX.....	153
4.1	Introdução à <i>Framework</i> Libgdx.....	153
4.1.1	Criação de um projeto libgdx.....	154
4.1.2	Estrutura de um projeto libgdx	155
4.1.3	Módulos	157
4.1.4	Ciclo de vida de uma aplicação libgdx.....	165
4.1.5	Classes de Arranque.....	167
4.1.6	API 3D.....	168
4.2	Jogo <i>Balloid</i>	175
4.2.1	Arquitetura base do jogo	176
4.2.2	Criação de modelos 3D	177
4.2.3	Estado do Jogo	180
4.2.4	Objeto de jogo	180
4.2.5	Lógica de jogo	181
5	UNITY 3D PARA ANDROID	195
5.1	Unity 3D.....	195
5.2	Projeto Unity 3D para Android	197
5.3	Cena base	200
5.4	Corpos Rígidos.....	202
5.5	Interação com o Utilizador	205
5.6	Interface com o Utilizador.....	210
5.7	Uso de Cenas Distintas	215
5.8	Prefabricados.....	219
5.9	Materiais	222
5.10	Som.....	224
5.11	Disponibilização	226
6	GOOGLE PLAY, SERVIÇOS E PUBLICAÇÃO	227
6.1	Google Play Services	227
6.1.1	Configuração do Google Play Services.....	228
6.1.2	Classe <code>GoogleApiClient</code>	229
6.1.3	Google Play Games Services.....	231
6.2	Integração do GPGS no jogo <i>Othelloid</i>	232
6.2.1	Configuração na Google Play Developer Console.....	232
6.2.2	Implementação.....	236
6.2.3	Teste e publicação dos serviços	257
6.3	Publicação do jogo na Google Play	259
6.3.1	Visão geral da publicação.....	259

6.32	Preparação da aplicação	260
6.33	Distribuição da aplicação	265
GLOSSÁRIO DE TERMOS		271
PORTUGUÊS EUROPEU / PORTUGUÊS DO BRASIL		271
ÍNDICE REMISSIVO		273

AGRADECIMENTOS

Em primeiro lugar, quero agradecer à minha **família**, em especial à minha esposa Márcia e filha Gabriela, pelo apoio e paciência que demonstraram durante todo o processo de escrita da obra. Uma palavra de apreço também ao meu amigo **Alberto** pela disponibilidade, camaradagem e competência.

Ricardo Queirós

Ao meu amigo **Ricardo** por me desafiar nesta empreitada e, claro, à minha família pelo constante apoio.

Alberto Simões

Agradecemos também à **equipa da FCA**, Paula Martins, Laura Faia, Sandra Correia e Eng.^o Frederico Annes, toda a simpatia e apoio prestado no decorrer desta aventura editorial.



INTRODUÇÃO AO DESENVOLVIMENTO EM ANDROID

Este capítulo faz uma breve introdução ao sistema operativo Android, apresentando a sua nova versão – o Android 5 (Lollipop) – lançada em novembro de 2014. De seguida, enumeram-se os requisitos de desenvolvimento para se começar a criar aplicações para dispositivos Android. Neste contexto, é apresentado o ambiente integrado para desenvolvimento chamado Android Studio. O editor da Google é usado para criar uma aplicação mínima através da qual são explicados todos os conceitos-chave da ferramenta e que servirá de base a todas as aplicações incluídas neste livro. O capítulo é finalizado com uma resenha dos principais componentes de uma aplicação Android.

1.1 INTRODUÇÃO

Os dispositivos inteligentes estão, cada vez mais, enraizados nas nossas vidas. Em casa ou no trabalho, a correr ou no carro, todos interagimos com esta nova realidade tecnológica. Na verdade, temos assistido nas últimas décadas a uma revolução digital, em que os dispositivos móveis assumem um papel-chave, potenciando o acesso à informação e à comunicação à escala global. Atualmente, o mercado móvel é dominado por duas categorias de dispositivos móveis: os *smartphones* e os *tablets*.

De acordo com a empresa de análise de mercado *International Data Corporation* (IDC), as previsões apontam nos próximos anos, relativamente ao mercado móvel, para a hegemonia dos *smartphones* e para o aparecimento de novos *players*, tais como os *phablets* (dispositivos entre 5 a 7 polegadas) e os dispositivos *wearable* (relógios inteligentes, aparelhos de *fitness*, etc.). Relativamente aos *smartphones*, de acordo com os resultados apresentados pela *Strategy Analytics*, cerca de 85% dos dispositivos vendidos, à escala global, no segundo trimestre de 2014 têm como sistema operativo (SO) o Android.

O **Android** é um SO para dispositivos móveis baseado em Linux e desenvolvido pela Open Handset Alliance, liderada pela Google e por outras empresas. O primeiro telemóvel comercialmente disponível a usar o Android (versão 1.0) foi o T-Mobile's G1 HTC Dream, lançado a 22 de outubro de 2008. Posteriormente, foram várias as versões do SO a surgir identificadas com nomes de bolos (em inglês) e seguindo uma lógica alfabética.

A Tabela 1.1 enumera as principais versões da plataforma Android, incluindo datas de lançamento, nomes de código, níveis da API e percentagens de uso.

VERSÃO	DATA DE LANÇAMENTO	NOME DE CÓDIGO	NÍVEL API	% DE USO ¹
2.2	maio de 2010	Froyo	8	0,4
2.3.3-2.3.7	fevereiro de 2011	Gingerbread	10	6,4
4.0.3-4.0.4	dezembro de 2011	Ice Cream Sandwich	15	5,7
4.1.x	julho de 2012	Jelly Bean	16	16,5
4.2.x	novembro de 2012		17	18,6
4.3	julho de 2013		18	5,6
4.4	outubro de 2013	KitKat	19	41,4
5.0	novembro de 2014	Lollipop	21	5,0
5.1	março de 2015		22	0,4

TABELA 1.1 – Histórico das versões do SO Android

1.2 ANDROID 5 (LOLLIPOP)

Neste contexto de claro domínio, a Google lançou, em novembro de 2014, uma nova versão do seu sistema operativo Android, batizando-a com o nome **Lollipop**, que inclui cinco mil novas API e onde se destacam as seguintes características:

☉ **Interface com o utilizador:**

→ *Material Design* – novo paradigma gráfico cuja missão é unificar os aspetos visuais e funcionais dos produtos da Google em diferentes tipos de dispositivos. A Google apostou num visual *flat* que transmite a ideia de “papel inteligente”, com um *design* mais apelativo, com animações mais realistas e naturais e com cores mais vibrantes (Figura 1.1). Com a implementação do *Material Design*, a Google vem garantir a flexibilidade e a harmonia visual do sistema operativo em qualquer dispositivo Android 5, seja um *smartphone*, *tablet*, *smartwatch* ou *smartTV*. Para além de melhorar a experiência do utilizador, o *Material Design* facilitará também a vida do programador: se as suas diretrizes forem seguidas, desenvolver uma *app* para qualquer tipo de dispositivo será uma tarefa uniforme e mais trivial;

¹ Baseada nos acessos à loja *online* Google Play durante uma semana até 6 de abril de 2015.

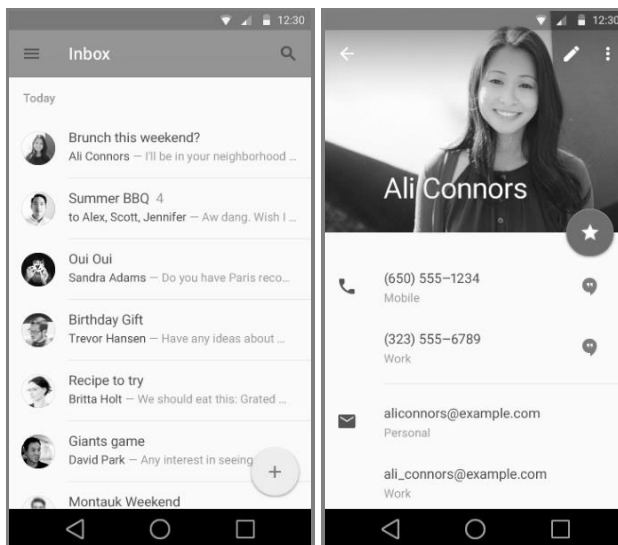


FIGURA 1.1 – Material Design

- **Notificações** – em versões anteriores, as notificações eram exibidas na barra de estado. Com o Android Lollipop, as notificações surgem no formato *pop-up*, não invasivas, e sem necessidade de abandonar a aplicação sempre que se tiver de interagir com elas (Figura 1.2). Pode-se também agora interagir com as notificações diretamente a partir do *lockscreen*, havendo opções para priorizar e/ou gerir a sua visualização (por exemplo, ocultar o conteúdo de mensagens recebidas).



FIGURA 1.2 – Notificações

⊙ **Performance:**

- **Android Runtime (ART)** – o Android 5 é executado exclusivamente no ART baseado numa compilação *ahead-of-time* (AOT), em tempo de instalação. Essa técnica elimina a sobrecarga de processamento associado à compilação *just-in-time* (JIT) usada pela máquina virtual *Dalvik*, melhorando o desempenho do sistema;
- **Suporte de 64 bits** – suporte para processadores de 64 bits. Um *smartphone* a 64 bits consegue lidar com 64 bits de dados por cada ciclo (em detrimento dos 32 bits atuais), o que reduz o número de operações de transferência e processamento de dados. Além do espaço de endereçamento, o outro grande benefício está relacionado com a precisão, inerente, por exemplo, ao tamanho dos registos (inteiros e de vírgula flutuante);
- **Projeto Volta** – com o projeto Volta, a vida útil de uma carga de bateria aumenta consideravelmente, através de uma utilização mais eficaz de vários componentes: o *Battery Saver* diminui o consumo de energia, desligando serviços de aplicações desnecessárias, diminuindo o brilho do ecrã e, por exemplo, desligando o *wi-fi* quando não existirem redes próximas; o *Battery Historian* permite visualizar um histórico do consumo de energia, evidenciando os momentos em que esse consumo é excessivo; a API `JobScheduler` permite que os programadores escolham os momentos exatos em que determinadas ações serão realizadas nas suas aplicações;
- **OpenGL ES 3.1** – nova versão da API que inclui novos *shaders* e texturas que vão tornar os jogos de computador mais sofisticados e realistas.

⊙ **Acesso:**

- **Múltiplos perfis** – possibilidade de acesso ao dispositivo através de vários perfis (Figura 1.3). Cada perfil tem associado um conjunto de permissões. O modo convidado (*guest*) é um perfil limitado que é adequado, por exemplo, quando queremos emprestar o telemóvel a uma pessoa sem que esta possa aceder aos seus dados;

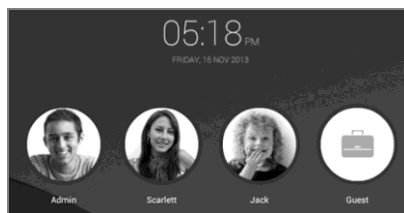


FIGURA 1.3 – Múltiplos perfis

- **Documentos simultâneos** – em versões anteriores, o ecrã **Recents** só poderia exibir uma tarefa para cada aplicação com a qual o utilizador tivesse interagido recentemente. Agora várias instâncias da mesma atividade contendo diversos documentos podem aparecer como tarefas no ecrã **Overview** (Figura 1.4). Exemplos de tais tarefas simultâneas podem incluir abas abertas de um *browser*, documentos de uma aplicação de produtividade ou *chats* de uma aplicação de mensagens.

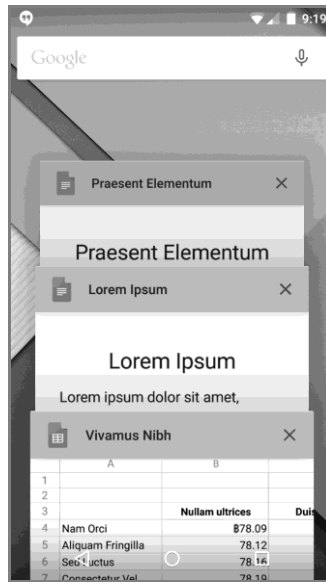


FIGURA 1.4 – Ecrã Overview

⊙ Segurança:

- Sistema de criptografia ativado por padrão e novas políticas de segurança *SELinux* que garantem maior proteção contra ataques por *malwares* e outras vulnerabilidades;
- **Android Smart Lock** – flexibilidade na gestão do ecrã de bloqueio sempre que estiver com um dispositivo com suporte *Bluetooth/NFC*, como um dispositivo *wearable* (por exemplo, um *smartwatch*), ou no carro.

⊙ Multimédia:

- Suporte do formato RAW que garante melhor qualidade de imagem devido à menor taxa de compressão quando comparado com o formato JPEG;
- Suporte do padrão de compressão de vídeo *High Efficiency Video Coding* (HEVC) para vídeos de 4K a 10-bit, com maior qualidade e fluidez;

- Integração com o Android TV;
 - Suporte de *audio* via porta USB.
- ⊙ **Conectividade:**
- **Bluetooth Low Energy (BLE)** – os dispositivos Android podem agora funcionar no modo periférico BLE;
 - **Multi-networking** – novas API de multi-rede que permitem que a aplicação procure dinamicamente por redes disponíveis de acordo com capacidades específicas e estabeleça uma conexão com elas. Esta funcionalidade é útil quando a aplicação requer uma rede especializada, como, por exemplo, a rede celular.

1.3 ANDROID STUDIO

As peças-chave para o desenvolvimento de aplicações Android são o SDK do Android (bibliotecas e ferramentas de desenvolvimento necessárias para construir e testar aplicações) e o IDE (editor de código integrado).

Em maio de 2013, na conferência anual da Google para programadores (Google I/O), foi anunciado um novo ambiente exclusivo para o desenvolvimento de aplicações Android, chamado **Android Studio**. Neste livro iremos usar este editor.



Outra alternativa seria usar o *plugin Android Developer Tools (ADT)* para o IDE Eclipse. Contudo, esteja ciente de que este não está mais em desenvolvimento ativo e que o Android Studio é agora o IDE oficial para o desenvolvimento de aplicações Android. Para obter ajuda na migração para o Android Studio, acesse ao *link* <http://developer.android.com/sdk/installing/migrate.html>.

O Android Studio está disponível gratuitamente (versão 1.2 – abril de 2015), através de uma licença Apache 2.0, para Windows, Linux e Mac OS X. Os requisitos que o computador deve ter para a sua instalação são agrupados por SO no Quadro 1.1.

WINDOWS	LINUX	MAC OS X
Microsoft Windows 8/7/Vista/2003 (32 ou 64 bits)	Desktop GNOME ou KDE Biblioteca GNU C (glibc) 2.11 ou superior	Mac OS X 10.8.5 ou superior, até 10.10 (Yosemite)
Mínimo de 2 GB RAM (recomendado 4 GB)		
Pelo menos 400 MB de espaço em disco para o Android Studio		
Pelo menos mais 1 GB para o SDK do Android, imagens dos emuladores e <i>caches</i>		
Resolução do ecrã mínima de 1280x800		
<i>Java Development Kit (JDK) 7</i>		

QUADRO 1.1 – Requisitos para a instalação do Android Studio



Antes de configurar o Android Studio, certifique-se de que instalou o Java. Para o desenvolvimento em Android 5 é necessário a *Java Development Kit (JDK)* versão 7 ou superior². Para verificar se tem o JDK instalado (e qual a versão), abra um terminal e digite **javac -version**.

Pode descarregar o Android Studio e o SDK para Android separadamente. Contudo, existe a opção de instalar ambos na forma de um pacote único. Esta opção instala o IDE Android Studio, o SDK do Android e as API e imagens do emulador para a versão 5.0 do Android.

Para instalar o pacote Android Studio no Windows siga os próximos passos:

- 1) Aceda à página oficial do Android Studio³ e clique no botão **Download Android Studio for Windows**. Aceite depois os termos e condições para iniciar o *download*.
- 2) Execute o ficheiro **android-studio-bundle-<version>-windows.exe** e siga as instruções para a instalação do Android Studio e do SDK (Figura 1.5).

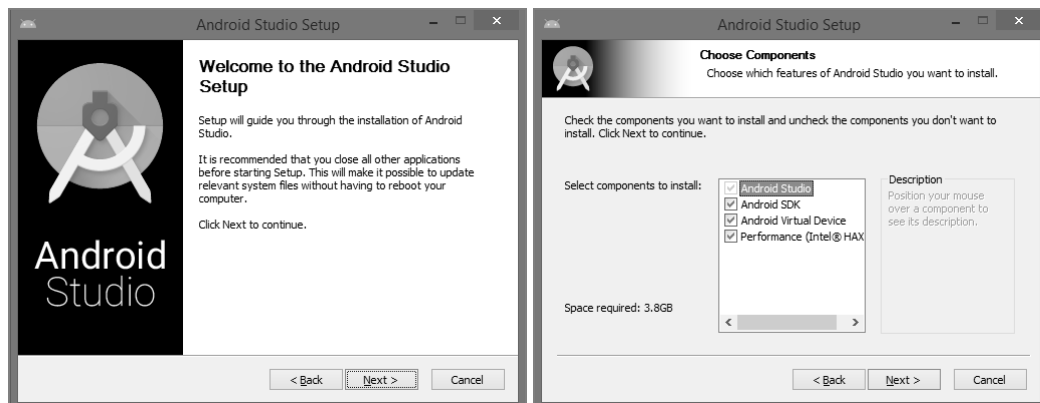


FIGURA 1.5 – Instalação do Android Studio



As ferramentas individuais e outros pacotes SDK são guardados fora da pasta da aplicação Android Studio. Por exemplo: `\Users\<user>\sdk\`.

- 3) Após a instalação do Android Studio, configure o Android SDK. O SDK separa ferramentas, plataformas e outros componentes em pacotes que se podem descarregar individualmente, usando a ferramenta **SDK Manager** (Figura 1.6). Por omissão, o SDK não inclui todos os componentes

² Link: <http://www.oracle.com/technetwork/java/javase/downloads/index.html>

³ Link: <http://developer.android.com/sdk>

necessários. Para aceder ao **SDK Manager**, inicie o Android Studio e escolha à opção **Tools → Android → SDK Manager**.

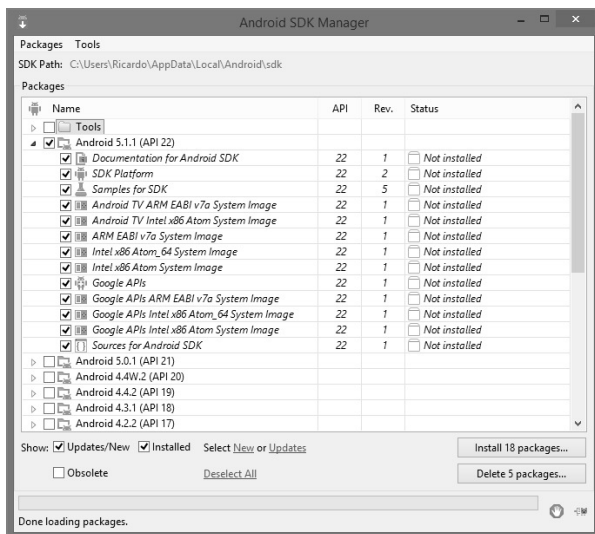


FIGURA 1.6 – SDK Manager

No mínimo, ao configurar o Android SDK, deve descarregar e instalar as mais recentes ferramentas, plataformas e API. Muitas delas já estão selecionadas por omissão, mas outras poderão não estar. A Tabela 1.2 lista todos os pacotes a instalar.

PASTA	PACOTE	DESCRIÇÃO
Tools	Android SDK Tools	Conjunto completo de ferramentas de desenvolvimento e <i>debugging</i> para Android (por exemplo, AVD Manager).
	Android SDK Platform-tools	Componentes com funcionalidades da mais recente plataforma Android.
	Android SDK Build-tools (última versão)	Componente do SDK para a construção de código (por exemplo, <i>zipalign</i>).
Android X.X (última versão)	SDK Platform	Conjunto de API para a versão Android.
	ARM EABI v7a System Image	Para testar a aplicação num emulador, deve-se importar pelo menos uma imagem do sistema para a versão Android respetiva. Cada versão da plataforma inclui imagens do sistema que suportam uma arquitetura de processador (por exemplo, ARM EABI, Intel x86, MIPS). Tipicamente, é incluída uma imagem do sistema que contém as API da Google.

PASTA	PACOTE	DESCRIÇÃO
Extras	Android Support Repository Android Support Library	A biblioteca Android Support Library fornece API compatíveis com a maioria das versões do Android e é necessária para Android Wear, Android TV, Google Cast. Fornece também suporte para os seguintes elementos estruturais gráficos: Navigation Drawer, Swipe views e Action bar.
	Google Repository Google Play Services	As API do Google Play Services fornecem uma variedade de recursos para as aplicações Android, tais como autenticação Google+, Maps, Cast, Games, entre outros. Nota: as API do Google Play Services estão disponíveis em todos os dispositivos com a Google Play Store. Para usar essas API no emulador Android, deve instalar a imagem do sistema com as Google API.

TABELA 1.2 – Pacotes a instalar através do SDK Manager

- 4) Instale os pacotes selecionados executando os seguintes passos:
- Clique no botão **Install X packages...**;
 - Na janela seguinte, dê um duplo clique em cada nome do pacote para aceitar o respetivo contrato de licença;
 - Clique no botão **Install**.



A instalação em ambiente Mac OS X é semelhante e feita da forma habitual para este sistema operativo, arrastando a aplicação para a pasta **Aplicações (Applications)**. Na primeira vez que a aplicação é executada deverá usar-se a opção **Configure → SDK Manager**. Aí deverão ser seguidas as instruções da versão Windows, instalando os pacotes previamente selecionados.

Com os pacotes anteriores instalados, está pronto para iniciar o desenvolvimento de aplicações para dispositivos Android. À medida que novas ferramentas e outras API se tornam disponíveis, basta iniciar o **SDK Manager** para descarregar os novos pacotes para o SDK.

1.4 A MINHA PRIMEIRA APLICAÇÃO

Para ilustrar o processo de desenvolvimento de uma aplicação Android no Android Studio, apresenta-se a criação de uma aplicação bastante simples que exhibe no ecrã o texto “Bem-vindo ao Android Studio!”.

1.4.1 CRIAÇÃO DE PROJETO

Para desenvolver uma aplicação no Android Studio tem de criar um projeto. Execute os seguintes passos:

- 1) Inicie o Android Studio (Figura 1.7).



FIGURA 1.7 – Arranque do Android Studio

- 2) Se não tiver nenhum projeto aberto, o Android Studio exibe a janela **Welcome to Android Studio** (Figura 1.8); clique na opção **Start a new Android Studio project**.

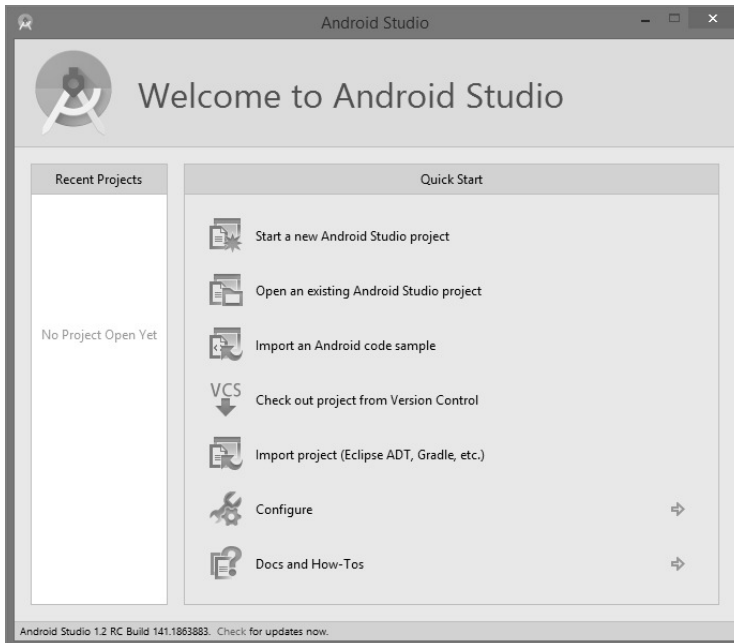


FIGURA 1.8 – Janela Welcome to Android Studio

- 3) Na nova janela (Figura 1.9), configure o novo projeto com os campos:
- **Application name** – nome pelo qual a aplicação será identificada no Android Studio; é também o nome que será usado na Google Play;
 - **Company Domain** – nome do domínio da empresa;
 - **Package name** – nome do pacote Java. É usado para identificar exclusivamente a aplicação dentro do ecossistema das aplicações Android. Baseia-se na URL invertida de nome de domínio, seguida do nome da aplicação;
 - **Project location** – localização do projeto no sistema de ficheiros. Por omissão, será criada uma subpasta com o nome da aplicação, na pasta **AndroidStudioProjects** localizada no seu diretório *home*. A localização pode ser alterada bastando, para isso, premir o botão à direita do campo de texto.

Preencha os campos de acordo com a Figura 1.9 e clique no botão **Next** para continuar.

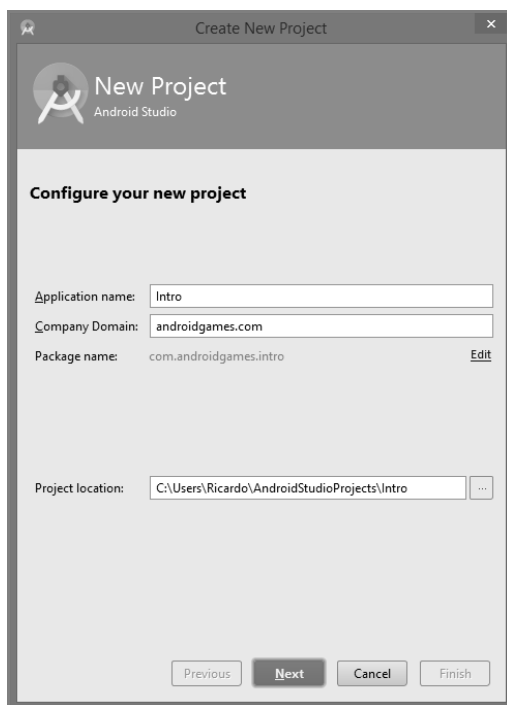


FIGURA 1.9 – Configuração de um novo projeto no Android Studio

- 4) Na janela seguinte (Figura 1.10), selecione as plataformas e as versões mínimas dos SDK onde a aplicação vai poder ser executada. Neste projeto,

mantenha todos os valores selecionados por omissão, nomeadamente a opção **Phone and Tablet** e o seu campo **Minimum SDK** com o valor API 15:Android 4.0.3 (IceCreamSandwich). Clique no botão **Next**.

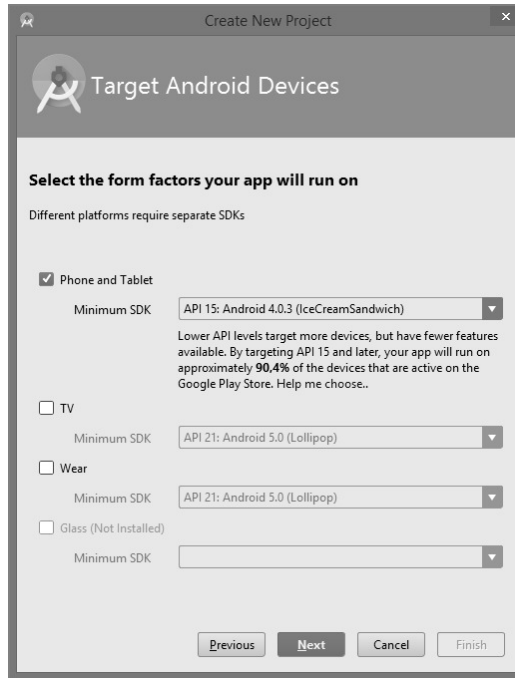


FIGURA 1.10 – Configuração das plataformas que vão poder executar a aplicação

Ao clicar no *link* **Help me choose** abre-se a janela **API Level** (Figura 1.11) com a distribuição acumulada de dispositivos para cada versão Android.

ANDROID PLATFORM VERSION	API LEVEL	CUMULATIVE DISTRIBUTION
2.2 Froyo	8	99,5%
2.3 Gingerbread	10	90,4%
4.0 Ice Cream Sandwich	15	82,6%
4.1 Jelly Bean	16	61,3%
4.2 Jelly Bean	17	40,9%
4.3 Jelly Bean	18	33,9%
4.4 KitKat	19	< 0,1%
5.0 Lollipop	21	

FIGURA 1.11 – Ajuda para a seleção do nível de API



Ao clicar num nível de API pode-se ver uma lista de recursos introduzidos na versão correspondente do Android. Esta é uma excelente ajuda para escolher o nível mínimo da API que contenha todas as características de que a aplicação a criar necessita. Assim, poder-se-á atingir o maior número de dispositivos possível. Na verdade, a escolha do nível mínimo da API deve ser um compromisso entre o número de utilizadores que se deseja atingir e os recursos de que a aplicação vai precisar.

- 5) O próximo passo é definir o tipo de atividade que será criada por omissão para a aplicação (Figura 1.12). Para tal, mantenha a opção **Blank Activity** selecionada para criar uma atividade em branco e clique no botão **Next**.

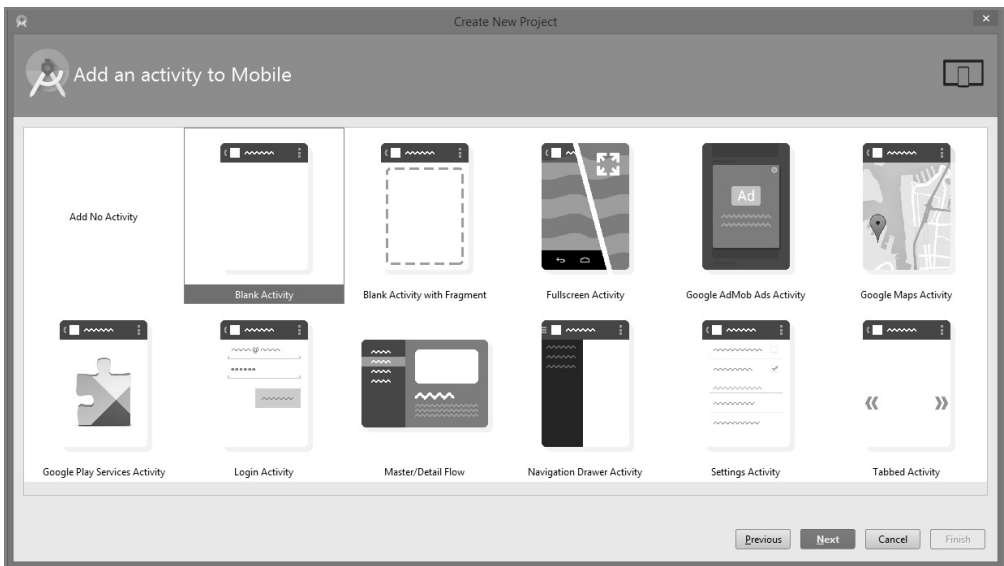


FIGURA 1.12 – Seleção do tipo de atividade

- 6) Na janela final preencha os campos de acordo com a Figura 1.13:
 - **Activity Name** – nome da classe que representa a atividade a criar;
 - **Layout Name** – nome do *layout* a criar para a atividade;
 - **Title** – texto a aparecer na barra de título da aplicação;
 - **Menu Resource Name** – nome do recurso do tipo menu.



FIGURA 1.13 – Configuração da atividade principal da aplicação

- 7) Clique no botão **Finish**. O Android Studio cria o projeto e apresenta a interface da Figura 1.14.

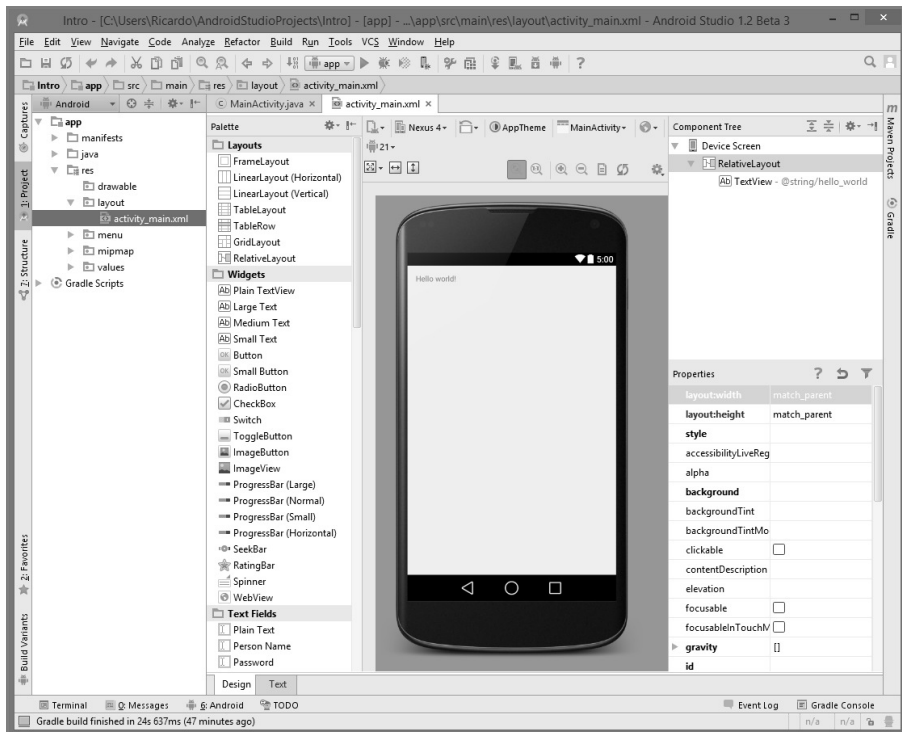


FIGURA 1.14 – Ambiente de trabalho do Android Studio

1.4.2 ESTRUTURA DE FICHEIROS

Um projeto Android é organizado em **módulos**. Existem vários tipos de módulos: módulos de aplicações, módulos de biblioteca e módulos de teste.



Quando cria um projeto Android, se seleccionar apenas uma plataforma de execução (por omissão, é a opção **Phone and Tablet**), será criado um módulo de aplicação único denominado **app** que representa a aplicação para a plataforma seleccionada. Se escolher mais do que uma plataforma (ver Figura 1.10), por exemplo, **Phone and Tablet** e **Wear**, serão criados dois módulos de aplicação com os nomes **mobile** e **wear**, respetivamente.

No Android Studio, a visualização dos módulos é feita através de vistas. A vista de projeto **Android** (Figura 1.15), ativada por omissão, destaca os ficheiros mais usados durante o desenvolvimento de aplicações Android. A vista inclui os seguintes elementos de nível de topo: o **app**, que é o módulo de aplicação, e o **Gradle Scripts**, que contém todos os ficheiros de compilação.

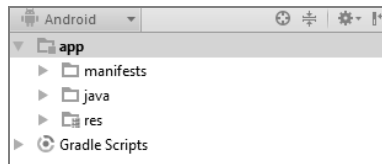


FIGURA 1.15 – A vista de projeto Android

O módulo de aplicação **app** contém três subelementos:

- **manifests** – ficheiro de manifesto do módulo. Cada módulo tem um ficheiro de manifesto, que descreve as características fundamentais da aplicação e define cada um dos seus componentes;
- **java** – ficheiros com código-fonte Java do módulo organizado por pacotes. Por omissão, é criada uma atividade principal para o projeto. No arranque da aplicação, é iniciada a atividade e carregado o *layout* associado;
- **res** – ficheiros de recurso do módulo. Os principais tipos de recursos são:
 - **drawable** – pasta para ficheiros *bitmap* (PNG, JPEG ou GIF), imagens *Nine-Patch* e ficheiros XML que descrevem formas e objetos *Drawable*;
 - **layout** – pasta com ficheiros XML que definem *layouts* gráficos. Estes ficheiros são depois associados, via código, às atividades do projeto;
 - **menu** – pasta com ficheiros XML representando menus da aplicação;
 - **mipmap** – pastas no formato **mipmap-[densidade]** que contêm ícones de lançamento de aplicações. Por omissão, são criadas quatro pastas com o ícone **ic_launcher.png** a diferentes densidades. Esta abordagem permite escolher o ícone a exibir de acordo com a densidade do dispositivo;

→ **values** – pasta com ficheiros XML que definem recursos por tipo de elemento XML (por exemplo, *strings* incluídas no *layout* gráfico).



Tenha a noção de que a estrutura do projeto no disco difere da representação da vista de projeto Android mantendo a estrutura do projeto tradicional.

O outro elemento de topo da vista de projeto Android é o **Gradle Scripts**. O Android Studio usa o *Gradle* para construir a aplicação.



O *Gradle* é uma ferramenta de automação de projeto que se baseia nos conceitos do Apache Ant e do Apache Maven. Usa uma *Domain-specific Language* (DSL), em detrimento das abordagens dos outros projetos, através de um grafo dirigido acíclico (*Directed Acyclic Graph* – DAG) que determina a ordem pela qual as tarefas são executadas.

Há um ficheiro **build.gradle** para cada módulo do projeto, bem como um ficheiro **build.gradle** para o projeto (Figura 1.16). Normalmente, só está interessado no ficheiro **build.gradle** para o módulo, já que é neste que as dependências (bibliotecas de código) de construção da aplicação são definidas, incluindo as configurações das versões SDK (*compiled*, *minimum* e *target*).

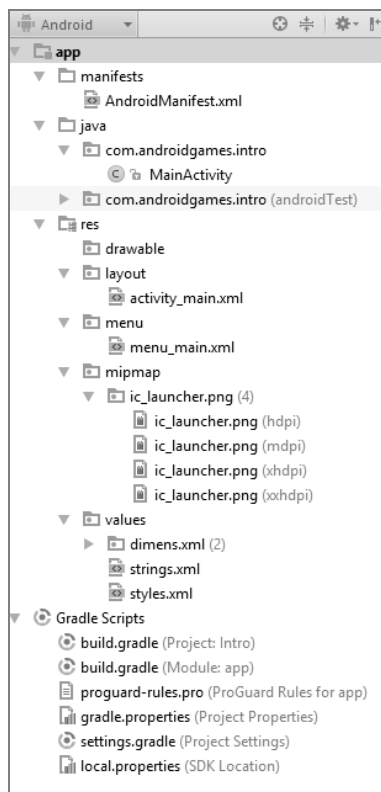


FIGURA 1.16 – A vista de projeto Android expandida

1.4.3 INTERFACE GRÁFICA

O *layout* associado à atividade principal do projeto inclui, por omissão, um objeto `TextView` com o texto “Hello World”. O próximo passo neste tutorial é alterar o texto para “Bem-vindo ao Android Studio!”. O *layout* da atividade está representado no ficheiro `activity_main.xml`, localizado na pasta `layout`. O *layout* pode ser editado no modo de desenho ou no modo de texto no formato XML. No modo de desenho, faça duplo clique sob o objeto `TextView` para editar o seu texto e identificador (Figura 1.17).



FIGURA 1.17 – Edição de propriedades nucleares de objetos gráficos

Os objetos inseridos automaticamente pelo Android Studio aderem às boas práticas de desenvolvimento de aplicações para Android, nas quais cadeias de texto são armazenadas separadamente sob a forma de recursos. Para alterar o texto aceda ao ficheiro de recurso `strings.xml`, localizado na pasta `res/values`, e altere o texto no elemento `string` que contém o atributo `name` com o valor `hello_world` (Figura 1.18).

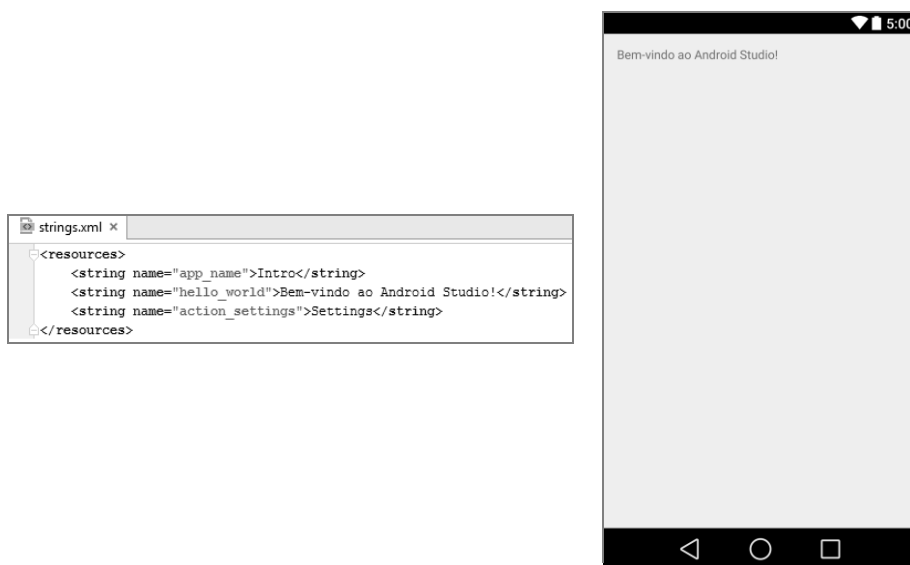


FIGURA 1.18 – Edição de ficheiros de recurso



Uma alternativa é clicar com o botão direito do rato na `TextView` e selecionar a opção **Go To Declaration (Ctrl+B)**, que abre automaticamente o ficheiro de *layout* XML e seleciona o elemento `TextView` respetivo. Depois, basta repetir a mesma ação no valor do atributo `android:text` de forma a poder abrir o ficheiro `strings.xml` e editar o texto no elemento `string` respetivo.

Até este momento, o *layout* só foi visualizado numa representação do dispositivo Nexus 4. Contudo, o *layout* pode ser testado para outros dispositivos, fazendo seleções no menu de topo do painel **Preview**. Outra opção fornecida por este menu é **Preview All Screen Sizes**, que mostra o *layout* em todos os dispositivos, tal como na Figura 1.19.

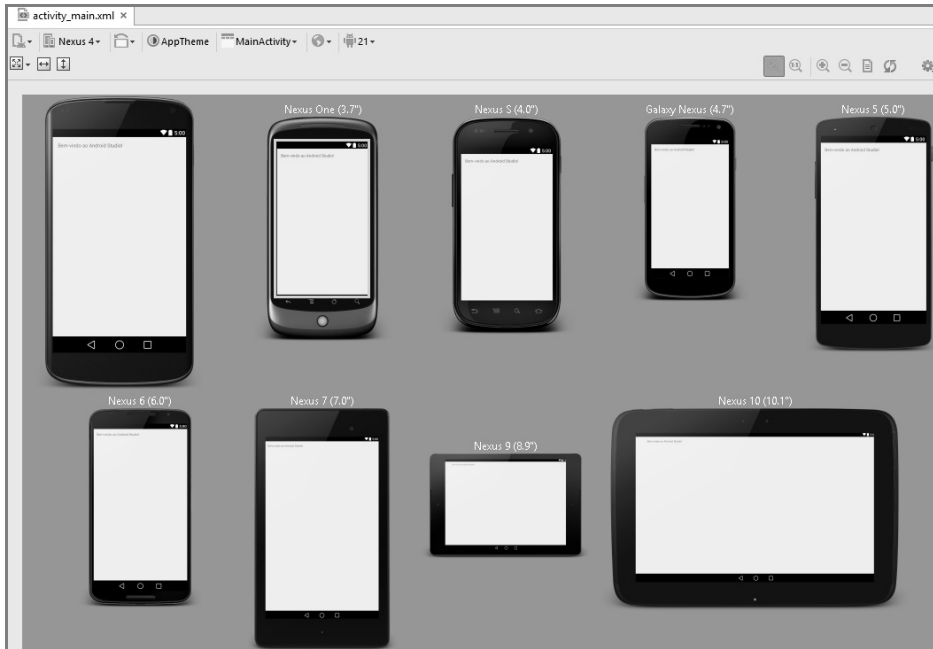


FIGURA 1.19 – Pré-visualização da aplicação em vários tipos de dispositivos

1.4.4 CRIAÇÃO DE UM *ANDROID VIRTUAL DEVICE*

Embora a capacidade de visualizar um *layout* a partir da ferramenta de desenho do Android Studio seja útil, não há melhor substituto para testar uma aplicação do que compilá-la e executá-la num dispositivo físico ou emulador, a partir daqui designado por *Android Virtual Device* (AVD).

Antes de um AVD poder ser usado, é necessário criá-lo e configurá-lo de forma a coincidir com a especificação de um modelo de dispositivo específico. Para criar um novo AVD, o primeiro passo é executar o **AVD Manager** selecionando a opção do menu principal **Tools** → **Android** → **AVD Manager**. Por omissão, nenhum AVD está configurado, conforme apresenta a Figura 1.20.

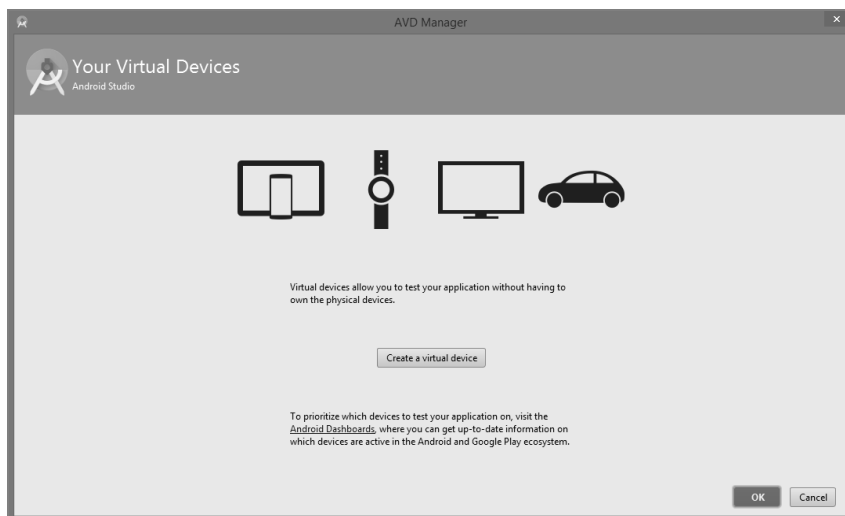


FIGURA 1.20 – AVD Manager

Clique no botão **Create a virtual device** para abrir a janela **Virtual Device Configuration**. Selecione a categoria de dispositivos (**Phone**) e o dispositivo físico a emular (**Nexus 5**), conforme mostra a Figura 1.21. De seguida, clique no botão **Next**.

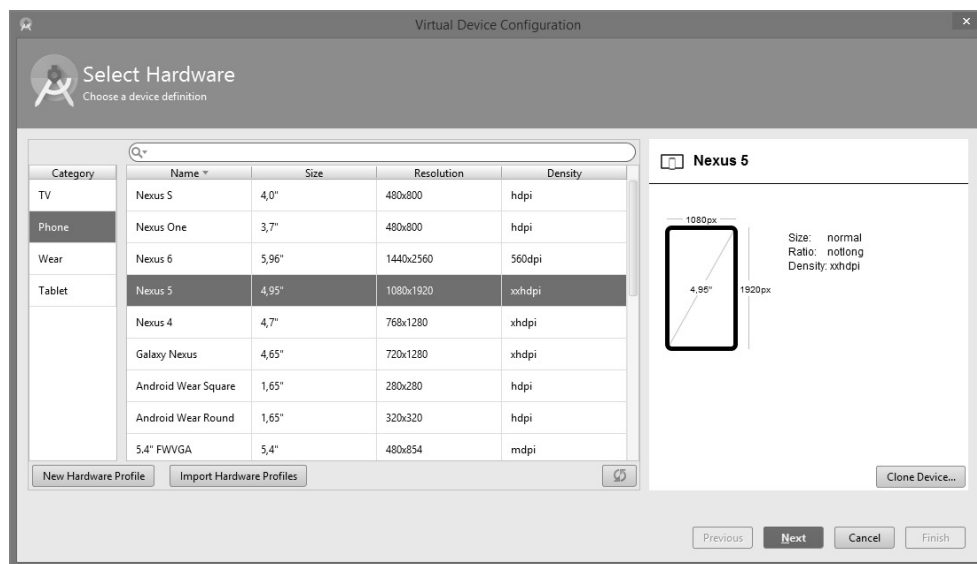


FIGURA 1.21 – Seleção da categoria e do dispositivo a emular

No próximo passo, selecione a imagem do sistema baseada no nível da API e na arquitetura do processador (Figura 1.22). Cada versão da plataforma inclui imagens do sistema que suportam uma arquitetura de processador específico, como ARM EABI, Intel x86 ou MIPS. Tipicamente, também é incluída uma imagem do sistema que contém as

API da Google. Neste caso, selecione a linha da tabela que inclui: a versão **Lollipop** (nível 22 da API) com as **API da Google** e a arquitetura **x86**. De seguida, clique no botão **Next**.

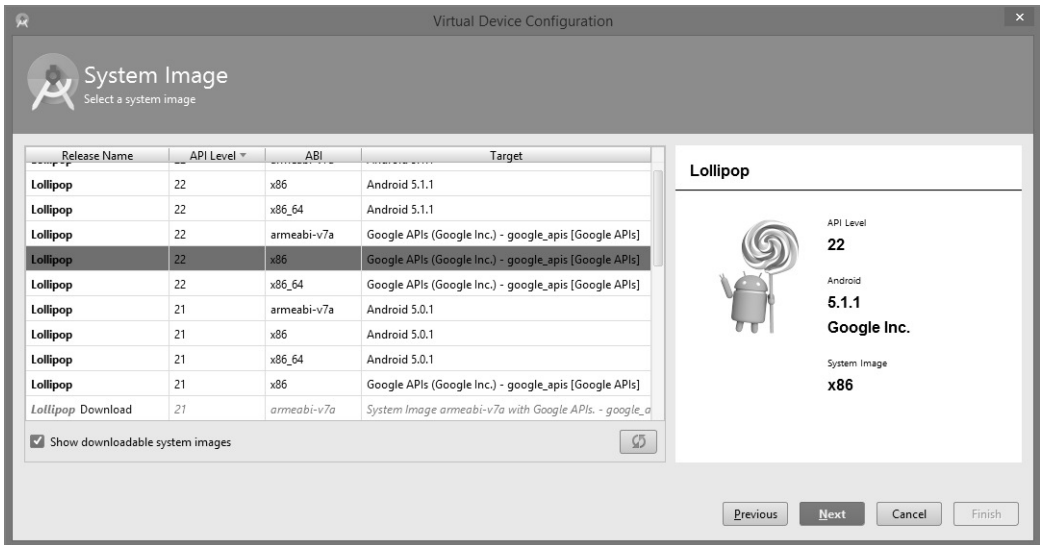


FIGURA 1.22 – Seleção de uma imagem do sistema

No próximo passo, definem-se as configurações do emulador. Para a edição de configurações mais avançadas clique no botão **Show Advanced Settings** (Figura 1.23). De entre os vários campos a editar, salientam-se os seguintes:

- O campo **AVD Name** identifica o AVD, não devendo conter espaços ou outros caracteres especiais;
- A secção **Startup size and orientation** permite testar a aplicação num ecrã que usa uma resolução ou densidade não suportada pelo *skin* predefinido do emulador e definir a orientação inicial do emulador;
- Na secção **Camera** (campos **Front** e **Back**), configuram-se as câmaras frontal e traseira do emulador. Se o computador *host* contiver uma *webcam*, a emulação da câmara pode ser configurada para usar esta câmara. Alternativamente, pode ser selecionada uma câmara emulada;
- A secção **Network** permite definir o estado inicial da taxa de transferência da rede utilizada pelo AVD e respetiva latência;

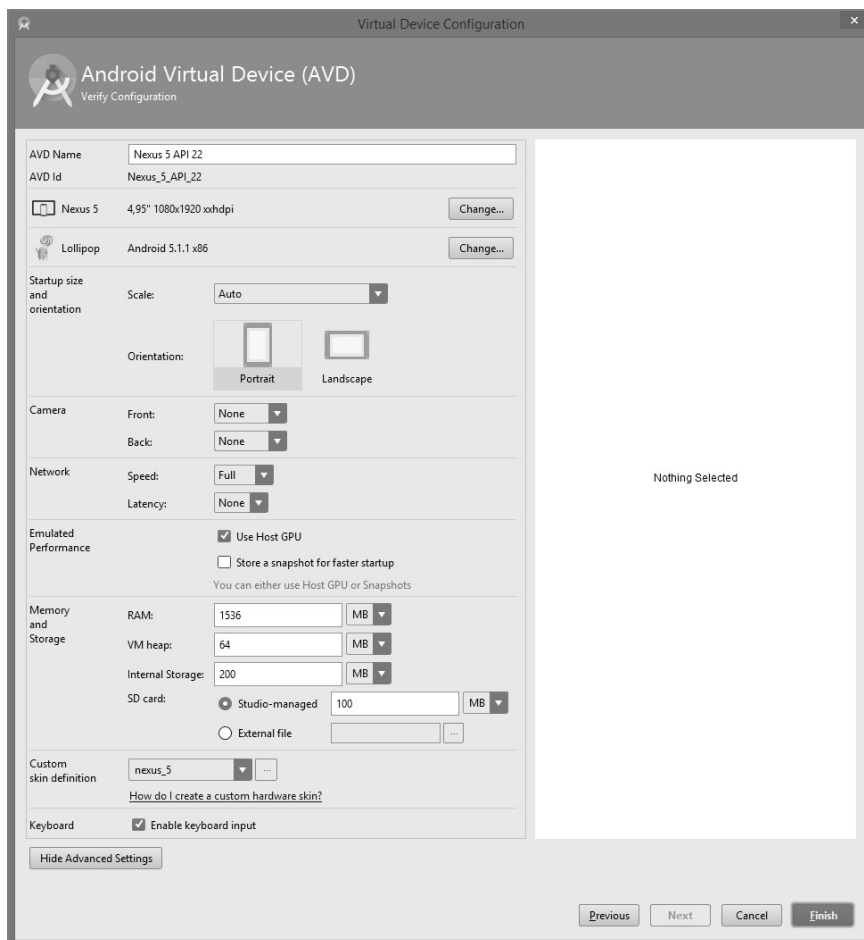


FIGURA 1.23 – Definição das configurações do emulador

- ⊙ A secção **Emulated Performance** inclui as seguintes opções mutuamente exclusivas, logo não pode usá-las em simultâneo:
 - A opção **Use Host GPU** usa a implementação OpenGL do computador *host* para avaliar os comandos OpenGL dentro do sistema emulado. Dito de outra forma, quando um programa dentro do emulador usa OpenGL para operações de gráficos, o trabalho vai para o GPU real, e o resultado vai voltar para o emulador, ao invés de emular uma GPU (o que é muito lento). O resultado é um grande aumento de velocidade, especialmente se considerarmos que a maioria dos desenhos na `View` e no `Canvas` usam OpenGL a partir da versão 4 do Android;
 - A opção **Store a snapshot for faster startup** acelera a inicialização do emulador, gravando o estado do AVD na altura do fecho do emulador e

usando o estado armazenado para iniciar o emulador em utilizações futuras. Desta forma, não se executa o processo de inicialização (que é lento, porque é emulado) todas as vezes que se iniciar o emulador.

- ⊗ A secção **Memory and Storage** define as configurações de memória e armazenamento do emulador. O campo **RAM** define a quantidade de memória RAM do emulador. O campo **VM heap** define a quantidade de RAM disponível para a *Java Virtual Machine* (JVM) alocar às aplicações em execução no emulador. Os campos **Internal Storage** e **SD Card** permitem definir a quantidade de armazenamento disponível (interno e externo) que o emulador vai providenciar. No campo **SD Card**, recomenda-se pelo menos 100 MB de forma a usar a câmara no emulador;
- ⊗ A secção **Custom skin definition** permite alterar a aparência/resolução do emulador;
- ⊗ O campo **Enable keyboard input** define a utilização, ou não, de um teclado físico. Quando esta opção estiver ativa, será possível utilizar o teclado físico no sistema onde o emulador está em execução. Como tal, o teclado do *software* Android não aparecerá dentro do emulador quando a entrada de texto for exigida por uma aplicação em execução.

Após preenchimento de todos os campos, clique no botão **Finish**. A janela **Your Virtual Devices** (Figura 1.24) apresenta a listagem dos AVD criados até ao momento.

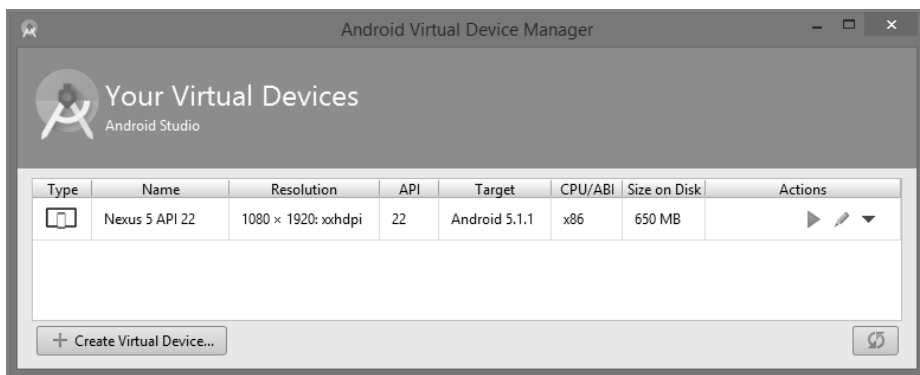


FIGURA 1.24 – Lista de dispositivos virtuais Android

São várias as ações, definidas como ícones na coluna **Actions**, que podem ser feitas nos AVD, tais como iniciar o emulador, editar as suas configurações, clonar e remover. Inicie o emulador clicando no ícone representado por uma seta (Figura 1.25).

Na primeira vez que é executado, o emulador pode demorar vários minutos a ser carregado. Em subsequentes invocações, a latência é inferior.



FIGURA 1.25 – Arranque do emulador

Podem-se instalar outros emuladores no Android Studio. Um dos mais populares é o **Genymotion**. Para instalar o seu *plugin* siga os próximos passos:

- 1) Crie uma conta no sítio Web da Genymotion e descarregue o pacote Genymotion + VirtualBox⁴.
- 2) Aceda ao Android Studio e selecione a opção **File → Settings**.
- 3) Selecione **Plugins** e clique no botão **Browse Repositories....**
- 4) Clique com o botão direito do rato no *plugin* **Genymotion** e selecione a opção **Download and Install**.
- 5) Reinicie o Android Studio e um novo ícone surgirá na barra de ferramentas.
- 6) Clique no ícone e crie um dispositivo virtual a partir da lista.
- 7) Depois, sempre que clicar no ícone surge a janela **Genymotion Device Manager** (Figura 1.26) com uma lista de emuladores instalados.

⁴ Link: <https://www.genymotion.com/#!/download>

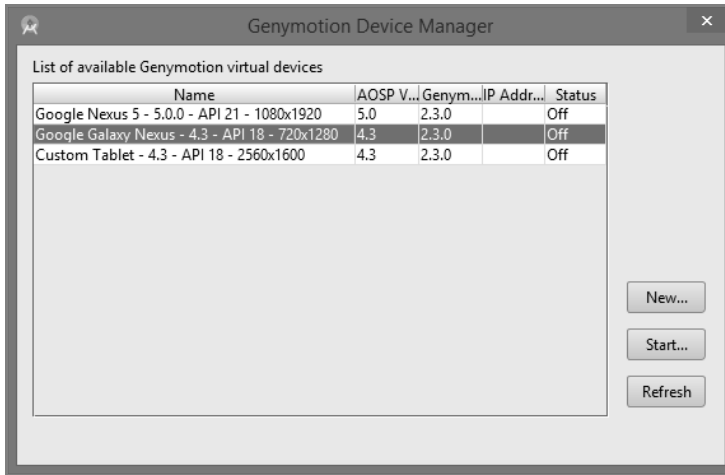


FIGURA 1.26 – Lista de emuladores disponíveis

- 8) Para iniciar um emulador selecione-o e clique no botão **Start** (Figura 1.27).

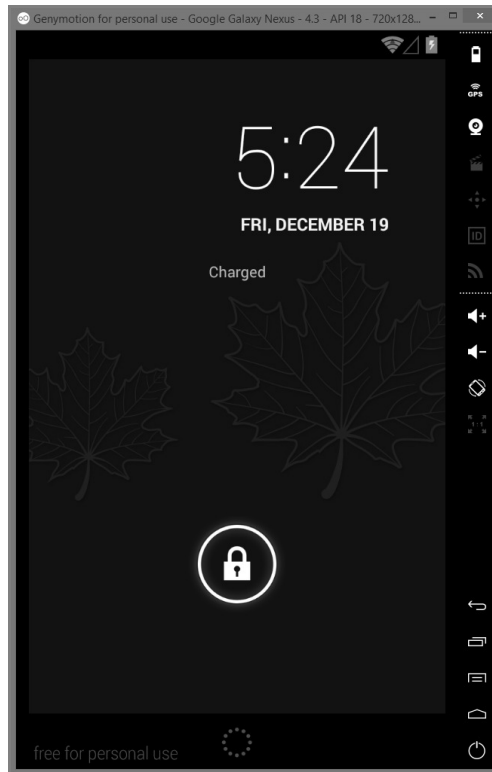


FIGURA 1.27 – Emulador Genymotion

1.4.5 EXECUÇÃO DA APLICAÇÃO

Com o AVD configurado, a aplicação **Intro** pode ser compilada e executada. No Android Studio, clique no botão de execução representado por um triângulo verde, localizado na barra de ferramentas do Android Studio (Figura 1.28), ou selecione **Run** → **Run** do menu, ou use a combinação de teclas **Shift + F10**.

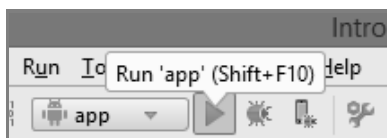


FIGURA 1.28 – Execução da aplicação

O Android Studio exibe a janela **Choose Device** (Figura 1.29), que oferece a opção de executar a aplicação numa instância AVD (já em execução) ou de lançar uma nova instância do emulador especificamente para esta aplicação. Selecione o AVD criado e executado anteriormente e clique no botão **OK**.



FIGURA 1.29 – Escolha de dispositivo para a execução da aplicação



Ative a caixa de verificação **Use same device for future launches** se o dispositivo onde pretende executar a aplicação for sempre o mesmo. Desta forma, a janela **Choose Device** não tornará a aparecer.

Finalmente, a aplicação **Intro** é instalada e executada no emulador com sucesso, conforme ilustrado na Figura 1.30. Cada vez que uma nova revisão da aplicação é compilada e executada, a instância anterior da aplicação em execução no dispositivo ou emulador será automaticamente terminada e substituída pela nova versão.

Para parar manualmente uma aplicação em execução a partir do Android Studio selecione **Run** → **Stop** (**Ctrl + F2**) do menu principal.

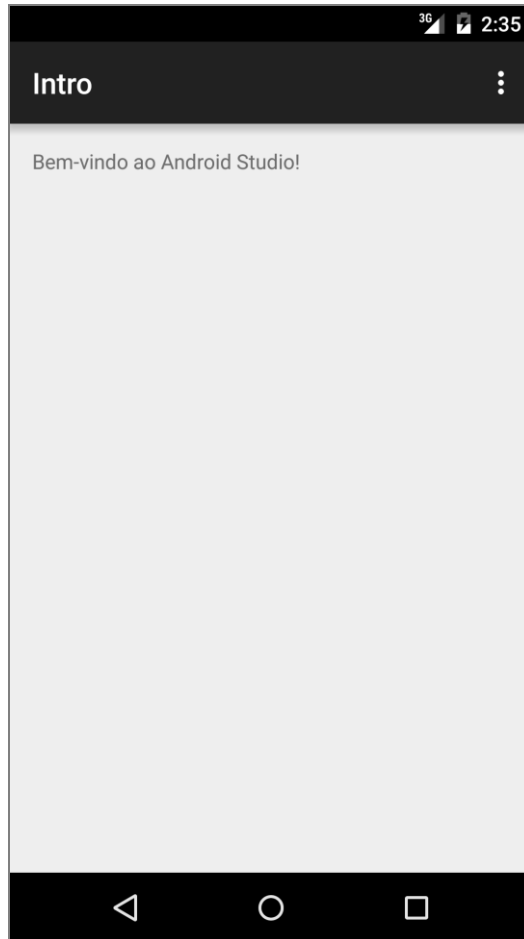


FIGURA 1.30 – A aplicação Intro executada no AVD

Pode também usar o **Android Device Monitor** através da opção **Tools → Android → Android Device Monitor**.



O **Android Device Monitor** é uma ferramenta independente que fornece uma GUI para várias ferramentas de *debugging* e análise de aplicações Android. A ferramenta não requer a instalação de um IDE e incorpora as seguintes ferramentas: Dalvik Debug Monitor Server (DDMS), Tracer for OpenGL ES, Hierarchy Viewer, Systrace, Traceview e Pixel Perfect magnification viewer.

Depois selecione o processo respetivo do painel lateral esquerdo, conforme a Figura 1.31, e clique no botão **Stop Process**.

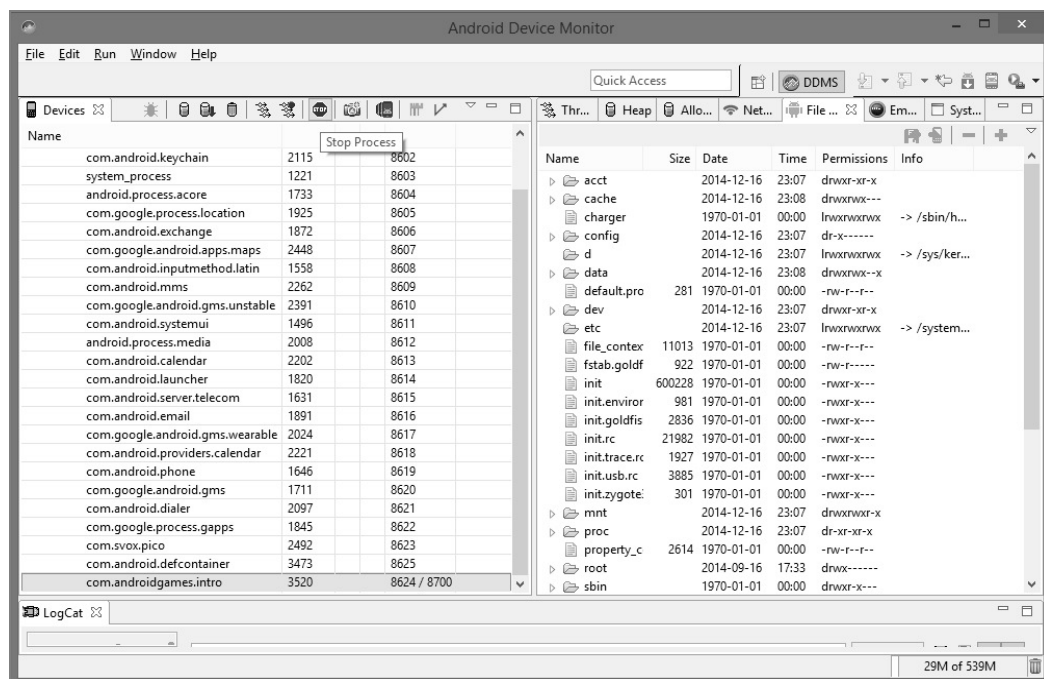


FIGURA 1.31 – Android Device Monitor

1.5 COMPONENTES PRINCIPAIS

As aplicações Android são compostas por um ou mais tipos de componentes. Cada tipo tem um papel diferente no comportamento geral da aplicação, e cada um pode ser ativado individualmente. O ficheiro de manifesto deve declarar todos os componentes e requisitos da aplicação. A Tabela 1.3 lista os tipos de componentes em Android.

FUNCIONALIDADE	CLASSE BASE JAVA	EXEMPLOS
Interação com o utilizador	Activity	Preencher um formulário
Execução de processos em <i>background</i>	Service	Tocar uma música
Resposta a eventos	BroadcastReceiver	Despoletar alarme
Armazenamento de dados	ContentProvider	Abrir contacto telefónico

TABELA 1.3 – Componentes de uma aplicação Android⁵

⁵ Exemplo retirado do livro *The Android Developer's Collection*, disponível no seguinte endereço: http://books.google.pt/books/about/The_Android_Developer_s_Collection.html?id=3Wi2gwGoZZ0C

As atividades (*Activity*) são responsáveis pela interface para com o utilizador, os serviços (*Service*) implementam operações com execução em *background*, os *BroadcastReceiver* respondem a eventos do sistema e os *ContentProvider* armazenam e/ou fornecem dados para a aplicação.

1.5.1 Activity

As **atividades** representam ecrãs de uma aplicação com uma interface gráfica de utilizador. Uma aplicação Android pode conter zero ou mais atividades. Por exemplo, uma aplicação *browser* pode ter uma atividade que permite a navegação em páginas Web, outra atividade para gestão de páginas favoritas e outra ainda para configurações, mas apenas uma atividade pode ser exibida num determinado momento.

Uma atividade é implementada como uma subclasse de *Activity*. As atividades devem lidar apenas com operações relacionadas com a interface gráfica do utilizador – *Graphical User Interface* (GUI). A gestão do ciclo de vida das atividades é controlada através da implementação de métodos de retorno (os chamados *callback methods*). O ciclo de vida de uma atividade é retratado na Figura 1.32.

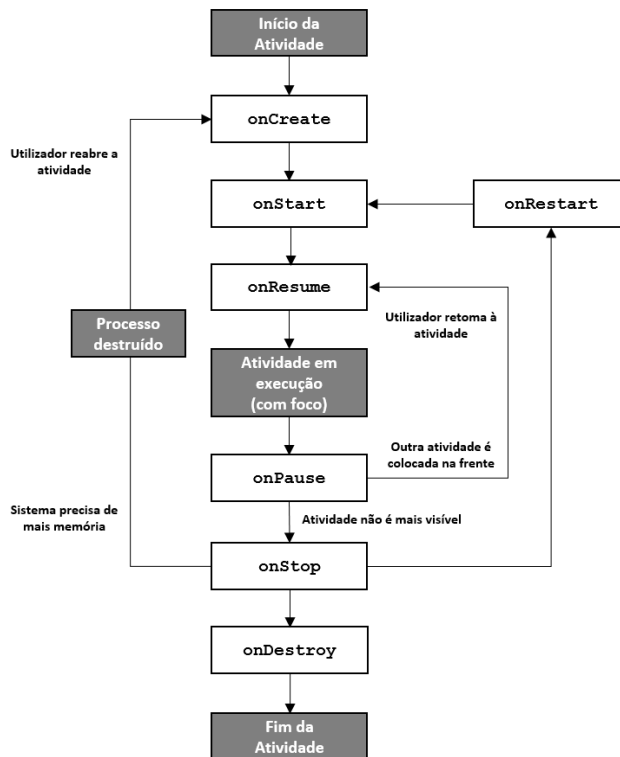


FIGURA 1.32 – Ciclo de vida de uma atividade

A partir do momento em que se tem mais do que uma atividade numa aplicação Android, ou se pretende navegar para uma atividade de outra aplicação, é necessário usar as *intents* (intenções) de forma a comunicar entre componentes. Uma *intent* é uma mensagem assíncrona que permite que uma aplicação peça funcionalidade a outros componentes do SO Android. Por exemplo, as atividades podem ser iniciadas de forma **explícita**, indicando diretamente o nome da atividade a ser chamada (classe Java associada), ou de forma **implícita**, descrevendo o tipo de ação que se deseja realizar. Neste último caso, o Android seleciona a atividade adequada, que pode até ser de uma aplicação diferente (Figura 1.33):

```
// COMUNICAÇÃO ENTRE DUAS ATIVIDADES (INTENT EXPLÍCITA)
Intent intent = new Intent(this, Activity2.class);
intent.putExtra("mensagem", "Olá atividade secundária!");
startActivity(intent);
// DEFINIÇÃO DE UMA MENSAGEM DE TEXTO SEM ESPECIFICAR O RESPONSÁVEL PELO ENVIO (INTENT IMPLÍCITA)
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, "Uma mensagem de texto...");
sendIntent.setType("text/plain");
startActivity(sendIntent);
```

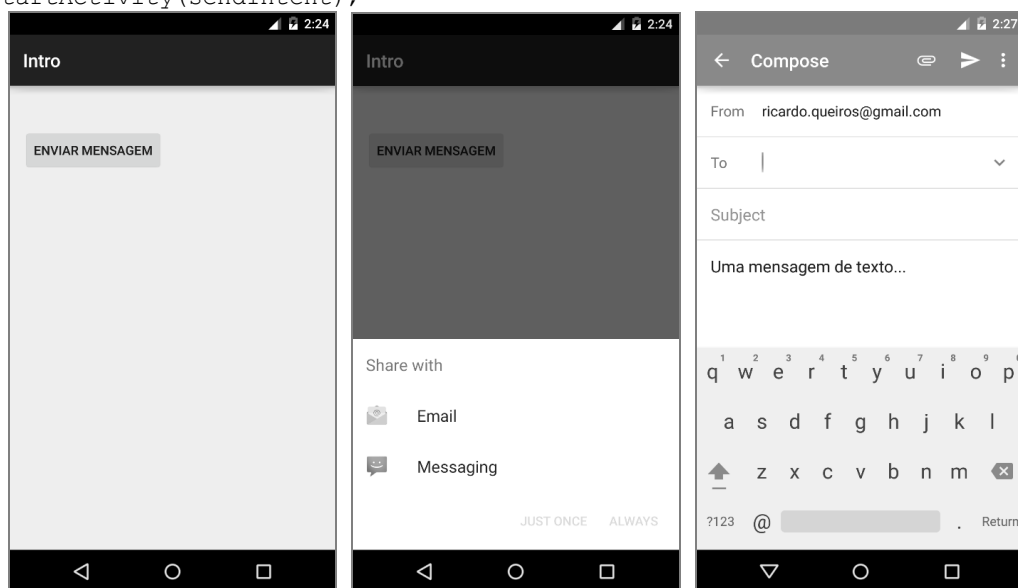


FIGURA 1.33 – Execução de uma atividade através de uma *intent* implícita com uma ação

Por vezes, necessitamos de obter um resultado a partir de uma atividade quando esta termina. Por exemplo, podemos iniciar uma atividade que permite ao utilizador escolher uma pessoa de uma lista de contactos e, após a seleção, devolver a pessoa selecionada. Para fazer isso, é necessário usar o método `startActivityForResult(Intent, int)` com um segundo parâmetro do tipo inteiro que identifica a chamada.

O próximo exemplo usa a aplicação **Contacts** do dispositivo Android e apresenta o número de telemóvel do contacto seleccionado (Figura 1.34). Defina um botão e inclua o próximo código de invocação:

```
Intent intent = new Intent(Intent.ACTION_PICK);
intent.setType(ContactsContract.CommonDataKinds.Phone.CONTENT_TYPE);
if (intent.resolveActivity(getPackageManager()) != null) {
    startActivityForResult(intent, REQUEST_SELECT_PHONE_NUMBER);
}
```

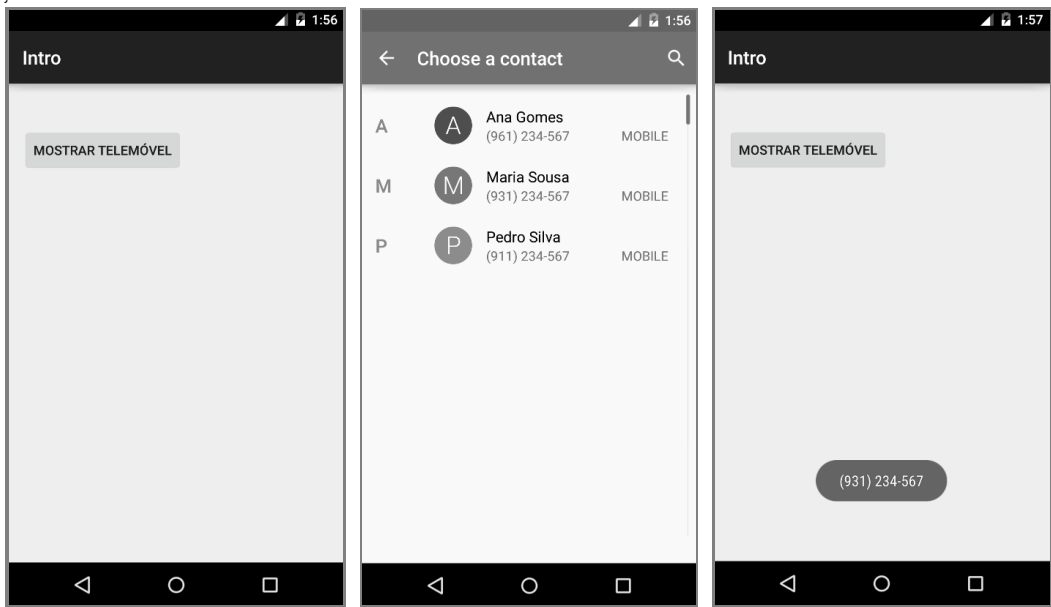


FIGURA 1.34 – Obtenção do resultado de uma atividade iniciada por uma *intent* implícita

Para lidar com o resultado da atividade use o método `onActivityResult`. No método obtém-se um `ContentResolver` que executa uma consulta no fornecedor do conteúdo devolvendo um objeto `Cursor`. Este objeto permite que os dados resultantes da consulta possam ser lidos. De seguida, o número de telemóvel do contacto seleccionado é apresentado através da classe `Toast`:

```
protected void onActivityResult(int reqCode,int resCode,Intent data) {
    if(reqCode==REQUEST_SELECT_PHONE_NUMBER && resCode==RESULT_OK) {
        // CONSULTA O FORNECEDOR DE CONTEÚDO PELO NÚMERO DE TELEMÓVEL
        Cursor c = getContentResolver().query(data.getData(),
            String[]{ContactsContract.CommonDataKinds.Phone.NUMBER},
            null, null, null);
        if (c != null && c.moveToFirst()) {
            int i=c.getColumnIndex(ContactsContract.CommonDataKinds.Phone.NUMBER);
            Toast.makeText(getApplicationContext(),cursor.getString(i),
                Toast.LENGTH_LONG).show();
        }
    }
}
```

Para outro tipo de operações (por exemplo, obter dados remotos), e embora se possam usar as classes `AsyncTask` ou `Handler` para executar operações fora da *thread* principal, recomenda-se delegar operações de longa duração para um componente especial, chamado serviço, e executá-las numa *thread* separada (prevenindo assim o término abrupto de uma operação por o utilizador ter pressionado o botão **Home**).

1.5.2 Service

Os **serviços** são componentes que devem ser utilizados para executar operações em *background* e que, tipicamente, não fornecem ou não têm de interagir com uma GUI (por exemplo, descarregar ficheiros da Web). Um serviço pode assumir duas formas:

- ⊙ **Started** – um serviço é “iniciado” (*started*) quando um componente da aplicação (por exemplo, uma atividade) o principia invocando o método `startService`. Uma vez iniciado, um serviço pode ser executado em *background* por tempo indeterminado, mesmo que o componente que o iniciou seja destruído. Normalmente, um serviço iniciado executa uma única operação e não devolve um resultado para quem o invocou. Quando a operação termina, o serviço deve parar por si mesmo;
- ⊙ **Bound** – um serviço é *bounded* quando um componente da aplicação se liga a ele invocando o método `bindService`. Um serviço vinculado oferece uma interface cliente-servidor que permite que os componentes possam interagir com o serviço, enviar pedidos, obter resultados, e até mesmo em serviços de diferentes processos, comunicar através do *InterProcess Communication* (IPC). Um serviço deste tipo é executado apenas enquanto o outro componente da aplicação a ele estiver ligado. Vários componentes podem-se ligar ao serviço de uma só vez, mas quando todos se desvincularem, o serviço é destruído.

Um serviço pode ser dos dois tipos em simultâneo, sendo apenas necessário implementar os métodos `onStartCommand` e `onBind`. Independentemente do tipo de serviço, qualquer componente da aplicação pode usar o serviço (mesmo a partir de uma outra aplicação). No entanto, pode-se declarar o serviço como privado, no ficheiro de manifesto, e bloquear o acesso a outras aplicações. Tipicamente, existem duas classes que servem de base para a criação de um serviço:

- ⊙ **Service** – esta é a classe base para todos os serviços. Um serviço definido desta forma é executado na *thread* principal e não cria a sua própria *thread*. Isto significa que, se o serviço vai fazer qualquer trabalho intensivo ou suscetível de bloqueio (por exemplo, reproduzir um ficheiro MP3, aceder à rede), é necessário criar uma nova *thread* dentro do serviço através da classe `Handler` ou da classe `AsyncTask`;

- `IntentService` – esta é uma subclasse da classe `Service` que usa uma *worker thread* para lidar com todos os pedidos de arranque do serviço. Esta abordagem é a ideal quando não se pretende que o seu serviço lide simultaneamente com vários pedidos. Esta abordagem também simplifica a criação do serviço, pois basta implementar o método `onHandleIntent`, que recebe a *intent* respetiva a cada pedido.



Ao usar uma *thread* separada, reduz-se o risco de erros do tipo *Application Not Responding* (ANR) e a *thread* principal pode dedicar-se à interação do utilizador com as suas atividades.

Ao usar como base a classe `Service` vai permitir a execução *multi-threading*, mas exige mais codificação por parte do programador, que necessita de criar manualmente uma nova *thread* para processar cada pedido. Ao invés, usando o `IntentService`, a execução dos pedidos é enviada para uma *worker thread* única, que processa um pedido de cada vez. O próximo código mostra a estrutura base de uma classe `IntentService`:

```
public class MyIntentService extends IntentService {
    private static final String NAME = "MyIntentService";

    // CONSTRUTOR (OBRIGATÓRIO) INVOCANDO O MÉTODO SUPER COM O NOME DA THREAD
    public MyIntentService() {
        super(NAME);
    }

    // MÉTODO EXECUTADO NA WORKER THREAD (UMA INTENT DE CADA VEZ)
    @Override
    protected void onHandleIntent(Intent intent) {
        // CÓDIGO DA AÇÃO A SER EXECUTADA NA WORKER THREAD
    }
}
```

Para além das vantagens mencionadas no uso da classe `IntentService` em detrimento da classe `Service`, existem outras que simplificam a criação do serviço, tais como:

- Terminar o serviço após todos os pedidos terem sido tratados, evitando assim a necessidade de usar o método `stopSelf`;
- Disponibilizar uma implementação por omissão do método `onBind` devolvendo nulo;
- Disponibilizar uma implementação por omissão do método `onStartCommand` que envia a *intent* para a *thread* e depois para o método `onHandleIntent`.

O ciclo de vida de um serviço (Figura 1.35) é mais simples do que o de uma atividade, podendo seguir caminhos diferentes tendo em conta os dois tipos de serviços.

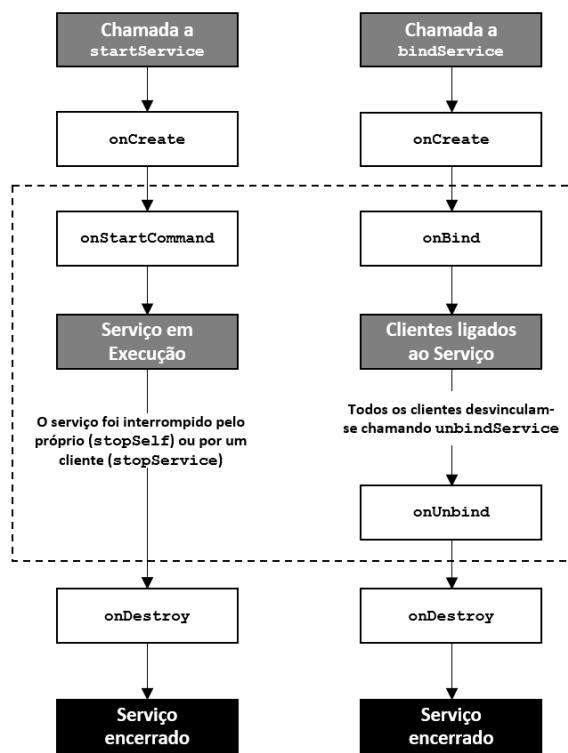


FIGURA 1.35 – Ciclo de vida de serviços iniciados pelos métodos `startService` e `bindService`

Um *started service* é iniciado pelo método `startService`. O serviço só pode ser encerrado por si próprio, recorrendo ao método `stopSelf`, ou por intermédio de outro componente, através do método `stopService`. Quando o serviço é parado, o sistema destrói-o.

Um *bound service* é criado quando um componente (cliente) invoca o método `bindService`. O cliente pode comunicar com o serviço através da interface `IBinder`. O cliente pode fechar a conexão através do método `unbindService`. Este tipo de serviço permite a conexão de vários clientes. Quando todos os clientes se desvinculam do serviço, o sistema destrói-o, sem haver a necessidade de o serviço se autodestruir.

A classe `IntentService` mostrada anteriormente fornece uma comunicação fácil entre um componente (normalmente uma atividade) e um serviço. Mas, geralmente, queremos saber o resultado da operação executada pelo serviço. Tal pode ser feito de várias maneiras, mas se quisermos manter o comportamento assíncrono da classe `IntentService`, a melhor abordagem é transmitir uma resposta através do serviço e implementar um `BroadcastReceiver` para receber e lidar com a mensagem de resposta (Figura 1.36).

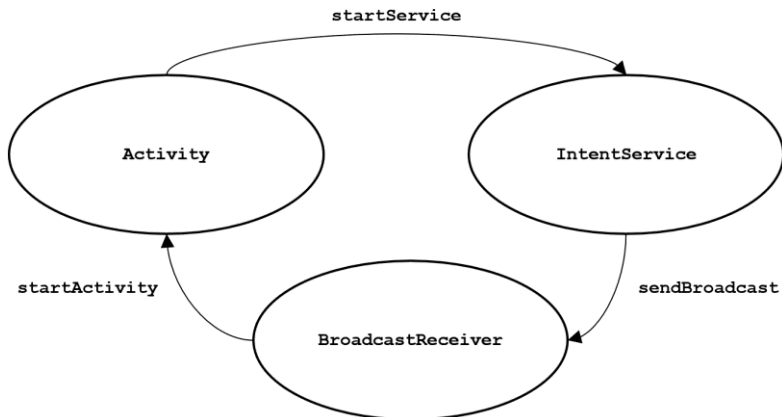


FIGURA 1.36 – Comunicação entre uma atividade, um serviço e um BroadcastReceiver

1.5.3 BroadcastReceiver

Um **recetor de broadcast** é um componente Android que responde a eventos enviados pelo sistema ou aplicação. Todos os recetores registados para um determinado evento são notificados em tempo de execução assim que o evento aconteça. Por exemplo, as aplicações podem-se inscrever para o evento do sistema `ACTION_BATTERY_LOW`, que é despoletado quando o dispositivo Android fica com pouca bateria. Nessa altura o recetor poderá estar programado para realizar tarefas que libertem os recursos do sistema, como, por exemplo, desligar o *wi-fi*.

A implementação de um recetor usa como base a classe `BroadcastReceiver`. Se o evento para o qual o recetor se registou acontecer, o seu método `onReceive(Context, Intent)` é chamado pelo sistema Android. Um objeto `BroadcastReceiver` só é válido durante a chamada ao método `onReceive`. A partir daqui, pode-se notificar o utilizador através de um objeto `Notification` ou delegar a execução para outro componente através dos métodos da classe `Context`: `startActivity` ou `startService`.

Um `BroadcastReceiver` pode ser registado de duas formas:

- ⊗ **Estática** – através do elemento `<receiver>` no ficheiro de manifesto;
- ⊗ **Dinâmica** – através do método `Context.registerReceiver`.

Exemplos para a primeira categoria são as aplicações que precisam de fazer uma determinada operação assim que o dispositivo ou a aplicação são inicializados. O próximo exemplo define um recetor de *broadcast* para escutar mudanças de estado do telefone. Se o telefone recebe uma chamada, o recetor será notificado e enviará uma mensagem usando a classe `Log`.

Crie um novo projeto Android. No ficheiro de manifesto registe o recetor e adicione permissões para acesso de leitura ao estado do telefone:

```
<manifest ...>
  <uses-permission android:name="android.permission.READ_PHONE_STATE" />
  <application ...>
    <activity .../>
    <receiver android:name="MyPhoneReceiver">
      <intent-filter>
        <action android:name="android.intent.action.PHONE_STATE"/>
      </intent-filter>
    </receiver>
  </application>
</manifest>
```

De seguida, crie a classe `MyPhoneReceiver`:

```
public class MyPhoneReceiver extends BroadcastReceiver {
    private final String TAG = "BROADCAST_TAG";
    @Override
    public void onReceive(Context context, Intent intent) {
        Bundle extras = intent.getExtras();
        if (extras != null) {
            String state = extras.getString(TelephonyManager.EXTRA_STATE);
            if (state.equals(TelephonyManager.EXTRA_STATE_RINGING)) {
                Log.w(TAG, extras
                    .getString(TelephonyManager.EXTRA_INCOMING_NUMBER));
            }
        }
    }
}
```

Após a instalação da aplicação no emulador, simula-se uma chamada de telefone na secção **Emulador Control** (Figura 1.37) da ferramenta de monitorização DDMS⁶.

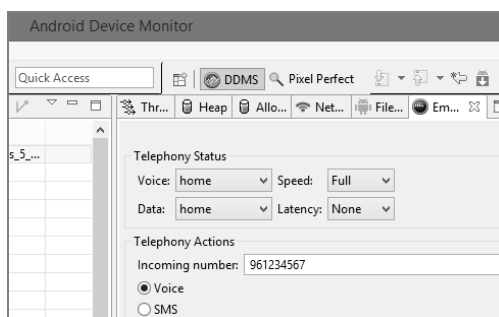


FIGURA 1.37 – Simulação de uma chamada telefónica através do Emulador Control

O recetor é chamado e envia uma notificação para o **LogCat** (Figura 1.38). Ao mesmo tempo, a chamada é recebida no emulador.

⁶ A DDMS é uma das várias ferramentas incluídas no *Android Device Monitor*.

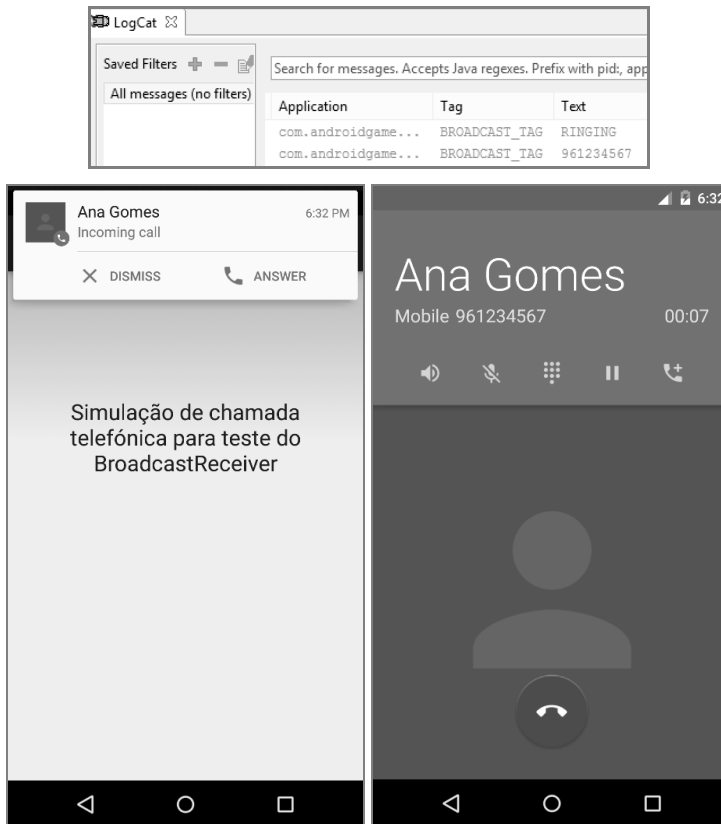


FIGURA 1.38 – Exemplo de um BroadcastReceiver para intercetar chamadas telefónicas

Como alternativa, pode-se registar o recetor dinamicamente no código através do método `registerReceiver`. Esta abordagem é a ideal para eventos que sinalizam uma mudança de estado da qual a aplicação depende para funcionar. Por exemplo, se uma aplicação depende de uma conexão *Bluetooth*, a aplicação deve reagir a uma mudança de estado deste tipo de conexão, mas somente quando está ativa.

O método `registerReceiver` tem dois parâmetros: o `BroadcastReceiver` que se pretende registar e o objeto `IntentFilter` que especifica qual o evento que o recetor deve ouvir. Sugere-se que seja feito o registo do recetor no método `onResume` da atividade e o respetivo cancelamento no método `onPause`.

O próximo exemplo mostra como criar um `BroadcastReceiver` para detetar uma mudança na conectividade de rede e como registar e cancelar o registo do recetor:

```
public class ConnectivityChangeReceiver extends BroadcastReceiver {
    public void onReceive(Context context, Intent intent) {
        // INVOCA UM SERVIÇO PARA LIDAR COM O EVENTO
        context.startService(new Intent(context, myService.class));
    }
}
```

O método `onReceive` inclui a invocação de um serviço sempre que o evento seja detetado. Os serviços são, provavelmente, uma das formas mais comuns para executar uma ação nos recetores, uma vez que é bastante provável que o utilizador não esteja a usar ativamente a aplicação no momento em que o evento ocorre. De seguida, cria-se a atividade com o registo do `BroadcastReceiver` e o seu cancelamento:

```
public class MainActivity extends Activity {
    private ConnectivityChangeReceiver myConnectivityChangeReceiver;
    protected void onResume() {
        super.onResume();
        myConnectivityChangeReceiver = new ConnectivityChangeReceiver();
        registerReceiver(myConnectivityChangeReceiver,
            new IntentFilter(ConnectivityManager.CONNECTIVITY_ACTION));
    }
    protected void onPause() {
        super.onPause();
        unregisterReceiver(myConnectivityChangeReceiver);
    }
}
```

Há também eventos que não são autorizados a registarem-se estaticamente. Por exemplo, o evento `Intent.ACTION_TIME_TICK`, que é transmitido a cada minuto, sendo uma boa decisão, já que um recetor neste contexto gastaria desnecessariamente a bateria.



No caso de se pretender enviar e receber mensagens apenas no processo da aplicação, considere o uso do método `LocalBroadcastManager.sendBroadcast` em vez do método genérico `Context.sendBroadcast`. Esta abordagem, denominada *broadcast local*, é mais eficiente, porque não necessitamos da gestão entre processos, nem de nos preocuparmos com questões de segurança inerentes ao *broadcasting*.

Os *broadcasts* são divididos em duas categorias (Figura 1.39) no que respeita à ordem pela qual são notificados os recetores: **normal** e **ordenado**.

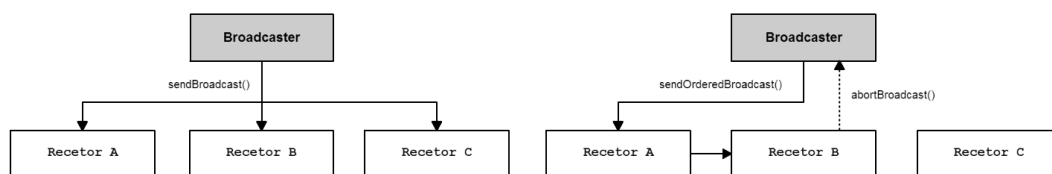


FIGURA 1.39 – *Broadcast* normal e ordenado

As transmissões normais são enviadas para todos os recetores de forma assíncrona através do método `sendBroadcast` e recebidas numa ordem não especificada. Este método é mais eficiente, mas carece de recursos avançados encontrados nas transmissões ordenadas, tal como o envio de *feedback* para quem iniciou a transmissão.

Uma transmissão ordenada é enviada através do método `sendOrderedBroadcast` e entregue aos recetores registados, um de cada vez, seguindo uma ordem específica através do atributo `android:priority` do elemento `<intent-filter>` respetivo. Outra

característica das transmissões ordenadas é a possibilidade de o recetor usar os métodos `setResultCode` e `setResultData` para definir um resultado que é devolvido a quem iniciou a transmissão. O recetor também pode usar o método `abortBroadcast` para cancelar uma transmissão de modo a que a *intent* não seja propagada para o próximo recetor na fila de recetores. Um recetor verifica que se trata de uma transmissão ordenada através do método `isOrderedBroadcast`. Um bom exemplo do uso de transmissões ordenadas é, por exemplo, a interceção de mensagens SMS.

Um *broadcast* normal não fica disponível depois de ter sido enviado e processado pelo sistema. Contudo, ao usar o método `sendStickyBroadcast(intent)`, a *intent* correspondente torna-se persistente mesmo após o *broadcast* estar terminado, permitindo futuros registos da *intent* para receber o mesmo *broadcast*. Este método é especialmente útil para sinalizar os estados do sistema (como, por exemplo, o estado da bateria). Para que uma aplicação possa enviar este tipo de transmissão deve-se incluir no ficheiro de manifesto a permissão `android.permission.BROADCAST_STICKY` e enviar o *broadcast* usando o método `Context.sendStickyBroadcast`.

Um exemplo de uma transmissão é `Intent.ACTION_BATTERY_CHANGED`, a qual é usada para indicar mudanças no nível da bateria do dispositivo. O próximo código mostra um exemplo de um recetor que interceta as mudanças de bateria. Vai também perceber se a transmissão persistente é nova, ou não:

```
public class BatteryChangeListener extends BroadcastReceiver {
    public void onReceive(Context context, Intent intent) {
        if(Intent.ACTION_BATTERY_CHANGED.equals(intent.getAction())) {
            if(isInitialStickyBroadcast()) { // EVENTO DA CACHE }
            else { // NOVO EVENTO }
        }
    }
}
```

O método `isInitialStickyBroadcast` devolve verdadeiro se o recetor estiver a processar o valor inicial do *broadcast*, ou seja, o valor que foi difundido pela última vez e que está atualmente na *cache*, de modo a que este não seja diretamente o resultado de uma transmissão atual. Outra variação das transmissões normais são as transmissões dirigidas onde se explicita, através do método `setComponent`, qual o recetor da *intent*. O componente deve incluir informação sobre o nome do pacote e da classe do recetor, conforme mostra o código seguinte:

```
public void sendDirectedBroadcast(String packageName, String className,
String action) {
    Intent intent = new Intent(action);
    intent.setComponent(new ComponentName(packageName, className));
    sendBroadcast(directedBroadcastIntent);
}
```

Ao enviar um *broadcast* deste tipo, o mesmo só será recebido pelo recetor especificado, ainda que existam outros recetores registados para a mesma ação.

1.5.4 ContentProvider

Os **fornecedores de conteúdo** fornecem dados às aplicações abstraindo o acesso aos dados através da interface `ContentResolver`. Um fornecedor de conteúdos só é necessário se precisarmos de partilhar dados entre múltiplas aplicações. Um bom exemplo é o fornecedor `Contacts`, que gere o repositório central do dispositivo de dados sobre pessoas. Caso contrário, podemos usar o objeto `SharedPreferences` para manter os dados no formato chave-valor, usar uma base de dados `SQLite` diretamente através da API `SQLite`, entre outras opções.



O Android inclui vários fornecedores de conteúdo: **Browser** – armazena dados sobre o *browser* (por exemplo, favoritos, histórico); **CallLog** – armazena informação sobre detalhes de ligações telefónicas, chamadas não atendidas, etc.; **MediaStore** – armazena ficheiros *media*, tais como áudio e vídeo. Para além dos `ContentProvider` predefinidos, é possível criar novos.

Contudo, se os dados da aplicação a armazenar forem adequados para `SQL`, normalmente é mais fácil implementar um `ContentProvider` mesmo que não partilhe os dados com outras aplicações. Ao fazê-lo, simplifica, por exemplo, a exibição dos dados da aplicação usando um objeto `AdapterView` (como uma lista ou uma grelha), porque a API `Loader` fornece implementações simples para carregamento de dados provenientes de um `ContentProvider`. Para criar um fornecedor de conteúdo estende-se a classe `ContentProvider` e implementam-se os métodos da Tabela 1.4.

MÉTODO	DESCRIÇÃO
<code>onCreate()</code>	Inicializa o fornecedor.
<code>query(Uri, String[], String, String[], String)</code>	Devolve os dados a quem o invocou.
<code>insert(Uri, ContentValues)</code>	Insere novos dados no <code>ContentProvider</code> .
<code>update(Uri, ContentValues, String, String[])</code>	Atualiza dados no <code>ContentProvider</code> .
<code>delete(Uri, String, String[])</code>	Remove dados do <code>ContentProvider</code> .
<code>getType(Uri)</code>	Devolve o tipo MIME dos dados no <code>ContentProvider</code> .

TABELA 1.4 – Métodos da interface `ContentResolver`

Um dos métodos mais importantes é o método `query`. Para consultar um fornecedor, por exemplo, o fornecedor `Contacts`, especifica-se a sequência da consulta na forma de um *Uniform Resource Identifier* (URI). O formato do URI é o seguinte:

```
<standard_prefix>://<authority>/<data_path>/<id>
```

As várias partes do URI são as seguintes:

- ⊙ `standard_prefix` – o prefixo dos fornecedores de conteúdo é `content://`;
- ⊙ `authority` – especifica o nome do fornecedor. No caso de ser um novo fornecedor, este deve conter o nome completo do pacote, como, por exemplo, `com.example.providerDemo`;
- ⊙ `data_path` – especifica o tipo de dados solicitados. Por exemplo, se estamos a obter todos os contactos do fornecedor **Contacts**, então o `data_path` seria `people` e o URI seria `content://contacts/people`;
- ⊙ `id` – indica o registo específico solicitado. Por exemplo, para procurar o contacto número dois do fornecedor **Contacts**, o URI anterior necessitaria de ser alterado para `content://contacts/people/2`.

A Tabela 1.5 lista alguns exemplos de *queries* a aplicar a um `ContentProvider`:

URI	DESCRIÇÃO
<code>content://media/external/images</code>	Devolve uma lista de todas as imagens armazenadas no armazenamento externo do dispositivo (por exemplo, cartão SD).
<code>content://call_log/calls</code>	Devolve uma lista de chamadas registadas no CallLog .
<code>content://browser/bookmarks</code>	Devolve uma lista de <i>bookmarks</i> armazenados no <i>browser</i> .

TABELA 1.5 – Exemplos de *queries* para um `ContentProvider`

Como exemplo, demonstra-se o uso de um `ContentProvider` através da listagem por ordem alfabética de todos os nomes dos contatos da aplicação **Contacts** (Figura 1.40). Neste caso, as informações básicas dos contactos estão armazenadas na tabela `Contacts`. A partir do Android 2.0, para consultar os registos dos contactos usa-se a URI armazenada em `ContactsContract.Contacts.CONTENT_URI`:

```
public TextView outputText;
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_intro);
    outputText = (TextView) findViewById(R.id.textView);
    fetchContacts();
}
public void fetchContacts() {
    Uri CONTENT_URI = ContactsContract.Contacts.CONTENT_URI;
    String DISPLAY_NAME = ContactsContract.Contacts.DISPLAY_NAME;
    StringBuffer output = new StringBuffer();
```



```
ContentResolver contentResolver = getContentResolver();
Cursor cursor = contentResolver.
    query(CONTENT_URI, null,null, null, null);
if (cursor.getCount() > 0) {
    while (cursor.moveToNext()) {
        String name = cursor.getString(cursor.getColumnIndex(DISPLAY_NAME));
        output.append(name + "\n");
    }
    outputText.setText(output);
}
}
```

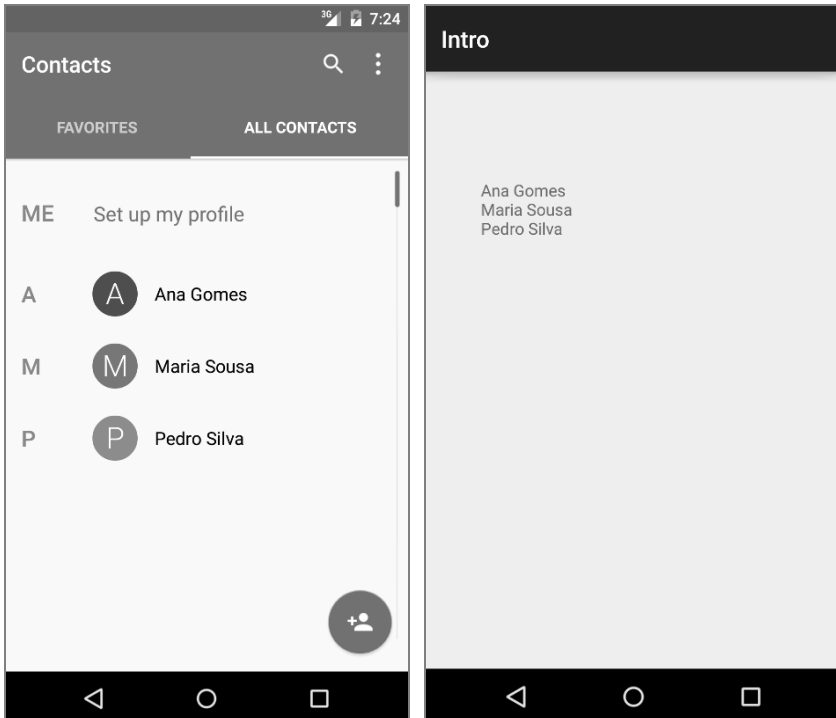


FIGURA 1.40 – Uso do fornecedor de conteúdo Contacts



Antes de executar o código anterior, deve incluir no ficheiro de manifesto as permissões de leitura para os contactos do dispositivo:

```
<uses-permission android:name="android.permission.READ_CONTACTS"/>
```


2

API DA GOOGLE E MOTORES DE JOGOS

A criação de um jogo digital requer o domínio de várias facetas desde a geração de gráficos até ao armazenamento de dados intrínsecos ao próprio jogo. Neste capítulo são apresentadas várias tecnologias para o desenvolvimento dessas facetas, nomeadamente, as API para a geração de gráficos 2D/3D, para a gestão do *input* do utilizador, para a manipulação de áudio e para a persistência de dados. Por fim, explica-se a importância dos motores de jogos no processo de criação de jogos mais complexos, enumerando-se os motores de jogos mais importantes e populares dos últimos tempos.

2.1 INTRODUÇÃO

Uma das grandes dúvidas de quem se está a iniciar no mundo do desenvolvimento de jogos digitais é qual o *software* a usar. Na verdade, a plataforma Android oferece um leque variado de soluções que cobrem grande parte das facetas de um jogo digital (Figura 2.1).

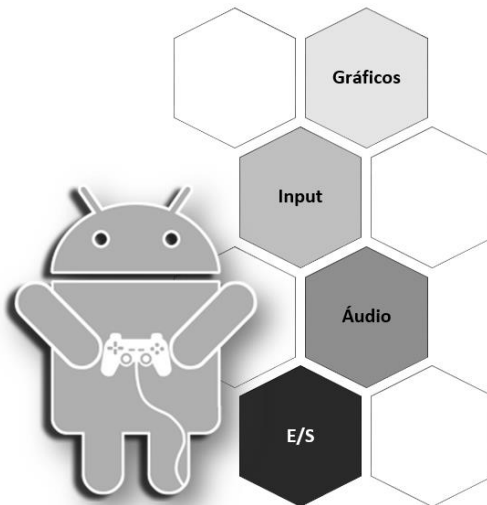


FIGURA 2.1 – As facetas de um videojogo

Por exemplo, no desenho de gráficos, o Android suporta várias soluções que vão desde uma simples API 2D até ao uso de um *game engine* (em português, motor de jogo). Neste caso, a melhor solução vai depender de vários fatores, como, por exemplo, tipo de jogo, efeito dimensional a dar, nível de detalhe gráfico e de física envolvido, entre outros.

De forma a dar a conhecer ao leitor as alternativas existentes na plataforma Android para o desenvolvimento de jogos, as duas secções seguintes focam as API da Google para a implementação das facetas dos videojogos e os motores de jogos que visam simplificar e abstrair o desenvolvimento de jogos mais exigentes.

2.2 API DA GOOGLE

A plataforma Android disponibiliza várias API que facilitam a implementação das facetas de um jogo digital, nomeadamente para geração de gráficos 2D/3D, gestão do *input* do utilizador, persistência de dados e manipulação de áudio.

2.2.1 GRÁFICOS

O SDK do Android inclui várias API para criação de gráficos. Apresentam-se as três mais populares:

- ⊙ **Drawable** – é uma API gráfica 2D que usa uma abordagem declarativa para a criação de gráficos incorporando as instruções de desenho em ficheiros XML;
- ⊙ **Canvas** – é uma API gráfica 2D que envolve o desenho diretamente numa superfície (por exemplo, um *bitmap*) fornecendo um maior controlo na forma como os gráficos são criados;
- ⊙ **OpenGL ES** – é uma API gráfica multiplataforma que permite a criação de gráficos de alto desempenho em 2D e 3D.

2.2.1.1 Drawable

O SDK do Android inclui uma API gráfica 2D, chamada **Drawable**, para desenho personalizado de imagens e formas. Um objeto **Drawable** é uma abstração geral para “algo que pode ser desenhado” e que é armazenado num projeto Android como um ficheiro (doravante chamado **recurso drawable**). Os recursos podem ser de dois tipos:

- ⊙ Ficheiro de imagem (por exemplo, GIF, JPG, PNG);
- ⊙ Ficheiro XML para descrever formas (cor, borda, gradiente), o estado, transições, entre outras características.

CRIAÇÃO A PARTIR DE UMA IMAGEM

O uso de imagens é bastante comum em aplicações Android, principalmente para o desenho de logotipos e ícones em aplicações, ou para gráficos em jogos. Uma forma simples de adicionar imagens a uma aplicação é através do seu armazenamento e respetiva referência a partir dos recursos *drawable* do projeto. A Tabela 2.1 enumera os vários tipos de recursos *drawable* baseados em imagens.

RECURSO <i>DRAWABLE</i>	DESCRIÇÃO
BitmapDrawable	Um <i>bitmap</i> . O Android suporta ficheiros <i>bitmap</i> em três formatos: PNG (preferencial), JPG (aceitável), GIF (desaconselhável).
NinePatchDrawable	Uma imagem PNG com regiões predefinidas para permitir o redimensionamento da imagem baseado no seu conteúdo.

TABELA 2.1 – Tipos de recursos *drawable* baseados em imagens

Por exemplo, se quiser exibir numa aplicação Android a imagem **mycar.png**, representada na Figura 2.2, comece por criar um novo projeto Android e copie o ficheiro respetivo para uma subpasta da pasta de recursos, por exemplo, **res/drawable-hdpi**. O armazenamento do ficheiro faz com que o Android crie automaticamente um recurso.



FIGURA 2.2 – O objeto BitmapDrawable

Para poder exibir o recurso *drawable* inclua no *layout* um objeto *ImageView*:

```
<ImageView android:id="@+id/myImageView" ... />
```

Por fim, associe o recurso ao objeto *ImageView*. Apresentam-se duas abordagens diferentes para esta ação:

- No *layout* XML onde é definida a *ImageView* referencie o recurso no atributo `android:src` usando a sintaxe **@drawable/filename**, em que **filename** é o nome do ficheiro sem a extensão:

```
<ImageView  
    android:id="@+id/myImageView"  
    android:src="@drawable/mycar" />
```

- No código Java use o método `setImageResource` do objeto *ImageView* passando uma referência ao recurso através da sintaxe **R.drawable.filename**:

```
ImageView imageView = (ImageView) findViewById(R.id.myImageView);  
imageView.setImageResource(R.drawable.mycar);
```



As boas práticas sugerem o armazenamento de recursos *drawable* nas subpastas da pasta *res*, mais concretamente **drawable-hdpi**, **drawable-mdpi**, **drawable-xhdpi**, **drawable-xxhdpi**. Estas subpastas representam as diferentes densidades de um *bitmap*. O sistema Android seleciona automaticamente qual o *bitmap* a carregar baseado na configuração do dispositivo. Caso contrário, se não forem fornecidas diferentes versões do *bitmap*, o sistema Android dimensiona o *bitmap* ajustando-o à resolução corrente. Esta solução é indesejável, já que o *bitmap* poderá ficar distorcido.

Em certas situações, poderá querer lidar com a imagem como um objeto *Drawable*. A vantagem de obter um objeto *Drawable* é ter disponível uma série de métodos genéricos para interagir com o que está a ser desenhado, como, por exemplo, os métodos *setBounds*, *setState*, entre outros. Para fazer isso, crie um *Drawable* a partir do recurso da seguinte forma:

```
Resources res = myContext.getResources();
Drawable myCar = res.getDrawable(R.drawable.mycar);
```



É possível também definir um recurso XML que aponta para um ficheiro *bitmap*. O ficheiro inclui um conjunto de instruções (por exemplo, *anti-aliasing*) a serem aplicadas à imagem. Um dos usos mais comuns é definir o modo de colocação de um *bitmap* no *background* de um *layout*, como demonstra o próximo exemplo:

```
<bitmap android:src="@drawable/mycar" android:tileMode="repeat" .../>
```

Por vezes, pode-se querer que um *bitmap*, que está associado a uma determinada *View*, se adapte ao conteúdo da própria *View*. Para tal, usa-se o recurso *NinePatch*, que não é mais do que um ficheiro de imagem, no formato PNG, onde podem ser definidas regiões que o Android redimensiona sempre que o conteúdo dentro da *View* ultrapassar os limites da imagem. A definição das regiões redimensionáveis é feita pela inclusão de uma borda extra de largura de 1 pixel. O próximo exemplo (Figura 2.3) mostra como redimensionar um botão com cantos arredondados.

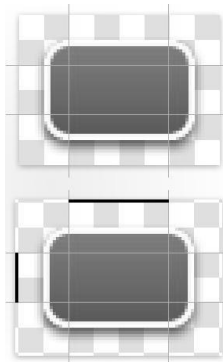


FIGURA 2.3 – Marcação da área redimensionável num recurso *NinePatch*

É definida uma grelha 3x3 sob a imagem. O Android usa a grelha para “cortar” a imagem em nove partes. As quatro peças dos cantos da imagem são mantidas, enquanto

as do meio são marcadas como redimensionáveis através de pixels pretos nas linhas da esquerda e superior da margem. O redimensionamento automático do recurso ocorre sempre que o conteúdo da `View` associada se altere (Figura 2.4).

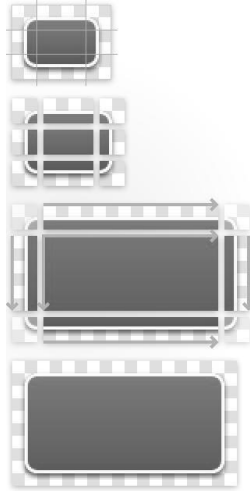


FIGURA 2.4 – Redimensionamento de um recurso *NinePatch*

Posteriormente, o recurso deve ser gravado com a extensão **9.png** na pasta de recursos do projeto Android. Devido à complexidade inerente à criação de recursos deste tipo, o SDK do Android inclui uma ferramenta chamada *Draw9patch*⁷, que permite criar imagens *NinePatch* usando um editor gráfico WYSIWYG (Figura 2.5).

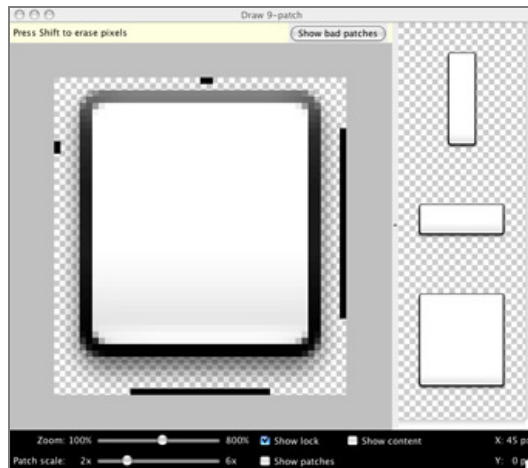


FIGURA 2.5 – A ferramenta *Draw9patch*

⁷ Localizada na pasta `SDK/tools`.

CRIAÇÃO A PARTIR DE UM FICHEIRO XML

Um recurso *drawable* pode também ser baseado num conjunto de instruções de desenho contidas num ficheiro XML. Em tempo de execução, uma aplicação Android carrega o recurso e usa essas instruções para criar gráficos 2D. O Android define vários tipos de recursos XML. A Tabela 2.2 lista os principais tipos.

RECURSO <i>DRAWABLE</i>	DESCRIÇÃO
ClipDrawable	Define um objeto Drawable com um determinado nível de <i>clipping</i> (recorte). Na maioria das vezes este tipo de recurso é usado para implementar barras de progresso.
InsetDrawable	Define um objeto Drawable para ser incluído a uma determinada distância. Isto é útil quando a View precisa de um recurso <i>drawable</i> como <i>background</i> que seja menor do que os limites atuais de exibição. A distância é garantida pelos atributos <code>android:inset Top Right Bottom Left</code> .
LayerDrawable	Gere uma série de outros objetos Drawable. Estes são desenhados na ordem como foram definidos no <i>array</i> , de modo que o elemento com o maior índice seja desenhado por cima de todos os outros.
LevelListDrawable	Gere uma série de objetos Drawable, cada um com um valor numérico máximo atribuído. Definindo o valor do nível do objeto com o método <code>setLevel</code> irá carregar a imagem na lista que tem um valor do atributo <code>android:maxLevel</code> maior ou igual ao valor passado para o método. Um bom exemplo seria um ícone indicador de nível de bateria, com imagens diferentes para indicar o nível atual da bateria.
ScaleDrawable	Altera o tamanho de outro objeto Drawable com base no seu nível atual.
ShapeDrawable	Define uma forma geométrica, incluindo cores e gradientes.
StateListDrawable	Usa várias imagens para representar o mesmo gráfico, dependendo do estado do objeto. Por exemplo, um objeto Button pode existir num de vários estados diferentes (pressionado, focado ou nenhum).
TransitionDrawable	Permite alternar entre dois recursos produzindo uma transição (<i>fading</i>). Cada objeto Drawable é representado por um elemento <code><item></code> dentro de um único elemento <code><transition></code> . São suportados não mais do que dois itens. Para fazer a transição para a frente invoca-se o método <code>startTransition</code> . Para fazer a transição para trás usa-se o método <code>reverseTransition</code> .

TABELA 2.2 – Tipos de recursos *drawable* baseados em ficheiros XML

O ClipDrawable é um objeto Drawable que define o nível de recorte (largura e altura) de uma imagem (outro objeto Drawable) com base no seu nível. É possível controlar também o seu alinhamento em relação ao objeto Drawable pai. Na maioria das

vezes este tipo de recurso é usado para implementar barras de progresso. Crie o ficheiro **clip.xml** na pasta de recursos de um projeto Android com o seguinte conteúdo:

```
<clip android:drawable="@drawable/mycar"
      android:clipOrientation="horizontal"
      android:gravity="left" />
```

Aplique o recurso anterior a um objeto `View`:

```
<ImageView android:id="@+id/myImageView"
           android:background="@drawable/clip"/>
```

Por fim, inclua o próximo código no método `onWindowFocusChanged` da atividade que notifica o utilizador de que a `View` já foi carregada, sendo possível obter os seus dados. O código obtém um objeto `Drawable` de um objeto `ImageView` através do método `getBackground`. Depois, o nível de visibilidade da imagem (Figura 2.6) é definido para metade através do método `setLevel`:

```
ImageView imageView = (ImageView) findViewById(R.id.myImageView);
ClipDrawable drawable = (ClipDrawable) imageView.getBackground();
drawable.setLevel(5000);
```



FIGURA 2.6 – O objeto `ClipDrawable`



Por omissão, o nível de visibilidade da imagem é 0, o que a torna invisível. Quando o nível é 10 000, a imagem não é cortada e fica completamente visível.

O `LayerDrawable` é um objeto `Drawable` que gere uma lista de outros objetos `Drawable`. O último elemento da lista é desenhado no topo da hierarquia de visualização. Cada objeto `Drawable` é representado por um elemento `<bitmap>` dentro de um elemento `<item>`. O elemento `<layer_list>` agrega todos os elementos `<item>`. O próximo exemplo define um recurso `LayerDrawable` chamado **layers.xml**, com três imagens:

```
<layer-list>
  <item>
    <bitmap android:src="@drawable/mycar" android:gravity="center" />
  </item>
  <item android:top="50dp" android:left="50dp">
    <bitmap android:src="@drawable/mycar" android:gravity="center" />
  </item>
  <item android:top="100dp" android:left="100dp">
    <bitmap android:src="@drawable/mycar" android:gravity="center" />
  </item>
</layer-list>
```

Depois, associa-se o recurso `LayerDrawable` a um `ImageView`. O resultado final é uma pilha de imagens sobrepostas (Figura 2.7):

```
<ImageView
    android:layout_height="wrap_content"
    android:layout_width="wrap_content"
    android:src="@drawable/layers" />
```



FIGURA 2.7 – O objeto `LayerDrawable`

O `StateListDrawable` é um objeto `Drawable` que permite definir várias imagens para diferentes estados do mesmo gráfico. Por exemplo, um botão pode ter vários estados diferentes (pressionado, focado ou nenhum). Usando uma lista de estados *drawable*, pode-se fornecer uma imagem de fundo diferente para cada estado. O próximo exemplo (Figura 2.8) mostra como exibir um *bitmap* diferente consoante o estado do botão (pressionado, focado ou nenhum). Primeiro, cria-se um recurso *drawable* chamado **buttons.xml**, onde são enumerados os vários estados:

```
<selector>
    <item android:drawable="@drawable/button_normal" />
    <item android:state_focused="true"
        android:drawable="@drawable/button_focused" />
    <item android:state_hovered="true"
        android:drawable="@drawable/button_focused" />
    <item android:state_pressed="true"
        android:drawable="@drawable/button_pressed" />
</selector>
```

De seguida, aplica-se o recurso criado a um objeto `Button`:

```
<Button
    android:layout_height="wrap_content"
    android:layout_width="wrap_content"
    android:background="@drawable/buttons" />
```



FIGURA 2.8 – O objeto `StateListDrawable`

Muitos mais exemplos sobre diferentes tipos de objetos `Drawable` poderiam ser incluídos nesta obra, contudo, o mais importante é perceber a lógica subjacente que é comum à maior parte deles.

Em suma, a API `Drawable` é a escolha ideal quando se pretende desenhar gráficos simples que não precisam de mudar sistematicamente. Para além disso, usa uma abordagem que favorece a manutenção do código ao potenciar uma clara separação entre o código e a interface gráfica da aplicação. Contudo, apesar de os recursos *drawable* serem úteis para grande parte dos requisitos gráficos, não é possível ter um controlo fino sobre o gráfico a ser criado. Nesse caso, a API `Canvas` é a escolha ideal, e será retratada na próxima secção.

2.2.1.2 Canvas

O `Canvas` é uma API que permite criar gráficos 2D diretamente numa superfície de desenho, fornecendo um grande controlo na forma como os gráficos são manipulados.

Antes do Android 3.0 esta API usava a biblioteca **Skia** para desenhar gráficos 2D, não permitindo tirar partido da aceleração via *hardware*. Desde esta versão e, por omissão, a partir da versão 4, a API `Canvas` usa a biblioteca **OpenGLRenderer** que traduz as operações da API para operações **OpenGL** de modo a serem executadas no *Graphics Processing Unit* (GPU).

A API tem métodos de desenho padrão de *bitmaps*, linhas, círculos, retângulos, texto, entre outros. O uso da API envolve normalmente a extensão de uma de duas classes:

- ⊙ `View` – se a aplicação não necessitar de um nível significativo de processamento ou velocidade de *frame-rate* (por exemplo, um jogo de xadrez, ou outra aplicação com animações lentas), então considere criar uma *custom view* e executar todo o desenho gráfico num objeto `Canvas` através do método `View.onDraw(Canvas canvas);`
- ⊙ `SurfaceView` – se a aplicação incluir movimentos rápidos, tais como jogos, que exigem atualizações com regularidade, então pondere o uso de uma `SurfaceView`. Ao contrário de uma `View` normal, a `SurfaceView` suporta o desenho numa *thread* diferente da *thread* da atividade principal, o que beneficia o constante redesenho gráfico e, conseqüentemente, a performance da aplicação.

USO DE UMA VIEW

O desenho de gráficos através da API `Canvas` é simples. Para tal, basta criar uma classe que deriva da classe `View` e implementar o método `View.onDraw(Canvas canvas)`. Este método é invocado pelo sistema Android de cada vez que a *view* precisa de ser redesenhada (ou seja, quando a atividade é criada ou restabelecida, ou quando uma janela que a sobrepõe fica invisível). A *framework* Android usa o objeto `Canvas` para desenhar um *bitmap* que é posteriormente tratado pelo sistema Android, responsável pela sua exibição no ecrã do dispositivo. O próximo código apresenta este cenário:

```
class RenderView extends View {
    public RenderView(Context context) {
        super(context);
    }
    protected void onDraw(Canvas canvas) {
        // A SER IMPLEMENTADO
    }
}
```



Há duas formas de obter um objeto `Canvas`. A primeira, como já se viu, é através da implementação do método `View.onDraw(Canvas canvas)`. A segunda envolve a definição de um objeto `Bitmap` e, de seguida, instanciar um objeto `Canvas` com ele. Por exemplo, o trecho de código seguinte cria um novo `Canvas` com um *bitmap* subjacente:

```
Bitmap bitmap = Bitmap.CreateBitmap (100, 100, Bitmap.Config.Argb8888);
Canvas canvas = new Canvas(bitmap);
```

O método `onDraw` recebe um objeto `Canvas` que contém uma série de métodos para desenho de formas, *bitmaps* e texto. A Tabela 2.3 enumera alguns deles.

MÉTODO	DESCRIÇÃO
<code>drawPoint(float x, float y, Paint p)</code>	Desenha um ponto.
<code>drawLine(float startX, float startY, float stopX, float stopY, Paint p)</code>	Desenha uma linha.
<code>drawRect(float tlX, float tlY, float brX, float brY, Paint p)</code>	Desenha um retângulo.
<code>drawCircle(float x, float y, float rad, Paint p)</code>	Desenha um círculo.
<code>drawBitmap(Bitmap b, Rect src, Rect dst, Paint p)</code>	Desenha um <i>bitmap</i> .
<code>drawText(String text, float x, float y, Paint p)</code>	Desenha um texto.

TABELA 2.3 – Métodos principais da classe `Canvas`

Um dos parâmetros comuns a todos os métodos de desenho é o objeto `Paint`. A *framework* `android.graphics` organiza o desenho em duas áreas: “o que desenhar” é gerido pela classe `Canvas` e o “como desenhar” é gerido pela classe `Paint`. Simplificando, a classe `Canvas` define as formas que o utilizador pode desenhar no ecrã, enquanto a classe `Paint` gere a informação sobre estilo, cor, fonte, entre outras características, das formas, texto e *bitmaps* a serem desenhados. Por exemplo, a classe `Canvas` disponibiliza

um método para desenhar retângulos, enquanto a classe `Paint` define se pretende preencher esse retângulo com alguma cor.

O próximo exemplo mostra como desenhar um círculo com borda preta e fundo amarelo no centro do ecrã (Figura 2.9). Comece por criar a classe `BallView`, responsável pelo desenho da bola, que estende a classe `View`. No seu construtor instancie um objeto `Paint`. Depois, todo o desenho é gerido no seu método `onDraw`:

```
class BallView extends View {
    private Paint paint;
    public BallView(Context context) {
        super(context);
        paint = new Paint();
    }
    protected void onDraw(Canvas canvas) {
        // OBTÉM A LARGURA E A ALTURA, EM PIXELS, DO TARGET PARA ONDE O OBJETO CANVAS FAZ O RENDER
        int largura = canvas.getWidth();
        int altura = canvas.getHeight();
        paint.setColor(Color.BLACK);
        paint.setStrokeWidth(3);
        canvas.drawCircle(largura/2, altura/2, 100, paint);
        paint.setColor(Color.YELLOW);
        canvas.drawCircle(largura/2, altura/2, 97, paint);
    }
}
```

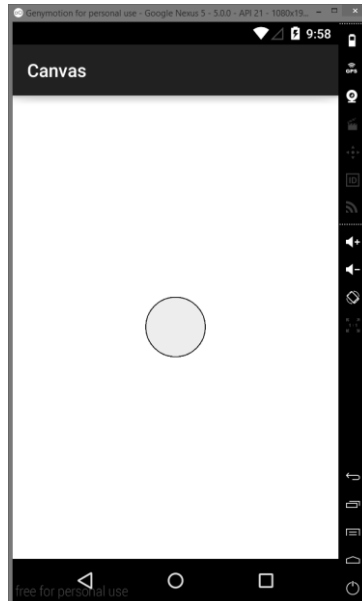


FIGURA 2.9 – Apresentação de um *bitmap* estático usando a classe `Canvas` e o método `onDraw`

Finalmente, no código da atividade, mais concretamente no seu método `onCreate`, crie um objeto `BallView` e use-o como argumento no método `setContentView` da atividade.

Este método é responsável por associar uma *view* a uma atividade:

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(new BallView(this));
}
```

Para já, este código apenas exibe um objeto gráfico estático. Se quisermos alterar a posição da bola e refletir esse movimento na interface gráfica, teríamos de esperar que a atividade fosse redesenhada. Para forçar o redesenho da *view* use o método `invalidate`:

```
protected void onDraw(Canvas canvas) {
    // INVOCAÇÃO DE MÉTODOS PARA O DESENHO
    invalidate();
}
```



Para solicitar a invalidação de uma *view* de uma *thread* diferente da *thread* da atividade, use o método `postInvalidate`.

De seguida, troca-se o círculo criado no exemplo anterior por uma bola de ténis representada por um *bitmap* e demonstra-se como gerir animações usando um objeto *Canvas* (Figura 2.10).

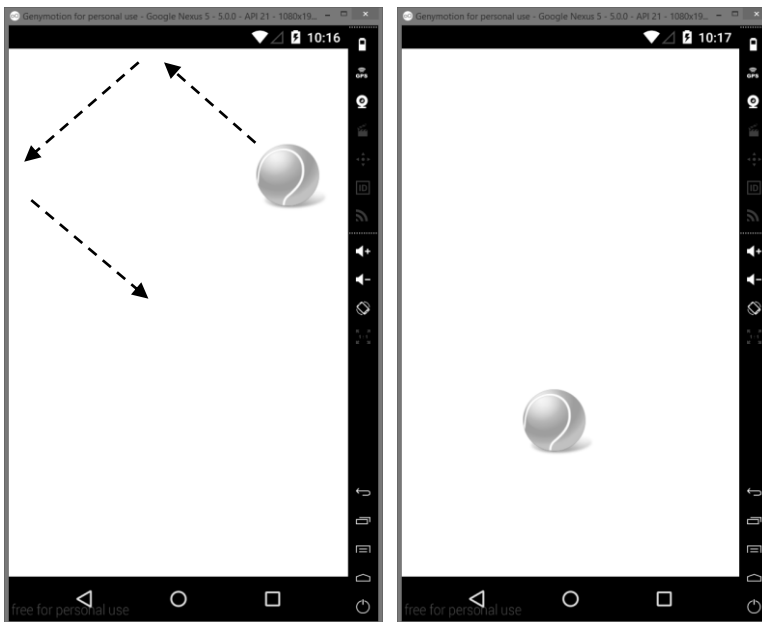


FIGURA 2.10 – Animações usando a classe *Canvas* e o método `onDraw`

Em primeiro lugar, define-se a classe `BallView` que estende a classe `View`. A classe inclui um construtor e o método `onDraw`. No construtor obtém-se uma referência para um objeto `BitmapDrawable` que representa o recurso `ball.png` anteriormente armazenado na pasta de recursos. Por sua vez, no método `onDraw` inclui-se a lógica para

o movimento da bola. A detecção da colisão entre a bola e os limites da *view* é baseada nos métodos `getWidth` e `getHeight` dos objetos `View` e do `BitmapDrawable` e nas coordenadas atuais da bola. Estas são mantidas nas variáveis `x` e `y`, e a “aceleração” da bola fica a cargo das variáveis `xSpeed` e `ySpeed`. Quando a bola atinge os limites da *view*, inverte-se o sentido do movimento multiplicando as variáveis de aceleração por `-1`. Depois de definidas as coordenadas atuais da bola, é altura de tratar da sua exibição. Para o desenho da bola, convertida num `BitmapDrawable`, usa-se o método `drawBitmap` do objeto `Canvas`. Depois disso, usa-se o método `invalidate` para que o método `onDraw` seja de novo chamado tão rápido quanto possível.

```
public class BallView extends View {
    BitmapDrawable ball;
    int x = -1, y = -1, xSpeed = 10, ySpeed = 5;

    public BallView(Context context) {
        super(context);
        ball = (BitmapDrawable)
            context.getResources().getDrawable(R.drawable.ball);
    }

    protected void onDraw(Canvas c) {
        if (x < 0 && y < 0) {
            x = this.getWidth()/2;
            y = this.getHeight()/2;
        } else {
            x += xSpeed;
            y += ySpeed;
            if ((x > this.getWidth() - ball.getBitmap().getWidth())
                || (x < 0)) {
                xSpeed = xSpeed * -1;
            }
            if ((y > this.getHeight() - ball.getBitmap().getHeight())
                || (y < 0)) {
                ySpeed = ySpeed * -1;
            }
        }
        c.drawBitmap(ball.getBitmap(), x, y, null);
        invalidate();
    }
}
```

A chamada do método `View.invalidate` no final do método `onDraw` indicará ao sistema Android para redesenhar a *view*. Contudo, não é garantido que o redesenho seja instantâneo. É preciso não esquecer que todas estas ações acontecem na mesma *thread* que a atividade principal, tornando todo este processo mais lento e imprevisível.

Finalmente, associe no método `onCreate` da atividade um objeto `BallView`:

```
setContentView(new BallView(this));
```

USO DE UMA *SURFACEVIEW*

A *SurfaceView* é uma subclasse da classe *View* que oferece uma superfície de desenho dedicada dentro da hierarquia das *views*. O objetivo desta classe é usar esta superfície de desenho numa *thread* secundária de uma aplicação, para que assim a aplicação não tenha de esperar até que a hierarquia de *views* do sistema esteja pronta a desenhar. Em vez disso, uma *thread* secundária com referência a um objeto *SurfaceView* pode desenhar para o seu próprio objeto *Canvas* e ao seu próprio ritmo.

Para começar, é necessário criar uma nova classe que estende a classe *SurfaceView*. A classe também deve implementar a interface *SurfaceHolder.Callback*, que irá notificá-lo com informações sobre a superfície subjacente, como quando ela é criada, alterada ou destruída. Estes eventos são importantes para que se saiba quando se pode começar a desenhar, quando é necessário fazer alguns ajustes com base em novas propriedades da superfície ou quando parar de desenhar e, potencialmente, parar algumas tarefas. No construtor da classe, deve-se obter um objeto *SurfaceHolder* através do método *getHolder*. Este objeto fornece acesso e controlo sobre a superfície subjacente da *SurfaceView*. Após isso, invoca-se o seu método *addCallback* para receber chamadas, via *SurfaceHolder.Callback*:

```
public MySurfaceView(Context context) {
    super(context);
    surfaceHolder = getHolder();
    surfaceHolder.addCallback(this);
}
```

Depois, basta substituir cada um dos métodos *SurfaceHolder.Callback* dentro da classe *SurfaceView*. No método *surfaceCreated*, que é chamado quando a superfície é criada, inicia-se uma nova *thread*:

```
public void surfaceCreated(SurfaceHolder holder) {
    new Thread(this).start();
}
```

De seguida, implementa-se o método *run* da interface *Runnable* que vai permitir trabalhar na segunda *thread*. Para começar a editar os pixels da superfície use o método *lockCanvas*, que devolve um objeto *Canvas* e que pode ser usado para desenhar no *bitmap* da superfície. Após o desenho, invoca-se o método *unlockCanvasAndPost*, que termina a edição de pixels na superfície. Após esta chamada, os pixels atuais da superfície são mostrados no ecrã:

```
@Override
public void run() {
    ...
    canvas = surfaceHolder.lockCanvas();
    // USAR O OBJETO CANVAS PARA DESENHAR
    surfaceHolder.unlockCanvasAndPost(canvas);
}
```


O próximo exemplo (Figura 2.11) demonstra o uso da `SurfaceView` através da implementação de uma aplicação que exibe uma série de carros, que surgem de forma aleatória, a cair desde o topo do ecrã.

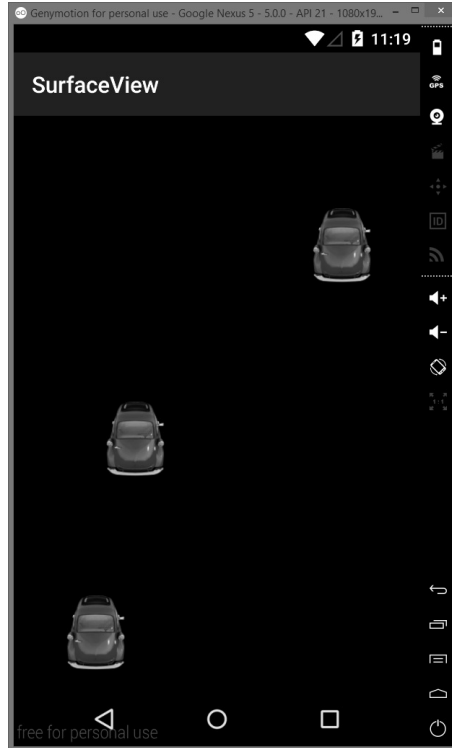


FIGURA 2.11 – Animações usando a classe `SurfaceView`

Comece por adicionar o ficheiro **mycar.png** na pasta de recursos. Depois, inicie a codificação da classe `Carro` que vai modelar a entidade `carro`. A classe inclui as coordenadas (x , y) do carro, um membro estático com uma referência para um objeto `Drawable` com a imagem do carro e os métodos de leitura e escrita (*getters* e *setters*) correspondentes.

```
public class Carro {
    private int x, y;
    static BitmapDrawable carro;
    Carro(Context context) {
        carro = (BitmapDrawable)
            context.getResources().getDrawable(R.drawable.mycar);
    }
    public static Bitmap getImage() { return carro.getBitmap(); }
    public int getX() { return x; }
    public void setX(int x) { this.x = x; }
    public int getY() { return y; }
    public void setY(int y) { this.y = y; }
}
```

De seguida, crie a classe `ChuvaCarrosView` que estende uma `SurfaceView` e implementa as interfaces `SurfaceHolder.Callback` e `Runnable`:

```
public class CarrosView extends SurfaceView implements
    SurfaceHolder.Callback, Runnable {
    private int nCarros = 10, salto = 5;
    private ArrayList<Carro> Cars = new ArrayList<Carro>();
    private Context context;
    private SurfaceHolder surfaceHolder;
    public CarrosView(Context context) {
        super(context);
        surfaceHolder = getHolder();
        surfaceHolder.addCallback(this);
        this.context = context;
    }
    @Override
    public void surfaceCreated(SurfaceHolder holder) {
        new Thread(this).start();
    }
    @Override
    public void surfaceChanged(SurfaceHolder sh, int f, int w, int h) {}
    @Override
    public void surfaceDestroyed(SurfaceHolder sh) {}
    @Override
    public void run() {
        Canvas canvas = null;
        while(Cars.size() < nCarros) {
            canvas = surfaceHolder.lockCanvas();
            render(canvas);
            surfaceHolder.unlockCanvasAndPost(canvas);
        }
    }
    private void render(Canvas canvas) {
        canvas.drawColor(Color.BLACK);
        if(Cars.size()==0 ||
            Cars.get(Cars.size()-1).getY() >
            canvas.getHeight()/3) {
            Carro myCar = new Carro(context);
            myCar.setX(1 + (int)(Math.random() * (canvas.getWidth() -
                Carro.getImage().getWidth())));
            myCar.setY(1);
            Cars.add(myCar);
        }
        for (int i = 0; i < Cars.size(); i++) {
            if(Cars.get(i).getY() < canvas.getHeight()) {
                canvas.drawBitmap(Carro.getImage(), Cars.get(i).getX(),
                    Cars.get(i).getY(), null);
                Cars.get(i).setY(Cars.get(i).getY() + salto);
            }
        }
    }
}
```

O método `run` será invocado assim que a superfície for criada. No seu interior é incluído um ciclo que irá controlar a duração do jogo. Neste caso, a condição de paragem do ciclo será quando o número de carros gerados em posições aleatórias atingir um número de carros fixado à partida. No seu interior, o método invoca a função `render` responsável pelo desenho no ecrã.

A geração de um novo carro e a sua respetiva adição a uma lista de carros é feita apenas quando é obedecida uma de duas condições: 1) ainda não foi gerado qualquer carro ou 2) o último carro gerado já atingiu um terço da altura do objeto `Canvas`.

Depois, basta iterar sobre a lista de carros e desenhá-los no `Canvas` de acordo com a sua posição. A posição do carro é também atualizada de forma a ser refletida no próximo desenho. Para terminar, associe uma instância da classe `CarrosView` que acabou de criar à atividade principal.

2.2.1.3 OPENGL ES

O **OpenGL ES** (*OpenGL for Embedded Systems*) é um subconjunto da biblioteca de gráficos tridimensionais OpenGL para o desenho 2D/3D de gráficos, tais como os que são usados em videojogos. O OpenGL ES foi projetado para sistemas embarcados, como *smartphones* e *tablets*. A Tabela 2.4 mostra uma distribuição das percentagens de dispositivos que usam as várias versões do OpenGL ES.

VERSÃO OPENGL ES	VERSÃO ANDROID	NÍVEL API	% DE USO ⁸
2.0	Android 2.2	8	65,9
3.0	Android 4.3	18	33,8
3.1	Android 5.0	21	0,3

TABELA 2.4 – Histórico das versões do OpenGL ES no Android



Para declarar qual a versão do OpenGL ES de que a sua aplicação precisa, deve usar o atributo `android:glEsVersion` do elemento `<uses-feature>` no ficheiro de manifesto. Adicionar essa declaração faz com que o Google Play restrinja a aplicação de ser instalada em dispositivos que não suportem o OpenGL ES.

O Android suporta o OpenGL ES através da sua *framework* incluída no pacote `android.opengl` ou de API nativas providenciadas pelo Android *Native Development Kit* (NDK). As próximas secções abordam a primeira alternativa.

⁸ Baseada nos acessos à loja *online* Google Play durante uma semana até 6 de abril de 2015.

CLASSES `GLSurfaceView` E `GLSurfaceView.Renderer`

Para criar e manipular gráficos usando o OpenGL ES é necessário dominar duas das suas classes principais: `GLSurfaceView` e `GLSurfaceView.Renderer`.

A classe `GLSurfaceView` é uma *view* onde se podem desenhar e manipular objetos gráficos (similar ao `SurfaceView`). Usa-se esta classe para criar uma instância de `GLSurfaceView` e adicionar-lhe um objeto `Renderer`.

A classe `GLSurfaceView.Renderer` é uma interface que define os métodos necessários para o desenho de gráficos numa `GLSurfaceView`. É necessário fornecer uma implementação desta interface como uma classe separada e associar a instância à `GLSurfaceView` através do método `GLSurfaceView.setRenderer`. A interface `GLSurfaceView.Renderer` necessita que se implementem os seguintes métodos:

- ⊗ `onSurfaceCreated` – chamado uma única vez, quando é criada a `GLSurfaceView`. Usar este método sempre que se queiram executar ações que devam acontecer apenas uma vez (por exemplo, configurar parâmetros de ambiente do OpenGL ES, inicializar objetos gráficos);
- ⊗ `onDrawFrame` – chamado cada vez que a `GLSurfaceView` necessita de ser redeseenhada. Usar este método como ponto principal de (re)desenho dos objetos gráficos;
- ⊗ `onSurfaceChanged` – chamado quando a geometria da `GLSurfaceView` é alterada (por exemplo, mudanças no tamanho da `GLSurfaceView`, alteração na orientação do ecrã do dispositivo, etc.). Usar este método para responder às mudanças no contentor da `GLSurfaceView`.



Muito mais haveria a dizer sobre o desenho usando o OpenGL ES. Para mais detalhes sobre o assunto consulte o *link*: <http://developer.android.com/training/graphics/opengl/index.html>.

A partir daqui pode-se começar a usar as API do OpenGL ES. Para usar a versão 2.0 use o pacote `android.opengl.GLES20`, que fornece uma interface de alto nível para o OpenGL ES 2.0, estando disponível a partir do Android 2.2 (API nível 8).

EXEMPLO PRÁTICO

Para exemplificar o uso do OpenGL ES 2.0 demonstra-se a criação de uma aplicação⁹ que desenha um triângulo no ecrã e que permite a sua rotação através do toque.

⁹ Código completo disponível na página do livro em www.fca.pt (até este se esgotar ou ser publicada nova edição atualizada ou com alterações).

Para tal siga os próximos passos:

- 1) Crie um novo projeto intitulado **OpenGLTriangle**.
- 2) Na atividade principal implemente os métodos `onCreate`, `onPause` e `onResume`.

```
public class OpenGLTriangle_Activity extends Activity {
    private GLSurfaceView myGLView;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // CRIA UMA INSTÂNCIA DE GLSURFACEVIEW E DEFINE O CONTEÚDO DA ATIVIDADE COMO SENDO A VIEW
        myGLView = new MyGLSurfaceView(this);
        setContentView(myGLView);
    }
    @Override
    protected void onPause() {
        super.onPause();
        // PARA A THREAD DE RENDERIZAÇÃO PARA POUPAR MEMÓRIA
        myGLView.onPause();
    }
    @Override
    protected void onResume() {
        super.onPause();
        // RETOMA A THREAD RESPONSÁVEL PELA RENDERIZAÇÃO GRÁFICA
        myGLView.onResume();
    }
}
```

- 3) Crie a classe `MyGLSurfaceView` que estende a classe `GLSurfaceView`, que não é mais do que uma *view* especializada onde se podem desenhar gráficos OpenGL ES. Esta *view* também é usada para capturar os eventos de toque.

```
public class MyGLSurfaceView extends GLSurfaceView {
    private final MyGLRenderer myRenderer;
    public MyGLSurfaceView(Context context) {
        super(context);
        // CRIA UM CONTEXTO PARA O OPENGL ES 2.0
        setEGLContextClientVersion(2);
        // DEFINE O CONFIGURADOR
        setEGLConfigChooser(8, 8, 8, 8, 16, 0);
        // DEFINE O RENDERIZADOR PARA DESENHAR NA GLSURFACEVIEW
        myRenderer = new MyGLRenderer();
        setRenderer(myRenderer);
        // FAZ O RENDER DA VIEW APENAS QUANDO HÁ UMA ALTERAÇÃO NOS DADOS DE DESENHO
        setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
    }
    // O MÉTODO MOTIONEVENT LIDA COM OS TOQUES NO ECRÃ. NESTE CASO, SÓ
    // NOS INTERESSAM OS EVENTOS ONDE A POSIÇÃO DE TOQUE É ALTERADA
    @Override
    public boolean onTouchEvent(MotionEvent e) {
        // CÓDIGO NÃO INCLUÍDO POR QUESTÕES DE LEGIBILIDADE
    }
}
```

- 4) Crie uma classe com o nome `MyGLRenderer` que implementa a interface `GLSurfaceView.Renderer`. Esta classe implementa um renderizador, isto é, define aquilo que vai ser desenhado no `GLSurfaceView`. Existem três métodos que necessitam de ser sobrepostos.

```
public class MyGLRenderer implements GLSurfaceView.Renderer {
    private Triangle myTriangle;
    public void onSurfaceCreated(GL10 unused, EGLConfig config) {
        // DEFINE A COR DE FUNDO DA FRAME
        GLES20.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
        // INSTANCIA O OBJETO GRÁFICO A DESENHAR
        myTriangle = new Triangle();
    }
    public void onDrawFrame(GL10 unused) {
        // REDESENHA A COR DE FUNDO
        GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT |
            GLES20.GL_DEPTH_BUFFER_BIT);
        // DEFINE A POSIÇÃO DA CÂMARA (VIEW MATRIX)
        Matrix.setLookAtM(mViewMatrix, 0, 0, 0, -3, 0f, 0f, 0f, 0f, 1.0f,
            0.0f);
        // CALCULA A PROJEÇÃO E A TRANSFORMAÇÃO DA VIEW
        Matrix.multiplyMM(mMVPMatrix, 0, mProjectionMatrix, 0,
            mViewMatrix, 0);
        // CRIA A ROTAÇÃO PARA O TRIÂNGULO
        Matrix.setRotateM(mRotationMatrix, 0, mAngle, 0, 0, 1.0f);
        // COMBINA A MATRIZ DE ROTAÇÃO COM A PROJEÇÃO
        Matrix.multiplyMM(scratch, 0, mMVPMatrix, 0, mRotationMatrix, 0);
        // DESENHA O OBJETO GRÁFICO, NESTE CASO UM TRIÂNGULO
        mTriangle.draw(scratch);
    }
    public void onSurfaceChanged(GL10 unused, int width, int height) {
        // AJUSTA O VIEWPORT BASEADO NAS ALTERAÇÕES À GEOMETRIA DA VIEW
        GLES20.glViewport(0, 0, width, height);
        float ratio = (float) width / height;
        // A MATRIZ DE PROJEÇÃO É APLICADA ÀS COORDENADAS DO OBJETO NO MÉTODO ONDRAWFRAME()
        Matrix.frustumM(mProjectionMatrix, 0, -ratio, ratio, -1, 1, 3, 7);
    }
}
```

- 5) Crie uma classe com o nome `MyTriangle` que representa um triângulo bidimensional para o uso como um objeto desenhado em OpenGL ES 2.0.

```
// CÓDIGO NÃO INCLUÍDO POR QUESTÕES DE LEGIBILIDADE
public class MyTriangle {
    // DEFINE OS DADOS DO OBJETO DE DESENHO PARA USO NO OPENGL ES
    public Triangle() {}
    // ENCAPSULA AS INSTRUÇÕES OPENGL ES PARA DESENHAR ESTA FORMA
    public void draw(float[].mvpMatrix) {}
}
```

- 6) Execute a aplicação no emulador (ou dispositivo). Após o aparecimento do triângulo (Figura 2.12), rode-o nos dois sentidos (esquerda e direita).

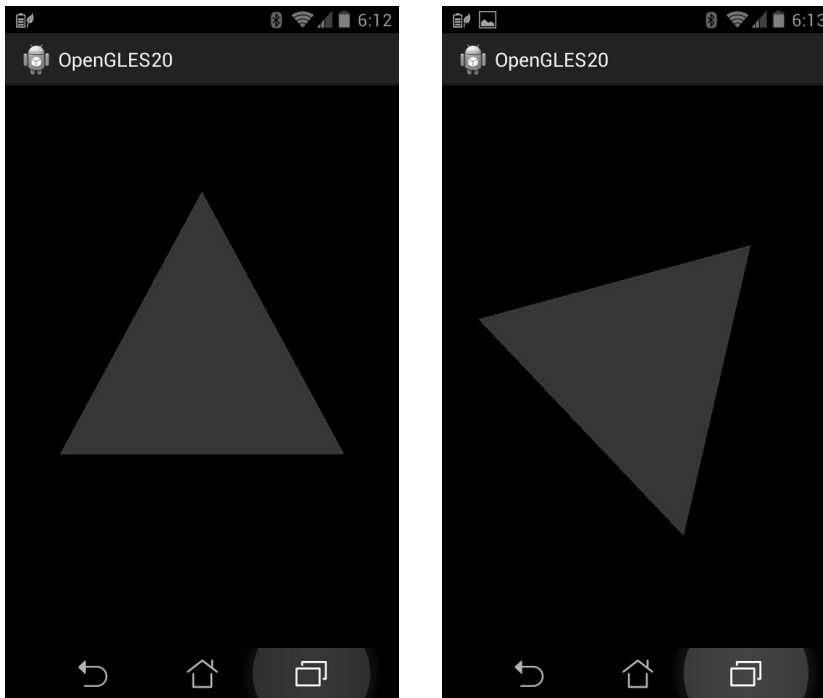


FIGURA 2.12 – Desenho 2D com o OpenGL ES

Podemos aplicar estes conceitos a uma aplicação exemplo que use o desenho a três dimensões (3D). Neste caso, o objeto gráfico é um cubo com a textura baseada no logotipo da editora FCA. A rotação do cubo é aleatória, sem qualquer intervenção humana (Figura 2.13). A maior novidade aqui é o carregamento da textura para o cubo. O método estático `loadTexture` da classe `Cube` é invocado a partir do método `onSurfaceCreated` da classe `MyGLRenderer`:

```
class MyGLRenderer implements GLSurfaceView.Renderer {
    public void onSurfaceCreated(GL10 gl, EGLConfig config) {
        // CARREGA A TEXTURA DO CUBO A PARTIR DE UM BITMAP
        MyCube.loadTexture(gl, context, R.drawable.fca);
    }
}

public class MyCube() {
    static void loadTexture(GL10 gl, Context context, int resource) {
        Bitmap bmp = BitmapFactory.decodeResource(
            context.getResources(), resource);
        GLUtils.texImage2D(GL10.GL_TEXTURE_2D, 0, bmp, 0);
        gl.glTexParameterx(GL10.GL_TEXTURE_2D,
            GL10.GL_TEXTURE_MIN_FILTER, GL10.GL_LINEAR);
        gl.glTexParameterx(GL10.GL_TEXTURE_2D,
            GL10.GL_TEXTURE_MAG_FILTER, GL10.GL_LINEAR);
        bmp.recycle();
    }
}
```



FIGURA 2.13 – Desenho 3D com o OpenGL ES

2.2.2 INPUT

Uma boa gestão da interação do utilizador com a aplicação favorece uma experiência de utilização mais rica. Ao invés, uma experiência pobre pode levar o utilizador ao descrédito e, conseqüentemente, ao abandono do uso da aplicação. Nesta secção discutem-se os três dispositivos de entrada mais relevantes para obter informação: o ecrã sensível ao toque (*touchscreen*), o teclado e os sensores.

2.2.2.1 EVENTOS DE TOQUE

O toque no ecrã é o tipo de *input* mais importante que os utilizadores têm para interagir com as aplicações. Apesar de não se dever depender exclusivamente do *touchscreen*, já que o mesmo pode não estar disponível para todos os utilizadores, ou em todos os contextos da aplicação, a adição de interação baseada em toque aumenta significativamente a utilidade e usabilidade de uma aplicação.

Até à versão 2.0, o Android só suportava o processamento de eventos de toque simples. A partir dessa versão, o multitoque foi introduzido. Ambas as modalidades de toque são explicadas de seguida.

PROCESSAMENTO DE EVENTOS DE TOQUE ÚNICO

Para intercetar eventos de toque numa *view* deve-se implementar o método `onTouchEvent(MotionEvent event)`. O único parâmetro é um objeto `MotionEvent`, que permite obter informações sobre o tipo do evento de toque. Os principais métodos da classe `MotionEvent` são enumerados na Tabela 2.5.

MÉTODO	DESCRIÇÃO
<code>getX() / getY()</code>	Devolvem as coordenadas <i>x</i> e <i>y</i> do evento de toque em relação à <i>view</i> . O sistema de coordenadas é definido com a origem no canto superior esquerdo do ecrã, com o eixo <i>X</i> apontando para a direita e o eixo <i>Y</i> apontando para baixo. As coordenadas são dadas em pixels. Note-se que as coordenadas têm precisão ao nível do subpixel, já que os métodos devolvem valores em <code>float</code> .
<code>getActionMasked()</code>	Este método devolve informação sobre o tipo do evento de toque. É um número inteiro que assume um dos seguintes valores: <ul style="list-style-type: none"> ⊗ <code>MotionEvent.ACTION_DOWN</code> – novo toque no ecrã; ⊗ <code>MotionEvent.ACTION_MOVE</code> – o ponteiro está a mover-se; ⊗ <code>MotionEvent.ACTION_UP</code> – toque libertado; ⊗ <code>MotionEvent.ACTION_CANCEL</code> – evento cancelado.

TABELA 2.5 – Principais métodos da classe `MotionEvent`

O método devolve verdadeiro se o evento foi tratado e falso, caso contrário. O próximo exemplo (Figura 2.14) mostra como criar uma aplicação que permite desenhar através do toque:

```
public class SingleTouchView extends View {
    private Paint paint = new Paint();
    private Path path = new Path();
    private float eventX, eventY;
    private boolean fingerDown = false;

    public SingleTouchView(Context context, AttributeSet attrs) {
        super(context, attrs);
        paint.setAntiAlias(true);
        paint.setStrokeWidth(6f);
        paint.setColor(Color.BLACK);
        paint.setStyle(Paint.Style.STROKE);
        paint.setStrokeJoin(Paint.Join.ROUND);
    }
    protected void onDraw(Canvas canvas) {
        canvas.drawPath(path, paint);
        if (fingerDown) {
            paint.setColor(Color.BLUE);
            canvas.drawCircle(eventX, eventY, 20, paint);
            paint.setColor(Color.BLACK);
        }
    }
}
```

```
    }  
}  
public boolean onTouchEvent(MotionEvent event) {  
    eventX = event.getX();  
    eventY = event.getY();  
    switch (event.getActionMasked()) {  
        case MotionEvent.ACTION_DOWN:  
            fingerDown = true;  
            path.moveTo(eventX, eventY);  
            break;  
        case MotionEvent.ACTION_MOVE:  
            path.lineTo(eventX, eventY);  
            break;  
        case MotionEvent.ACTION_UP:  
            fingerDown = false;  
            break;  
        default: return false;  
    }  
    // MARCA UM REPAINT  
    invalidate();  
    return true;  
}  
}
```



FIGURA 2.14 – A aplicação SingleTouch

No construtor da classe `SingleTouchView`, que estende a classe `View`, trata-se da configuração do objeto `Paint`. No método `onDraw`, desenha-se o caminho percorrido pelo

dedo através do método `canvas.drawPath(path, paint)`. Um dos seus parâmetros é um objeto `Path`, que permite a definição de caminhos geométricos (por exemplo, segmentos de reta, curvas quadráticas e cúbicas). No método `onTouchEvent`, deteta-se o tipo do evento de toque para controlar o desenho do caminho através dos métodos `moveTo` e `lineTo` do objeto `Path`. Por fim, associe a nova *view* à atividade:

```
setContentView(new SingleTouchView(this, null));
```



Como uma alternativa ao método `onTouchEvent`, pode-se associar um objeto `View.OnTouchListener` a qualquer objeto `View` usando o método `setOnTouchListener`. Isso torna possível escutar eventos de toque através do método `onTouch` sem necessidade de estender uma *view* existente. Por exemplo:

```
View myView = findViewById(R.id.my_view);
myView.setOnTouchListener(new OnTouchListener() {
    public boolean onTouch(View v, MotionEvent event) {...}
});
```

Contudo, se a aplicação usa gestos como o duplo toque, o toque longo, o arrastar, entre outros, pode-se tirar proveito da classe `GestureDetector`. Começa-se por instanciar um objeto da classe `GestureDetectorCompat`, que usa dois parâmetros: o contexto da aplicação e a classe que implementa a interface `GestureDetector.OnGestureListener`. De forma a que o objeto `GestureDetector` receba eventos de toque, sobrepõe-se o método `onTouchEvent` e passam-se todos os eventos para a instância do detetor criado. Alguns dos métodos públicos mais usados desta interface são: `onDown`, `onFling`, `onLongPress`, `onScroll`, `onShowPress` e `onSingleTapUp`. O esqueleto de código padrão numa aplicação de deteção de gestos comuns é o seguinte:

```
public class MainActivity extends Activity implements
    GestureDetector.OnGestureListener {
    private GestureDetectorCompat myDetector;
    public void onCreate(Bundle savedInstanceState) {
        // INSTANCIÇÃO DO DETETOR DE GESTOS
        myDetector = new GestureDetectorCompat(this, this);
    }
    @Override
    public boolean onTouchEvent(MotionEvent event) {
        this.myDetector.onTouchEvent(event);
        return super.onTouchEvent(event);
    }
    public boolean onDown(MotionEvent e) { ... }
    public boolean onFling(MotionEvent e1, MotionEvent e2,
        float velocityX, float velocityY) { ... }
    public void onLongPress(MotionEvent e) { ... }
    public boolean onScroll(MotionEvent e1, MotionEvent e2, float
        distanceX, float distanceY) { ... }
    public void onShowPress(MotionEvent e) { ... }
    public boolean onSingleTapUp(MotionEvent e) { ... }
}
```

PROCESSAMENTO DE EVENTOS DE MULTITOQUE

Um gesto multitoque acontece quando vários ponteiros (dedos) tocam no ecrã ao mesmo tempo. Neste cenário, o sistema gera vários eventos de toque. A Tabela 2.6 refere alguns dos eventos mais usados.

EVENTO	DESCRIÇÃO
<code>MotionEvent.ACTION_DOWN</code>	Para o primeiro ponteiro que toca no ecrã. Inicia o gesto e obtém o índice 0 no <code>MotionEvent</code> .
<code>MotionEvent.ACTION_POINTER_DOWN</code>	Para ponteiros extra que tocam no ecrã para além do primeiro. Os dados do ponteiro são obtidos através do índice devolvido pelo método <code>getActionIndex</code> .
<code>MotionEvent.ACTION_MOVE</code>	O ponteiro moveu-se.
<code>MotionEvent.ACTION_POINTER_UP</code>	Enviado quando um ponteiro não-primário sobe.
<code>MotionEvent.ACTION_UP</code>	Enviado quando o último ponteiro abandona o ecrã.

TABELA 2.6 – Eventos multitoque na classe `MotionEvent`

Os vários movimentos em simultâneo são chamados ponteiros. Para lidar com cada ponteiro individualmente é necessário ter em conta o seu identificador e índice. Cada ponteiro tem uma identificação única que é atribuída quando o ponteiro fica no estado `ACTION_DOWN` (ou `ACTION_POINTER_DOWN`). O identificador do ponteiro permanece válido até que o ponteiro suba (indicado por `ACTION_UP` ou `ACTION_POINTER_UP`) ou quando o movimento é cancelado (indicado por `ACTION_CANCEL`). A classe `MotionEvent` fornece muitos métodos para consultar a posição e outras propriedades dos ponteiros, como os métodos `getX`, `getY`, `getPointerId`, `getToolType`, entre muitos outros. A maior parte destes métodos aceita o índice de ponteiro como um parâmetro em vez do próprio identificador do ponteiro. O índice de ponteiro de cada ponteiro varia entre 0 e o valor devolvido por `getPointerCount-1`.

A ordem pela qual os ponteiros individuais aparecem dentro de um evento de movimento é indefinida. Assim, o índice de um ponteiro pode mudar de um evento para o seguinte, mas o identificador do ponteiro é garantido que permaneça constante, desde que o ponteiro permaneça ativo. Use o método `getPointerId` para obter o ID de um ponteiro de forma a segui-lo em todos os eventos de movimento subsequentes num gesto. Então, para eventos de movimento sucessivo use o método `findPointerIndex` para obter o índice de ponteiro para um determinado ID de ponteiro nesse evento de

movimento. O próximo exemplo mostra uma aplicação que recebe e manipula vários ponteiros:

```
public class MultiTouchView extends View {
    private static final int SIZE = 60;
    private SparseArray<PointF> mActivePointers;
    private Paint mPaint;
    private int[] colors = { Color.BLUE, Color.GREEN, Color.MAGENTA,
        Color.BLACK, Color.CYAN, Color.GRAY, Color.RED, Color.DKGRAY,
        Color.LTGRAY, Color.YELLOW };
    private Paint textPaint;
    public MultiTouchView(Context context, AttributeSet attrs) {
        super(context, attrs);
        initView();
    }
    private void initView() {
        mActivePointers = new SparseArray<PointF>();
        mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
        mPaint.setColor(Color.BLUE);
        mPaint.setStyle(Paint.Style.FILL_AND_STROKE);
        textPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
        textPaint.setTextSize(20);
    }
    public boolean onTouchEvent(MotionEvent event) {
        // OBTÉM ÍNDICE E IDENTIFICADOR DO PONTEIRO
        int pointerIndex = event.getActionIndex();
        int pointerId = event.getPointerId(pointerIndex);
        // OBTÉM A AÇÃO
        int maskedAction = event.getActionMasked();
        switch (maskedAction) {
            case MotionEvent.ACTION_DOWN:
            case MotionEvent.ACTION_POINTER_DOWN: {
                // NOVO PONTEIRO, LOGO ADICIONA-SE À LISTA DE PONTEIROS
                PointF f = new PointF();
                f.x = event.getX(pointerIndex);
                f.y = event.getY(pointerIndex);
                mActivePointers.put(pointerId, f);
                break;
            }
            case MotionEvent.ACTION_MOVE: {
                // UM PONTEIRO MOVEU-SE
                for (int size = event.getPointerCount(), i = 0; i < size; i++) {
                    PointF point = mActivePointers.get(event.getPointerId(i));
                    if (point != null) {
                        point.x = event.getX(i);
                        point.y = event.getY(i);
                    }
                }
                break;
            }
            case MotionEvent.ACTION_CANCEL: {
                mActivePointers.remove(pointerId);
                break;
            }
        }
    }
}
```

```
    }
    invalidate();
    return true;
}
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    // DESENHA TODOS OS PONTEIROS
    for (int size = mActivePointers.size(), i = 0; i < size; i++) {
        PointF point = mActivePointers.valueAt(i);
        if (point != null) {
            mPaint.setColor(colors[i % 9]);
            canvas.drawCircle(point.x, point.y, SIZE, mPaint);
        }
    }
    canvas.drawText("Total pointers: " + mActivePointers.size(), 10, 40 ,
        textPaint);
}
}
```

Altere o *layout* da aplicação de forma a incluir a nova *view*, modificando novamente o método `setContentView` no evento `onCreate` da atividade:

```
setContentView(new MultiTouchView(this,null));
```

Execute a aplicação num dispositivo real e interaja com a aplicação conforme demonstrado na Figura 2.15.

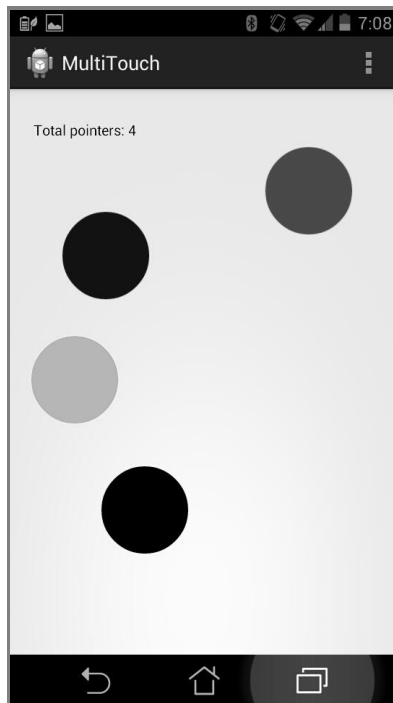


FIGURA 2.15 – A aplicação MultiTouch

O Android suporta outros gestos multitoque, nomeadamente o de “arrastar e soltar” (mais conhecido como *drag-and-drop*). A operação *drag-and-drop* inicia-se quando o utilizador faz um gesto de arrastamento de um objeto no *layout* da aplicação. Em resposta, a aplicação informa o sistema que o processo de arrastamento está a iniciar-se. O sistema obtém uma representação dos dados que estão a ser arrastados. À medida que o dedo do utilizador move a representação (*drag shadow*) sobre o *layout* atual, o sistema envia eventos de arrastamento representados pela classe `DragEvent`. Uma vez que o utilizador solta o *drag shadow*, o sistema termina a operação de arrastamento.

Um objeto `View` pode manipular eventos de *drag-and-drop* através da implementação da interface `View.OnDragListener` passada como parâmetro no método `setOnDragListener`. Na implementação da interface sobrepõe-se o método `onDrag` que recebe um objeto `DragEvent`. O objeto representa os diferentes estados do evento podendo-se obter o estado atual através do método `getAction`.

A Tabela 2.7 enumera os seis valores possíveis definidos como constantes na classe `DragEvent`.

VALOR PARA O <code>GETACTION</code>	DESCRIÇÃO
<code>ACTION_DRAG_STARTED</code>	Sinaliza o início de uma operação de arrastar e soltar. Executado logo após a invocação de <code>startDrag</code> e a obtenção do <i>drag shadow</i> .
<code>ACTION_DRAG_ENTERED</code>	Informa a <i>view</i> que o <i>drag shadow</i> entrou na sua área.
<code>ACTION_DRAG_LOCATION</code>	Enviado à <i>view</i> enquanto o <i>drag shadow</i> se mantiver dentro dos seus limites.
<code>ACTION_DRAG_EXITED</code>	Enviado à <i>view</i> quando o <i>drag shadow</i> sair dos seus limites.
<code>ACTION_DROP</code>	Enviado à <i>view</i> quando o <i>drag shadow</i> é solto sob o objeto <code>View</code> . Note que este tipo de ação não é enviado se o utilizador soltar o <i>drag shadow</i> numa <i>view</i> que não tem um <i>listener</i> registado.
<code>ACTION_DRAG_ENDED</code>	Sinaliza o fim de uma operação de “arrastar e soltar”.

TABELA 2.7 – Tipos de ação devolvidos pelo evento de “arrastar e soltar”

No próximo exemplo mostra-se como usar o gesto de *drag-and-drop* através do arrastamento de uma bola para um cesto (Figura 2.16). É exibida uma mensagem por debaixo do cesto indicando o número de bolas que foram soltadas para o cesto em comparação com o número de tentativas.

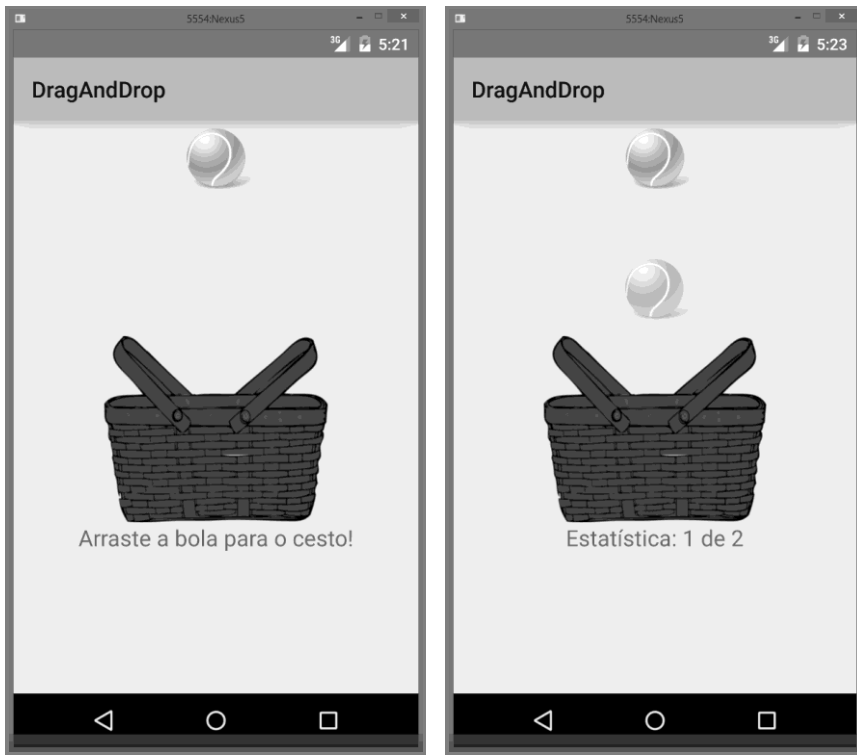


FIGURA 2.16 – Aplicar *drag-and-drop* numa imagem

Comece por criar um novo projeto. De seguida, adicione na pasta de recursos as imagens **basket.png** e **ball.png**, que representam o cesto e a bola, respetivamente. Depois, insira no *layout* da aplicação dois objetos `ImageView` para a bola e o cesto e um objeto `TextView` para as mensagens. Finalmente, codifica-se a atividade conforme o próximo excerto de código:

```
public class MainActivity extends Activity {
    ImageView drag, drop;
    TextView message;
    int total, sucesso = 0;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        drag = (ImageView) findViewById(R.id.imageViewDrag);
        drop = (ImageView) findViewById(R.id.imageViewDrop);
        message = (TextView) findViewById(R.id.message);
        drop.setOnDragListener(new View.OnDragListener() {
            @Override
            public boolean onDrag(View v, DragEvent event) {
                final int action = event.getAction();
                switch(action) {
                    case DragEvent.ACTION_DRAG_STARTED: break;
                }
            }
        });
    }
}
```



```
case DragEvent.ACTION_DRAG_EXITED: break;
case DragEvent.ACTION_DRAG_ENTERED: break;
case DragEvent.ACTION_DROP:
{
    sucesso = sucesso + 1;
    return(true);
}
case DragEvent.ACTION_DRAG_ENDED:{
    total = total +1;
    message.setText("Estatística: "+ sucesso + " de " + total);
    return(true);
}
default: break;
}
return true;
}});

drag.setOnTouchListener(new OnTouchListener() {
@Override
public boolean onTouch(View v, MotionEvent arg1) {
    ClipData data = ClipData.newPlainText("", "");
    View.DragShadowBuilder shadow = new
        View.DragShadowBuilder(drag);
    v.startDrag(data, shadow, null, 0);
    return false;
}
});
}
```

Falta explicar a criação do *drag shadow*. Primeiro cria-se um objeto `ClipData`, que não é mais do que uma representação dos dados a serem movidos através do método `newPlainText`. De seguida, usa-se o método `View.DragShadowBuilder(View)` para criar um *drag shadow* por omissão, que tem o mesmo tamanho da *view* passada como argumento. Finalmente, usa-se o método `startDrag` que inclui vários argumentos, entre os quais os dados a serem movidos e o respetivo *drag shadow*.

2.2.2.2 EVENTOS DE TECLADO

Quando o utilizador interage com uma caixa de texto e tem um teclado de *hardware* no dispositivo, a entrada de dados é tratada pelo sistema. Para intercepar e lidar diretamente com a pressão de teclas de um teclado a partir de uma atividade ou de uma *view*, implementa-se uma série de métodos pertencentes à interface `KeyEvent.Callback`, como, por exemplo, o método `onKeyUp`, cuja assinatura é a seguinte:

```
public boolean onKeyUp (int keyCode, KeyEvent event)
```

O primeiro parâmetro contém um valor inteiro identificando a tecla pressionada. O segundo parâmetro contém uma descrição sobre o evento. Normalmente, deve usar-se o método `onKeyUp` de forma a ter a certeza de que se recebe apenas um evento. Se o utilizador pressionar e segurar o botão durante algum tempo, então o método `onKeyDown` é chamado várias vezes. Por exemplo, esta aplicação responde a algumas teclas do teclado para controlar um jogo:

```
@Override
public boolean onKeyUp(int keyCode, KeyEvent event) {
    switch (keyCode) {
        case KeyEvent.KEYCODE_D:
            moveShip(MOVE_LEFT);
            return true;
        case KeyEvent.KEYCODE_F:
            moveShip(MOVE_RIGHT);
            return true;
        case KeyEvent.KEYCODE_J:
            fireMachineGun();
            return true;
        default:
            return super.onKeyUp(keyCode, event);
    }
}
```

Para responder a eventos de teclado que combinem teclas, como quando uma tecla é combinada com **Shift** ou **Control**, pode-se consultar o objeto `KeyEvent` que é passado no método `onKeyUp` e verificar qual a tecla modificadora que está a ser pressionada através dos métodos `isShiftPressed` ou `isCtrlPressed`.

Por exemplo, a próxima aplicação verifica se a tecla **Shift** foi pressionada com a tecla **J** para executar uma determinada ação:

```
@Override
public boolean onKeyUp(int keyCode, KeyEvent event) {
    switch (keyCode) {
        case KeyEvent.KEYCODE_J:
            if (event.isShiftPressed()) {
                fireLaser();
            }
            else {
                fireMachineGun();
            }
            return true;
        default:
            return super.onKeyUp(keyCode, event);
    }
}
```

2.2.2.3 EVENTOS DE SENSORES

A plataforma Android suporta três grandes categorias de sensores:

- ⊙ **Sensores de movimento** – medem as forças de aceleração e de rotação ao longo de três eixos. Esta categoria inclui, entre outros, acelerómetros, sensores de gravidade e giroscópios;
- ⊙ **Sensores ambientais** – medem vários parâmetros ambientais, tais como a temperatura ambiente do ar e da pressão, a iluminação e a humidade. Esta categoria inclui barómetros, fotómetros e termómetros;
- ⊙ **Sensores de posição** – medem a posição física de um dispositivo. Esta categoria inclui sensores de orientação e magnetómetros.

Os sensores são baseados em *hardware* (H) e/ou em *software* (S). Os sensores de *hardware* são componentes físicos que derivam os seus dados através da medição direta de propriedades específicas ambientais, tais como a aceleração, a força do campo geomagnético ou a mudança angular. Os sensores de *software* (virtuais) obtêm os seus dados a partir de sensores de *hardware*. O sensor de aceleração linear e o de gravidade são alguns exemplos. A Tabela 2.8 resume alguns dos sensores suportados pelo Android.

SENSOR	TIPO	DESCRIÇÃO
TYPE_ACCELEROMETER	H	Mede a aceleração em metros por segundo ao quadrado (m/s^2) aplicada ao dispositivo nos três eixos, incluindo a força da gravidade. Uso: detetar movimento (vibração/inclinação).
TYPE_AMBIENT_TEMPERATURE	H	Mede a temperatura ambiente em graus <i>Celsius</i> ($^{\circ}C$). Uso: monitorizar a temperatura do ar.
TYPE_GRAVITY	H/S	Mede a força da gravidade em m/s^2 aplicada ao dispositivo nos três eixos. Uso: detetar movimento (vibração/inclinação).
TYPE_GYROSCOPE	H	Mede a taxa de rotação do dispositivo em radianos por segundo (rad/s) nos três eixos. Uso: detetar rotação.
TYPE_LIGHT	H	Mede a intensidade da iluminação em Lux (lx). Uso: controlar o brilho do ecrã.
TYPE_MAGNETIC_FIELD	H	Mede o campo geomagnético para cada um dos três eixos (x, y, z) em microtesla (μT). Uso: criar um compasso.
TYPE_PRESSURE	H	Mede a pressão do ar em hectopascal (hPa) ou milibares (mbar). Uso: monitorizar alterações da pressão do ar.
TYPE_PROXIMITY	H	Mede a proximidade de um objeto relativamente ao ecrã de um dispositivo em centímetros (cm). Uso: determinar se um dispositivo está perto do ouvido de uma pessoa.

TABELA 2.8 – Tipos de sensores suportados pela plataforma Android¹⁰

¹⁰ Link: http://developer.android.com/guide/topics/sensors/sensors_overview.html

Para aceder a estes sensores usa-se a *framework* de sensores do Android, incluída no pacote `android.hardware`. Começa-se por criar uma atividade que implementa a interface `SensorEventListener`. Esta interface inclui dois métodos obrigatórios: `onSensorChanged` e `onAccuracyChanged`. O primeiro é chamado sempre que há uma alteração dos valores associados ao sensor registado. O segundo é chamado quando a precisão do sensor é alterada. A classe principal da *framework* de sensores é a classe `SensorManager` que permite aceder aos sensores do dispositivo. Para obter uma instância da classe, invoca-se o método `Context.getSystemService` com o argumento `Context.SENSOR_SERVICE`. Depois, obtém-se a referência a um determinado sensor através do método `getDefaultSensor(Sensor)`.

Certifique-se, antes de usar um sensor, de que este existe e está disponível. De seguida, regista-se o sensor usando o método `registerListener`, que aceita três argumentos: o contexto da atividade, o sensor e a taxa de entrega dos eventos dos sensores. É também uma boa prática cancelar o registo do sensor quando a aplicação hiberna (`onPause`) e registar o sensor novamente quando a aplicação é retomada (`onResume`). Certifique-se sempre de que desativa os sensores de que não precisa, principalmente quando a atividade é interrompida. Não fazer isso incorre em grandes débitos da bateria, já que o sistema não desativa automaticamente os sensores quando o ecrã se desliga:

```
public class MainActivity extends Activity implements SensorEventListener
{
    private SensorManager sManager;
    private Sensor sAclm;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        sManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
        sAclm = sManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
        if (sAclm != null){
            // TENHO UM ACELERÓMETRO!
            sManager.registerListener(this,
                sAclm, SensorManager.SENSOR_DELAY_NORMAL)
        }
    }
    public void onSensorChanged(SensorEvent sensorEvent) { }
    public void onAccuracyChanged(Sensor sensor, int accuracy) { }
    protected void onPause() {
        super.onPause();
        sManager.unregisterListener(this);
    }
    protected void onResume() {
        super.onResume();
        sManager.registerListener(this, sAclm,
            SensorManager.SENSOR_DELAY_NORMAL);
    }
}
```

De seguida, codifica-se o método `onSensorChanged` de forma a obter os valores do sensor e fazer algo com eles. O código padrão deste método é o seguinte:

```
public void onSensorChanged(SensorEvent sensorEvent) {  
    Sensor mySensor = sensorEvent.sensor;  
    if (mySensor.getType() == Sensor.TYPE_ACCELEROMETER) { }  
}
```

Usa-se o objeto `SensorEvent` passado como argumento para obter uma referência ao sensor e, usando o método `getType`, verifica-se o seu tipo, neste caso, se o tipo de sensor é um acelerómetro. O próximo passo é extrair a posição do dispositivo no espaço (os eixos X, Y e Z). Observe a Figura 2.17.

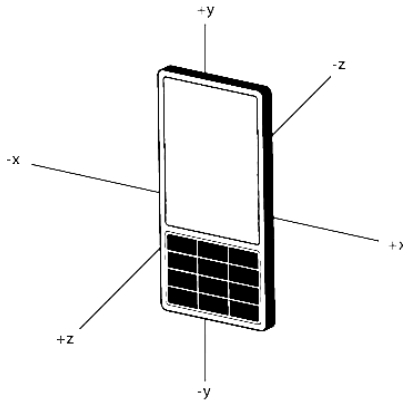


FIGURA 2.17 – Os eixos a ler para o sensor acelerómetro

O eixo X define o movimento lateral, enquanto o eixo Y define o movimento vertical. O eixo Z define o movimento para dentro e para fora do plano definido pelos eixos X e Y. Para obter os valores de cada eixo, usa-se o objeto `SensorEvent`, que devolve as posições como um *array* de valores do tipo `float`:

```
float x = sensorEvent.values[0];  
float y = sensorEvent.values[1];  
float z = sensorEvent.values[2];
```

A partir daqui e na posse destes valores, pode-se fazer muita coisa, desde alterar a posição de um objeto gráfico de acordo com os valores obtidos até ao desencadear de uma ação predefinida mediante um determinado movimento.

De seguida, apresenta-se uma aplicação que mostra a aceleração de um dispositivo físico em todos os seus eixos. A aceleração influenciará os números de cada eixo que se alteram sempre que o dispositivo é inclinado para qualquer direção:

```
public class MyActivity extends Activity implements SensorEventListener {  
    private SensorManager sensorManager;  
    private Sensor accelerometer;  
    public Vibrator vib;  
    private TextView valorX, valorY, valorZ;
```

```

private float deltaX = 0, deltaY = 0, deltaZ = 0;
private float lastX = 0, lastY = 0, lastZ = 0;
private float vibThreshold = 0;
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    valorX = (TextView) findViewById(R.id.textviewX);
    valorY = (TextView) findViewById(R.id.textviewY);
    valorZ = (TextView) findViewById(R.id.textviewZ);
    sm = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
    if (sm.getDefaultSensor(Sensor.TYPE_ACCELEROMETER) != null) {
        // SUCESSO, TEMOS UM ACELERÓMETRO!
        acc = sm.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
        sm.registerListener(this, acc,
            SensorManager.SENSOR_DELAY_NORMAL);
        vibThreshold = acc.getMaximumRange() / 2;
    } else { // NÃO TEMOS UM ACELERÓMETRO! }

    // INICIALIZA O VIBRADOR
    vib = (Vibrator) this.getSystemService(Context.VIBRATOR_SERVICE);
}

@Override
public void onSensorChanged(SensorEvent event) {
    // MOSTRA OS VALORES NAS TEXTVIEW
    valorX.setText(Float.toString(deltaX));
    valorY.setText(Float.toString(deltaY));
    valorZ.setText(Float.toString(deltaZ));

    // OBTÉM A MUDANÇA DOS VALORES X, Y, Z DO ACELERÓMETRO
    deltaX = Math.abs(lastX - event.values[0]);
    deltaY = Math.abs(lastY - event.values[1]);
    deltaZ = Math.abs(lastZ - event.values[2]);
    // SE A DIFERENÇA FOR MENOR DO QUE 2, CONCLUI-SE QUE É RUÍDO E DESCARTA-SE
    if (deltaX < 2) deltaX = 0;
    if (deltaY < 2) deltaY = 0;
    if (deltaZ < 2) deltaZ = 0;

    // DEFINE OS ÚLTIMOS VALORES CONHECIDOS DE X, Y, Z
    lastX = event.values[0];
    lastY = event.values[1];
    lastZ = event.values[2];

    // O DISPOSITIVO VIBRA SE UM VALOR DO EIXO SUPERAR UM VALOR THRESHOLD PREVIAMENTE DEFINIDO
    if (deltaZ > vibThreshold || (deltaY > vibThreshold) || (deltaX >
        vibThreshold)) {
        vib.vibrate(50);
    }
}
@Override
public void onAccuracyChanged(Sensor sensor, int accuracy) { }
}

```

A Figura 2.18 mostra a aplicação **Sensores** em execução num dispositivo físico.

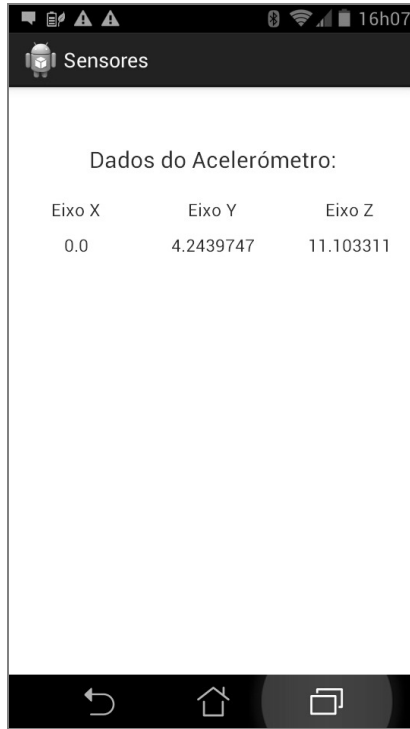


FIGURA 2.18 – A aplicação Sensores em execução num dispositivo Android

Para aplicações profissionais deve ter em conta a força da gravidade. Um sensor de aceleração mede a aceleração aplicada ao dispositivo, incluindo a força da gravidade. Para medir a aceleração real do aparelho, a força de gravidade tem de ser removida a partir dos dados do acelerómetro. Isto pode ser conseguido através da aplicação de um filtro *high-pass*. Por outro lado, um filtro *low-pass* pode ser utilizado para isolar a força da gravidade. O exemplo a seguir mostra como se faz isso:

```
public void onSensorChanged(SensorEvent event) {  
    // ALPHA É CALCULADO COMO T / (T + DT), ONDE T É O FILTRO LOW-PASS  
    // E DT A TAXA E ENTREGA DE EVENTOS  
    final float alpha = 0.8;  
  
    // ISOLA A FORÇA DE GRAVIDADE COM O FILTRO LOW-PASS  
    gravity[0] = alpha * gravity[0] + (1 - alpha) * event.values[0];  
    gravity[1] = alpha * gravity[1] + (1 - alpha) * event.values[1];  
    gravity[2] = alpha * gravity[2] + (1 - alpha) * event.values[2];  
  
    // REMOVE A GRAVIDADE COM O FILTRO HIGH-PASS  
    linear_acceleration[0] = event.values[0] - gravity[0];  
    linear_acceleration[1] = event.values[1] - gravity[1];  
    linear_acceleration[2] = event.values[2] - gravity[2];  
}
```



Pode obter uma listagem de todos os sensores num dispositivo através do método `getSensorList` e usando a constante `TYPE_ALL`. Por exemplo:

```
List <Sensor> deviceSensors = sm.getSensorList(Sensor.TYPE_ALL);
```

Se quiser a lista de todos os sensores de um determinado tipo, deve trocar a constante `TYPE_ALL` pelo tipo respetivo (por exemplo, `TYPE_GYROSCOPE`).

2.2.3 ÁUDIO

Um dos tipos de *media* mais importantes num jogo é o áudio. A plataforma Android oferece duas classes para a manipulação de áudio: a classe `SoundPool` e a classe `MediaPlayer`. A primeira é adequada para a execução de efeitos sonoros curtos, como, por exemplo, o som de uma explosão ou da colheita de uma moeda. A segunda classe deve ser usada para sons mais longos, como a música de fundo de um jogo.

2.2.3.1 EFEITOS SONOROS

A classe `SoundPool` é adequada para a reprodução de ficheiros áudio pequenos e que são executados com bastante frequência. É essencial que os ficheiros tenham um tamanho pequeno (menos do que 1 MB), pois o Android mantém o ficheiro na memória RAM do dispositivo durante a execução da aplicação. Por exemplo, num jogo de ação, o efeito sonoro de um soco seria um bom momento para utilizar esta classe, sabendo que se trata de um ficheiro curto e que será executado muitas vezes no jogo.

Para instanciar a classe `SoundPool` usa-se o seu construtor da seguinte forma:

```
new SoundPool(int maxStreams, int streamType, int srcQuality);
```



Este construtor foi preterido no Android 5.0. Para criar e configurar uma instância da classe `SoundPool` use a classe `SoundPool.Builder`.

O primeiro parâmetro define o número máximo de efeitos sonoros que podem tocar simultaneamente. Isso não significa que não se possa ter mais efeitos de som carregados, mas restringe quantos efeitos sonoros podem ser reproduzidos simultaneamente. O segundo parâmetro define o fluxo de áudio para onde o `SoundPool` irá fazer o *output* do áudio. O parâmetro final define a qualidade de execução do ficheiro e deve usar como padrão o valor 0. De seguida, é necessário carregar os efeitos sonoros através do método `load` da classe `SoundPool`:

```
load(Context context, int resId, int priority)
```

No primeiro parâmetro inclui-se o contexto onde o som irá ser reproduzido. No segundo parâmetro define-se o identificador do recurso áudio. No último parâmetro

inclui-se a prioridade do som que neste momento não tem qualquer influência. Use o valor 1 como valor padrão. O passo final é reproduzir o som através da invocação do método `play`. O método tem a seguinte sintaxe:

```
play (int soundID, float leftVolume, float rightVolume, int priority, int loop, float rate)
```

Os parâmetros deste método são os seguintes:

- ⊙ `soundID` – um identificador do som devolvido pelo método `load`;
- ⊙ `leftVolume` – volume da esquerda (variação entre 0.0 e 1.0);
- ⊙ `rightVolume` – volume da direita (variação entre 0.0 e 1.0);
- ⊙ `priority` – prioridade (0 = menor prioridade);
- ⊙ `loop` – modo de *loop* (por exemplo, 0 = sem loop, -1 = loop infinito, outro valor = número de repetições);
- ⊙ `rate` – taxa de reprodução (variação entre 0.5 e 2.0). Por exemplo, 1.0 = reprodução normal, 2.0 = reprodução duas vezes mais rápida do que a original.

Segue-se um exemplo do uso desta classe para reproduzir um ficheiro áudio:

```
int soundIds[] = new int [10];
SoundPool sp = new SoundPool(10, AudioManager.STREAM_MUSIC, 0);
// CARREGA O FICHEIRO SOM1.MP3 ARMAZENADO NA PASTA RES/RAW
soundIds[0] = sp.load(context, R.raw.som1, 1);
// ... PODE CARREGAR OUTROS SONS AQUI
// REPRODUÇÃO DO PRIMEIRO FICHEIRO
sp.play(soundIds[0], 1, 1, 1, 0, 1);
```

2.2.3.2 STREAMING

A classe `MediaPlayer` é responsável pela reprodução de ficheiros *media* numa aplicação Android, tais como vídeo e áudio de longa duração. A reprodução pode ser feita a partir de ficheiros armazenados como recursos do projeto (pasta **res/raw**), a partir de ficheiros no sistema de ficheiros do dispositivo ou através de fluxo de dados (*stream*) que chegam através de uma conexão de rede.



Antes de iniciar o desenvolvimento de uma aplicação usando a classe `MediaPlayer`, verifique se o manifesto tem as declarações necessárias para permitir o uso dos recursos. Por exemplo, se a classe `MediaPlayer` está a ser utilizada para transmitir conteúdo baseado na rede, a aplicação deve solicitar o acesso à rede incluindo a seguinte linha no ficheiro de manifesto do projeto Android:

```
<uses-permission android:name="android.permission.INTERNET" />
```

Um objeto desta classe permite obter, decodificar e reproduzir ficheiros de áudio e vídeo com configuração mínima. Os ficheiros podem ter várias fontes:

- ⊙ Recursos locais;
- ⊙ URI internos;
- ⊙ URL externos (*streaming*).

Segue-se um exemplo de como reproduzir um ficheiro áudio disponível como um recurso local:

```
MediaPlayer mediaPlayer = MediaPlayer.create(context, R.raw.som1);
mediaPlayer.start(); // O MÉTODO CREATE EXECUTA O MÉTODO PREPARE
```

Este segundo exemplo mostra como reproduzir um ficheiro a partir de um URI disponível localmente no sistema:

```
Uri myUri = ...; // INICIALIZAR O URI
MediaPlayer mediaPlayer = new MediaPlayer();
mediaPlayer.setAudioStreamType(AudioManager.STREAM_MUSIC);
mediaPlayer.setDataSource(getApplicationContext(), myUri);
mediaPlayer.prepare();
mediaPlayer.start();
```

O próximo exemplo mostra como reproduzir um som a partir de um URL remoto via *streaming* HTTP:

```
String url = "http://myserver.com/myMusics/music01.wav";
MediaPlayer mediaPlayer = new MediaPlayer();
mediaPlayer.setAudioStreamType(AudioManager.STREAM_MUSIC);
mediaPlayer.setDataSource(url);
mediaPlayer.prepare(); // ESTE PROCESSO PODE DEMORAR (POR EXEMPLO, BUFFERING)
mediaPlayer.start();
```

Em todos estes exemplos, o método `prepare` é usado de forma implícita ou explícita. Este método é responsável por obter e decodificar os recursos de áudio que se pretendem reproduzir. Esta tarefa, por vezes, pode ser demorada e, como tal, nunca deve ser executada na *thread* principal. Isto porque enquanto o método não for plenamente executado, a interface gráfica não reage à interação do utilizador, dando-lhe a impressão de que a aplicação é lenta e provocando assim uma má experiência de utilização. Para usar um objeto `MediaPlayer` numa *thread* diferente da *thread* principal, a *framework* inclui o método `prepareAsync`, que prepara os recursos áudio em *background* e invoca o método `onPrepared` da interface `MediaPlayer.OnPreparedListener` após terminar a preparação.

O controlo da reprodução de ficheiros de áudio/vídeo é gerido como uma máquina de estados. A Figura 2.19 mostra um diagrama com o ciclo de vida e os estados de um objeto `MediaPlayer` conduzido pelas operações de controlo de reprodução suportadas. As formas ovais representam os estados de um objeto `MediaPlayer`. Os arcos

representam as operações de controlo de reprodução que permitem a transição do estado do objeto. Existem dois tipos de arcos. Os arcos com uma única seta representam chamadas de métodos síncronos, enquanto os de seta dupla representam chamadas de métodos assíncronos.

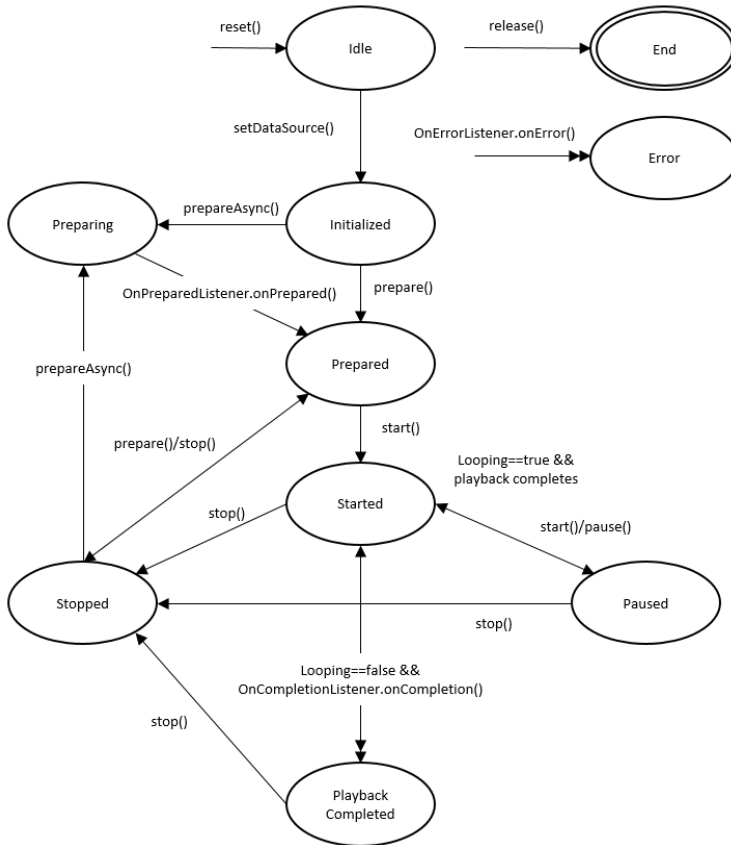


FIGURA 2.19 – Diagrama de estados de um objeto `MediaPlayer`

Após a instanciação de um objeto `MediaPlayer`, este encontra-se no estado *Idle*. Ao inicializá-lo através do método `setDataSource`, o objeto fica no estado *Initialized*. Depois disso, é necessário iniciar a preparação do objeto (obter e decodificar o recurso áudio) usando o método `prepare` ou `prepareAsync`. Após a preparação, o estado do objeto altera-se para *Prepared*. Finda a preparação, pode a qualquer momento reproduzir o som através do método `start`. Nesse ponto, como o diagrama da Figura 2.19 mostra, pode-se mover entre os estados *Started*, *Paused* e *PlaybackCompleted* através da invocação dos métodos `start`, `pause` e `seekTo`, entre outros. Outro ponto importante que pode ser constatado no diagrama é que, quando o método `stop` é invocado, o objeto passa ao estado *Stopped*, e para reproduzir de novo o som torna-se necessário prepará-lo antes de ser novamente chamado o método `start`.

Para libertar os recursos usados pelo objeto `MediaPlayer` deve usar o seguinte trecho de código:

```
mediaPlayer.release();
mediaPlayer = null;
```

Pode também usar a classe `MediaPlayer` como um serviço. Imagine que deseja que o som seja reproduzido mesmo quando o utilizador interage com outras aplicações; então, deve-se iniciar um serviço e controlar a instância da classe `MediaPlayer` a partir de lá:

```
public class MyService extends Service implements
    MediaPlayer.OnPreparedListener {
    private static final String ACTION_PLAY = "com.example.action.PLAY";
    MediaPlayer mMediaPlayer = null;
    public int onStartCommand(Intent intent, int flags, int startId) {
        ...
        if (intent.getAction().equals(ACTION_PLAY)) {
            mMediaPlayer = ... // INICIALIZAÇÃO DO MEDIAPLAYER
            mMediaPlayer.setOnPreparedListener(this);
            // PREPARAR ASYNC PARA NÃO BLOQUEAR A THREAD PRINCIPAL
            mMediaPlayer.prepareAsync();
        }
    }
    // CHAMADO QUANDO O MEDIAPLAYER ESTÁ PRONTO
    public void onPrepared(MediaPlayer player) {
        player.start();
    }
}
```

2.2.4 INPUT/OUTPUT

O Android suporta várias opções para guardar os dados da aplicação de forma persistente. A escolha depende das necessidades, por exemplo, se os dados devem ser privados ou públicos, ou qual o espaço de armazenamento requerido. As opções de armazenamento usam o sistema de ficheiros e são organizadas da seguinte forma:

- ⊗ **Shared Preferences** (ficheiros de preferências partilhadas) – dados primitivos em pares chave-valor;
- ⊗ **Sistema de ficheiros** – dados privados armazenados na memória interna do dispositivo ou dados públicos em suporte externo (por exemplo, cartão de memória);
- ⊗ **Base de dados** – dados estruturados em base de dados (**SQLite**).

Nesta secção destacamos a primeira abordagem – os ficheiros de preferências partilhadas. Abordam-se também o serviço de *backup* que o Android oferece e a nova *framework Storage Access Framework* (SAF), incluída no Android 4.4 e que vem abstrair e

simplificar o acesso dos utilizadores a qualquer tipo de ficheiro proveniente de vários fornecedores de documentos (*document providers*).



Para obter informações sobre as duas outras abordagens para a persistência dos dados consulte a obra *Desenvolvimento de Aplicações Profissionais em Android*, também da editora FCA.

2.2.4.1 PREFERÊNCIAS

A classe `SharedPreferences` fornece uma *framework* para guardar e recuperar pares chave-valor de tipos de dados primitivos (por exemplo, `int`, `float`, `long`, `boolean` e `string`). Estes dados poderão ser criados e recuperados em diferentes sessões do mesmo utilizador. As principais formas de obter um objeto `SharedPreferences` são:

- ⊙ `Context.getSharedPreferences(String name, int mode)` – usado quando temos vários ficheiros de preferências partilhados na mesma aplicação. O nome do ficheiro de preferências será dado pelo primeiro argumento do método;
- ⊙ `Activity.getPreference(int mode)` – usado quando queremos ter um ficheiro de preferências que seja privado e para uso exclusivo de uma atividade. O nome do ficheiro de preferências será o nome da atividade respetiva.

O próximo exemplo mostra como aceder a um ficheiro de preferências identificado pelo recurso `R.string.file_key` e abre-o usando o modo privado de modo que o ficheiro seja acessível apenas pela aplicação:

```
Context context = getActivity();
SharedPreferences sharedPref =
    context.getSharedPreferences(getString(R.string.file_key),
                               Context.MODE_PRIVATE);
```

Alternativamente, se precisar apenas de um ficheiro de preferências partilhado para a sua atividade, pode-se usar o método `getPreferences()`:

```
SharedPreferences sharedPref =
    getActivity().getPreferences(Context.MODE_PRIVATE);
```

Depois de obter o objeto, pode-se fazer uma de duas ações: escrever ou ler valores. Para escrever valores no ficheiro de preferências usa-se o método `edit`, que devolve uma nova instância da interface `SharedPreferences.Editor`. Com este novo objeto podem adicionar-se valores usando métodos como `putString` ou `putBoolean` (de acordo com o tipo de dados a guardar). Para concluir a edição dos dados e o respetivo armazenamento usa-se o método `commit`.

O próximo exemplo mostra como armazenar o *high score* de um jogador:

```
SharedPreferences sharedPref =
    getActivity().getPreferences(Context.MODE_PRIVATE);
SharedPreferences.Editor editor = sharedPref.edit();
editor.putInt(getString(R.string.high_score), newHighScore);
editor.commit();
```

A leitura de valores de um ficheiro de preferências é também bastante simples, sendo apenas necessário invocar os métodos de leitura, tais como `getString` ou `getBoolean`, fornecendo a chave para o valor desejado, e, opcionalmente, um valor por omissão para ser devolvido caso a chave não esteja presente, por exemplo:

```
SharedPreferences sharedPref =
    getActivity().getPreferences(Context.MODE_PRIVATE);
long highScore = sharedPref.getInt(getString(R.string.high_score), 0);
```

Para obter uma notificação quando um valor é atualizado num ficheiro de preferências regista-se um *listener*, que é despoletado quando o método `commit` é chamado. Um exemplo disso é quando um valor de um ficheiro de preferências é alterado a partir de uma atividade interferindo no comportamento de um serviço a correr em *background*. Outro exemplo é quando um valor é alterado violando as boas práticas de usabilidade. O próximo excerto demonstra como codificar este último exemplo:

```
public class NetworkService implements
    SharedPreferences.OnSharedPreferenceChangeListener {
    private static final String FONT_SIZE_KEY = "fontsize";
    private float mFont, recommendFontSize = 10;
    @Override
    public void onCreate() {
        SharedPreferences pref = PreferenceManager
            .getDefaultSharedPreferences(this);
        pref.registerOnSharedPreferenceChangeListener(this);
        mFont = preferences.getFloat(FONT_SIZE_KEY, recommendFontSize);
    }
    @Override
    public void onDestroy() {
        SharedPreferences pref = PreferenceManager
            .getDefaultSharedPreferences(this);
        pref.unregisterOnSharedPreferenceChangeListener(this);
    }
    @Override
    public void onSharedPreferenceChanged(SharedPreferences pref,
        String key) {
        if (FONT_SIZE_KEY.equals(key)) {
            mFont = pref.getFloat(FONT_SIZE_KEY, recommendFontSize);
            if (mFont < recommendFontSize) {
                // ENVIAR UMA NOTIFICAÇÃO A INFORMAR QUE O TAMANHO DA FONTE NÃO É ACONSELHÁVEL
            }
        }
    }
}
```



O objeto `SharedPreferences` não deve ser usado diretamente para armazenar as preferências do utilizador (por exemplo, o tipo de toque). Para criar as preferências do utilizador numa aplicação deve-se usar a classe `PreferenceActivity`, que fornece uma *framework* de persistência de preferências que usa o objeto `SharedPreferences`.

As aplicações geralmente incluem configurações que permitem aos utilizadores modificarem funcionalidades e comportamentos das aplicações. Por exemplo, algumas aplicações permitem aos utilizadores especificar quantas vezes a aplicação sincroniza dados com a *cloud*.

No caso de se desejar ter uma opção de configurações da aplicação deve-se usar a API `Preference`, de forma a construir uma interface que seja consistente com a experiência do utilizador com outras aplicações Android (Figura 2.20), incluindo as configurações do sistema.

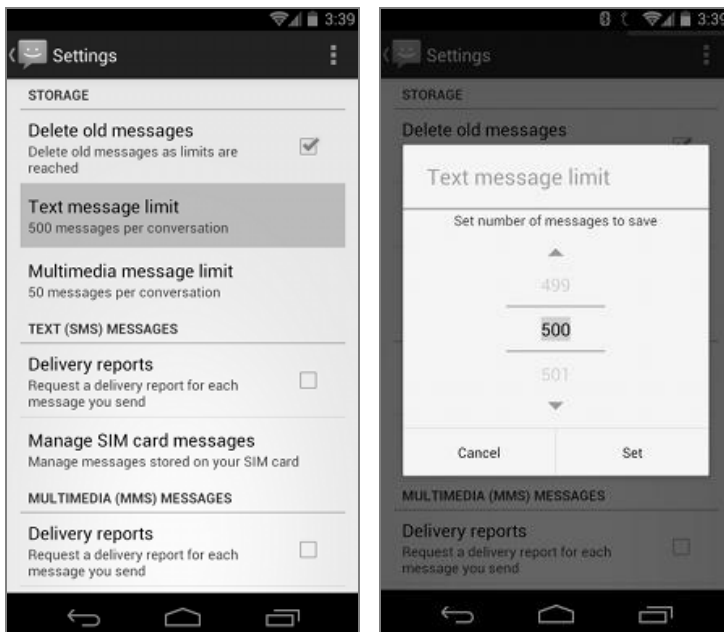


FIGURA 2.20 – Exemplo do ecrã de configurações de uma aplicação que usa a API `Preference`

Em vez de se usar um objeto `View` para construir a interface com o utilizador, as configurações são construídas utilizando várias subclasses da classe `Preference` que se declara num ficheiro XML.

Um objeto `Preference` é o alicerce para uma única configuração. Cada preferência aparece como um item numa lista e fornecendo uma GUI que permite a sua manipulação. Por exemplo, um `CheckBoxPreference` cria um item de lista que mostra uma caixa de seleção e um `ListPreference` cria um item que abre uma caixa de diálogo com uma lista de opções.

Cada objeto `Preference` que se adicionar tem um par chave-valor associado que é armazenado num ficheiro de preferência, via `SharedPreferences`. Quando o utilizador altera uma configuração, o sistema atualiza o valor correspondente no ficheiro de preferências. A única vez que se terá de interagir diretamente com o objeto `SharedPreferences` associado é quando for necessário ler o valor, a fim de determinar o comportamento da aplicação com base na configuração do utilizador.

Como as configurações da interface do utilizador na aplicação são construídas usando objetos `Preference` em detrimento de objetos `View`, é necessário especializar uma atividade ou fragmento para exibir a lista de configurações.

A classe `PreferencesActivity` é a classe base para uma atividade mostrar uma hierarquia de preferências para o utilizador. Antes do Android 3.0, esta classe só permitia a exibição de um único conjunto de preferências (essa funcionalidade deve agora ser encontrada na nova classe `PreferenceFragment`).

Atualmente, a classe `PreferencesActivity` mostra um ou mais cabeçalhos de preferência, cada um dos quais está associado a um `PreferenceFragment` para exibir as preferências desse cabeçalho. O *layout* e a visualização dessas associações podem variar:

- ⊗ Num ecrã pequeno exibe os cabeçalhos numa única lista. Após a seleção de um dos itens de cabeçalho irá relançar a atividade mostrando apenas o `PreferenceFragment` desse cabeçalho;
- ⊗ Num ecrã grande exibe simultaneamente os cabeçalhos e os `PreferenceFragment` como painéis. A seleção de um item de cabeçalho mostra o `PreferenceFragment` correspondente.

Para preencher a lista de cabeçalhos com os itens desejados as subclasses de `PreferenceActivity` devem implementar o método `onBuildHeaders`.

Embora se possam instanciar novos objetos `Preference` em tempo de execução, deve-se definir a lista de configurações em XML como uma hierarquia de objetos `Preference`, facilitando assim a sua manutenção. Além disso, as configurações da aplicação são geralmente pré-determinadas, embora se possa modificar a coleção em tempo de execução. Cada subclasse de `Preference` é declarada num elemento XML correspondente ao nome da classe, como, por exemplo, `<CheckBoxPreference>`.

O ficheiro de preferências deve ser armazenado na pasta **res/xml**. Tipicamente, usa-se o nome **preferences.xml**. O elemento raiz do ficheiro XML é o elemento `<PreferenceScreen>`. Dentro deste elemento adiciona-se cada `Preference`. Cada elemento filho adicionado aparecerá como um único item na lista de configurações.

Alguns dos mais comuns são as preferências:

- ⊙ `CheckBoxPreference` – mostra um item com uma caixa de seleção para ativação/desativação. É devolvido um valor lógico;
- ⊙ `ListPreference` – abre uma janela com uma lista de botões de rádio devolvendo um valor da enumeração do atributo `android:entryValues`;
- ⊙ `EditTextPreference` – abre uma caixa de diálogo com um objeto `EditText`. O valor devolvido é a *string* digitada.

Segue-se um exemplo de um ficheiro XML de preferências:

```
<PreferenceScreen>
  <CheckBoxPreference
    android:key="pref_sync"
    android:title="@string/pref_sync"
    android:summary="@string/pref_sync_summ"
    android:defaultValue="true"
  />
  <ListPreference
    android:dependency="pref_sync"
    android:key="exemple_list"
    android:title="@string/pref_syncConnectionType"
    android:entries="@array/pref_syncConnectionTypes_entries"
    android:entryValues="@array/pref_syncConnectionTypes_values"
    android:defaultValue="@string/pref_syncConnectionTypes_default"
  />
</PreferenceScreen>
```

Para implementar um `PreferenceActivity` inclui-se no método `onPostCreate`, entre outras chamadas, uma invocação ao método `addPreferencesFromResource`, que vai carregar o ficheiro XML de preferências criado anteriormente:

```
public class Opcoes extends PreferenceActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        // ADICIONA AS PREFERÊNCIAS A PARTIR DE UM RECURSO XML
        addPreferencesFromResource(R.xml.pref_general);

        // ASSOCIA OS SUMÁRIOS DAS PREFERÊNCIAS LIST PARA OS SEUS VALORES
        // QUANDO O VALOR SE ALTERA, O SUMÁRIO É ATUALIZADO
        bindPreferenceSummaryToValue(findPreference("exemple_list"));
    }

    @Override
    public void onBuildHeaders(List<Header> target) {
        loadHeadersFromResource(R.xml.pref_headers, target);
    }
}
```

```
//ESTE FRAGMENTO MOSTRA APENAS AS PREFERÊNCIAS. É USADO QUANDO A ATIVIDADE É MOSTRADA NUM
LAYOUT COM DOIS PAINÉIS
public static class GeneralPreferenceFragment extends PreferenceFragment
{
    @Override
    public void onCreate(Bundle savedInstanceState) {
        addPreferencesFromResource(R.xml.pref_general);
    }
}
```

Por questões de legibilidade, foram retirados, no excerto de código anterior, métodos de verificação do tamanho do ecrã e da atualização dos sumários.



No Android Studio é bastante simples trabalhar com ficheiros de preferências. Basta adicionar uma nova atividade do tipo **Settings** e todos estes ficheiros são criados automaticamente sendo apenas necessário configurar os itens que queremos que surjam no ecrã de configurações.

2.2.4.2 *BACKUP* DE DADOS

Uma prática comum e recomendada quando lidamos com dados é a possibilidade de fazer *backup* dos mesmos. O Android possui um serviço de *backup* que permite copiar os dados persistentes de uma aplicação para um armazenamento remoto na nuvem. Se um utilizador realiza uma reposição de fábrica ou se adquire um novo dispositivo Android, o sistema restaura automaticamente os dados de *backup* quando a aplicação for reinstalada. Desta forma, os utilizadores não necessitam de reproduzir os seus dados ou configurações da aplicação. Este processo é completamente transparente para o utilizador, não afetando a funcionalidade ou a experiência do utilizador com a sua aplicação.

Para fazer *backup* dos dados da aplicação é necessário implementar um agente de *backup*. O agente de *backup* é chamado pelo *Backup Manager* do Android para fornecer os dados para cópia. O mesmo é também chamado para restaurar os dados de *backup* quando a aplicação é reinstalada. Para implementar um agente de *backup* é necessário:

- 1) Declarar o agente de *backup* no ficheiro de manifesto.
- 2) Registrar a aplicação no serviço *Android Backup Service* (Figura 2.21).
- 3) Definir um agente de *backup* estendendo a classe `BackupAgent` ou `BackupAgentHelper`.

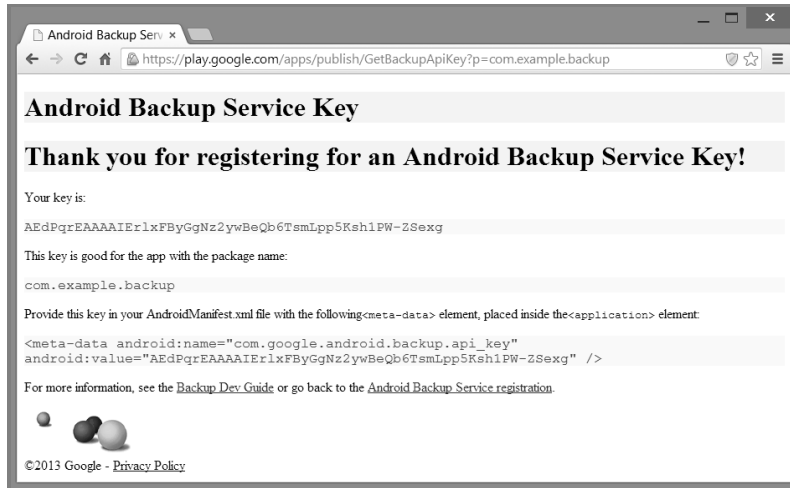


FIGURA 2.21 – *Android Backup Service*



O serviço de *backup* e as API a usar para fazer *backup* dos dados de uma aplicação estão disponíveis apenas para dispositivos com suporte para a API nível 8 (Android 2.2 ou superior).

Comece por criar uma nova aplicação através da criação de um novo projeto Android no Eclipse chamado **Backup**. De seguida, declare o agente de *backup* no ficheiro de manifesto através do atributo `android:backupAgent` no elemento `<application>`:

```
<application ... android:backupAgent="MyAgent">...</application/>
```

Posteriormente, registe a aplicação num serviço de *backup*. A Google oferece um serviço de *backup* chamado *Android Backup Service* (ABS) para as aplicações Android, que obriga a um registo prévio¹¹. Para efetivar o registo deve aceitar as condições de uso e identificar o pacote da aplicação (`com.example.backup`).

Após o registo, obtém-se uma chave e a descrição de um elemento `<metadata>` que inclui a chave gerada e que deve ser incluído no ficheiro de manifesto da aplicação. Se tiver várias aplicações, deve registar cada uma através do nome do pacote respetivo. Inclua o próximo excerto no ficheiro de manifesto da aplicação:

```
<meta-data android:name="com.google.android.backup.api_key"
  android:value="AEdPqrEAAAAIEr1xFBYGgNz2ywBeQb6TsmLpp5Ksh1PW-ZSexg" />
```

Por fim, deve implementar um agente de *backup*. A implementação pode ser feita a dois níveis: estendendo a classe `BackupAgent` ou a classe `BackupAgentHelper`.

¹¹ Para se registar no serviço ABS aceda a: <https://developer.android.com/google/backup/signup.html>

A primeira opção usa a classe `BackupAgent`. A classe disponibiliza uma interface através da qual a aplicação comunica com o *Backup Manager*. A extensão obriga à implementação dos métodos `onBackup` e `onRestore`. Esta abordagem é recomendada quando se quer fazer o *backup* de uma base de dados **SQLite**, fazer o *backup* parcial de ficheiros ou mesmo controlar versões dos ficheiros de dados.

A segunda opção usa a classe `BackupAgentHelper`. A classe disponibiliza um *wrapper* da classe `BackupAgent` que minimiza a quantidade de código necessária para implementar o agente. Na instância de um `BackupAgentHelper` deve-se usar um ou mais objetos *helper*, que automaticamente fazem o *backup/restore* dos dados sem necessidade de implementar os métodos `onBackup` e `onRestore`. O Android suporta *helpers* para fazer *backup* de ficheiros completos oriundos quer de objetos `SharedPreferences`, quer do armazenamento interno.

Neste exemplo vamos fazer o *backup* do ficheiro de preferências gerado por omissão através do método `PreferenceManager.getDefaultPreferences`. O nome do ficheiro é **<package-name>_preferences**:

```
public class MyAgent extends BackupAgentHelper {
    public static final String PREFS_BACKUP_KEY = "prefs";
    @Override
    public void onCreate() {
        super.onCreate();
        SharedPreferencesBackupHelper sharedPreferencesBackupHelper
            = new SharedPreferencesBackupHelper(this,
                getPackageName() + "_preferences");
        addHelper(PREFS_BACKUP_KEY, sharedPreferencesBackupHelper);
    }
}
```

A implementação da classe `BackupAgentHelper` deve usar um ou mais *helpers*. Um *helper* de *backup* é um componente especializado que executa ações de *backup/restore* para tipos de dados particulares. O Android disponibiliza dois:

- ⊙ `SharedPreferencesBackupHelper` – para operações em ficheiros de preferências;
- ⊙ `FileBackupHelper` – para operações em ficheiros de armazenamento interno.

Só necessita de um *helper* para cada tipo de dados. Para cada *helper* deve, no método `onCreate`, instanciar o *helper* indicando no construtor o ficheiro e invocar o método `addHelper` para adicionar o componente à implementação da classe `BackupAgentHelper`. Quando o `BackupManager` chamar os métodos `onBackup` ou `onRestore`, a implementação do `BackupAgentHelper` chama o *helper* definido para executar as operações de *backup/restore* do tipo de dados especificado. Para lidar com o processo de *backup/restore* de outros ficheiros deve usar a classe `FileBackupHelper`.

2.2.4.3 STORAGE ACCESS FRAMEWORK

Atualmente, a capacidade de armazenamento interno de um *smartphone/tablet* em comparação com um PC é ainda bastante inferior. Sendo assim, a necessidade do armazenamento remoto de ficheiros é um requisito fundamental para muitas aplicações. Perante isto, a Google introduziu a *Storage Access Framework* (SAF) no Android SDK 4.4.

A SAF simplifica o acesso dos utilizadores a qualquer tipo de ficheiro proveniente de vários fornecedores de documentos (*document providers*). Através de uma GUI (Figura 2.22), os utilizadores acedem aos documentos de uma forma consistente.

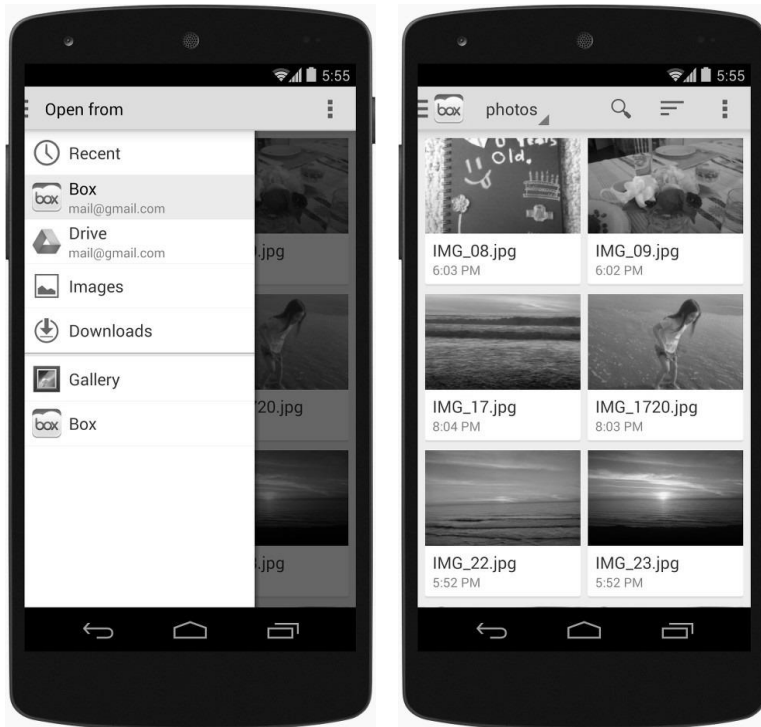


FIGURA 2.22 – Componente *picker* da SAF

Os fornecedores de documentos implementam a classe `DocumentsProvider` que encapsula os seus serviços. Do lado do cliente, bastam poucas linhas de código para aceder aos documentos usando a SAF. Esta é constituída por:

- **Fornecedor de documentos** – permite que um serviço de armazenamento (como o Google Drive) revele os ficheiros que gere. É implementado como uma subclasse da classe `DocumentsProvider`;
- **Aplicação cliente** – é uma aplicação personalizada que invoca as *intents* `ACTION_OPEN_DOCUMENT` e/ou `ACTION_CREATE_DOCUMENT` e recebe os ficheiros enviados pelos fornecedores de documentos;

- ⊗ *Picker* – permite que os utilizadores acedam aos documentos de todos os fornecedores que satisfaçam os critérios de pesquisa da aplicação cliente.

O fluxo da informação é representado pelo diagrama da Figura 2.23. Na SAF, os fornecedores e os clientes não interagem diretamente. Um cliente pede permissão para interagir com ficheiros (isto é, ler, editar, criar ou remover ficheiros).

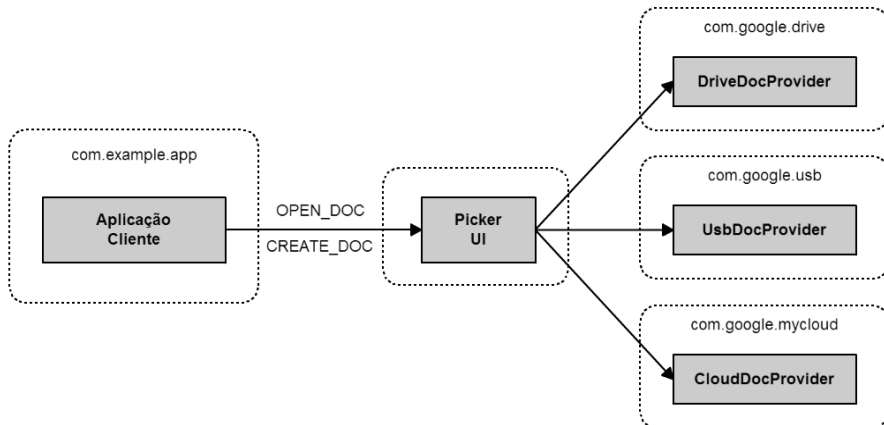


FIGURA 2.23 – Arquitetura SAF

A interação começa quando a aplicação cliente invoca as *intents* `ACTION_OPEN_DOCUMENT` ou `ACTION_CREATE_DOCUMENT`. A *intent* pode incluir filtros para refinar ainda mais a pesquisa (por exemplo, “dá-me todos os ficheiros que podem ser abertos e que têm o tipo MIME image”). De seguida, o *picker* (um componente gráfico controlado pelo sistema) vai a cada fornecedor de documentos registado (por exemplo, Google Drive, USB) e mostra ao utilizador os conteúdos correspondentes numa interface única.

Para trabalhar com ficheiros na SAF deve invocar um conjunto de *intents*, dependendo da ação específica a ser executada. Independentemente da ação, a *framework* exibirá um *picker* de modo a que o utilizador possa especificar o local de armazenamento (como uma pasta no Google Drive e o nome de um ficheiro). Quando o trabalho da *intent* é concluído, a aplicação é notificada por uma chamada ao método `onActivityResult`.

Para exemplificar o uso da SAF vamos criar uma aplicação que permite a criação e o armazenamento de ficheiros de texto. Execute os seguintes passos:

- 1) Crie um projeto Android denominado **SAF**.
- 2) No campo **Target SDK** selecione a opção **API nível 19**.
- 3) No campo **Navigation Type** selecione a opção **Dropdown**.
- 4) Altere o *layout* de acordo com a Figura 2.24.

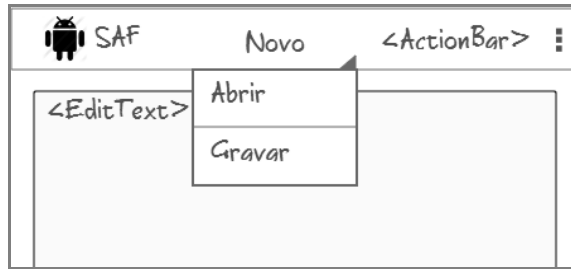


FIGURA 2.24 – Interface gráfica da aplicação SAF

- 5) Adicione código para criar, abrir e guardar ficheiros através da definição das *intents* `ACTION_CREATE_DOCUMENT` e `ACTION_OPEN_DOCUMENT` configuradas para criar/abrir ficheiros de texto simples. Estas ações permitem que o sistema apresente apenas os fornecedores de documentos que possibilitam a criação/abertura de ficheiros. A *intent* de criação de ficheiro inclui também um nome para o ficheiro que pode ser alterado no *picker* e adiciona a categoria `CATEGORY_OPENABLE`, que filtra os resultados para exibir apenas ficheiros que podem ser abertos. O resultado será recebido no método `onActivityResult` e diferenciado através da variável `requestCode`:

```
public void newFile(View view) {
    Intent intent = new Intent(Intent.ACTION_CREATE_DOCUMENT);
    intent.addCategory(Intent.CATEGORY_OPENABLE);
    intent.setType("text/plain");
    intent.putExtra(Intent.EXTRA_TITLE, "meuFicheiro.txt");
    startActivityForResult(intent, CREATE_REQUEST_CODE);
}
public void openFile(View view) {
    Intent intent = new Intent(Intent.ACTION_OPEN_DOCUMENT);
    intent.addCategory(Intent.CATEGORY_OPENABLE);
    intent.setType("text/plain");
    startActivityForResult(intent, OPEN_REQUEST_CODE);
}
public void saveFile(View view) {
    Intent intent = new Intent(Intent.ACTION_OPEN_DOCUMENT);
    intent.addCategory(Intent.CATEGORY_OPENABLE);
    intent.setType("text/plain");
    startActivityForResult(intent, SAVE_REQUEST_CODE);
}
```

- 6) Implemente o método `onActivityResult`. Na criação do ficheiro, o código deste método limita-se a limpar a caixa de texto. Na ação de gravação do ficheiro é obtido o URI do ficheiro através do método `getData` da *intent*, e esse mesmo URI (para onde o texto deverá ser escrito) é passado como parâmetro no método `writeFileContent`, cuja missão é gravar o texto no ficheiro selecionado. Na ação de abertura é obtido o URI do ficheiro selecionado, e o URI é usado no método `readFileContent`, que obtém o

texto do ficheiro. Posteriormente, é atualizada a caixa de texto com o texto do ficheiro selecionado.

```
public void onActivityResult(int requestCode, int resultCode, Intent
    resultData) {
    Uri currentUri = null;
    if (resultCode == Activity.RESULT_OK) {
        if (requestCode == CREATE_REQUEST_CODE) {
            if (resultData != null) {
                textView.setText("");
            }
        } else if (requestCode == SAVE_REQUEST_CODE) {
            if (resultData != null) {
                currentUri = resultData.getData();
                writeFileContent(currentUri);
            }
        } else if (requestCode == OPEN_REQUEST_CODE) {
            if (resultData != null) {
                currentUri = resultData.getData();
                try {
                    String content = readFileContent(currentUri);
                    textView.setText(content);
                } catch (IOException e) { //TRATAMENTO DE ERROS }
            }
        }
    }
}
```

7) Defina os métodos auxiliares de abertura e escrita em ficheiro:

```
private void writeFileContent(Uri uri) {
    try{
        ParcelFileDescriptor pfd =
            this.getContentResolver().openFileDescriptor(uri, "w");
        FileOutputStream fos = new FileOutputStream(pfd.getFileDescriptor());
        String textContent = textView.getText().toString();
        fos.write(textContent.getBytes());
        fos.close();
        pfd.close();
    } catch (FileNotFoundException e) { e.printStackTrace(); }
    catch (IOException e) { e.printStackTrace(); }
}

private String readFileContent(Uri uri) throws IOException {
    InputStream is = getContentResolver().openInputStream(uri);
    BufferedReader rd = new BufferedReader(new InputStreamReader(is));
    StringBuilder stringBuilder = new StringBuilder();
    String currentline;
    while ((currentline = rd.readLine()) != null) {
        stringBuilder.append(currentline + "\n");
    }
    is.close();
    return stringBuilder.toString();
}
```


8) Atualize o método `onNavigationItemSelected`:

```
public boolean onNavigationItemSelected(int position, long id) {  
    // QUANDO O ITEM DO MENU É SELECIONADO, MOSTRA O CONTEÚDO NA VIEW  
    switch(position) {  
        case 0: newFile(textView); break;  
        case 1: openFile(textView); break;  
        case 2: saveFile(textView); break;  
    }  
    return true;  
}
```

- 9)** Execute a aplicação no AVD (Figura 2.25). Inicialmente, surge o componente *picker*, que permite ao utilizador seleccionar o fornecedor de documentos e o nome do ficheiro. Selecione **Downloads**, confirme o nome do ficheiro e clique no botão **Save**. De seguida, escreva texto na caixa de texto.

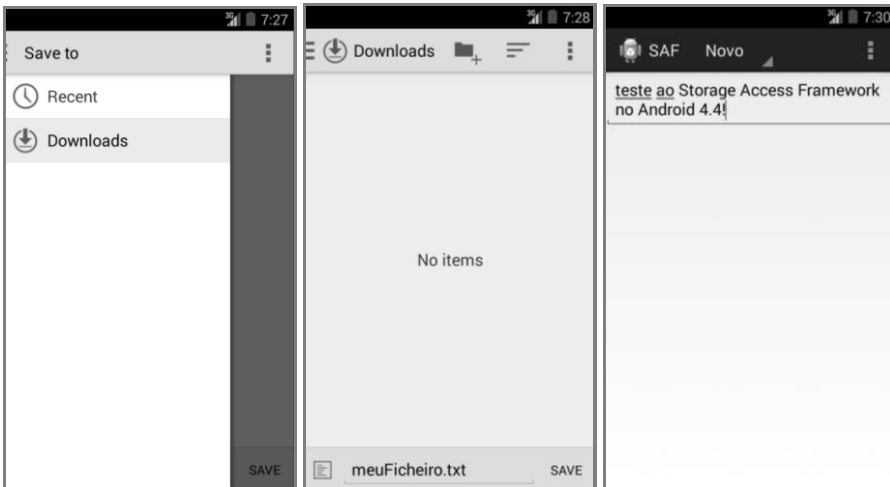


FIGURA 2.25 – A aplicação SAF (criação de ficheiro)

- 10)** Após a escrita, grave o ficheiro seleccionando a opção **Gravar** no menu flutuante da *Action Bar* e confirme no *picker* qual o ficheiro no qual pretende guardar o texto inserido (Figura 2.26).

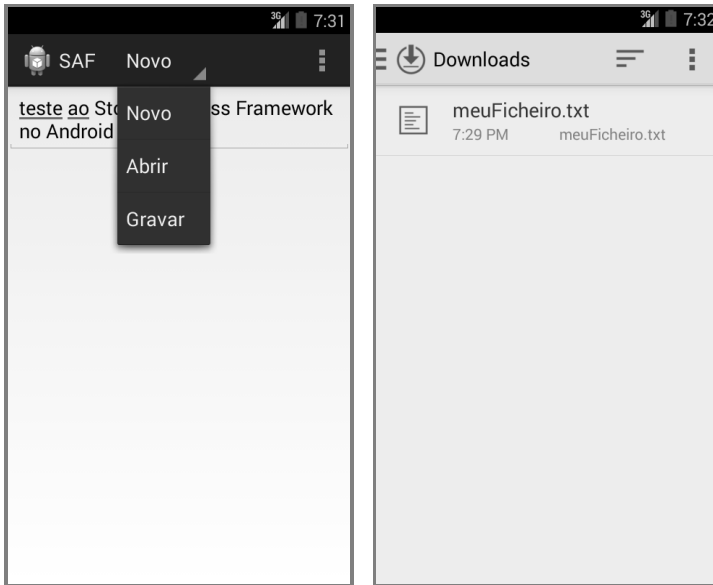


FIGURA 2.26 – A aplicação SAF (gravação de ficheiro)

- 11) Por fim, limpe o texto escrito e selecione a opção **Abrir** do menu. É exibido novamente o *picker*. Selecione o ficheiro anteriormente gravado. Se tudo correr bem, o texto inicial é apresentado na caixa de texto (Figura 2.27).

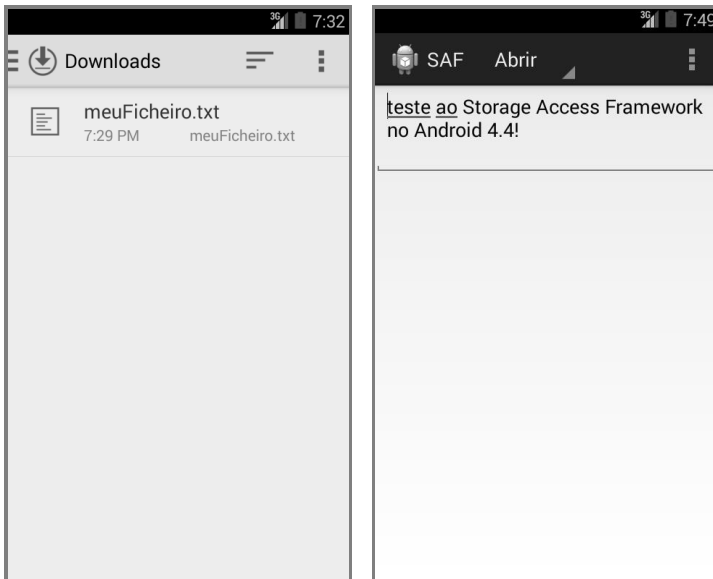


FIGURA 2.27 – A aplicação SAF (abertura de ficheiro)

Também é possível implementar um novo fornecedor de documentos necessitando, para tal, de criar uma classe que estende a classe `DocumentsProvider` e implementar quatro métodos, enumerados na Tabela 2.9.

MÉTODO	DESCRIÇÃO
<code>openDocument(String docId, String mode, Cancellation signal)</code>	Abre e devolve o documento pedido como um objeto <code>ParcelFileDescriptor</code> .
<code>queryChildDocuments(String parentDocId, String[] projection, String sortOrder)</code>	Devolve um objeto <code>Cursor</code> para os documentos filho contidos na pasta especificada.
<code>queryDocument(String docId, String[] projection)</code>	Devolve um objeto <code>Cursor</code> para os metadados associados ao documento pedido.
<code>queryRoots(String[] projection)</code>	Devolve um objeto <code>Cursor</code> para todas as raízes atualmente disponíveis.

TABELA 2.9 – Métodos principais da classe `DocumentsProvider`

Estes métodos permitem a *browsing*, a leitura e a escrita de dados locais/remotos que podem ser representados como ficheiros/documentos. Após a implementação dos métodos, é necessário registar o fornecedor de documentos no ficheiro de manifesto do projeto Android. Entre as principais ações, destacam-se a definição do *intent filter* `DOCUMENTS_PROVIDER` e o pedido de permissão `MANAGE_DOCUMENTS`:

```
<manifest>
  <application>
    ...
    <provider
      android:name="com.example.MyCloudProvider"
      android:authorities="com.example.mycloudprovider"
      android:exported="true"
      android:grantUriPermissions="true"
      android:permission="android.permission.MANAGE_DOCUMENTS">
      <intent-filter>
        <action
          android:name="android.content.action.DOCUMENTS_PROVIDER" />
        </intent-filter>
      </provider>
      ...
    </application>
</manifest>
```

2.3 MOTORES DE JOGO

O processo de construção de um jogo é complicado e multidisciplinar. Mesmo que se remova o processo criativo (definição da história e guião de jogo) e de *design* (criação e animação das personagens, desenho de paisagens, etc.) e se discuta apenas o processo de desenvolvimento, este não deixa, de todo, de ser multidisciplinar. Existem tarefas relacionadas com a representação gráfica dos objetos, com a inteligência artificial, com o som, com a simulação de leis de física, entre muitas outras características.

Os motores de jogos são pacotes que incluem bibliotecas no sentido de agilizar todos estes processos, tornando mais simples o desenvolvimento de jogos para equipas pequenas. Um motor de jogo pode ser mais ou menos completo, como se verá nos vários exemplos apresentados. Podem contemplar apenas bibliotecas gráficas, incluir também a gestão de interação ou a simulação física e, ainda, ser não só um conjunto de bibliotecas, mas também uma interface de desenvolvimento poderosa.

O conceito de motor de jogo surgiu há algum tempo, quando empresas de jogos tentaram capitalizar todo o investimento no desenvolvimento de um jogo complicado. Exemplo disso foi o desenvolvimento da família de jogos *Quake* (da Id Software), em que cada um deles deu origem a um motor de jogo específico para esse tipo de jogos (*first person shooter*), os quais foram vendidos a outras empresas para o desenvolvimento de jogos semelhantes (por exemplo, o *Half Life* da Valve Corporation, que usou o motor da primeira versão do *Quake*, o *Return to Castle Wolfenstein* da Gray Matter/Nerve Software ou o *Call of Duty* da Infinity Ward, que foram construídos através do motor idTech 3 do *Quake 3*).

2.3.1 COMPONENTES

Como já se referiu, existem diferentes componentes ou módulos nos motores de jogos, embora nem todos os motores de jogos incluam todos os componentes. Na verdade, isto leva à discussão do que poderá, ou não, ser considerado um motor de jogo, já que alguns incluem apenas um componente e, portanto, poderiam ser considerados apenas como uma biblioteca. Os componentes podem ser:

- ⊗ **Input** – nível de abstração sobre a gestão de entrada de dados, seja através do teclado, de um rato, *joystick*, comando ou de um ecrã com suporte ao toque, como os existentes em dispositivos móveis. Este componente poderá ser capaz de tornar a gestão do uso de um teclado, *joystick* ou rato de forma uniforme, sem que o programador tenha de escrever código distinto para cada um deles;
- ⊗ **Gráficos** – talvez seja o componente mais comum. É responsável pelo desenho de imagens ou modelos tridimensionais, o seu posicionamento no

ecrã (ou num mundo virtual), a animação de personagens, entre outros. Estes componentes também são capazes de carregar modelos desenhados em modeladores tridimensionais (como o Maya ou o Blender) de forma automática;

- ⊗ **Som** – os motores de som, menos comuns, são capazes de reproduzir som, tratando de todo o processo de *surround* e desvanecimento de som, de acordo com a posição do jogador em relação às fontes de som. Muitos destes sistemas também são capazes de aplicar filtros de som, bem como reproduzir áudio armazenado em diferentes formatos;
- ⊗ **Rede** – cada vez mais, os jogos são utilizados em rede. A gestão de rede, diretamente através de um protocolo da pilha TCP/IP, como sejam o TCP (*Transmission Control Protocol*) ou o UDP (*User Datagram Protocol*), é complicada e sujeita a diferentes tipos de restrições. Muitos motores de jogos incluem funcionalidades de alto nível para a gestão de redes e de jogadores, sem que o programador tenha de se preocupar com os detalhes subjacentes à implementação de baixo nível;
- ⊗ **Física** – a simulação física (como as simulações de gravidade ou de colisões entre objetos) é complicada, já que recorre a fórmulas físicas, o que obriga que o programador tenha alguma familiaridade com esta área do conhecimento. Este é um dos componentes mais importantes na preparação de um jogo que tente modelar a realidade. O programador apenas precisa de definir um conjunto de forças e de aplicá-las aos objetos sujeitos às leis físicas;
- ⊗ **Interface com o utilizador (GUI)** – não existem muitos motores de jogos que se dediquem à definição da GUI, nomeadamente na criação de botões, formulários, etc., uma vez que as bibliotecas nativas dos vários sistemas operativos já incluem esta funcionalidade. No entanto, existem outros que incluem mecanismos para o programador definir uma interface gráfica com um aspeto independente do disponibilizado pelo sistema operativo;
- ⊗ **Inteligência artificial** – os motores de jogos mais recentes incluem também funcionalidades relacionadas com a implementação de algoritmos de inteligência artificial, desde os habituais cálculos de caminhos (nomeadamente usando a técnica denominada *navmeshes*), bem como a deteção de movimento, algoritmos de movimento inteligente, ou ainda a modelação de comportamento usando diferentes técnicas (como máquinas de estado, árvores de decisão e árvores de comportamento);
- ⊗ **Scripting** – finalmente, o desenvolvimento de jogos passa muito pela criação de guiões em que se descreve o fluxo da história ou, pelo menos, de jogo.

Cada vez mais, os guiões de jogos complicados deixam de ser definidos diretamente no código do jogo e passam a ser descritos em linguagens de alto nível (designadas por linguagens de *scripting*) que possam ser utilizadas por programadores pouco experientes de forma rápida e cómoda. Talvez um dos exemplos mais carismáticos do uso de *scripting* seja o *World of Warcraft* (da Blizzard), que usa a linguagem Lua para a definição de comportamento dos vários elementos de jogo.

Existem alguns componentes que são distribuídos de forma independente. Por exemplo, o **BulletPhysics**¹² é uma biblioteca dedicada apenas à simulação de elementos físicos, e a **RakNet**¹³, que é uma biblioteca para a programação de jogos em rede.

2.3.2 EXEMPLOS

Esta secção pretende apresentar alguns motores de jogos preparados para o desenvolvimento de jogos em Android. Não se pretende que seja uma lista exaustiva (até porque todos os dias surgem novos projetos), e é claro que não poderão ser todos apresentados em detalhe. No entanto, foram preparadas duas secções com a documentação de criação de jogos em dois destes motores: o **Unity 3D** e o **libgdx**.

2.3.2.1 UNITY 3D

O Unity 3D¹⁴ é um dos motores de jogos mais conhecidos e mais completos. Não se baseia apenas num conjunto de bibliotecas, mas em todo um ambiente de desenvolvimento. Inclui um editor interativo de cenas de jogo, em que o utilizador pode colocar os objetos, sejam tridimensionais ou bidimensionais, adicionar comportamento independente a cada objeto usando código ou técnicas de animação (como, por exemplo, máquinas de estado), importar modelos em vários formatos, bem como som ou imagens. Para além de tudo isso, inclui um editor de código baseado na *framework Mono*¹⁵, permitindo assim a implementação quer em Microsoft C#, quer em Microsoft JScript (também conhecido por JavaScript). Ao fazer uso do *Mono*, permite exportar os jogos nas mais diversas arquiteturas, seja para computadores pessoais a executar Microsoft Windows, Mac OS X ou Linux, para arquiteturas móveis, como Android ou iOS, e ainda para uma variedade de consolas, como Xbox e PlayStation.

¹² <http://bulletphysics.org/wordpress/>

¹³ <http://www.jenkinssoftware.com/>

¹⁴ <http://unity3d.com/>

¹⁵ <http://www.mono-project.com/>

Existem duas versões: uma gratuita e uma outra, denominada Profissional, paga. Enquanto nas versões anteriores à 5.0.0 o Unity incluía funcionalidades avançadas, na versão profissional, a partir da versão 5.0.0 a diferença deixou de ser a nível de funcionalidades e passou a ser a nível de suporte. A Figura 2.28 apresenta o Editor do Unity 3D.

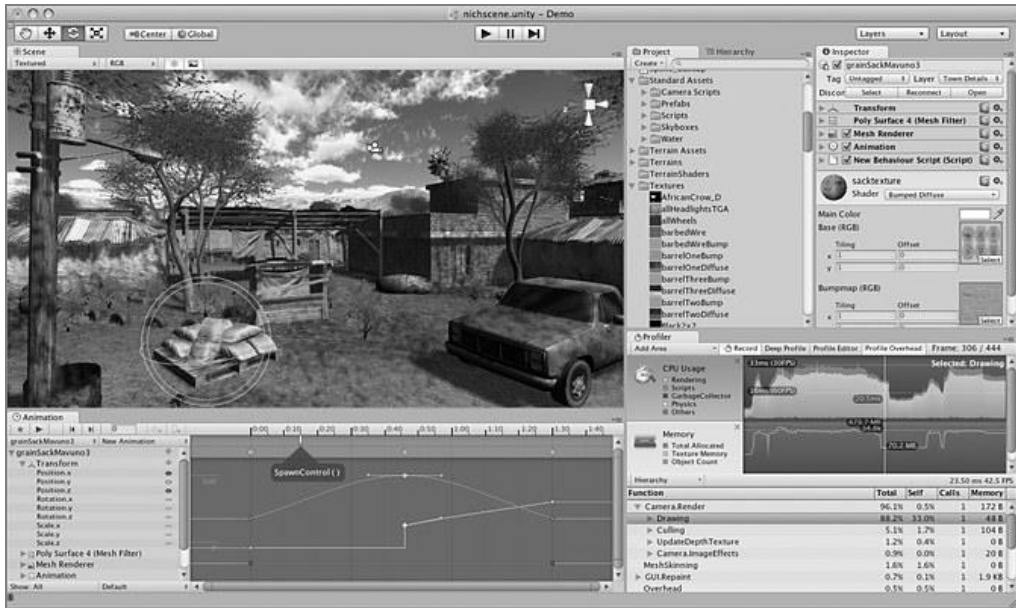


FIGURA 2.28 – Editor Unity 3D

Sendo uma ferramenta tão versátil, foi escolhida para integrar um capítulo deste livro (Capítulo 5), no qual será desenvolvido um jogo tridimensional para Android.

2.3.2.2 CRY ENGINE

Não há quem se interesse por jogos digitais que não tenha ficado espantado com a qualidade gráfica do jogo *Far Cry* da Ubisoft. Tal como era esperado, a empresa investiu na generalização do motor usado neste jogo e respetivas sequelas, surgindo o Cry Engine, um motor de jogos poderoso e apresentado como um produto.

O Cry Engine é multiplataforma, permitindo a geração de jogos para Xbox One, PlayStation 4, WiiU, Windows, Android e iOS.

Inclui um motor gráfico com diferentes tipos de renderização, reflexos locais, sistemas de partículas, nevoeiro volumétrico, entre várias outras funcionalidades, bem como um motor gráfico dedicado apenas ao desenho de personagens e à sua animação.

Tal como o Unity 3D, inclui um potente motor de física com diferentes algoritmos, como deformação de objetos através de forças, simulação de cordas ou ainda a simulação de tecido.

Também disponibiliza ferramentas para o desenvolvimento de inteligência artificial, interface com o utilizador, gerador de terrenos, editor dedicado para veículos, entre muitas outras (Figura 2.29).

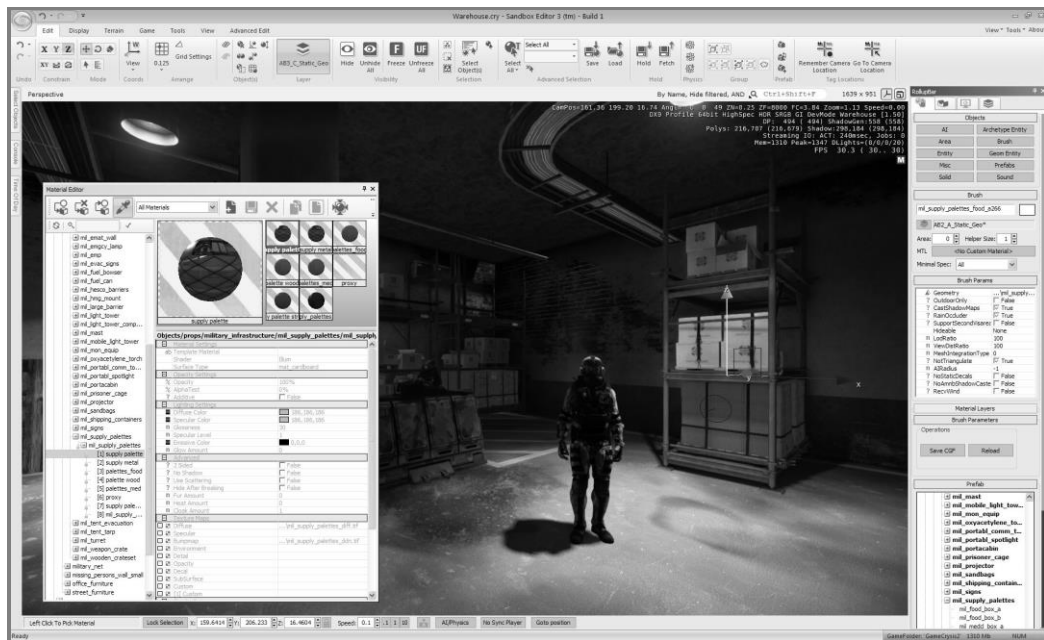


FIGURA 2.29 – Editor Cry Engine

Infelizmente, este motor não tem uma versão gratuita, existindo duas formas de o adquirir, seja por subscrição com pagamentos mensais ou através de uma licença completa, com acesso a código-fonte e suporte por parte da Crytech.

2.3.2.3 COCOS 2D

O motor Cocos2d-x¹⁶ é um motor de jogo escrito em C++ que funciona numa camada computacional dependente de plataforma. Isto permite-lhe ser independente de plataforma, suportando iOS, Android, Windows Phone, Mac OS X, Windows e Linux. Existem duas variantes, denominadas Cocos2d-x + Lua e Cocos2d-js, que permitem o desenvolvimento usando as linguagens Lua e JavaScript, respetivamente.

¹⁶ <http://www.cocos2d-x.org/products#cocos2dx>

É especialmente dedicado ao desenvolvimento de jogos 2D, com suporte OpenGL ES 2.0 e OpenGL ES 2.1, transições entre cenas, *sprites*, efeitos de imagem, ações compostas (baseadas em ações simples), sistemas de partículas, animações de esqueletos, suporte para mapas de azulejos (*tile map*), sejam ortogonais, isométricos ou hexagonais, câmara lenta (*slow motion*), etc.

Inclui também uma biblioteca para o desenho de interfaces, um motor de física 2D, suporte para som e desenvolvimento de jogos em rede. Existe, ainda, um editor gráfico denominado Cocos Studio (Figura 2.30), que permite agilizar o desenvolvimento das interfaces, e um IDE, denominado Cocos Code IDE, personalizado para a escrita de jogos usando este motor.

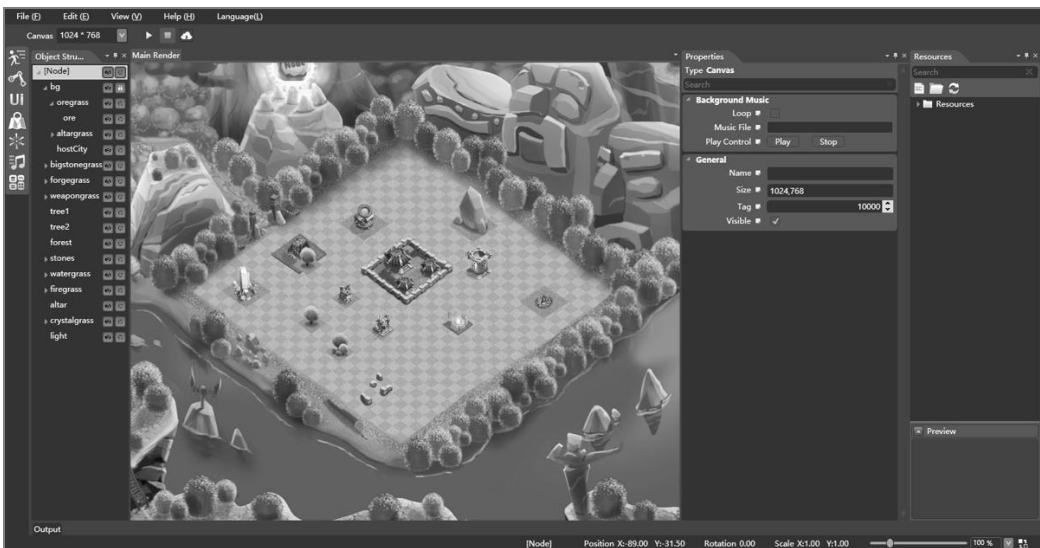


FIGURA 2.30 – Cocos Studio

2.3.2.4 CITRUS GAME ENGINE

O Citrus Game Engine¹⁷ é um motor gráfico para Flash, de código aberto e gratuito. Mais uma vez, funciona sobre uma camada dependente de plataforma que torna as aplicações desenvolvidas na Citrus capazes de serem executadas em Android, iOS, Linux, Mac OS X ou Windows (Figura 2.31).

¹⁷ <http://citrusengine.com/>

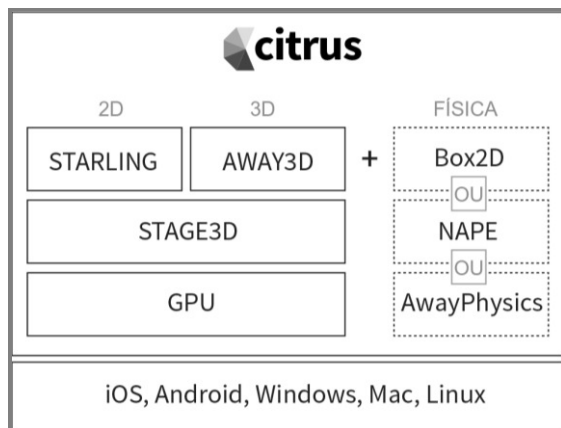


FIGURA 2.31 – Componentes citrus, por camadas

O seu componente gráfico é acelerado por *hardware*, via biblioteca **Stage3D**, nomeadamente quando existe disponível um processador gráfico (GPU). Sobre este, é possível desenvolver jogos em 2D usando a biblioteca **Starling** e jogos em 3D com a biblioteca **Away3D**. Para além do componente gráfico inclui uma biblioteca para a gestão de som e três bibliotecas para a simulação física: a **Box2D**, a **NAPE** e a **AwayPhysics**.

2.3.2.5 HAVOK VISION ENGINE

O Vision Engine¹⁸ da Havoc é um motor preparado para o desenvolvimento em C++, com suporte para Windows, Xbox 360, PlayStation 3, Nintendo Wii, WiiU, PlayStation Vita, iOS e Android. Embora o motor completo não seja gratuito, existem dois componentes disponíveis gratuitamente: o módulo de inteligência artificial e o *kit* de desenvolvimento de animações.

A versão completa inclui, para além dos dois módulos já mencionados, um motor gráfico, um motor para física, um simulador de tecido, uma ferramenta para a destruição de objetos e suporte para a linguagem de *scripting* Lua.

Sendo uma ferramenta profissional, existem vários jogos bem conhecidos que foram desenvolvidos com base em alguns destes componentes, como o *Assassin's Creed IV* para Xbox 360 (motor de física) ou o *Call of Duty 4* para PlayStation 4 (motor para *scripting*).

¹⁸ <http://www.havok.com/products/vision-engine>

Com base no Vision Engine da Havoc foi criado o projeto Anarchy¹⁹ (Figura 2.32), que disponibiliza acesso gratuito aos componentes gráfico, de física, de inteligência artificial e de animação para o desenvolvimento para Android, iOS e Tizen. Embora o uso destes componentes seja gratuito, a licença obriga a um conjunto de restrições, essencialmente no que toca ao *marketing* da própria Havoc.

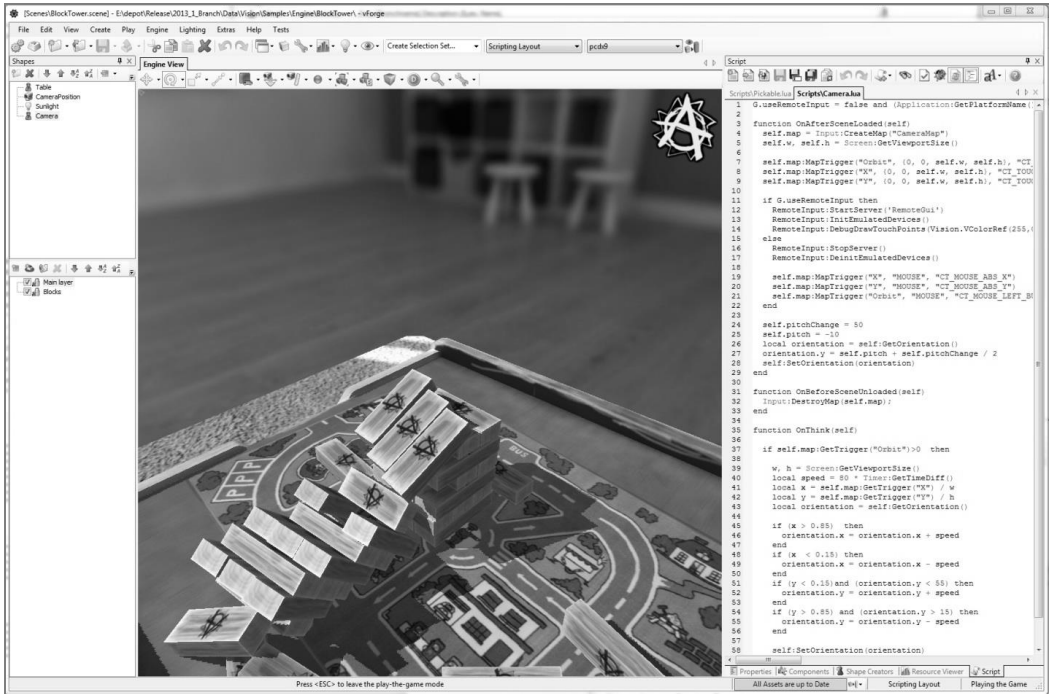


FIGURA 2.32 – Editor Havok Vision Engine

2.3.2.6 LIBGDx

O libgd²⁰, ao qual é dedicado o Capítulo 4 deste livro, é um motor de jogo que permite o desenvolvimento para Windows, Mac OS X, Linux, Android, iOS, BlackBerry ou HTML5, usando sempre o mesmo código. É desenvolvido com cuidados de eficiência, de forma gratuita e código aberto, e com suporte para 2D e 3D.

O desenvolvimento é realizado em Java; embora para isso seja sugerido o uso de um IDE comum, como o Eclipse ou o Android Studio, é disponibilizada uma ferramenta para criar um esqueleto base de aplicação com libgd (Figura 2.33).

¹⁹ <http://www.projectanarchy.com/>

²⁰ <http://libgd.badlogicgames.com/>

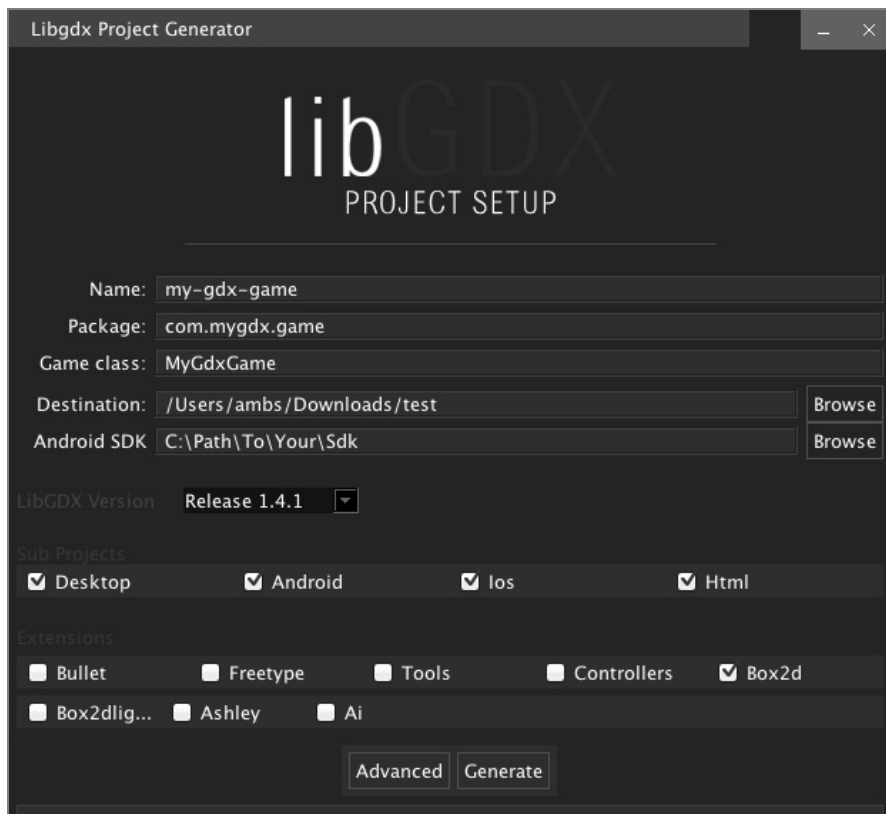


FIGURA 2.33 – Gerador de projetos libgdx

Embora o seu foco principal seja o componente gráfico, inclui suporte para áudio, gestão de *input*, bibliotecas matemáticas e físicas, bem como algumas outras de utilidade, como o acesso a ficheiros e a dados, a serialização eficiente de XML e JSON. Para além destes componentes inclui um editor de sistemas de partículas, um empacotador de texturas e um gerador de tipos de letra *raster*.

Também integra com o Spine, um animador 2D de esqueletos, o Nextpeer – uma biblioteca para o desenvolvimento de jogos em rede – e os otimizadores DexGuard e ProGuard da Saikoa.

2.3.2.7 MARMALADE

O Marmalade SDK²¹ permite o desenvolvimento em C++, Objective-C, Lua e HTML5 para Android, iOS, BlackBerry, Windows Mobile, Windows, Mac OS X e Tizen.

²¹ <https://www.madewithmarmalade.com/>

Não é considerado realmente um motor de jogos, mas uma biblioteca especialmente dedicada ao componente gráfico que tira partido, sempre que possível, do *hardware* e das bibliotecas existentes em cada sistema operativo. Outros motores de jogos, como o Cocos2d-x, integram com o Marmalade.

Existem alguns bons exemplos de jogos desenvolvidos em Marmalade, como o *Kingdom Clash* da Storm8, o *The Activision Decathlon* da Activision ou o *Plants vs Zombies* da Popcap.

2.3.2.8 GAME SALAD

O Game Salad²² é uma ferramenta para o desenvolvimento interativo de jogos, construindo-os por blocos e aplicando regras de comportamento a diferentes objetos (Figura 2.34).

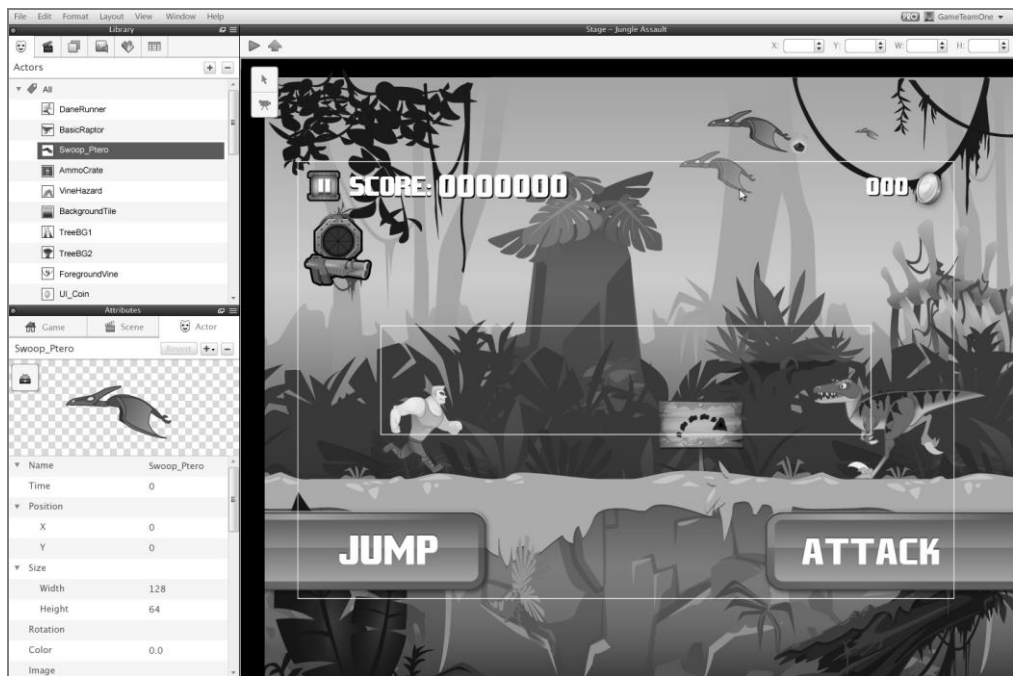


FIGURA 2.34 – Editor Game Salad

Inclui uma versão gratuita com funcionamento por *drag-and-drop*, uma biblioteca de comportamentos e um motor de física integrado. Permite a criação de jogos para Android, iOS e HTML.

²² <http://gamesalad.com/>

A versão profissional permite a integração com outros serviços, como a venda de funcionalidades (*in-app purchases*), o acesso a redes de publicidade e a redes sociais, possibilidade de desenvolvimento de jogos em rede, e inclui a exportação de jogos para Mac OS X, Windows 8 e Tizen.

2.3.2.9 SHIVA

O ShiVa²³ é um motor de jogos comercial que dispõe de uma versão gratuita, na Web, para jogos *online*, via Flash. Existe também uma versão comercial que permite a publicação de jogos em iOS, Android, BlackBerry, WindowsPhone, Windows, Mac OS X, Linux, Flash, Xbox 360, PlayStation 3 e Nintendo Wii. A versão paga inclui um editor interativo (Figura 2.35). O desenvolvimento é feito usando Lua ou C++.



FIGURA 2.35 – Editor ShiVa

Em termos de componentes, o ShiVa inclui um motor gráfico com suporte até 15 000 000 vértices por *frame*, importando modelos de várias principais aplicações de modelação 3D. Suporta diferentes sobreadores, suporte para luzes e sombras dinâmicas, sistemas de partículas e efeitos especiais, como sejam reflexo e refração nas texturas, efeito de distorção na água, espelhos, nevoeiro de calor e atenuação por nevoeiro.

²³ <http://www.shivaengine.com/>

Para a criação de animações apresenta um sistema de animação hierárquico com suporte para um número infinito de juntas. Também permite o desenvolvimento de jogos 2D, bem como um editor integrado de interface.

O componente de jogos em rede suporta até 16 jogadores e interface para serviços Web. Mediante licença adicional inclui um componente de jogos em rede com suporte até centenas de jogadores.

A simulação física tem gestão integrada de massa, fricção, movimento, diferentes tipos de materiais, forças lineares e angulares e vários tipos de juntas.

Finalmente, o motor de som suporta som 2D e 3D, incluindo suporte para 5.1 *surround* e posicionamento tridimensional de fontes de som.

2.3.2.10 GAMEMAKER STUDIO

O GameMaker Studio²⁴, da YoYo Games, é disponibilizado de forma gratuita apenas para pequenos jogos a serem executados sobre uma aplicação de nome **GameMaker Player**. Para a criação de jogos independentes é necessário adquirir uma licença profissional, em que os módulos são vendidos de forma independente (permitindo a quem os adquire configurar o seu ambiente de desenvolvimento de acordo com as suas necessidades).

Considerando a versão paga, o GameMaker permite a criação de jogos para várias plataformas, como Windows, Mac OS X, Linux, HTML5, Android, iOS, Windows Phone, Tizen, Xbox One, PlayStation Vista, 3 e 4.

O desenvolvimento é realizado numa aplicação específica, usando a linguagem GML para a escrita de código (Figura 2.36). Para além do componente gráfico inclui o motor de física Box2D, som via OpenAL e suporte para jogos em rede.

²⁴ <https://www.yoyogames.com/studio>

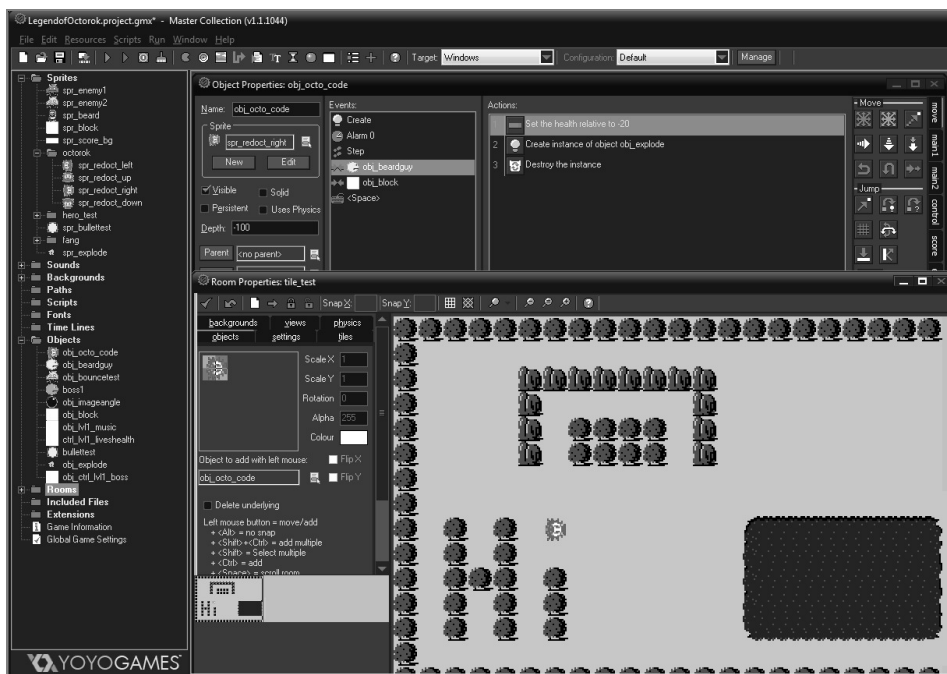


FIGURA 2.36 – GameMaker Studio

2.3.2.11 APP GAME KIT

O App Game Kit²⁵ permite o desenvolvimento de jogos numa linguagem inspirada em Basic sobre um IDE próprio ou, para programadores com mais experiência, diretamente em C++ (Figura 2.37). Em termos de portabilidade, suporta Windows, Max OS X, iOS, Android e BlackBerry. Embora seja um sistema pago, apresenta um preço acessível.

A versão atual inclui motores gráficos para 2D e 3D, com diferentes tipos de sombreadores, luzes, sistemas de partículas ou mesmo o uso de esqueletos para animação 2D. Embora, atualmente, já inclua o motor de física Box2D, ainda não tem suporte para física em 3D, estando planeado para a próxima versão, usar a conhecida biblioteca **BulletPhysics**.

O componente áudio é básico, suportando efeitos sonoros e música. O controlo de *input* é feito através de dois módulos, um que torna a programação para diferentes tipos de controladores transparente e outro para o controlo de sensores.

²⁵ <http://www.appgamekit.com/>

O suporte para comunicação em rede também é básico, suportando apenas o envio de mensagens. Inclui ainda um conjunto de utilitários, como a gestão de códigos QR ou o acesso ao Facebook.

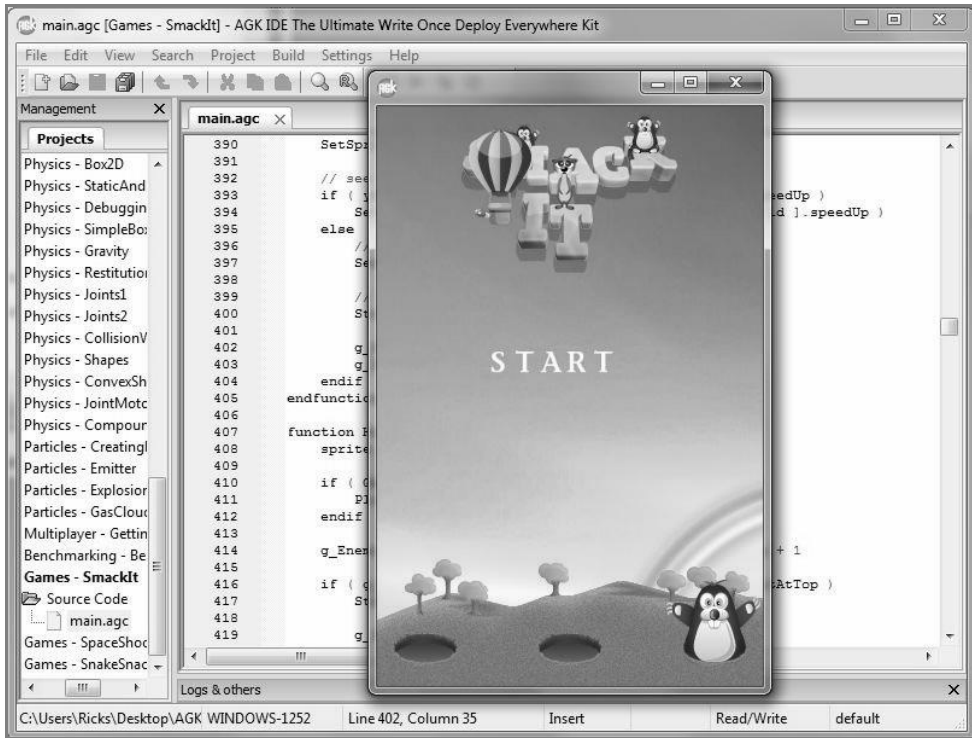


FIGURA 2.37 – IDE App Game Kit

2.3.2.12 REACH 3DX

O Reach 3DX²⁶ é um motor para o desenvolvimento de jogos com uma versão gratuita, sem integração no Visual Studio, e uma versão paga. No entanto, para poder vender comercialmente um jogo, o programador é obrigado a comprar a versão paga.

Mais uma vez, este motor é independente de plataforma, sendo capaz de criar executáveis para HTML5, iOS, Android, Flash e Tizen. Para ajudar no desenvolvimento disponibiliza um editor de cenas, um editor de terrenos e uma ferramenta para a otimização de eventos, *frames* e memória (Figura 2.38).

²⁶ <http://reach3dx.com/reach3dx/overview/>

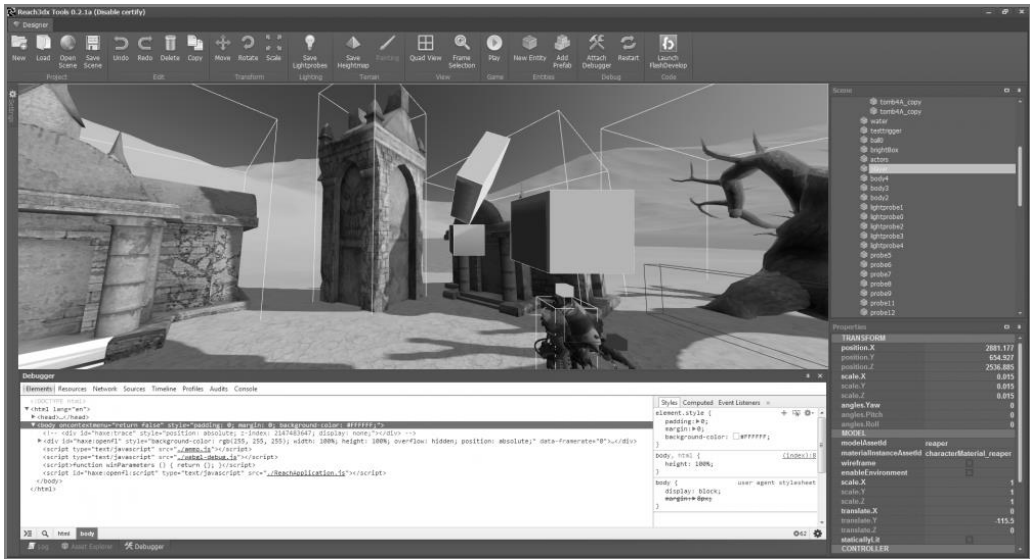


FIGURA 2.38 – Editor Reach 3DX

O motor gráfico suporta renderização de qualidade com iluminação e sombras dinâmicas e reflexos. Permite também o *streaming* de terrenos, no sentido de diminuir a memória necessária no dispositivo cliente. O motor de simulação física é baseado na biblioteca **BulletPhysics**. A programação pode ser feita usando *scripts* Lua ou JavaScript, ou ainda a linguagem Haxe, que é capaz de gerar código nativo em C++.

2.3.2.13 CONSTRUCT 2

A Scirra também desenvolve um motor de jogos 2D, de nome Construct 2²⁷. Este motor também é pago, embora esteja disponível uma versão gratuita para desenvolver jogos que funcionam em modo *offline* usando um *browser*. Para criar verdadeiramente jogos independentes será necessária uma versão paga. Estas versões incluem geração de executáveis para iOS, Android, Windows, Mac OS X, Linux e Wii U. Algumas das próprias funcionalidades do motor também estão limitadas na versão gratuita.

O desenvolvimento é realizado com um editor proprietário, visual, que permite a definição gráfica do jogo, a criação de eventos (programação visual) e a escolha de entre vários comportamentos predefinidos e configuráveis (Figura 2.39). No entanto, é possível estender as funcionalidades básicas usando JavaScript ou o desenho de efeitos visuais usando a linguagem GLSL.

²⁷ <https://www.scirra.com/construct2>



FIGURA 2.39 – Painel de comportamentos do Construct 2

2.3.2.14 CORONA SDK

O Corona SDK²⁸ é um motor para jogos 2D, com versão gratuita e versões pagas. A versão gratuita permite a criação de jogos para Android, iOS e Kindle. Algumas das versões pagas alargam o leque para incluir o Windows Phone.

À parte desta diferença, a versão gratuita não inclui o motor gráfico denominado *premium*, nem permite o uso nativo de bibliotecas.

O desenvolvimento via *scripting* é feito em Lua, o motor gráfico é implementado diretamente sobre a API OpenGL, o controlo de física é feito usando o Box2D e o sistema de som é implementado sobre OpenAL.

2.3.3 REFLEXÃO FINAL

Nas páginas anteriores foram referidos vários motores de jogos. Embora existam bastantes mais, a sua principal diferença centra-se, habitualmente, na interface de

²⁸ <http://coronalabs.com/products/corona-sdk/>

desenvolvimento, seja interface gráfica, linguagem de *scripting* ou linguagem nativa escolhida, e, no caso dos motores pagos, no tipo de suporte.

Como se viu, a maioria dos motores, sejam gratuitos ou pagos, baseia os seus motores em bibliotecas comuns. Por exemplo, para a simulação física a maioria usa a biblioteca **Box2D** para jogos a duas dimensões e a **BulletPhysics** para jogos tridimensionais. Apenas um par de motores inclui o poderoso motor da NVIDIA PhysX. Do mesmo modo, para suporte áudio é usada, essencialmente, a biblioteca **OpenAL**.

As principais razões para a escolha do Unity 3D e do libgdx para este livro são:

- ⊗ **Unity 3D** – existe uma versão gratuita, completa, que permite a criação de jogos para as várias arquiteturas e que se possa comercializar o jogo (embora exista um valor máximo de lucro permitido). Além disso, a quantidade de livros escritos e de informação na Internet sobre esta ferramenta é grande. O facto de permitir o desenvolvimento quer em JScript ou C# foi, também, importante;
- ⊗ **libgdx** – pareceu-nos importante escolher também uma ferramenta gratuita. O libgdx tem como vantagens o facto de permitir o desenvolvimento em Java (o que possibilitou manter o livro centrado em Java e nas suas variantes) e de ser fácil de usar com um IDE comum, como o Android Studio, que foi escolhido para guiar o desenvolvimento deste livro. Além disso, não é apenas dedicada ao componente gráfico, já que inclui suporte para motores de física 2D e 3D (mais uma vez, **Box2D** e **BulletPhysics**).

3

API 2D PARA ANDROID

Neste capítulo aplicam-se as API da Google para a programação em 2D – `Drawable` e `Canvas` – no desenvolvimento de uma aplicação gráfica. Como exemplo, é desenvolvida uma versão do jogo *Othello* (também conhecida por *Reversi*) que denominaremos *Othelloid*. A explicação do jogo é organizada da seguinte forma: em primeiro lugar, apresentam-se as classes principais do jogo; de seguida, dá-se ênfase à integração da inteligência artificial no jogo através do algoritmo *Minimax*, um algoritmo genérico e de fácil aplicação à maioria dos jogos de tabuleiro; e, finalmente, detalham-se as interfaces aplicacionais e gráficas do jogo.

3.1 INTRODUÇÃO

A plataforma Android disponibiliza várias API que facilitam a implementação das principais facetas de um jogo digital. Algumas delas facilitam a geração de gráficos 2D/3D. Neste capítulo destacam-se as API `Drawable` e `Canvas`:

- ⊙ `Drawable` é uma API gráfica 2D que usa uma abordagem declarativa para a criação de gráficos incorporando as instruções de desenho em ficheiros XML;
- ⊙ `Canvas` é uma API gráfica 2D que envolve o desenho diretamente numa superfície (por exemplo, um *bitmap*) fornecendo um maior controlo no modo como os gráficos são criados.

Para demonstrar a aplicabilidade prática destas API será desenvolvida uma versão do jogo *Othello* (ou *Reversi*) denominada *Othelloid* (Figura 3.1). Neste contexto, dar-se-á ênfase às classes que modelam o jogo em termos de lógica e grafismo. No primeiro caso, explicam-se as classes que manipulam o tabuleiro, a sua representação e o cálculo das jogadas. Dá-se também particular importância ao algoritmo que serve de base às jogadas efetuadas pelo computador, denominado *Minimax*. No segundo caso, expõem-se as interfaces gráficas do jogo, mais concretamente o ecrã inicial que contém o menu do jogo e o ecrã das opções no qual se pode configurar o jogo.

FIGURA 3.1 – O jogo *Othelloid*

3.2 TABULEIRO

Esta primeira secção é dedicada à representação interna do tabuleiro e à criação de algumas funções auxiliares. Em primeiro lugar, será apresentada a classe `Posicao`, que representa a posição de uma peça no tabuleiro, como sendo um par de inteiros. Posteriormente, será apresentada a classe `Tabuleiro` e toda a sua funcionalidade associada.

3.2.1 CLASSE `Posicao`

A classe `Posicao` é um par de inteiros, as coordenadas x e y de uma peça no tabuleiro, e inclui muito pouca funcionalidade extra. Uma vez que é uma classe auxiliar, que funciona apenas como estrutura de dados, foi decidido que os seus membros seriam públicos.

```
public class Posicao {
    public int x;
    public int y;
    ...
}
```

Em relação aos construtores, foram definidos dois: um construtor capaz de produzir uma posição com base num par de coordenadas, e outro com base noutra posição, copiando-lhe as coordenadas.

```
public Posicao(int x, int y) {
    this.x= x;
    this.y= y;
}
public Posicao(Posicao p) {
    this.x = p.x;
    this.y = p.y;
}
```

A funcionalidade desta classe é limitada. Implementa apenas um incrementador, que, com base nas coordenadas de uma outra posição, atualiza as coordenadas do objeto invocador.

```
public void Incrementa(Posicao p) {
    this.x += p.x;
    this.y += p.y;
}
```

Além deste método, é redefinido o método `equals`, que confirma se uma posição representa as mesmas posições do objeto invocador. O método começa por confirmar se algum dos objetos é nulo, verificando depois o seu tipo e, no caso de ser uma posição, verifica se as coordenadas estão corretas.

```
@Override
public boolean equals(Object other) {
    if (other == null) return false;
    if (other == this) return true;
    if (!(other instanceof Posicao)) return false;
    Posicao o = (Posicao)other;
    return o.x == this.x && o.y == this.y;
}
```

3.2.2 CLASSE Tabuleiro: REPRESENTAÇÃO BASE

A classe `Tabuleiro` é responsável pela representação interna do tabuleiro do *Othelloid* (Figura 3.2). Em primeiro lugar, são definidas algumas constantes (em maiúsculas) que representam, pela ordem apresentada, o número de colunas e de linhas do tabuleiro (que embora seja fixo, e igual a 8, torna a leitura do código mais simples em determinadas circunstâncias) e os estados de uma posição do tabuleiro (vazia, com uma peça branca ou com uma peça preta).

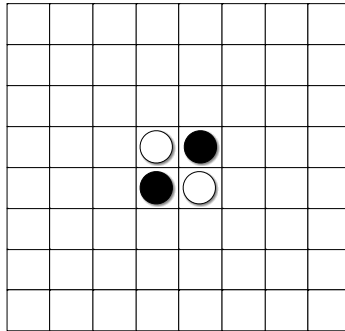


FIGURA 3.2 – Tabuleiro *Othelloid* na posição inicial

Em relação a variáveis de instância, é guardada uma matriz com as 64 posições do tabuleiro, qual o jogador que deve fazer a próxima jogada e, finalmente, um contador com o número de posições livres no tabuleiro:

```
public class Tabuleiro {
    public static final int COLS = 8;
    public static final int LINS = 8;
    public static final int VAZIA = 0;
    public static final int BRANCA = 1;
    public static final int PRETA = 2;
    private int[][] tabuleiro;
    private int jogador;
    private int nrPosLivres;
    ...
}
```

Existem outras variáveis de classe que serão definidas mais adiante, e que deverão ser adicionadas após o código aqui apresentado. O construtor do tabuleiro não recebe qualquer parâmetro e é responsável pela inicialização da matriz, colocando todas as posições a vazio e adicionando as primeiras quatro peças (duas de cada cor) tal como representado na Figura 3.2.

```
public Tabuleiro() {
    tabuleiro = new int[LINS][COLS];
    for (int linha = 0; linha < LINS; linha++) {
        for (int coluna = 0; coluna < COLS; coluna++) {
            tabuleiro[linha][coluna] = VAZIA;
        }
    }
    colocarPedra(3, 3, BRANCA);
    colocarPedra(4, 4, BRANCA);
    colocarPedra(3, 4, PRETA);
    colocarPedra(4, 3, PRETA);
    jogador = BRANCA;
    nrPosLivres = LINS * COLS - 4;
}
```


O método `colocarPedra` é usado para preencher uma posição específica do tabuleiro com a cor. Do mesmo modo, também se implementou o método `obterPedra` para obter a cor da pedra em determinada posição do tabuleiro (ou se a posição se encontra vazia).

```
public void colocarPedra(int linha, int coluna, int jogador) {
    tabuleiro[linha][coluna] = jogador;
}
public int obterPedra(int linha, int coluna) {
    return tabuleiro[linha][coluna];
}
```

Para além deste acessor foram criados outros dois que apenas verificam se determinada posição do tabuleiro se encontra vazia ou se a pedra que lá se encontra tem a cor branca.

```
public boolean Vazia(int linha, int coluna) {
    return tabuleiro[linha][coluna] == VAZIA;
}
public boolean Branca(int linha, int coluna) {
    return tabuleiro[linha][coluna] == BRANCA;
}
```

O jogo termina quando o tabuleiro se encontra totalmente preenchido ou se ambos os jogadores não têm jogadas válidas.

```
public boolean FimDeJogo() {
    return nrPosLivres == 0 ||
        (jogadas(BRANCA).size() == 0 && jogadas(PRETA).size() == 0);
}
```

O método `jogadas`, ainda não implementado, retorna uma lista das jogadas possíveis para determinado jogador. Este método será discutido na secção 3.2.3.

Para terminar a estrutura base desta classe serão implementados dois novos métodos: um para obter qual o próximo jogador e outro para alternar o jogador atual.

```
public int jogadorAtual() {
    return jogador;
}
public void alternaJogador() {
    jogador = (jogador == BRANCA) ? PRETA : BRANCA;
}
```

3.2.3 CLASSE Tabuleiro: CÁLCULO DE JOGADAS

Para implementar o jogo será necessário, por um lado, calcular quais as jogadas válidas (para não permitir que o jogador possa realizar uma inválida) e, por outro,

realizar a jogada propriamente dita. A Figura 3.3 mostra as jogadas possíveis para o jogador de cor preta.

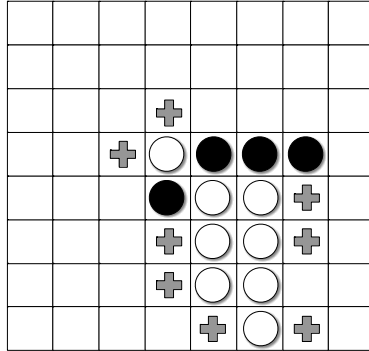


FIGURA 3.3 – Tabuleiro *Othelloid* com movimentos válidos para a cor preta

É possível que a implementação destes métodos seja um pouco estranha. O principal objetivo, no modo como foram implementados, foi a redução na quantidade de código.

Em primeiro lugar, considere-se o problema de, dado um jogador e uma posição vazia, verificar se esta é, ou não, uma jogada válida. Para que uma posição seja uma jogada válida terá de existir uma linha (horizontal, vertical ou diagonal) de peças da cor do oponente, a começar numa casa vizinha à casa vazia e a terminar numa peça da cor do jogador (Figura 3.4).

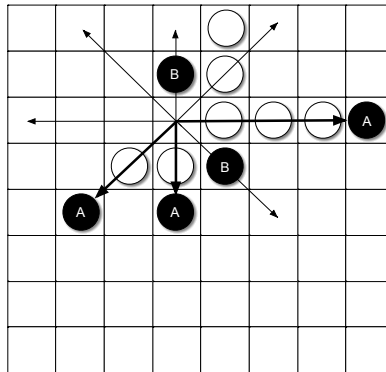


FIGURA 3.4 – Esquema de cálculo de validade de uma jogada

Nesta figura, considerando a jogada de uma pedra preta, repare-se que apenas as pedras A validam a jogada na posição de onde partem as setas. Embora as pedras B também sejam limites, são vizinhas da posição inicial e, portanto, não são suficientes para validar a jogada.

O método `jogadas`, apresentado de seguida, retorna uma lista de posições de jogadas válidas para determinado jogador. O algoritmo baseia-se em percorrer todo o tabuleiro e, para cada uma das posições vazias, invocar o método `limita` nas oito direções possíveis, no sentido de obter uma pedra que valide uma jogada nessa posição.

```
public ArrayList<Posicao> jogadas(int jogador) {
    ArrayList<Posicao> possiveisJogadas = new ArrayList<Posicao>();
    for (int x = 0; x < 8; x++) {
        for (int y = 0; y < 8; y++) {
            if (!Vazia(x, y)) continue;
            boolean paraSair = false;
            for (int i = -1; i <= 1; i++) {
                for (int j = -1; j <= 1; j++) {
                    if (i == 0 && j == 0) continue;
                    Posicao limite = limita(new Posicao(x, y),
                        new Posicao(i, j), jogador);
                    if (limite != null && tabuleiro[x + i][y + j] != jogador) {
                        possiveisJogadas.add(new Posicao(x, y));
                        paraSair = true;
                    }
                }
                if (paraSair) break;
            }
            if (paraSair) break;
        }
    }
    return possiveisJogadas;
}
```

Os dois ciclos externos (sobre as variáveis x e y) são usados para percorrer todas as posições do tabuleiro. Sempre que uma casa vazia é encontrada, são procurados limites que validem a jogada em cada uma das oito direções possíveis. Para isso, são usados os dois ciclos internos (sobre as variáveis i e j) que irão gerar vários incrementos: $(-1, -1)$, $(-1, 0)$, $(-1, 1)$, $(0, -1)$, etc. (Figura 3.5). Para cada um destes possíveis incrementos é usado o método `limita`, que verifica se existem condições, nessa direção, para que a jogada seja válida. Quando um limite é encontrado, e esse limite não é uma posição diretamente vizinha da posição vazia, então essa posição é uma jogada válida. Nessa altura, deixa de ser necessário testar mais direções, pelo que a variável `paraSair` é colocada a verdadeiro, forçando os dois ciclos internos a terminar.

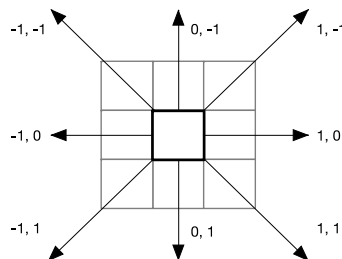


FIGURA 3.5 – Direções baseadas nos incrementos i e j

O método `limita` recebe uma posição de origem, uma direção (um par de incrementos, tal como calculados pelo método anterior) e a informação de qual o jogador a colocar a próxima pedra. A posição será incrementada (ou decrementada) enquanto se encontrar dentro dos limites do tabuleiro e enquanto não encontrar uma posição vazia, ou da cor do jogador atual. Logo que um limite seja encontrado, o método retorna-o. Se nenhum limite for encontrado, é retornado um objeto nulo.

```
private Posicao limita(Posicao origem, Posicao direcao, int jogador) {
    Posicao pos = new Posicao(origem);
    pos.Incrementa(direcao);
    while (pos.x >= 0 && pos.y >= 0 && pos.x < 8 && pos.y < 8 &&
        !Vazia(pos.x, pos.y)) {
        if (tabuleiro[pos.x][pos.y] == jogador) return pos;
        pos.Incrementa(direcao);
    }
    return null;
}
```

Com estes dois métodos implementados torna-se fácil verificar se determinada posição é uma jogada válida: basta calcular a lista de jogadas possíveis e verificar se a posição desejada se encontra na lista.

```
public boolean jogadaValida(int linha, int coluna, int jogador) {
    ArrayList<Posicao> listaJogadas = jogadas(jogador);
    return listaJogadas.contains(new Posicao(linha, coluna));
}
```

O passo seguinte é implementar o método que realiza, verdadeiramente, a jogada escolhida (e já validada). Embora para a simulação do tabuleiro não fosse necessário, para a implementação do algoritmo de inteligência artificial é importante que este método crie uma nova cópia do tabuleiro com o novo estado (depois de realizada a jogada). Para facilitar este processo será implementado o método `duplica`, que cria uma nova cópia do tabuleiro.

```
private Tabuleiro duplica() {
    Tabuleiro novoTabuleiro = new Tabuleiro();

    novoTabuleiro.jogador = jogador;
    novoTabuleiro.nrPosLivres = nrPosLivres;
    novoTabuleiro.tabuleiro = new int[LINS][COLS];
    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            novoTabuleiro.tabuleiro[i][j] = tabuleiro[i][j];
        }
    }
    return novoTabuleiro;
}
```

Portanto, o método `realizaJogada` retornará um novo tabuleiro depois de realizar a jogada solicitada. O algoritmo é bastante simples: é criada a cópia do tabuleiro,

decrementado o número de posições livres, colocada a pedra da cor do jogador na posição desejada, alterando as pedras envolvidas na jogada, e alternado o jogador atual.

Esta alteração das pedras é realizada usando a mesma técnica do método `jogadas`, verificando as oito direções possíveis e alterando a cor das pedras das direções em que existe um limite.

```
public Tabuleiro realizaJogada(int x, int y, int jogador) {
    Tabuleiro novoTabuleiro = duplica();

    novoTabuleiro.nrPosLivres--;
    novoTabuleiro.colocarPedra(x, y, jogador);

    for (int i = -1; i <= 1; i++) {
        for (int j = -1; j <= 1; j++) {
            if (i == 0 && j == 0) continue;
            Posicao limite = novoTabuleiro.limita(new Posicao(x, y),
                                                new Posicao(i, j),
                                                jogador);

            if (limite != null)
                novoTabuleiro.atualizaDirecao(new Posicao(x, y),
                                                limite,
                                                new Posicao(i, j),
                                                jogador);
        }
    }
    novoTabuleiro.alternaJogador();
    return novoTabuleiro;
}
```

O método `atualizaDirecao` é semelhante ao método `limita`, mas enquanto procura um limite também altera as posições pelas quais vai iterando.

```
private void atualizaDirecao(Posicao origem, Posicao destino,
                           Posicao incremento, int jogador) {
    Posicao pos = new Posicao(origem);

    while (pos.x != to.x || pos.y != to.y) {
        tabuleiro[pos.x][pos.y] = jogador;
        pos.Incrementa(incremento);
    }
}
```

3.3 INTELIGÊNCIA ARTIFICIAL

Para a implementação de inteligência artificial (IA) neste jogo será usado o algoritmo *Minimax*, um algoritmo genérico e de fácil aplicação à maioria dos jogos de tabuleiro. De seguida, discutem-se os detalhes teóricos deste algoritmo, apelando à intuição do leitor, para melhor perceber o algoritmo, que é apresentado em detalhe. Finalmente, aplica-se o *Minimax* ao caso concreto do *Othelloid*.

3.3.1 MINIMAX: INTUIÇÃO

Ao jogar algum tipo de jogo de tabuleiro, baseado em turnos (cada jogador faz a sua jogada de cada vez), o jogador não decide realizar, apenas, aquela jogada que produz uma melhor posição (ou uma posição dominante), mas também aquela jogada que coloca o oponente na pior posição possível.

Ou seja, ao mesmo tempo, tenta-se maximizar a própria classificação e minimizar a classificação do adversário. Do mesmo modo, o adversário tem como objetivo maximizar a sua própria classificação e minimizar a do oponente.

O algoritmo *Minimax* baseia-se na construção de uma árvore de jogadas possíveis (com base num limite máximo de jogadas, por questões de eficiência). O primeiro nível corresponde à jogada do motor de IA e, portanto, à maximização da sua própria classificação. O segundo nível corresponde à jogada do jogador, ou seja, à minimização da classificação do motor de IA. A árvore continua, sendo o terceiro nível de maximização e o quarto de minimização, e assim sucessivamente para os restantes níveis. Esta dualidade, de minimização e maximização, justifica o nome do algoritmo, e é habitualmente denominada *minimaxing*.

O objetivo final é maximizar a pontuação de um conjunto de jogadas da IA e minimizar a pontuação de um conjunto de jogadas do utilizador, no sentido de obter a sequência de jogadas mais promissora.

O algoritmo cria uma árvore em que cada nodo corresponde a um tabuleiro, e a transição entre tabuleiros corresponde a uma jogada de um dos jogadores. No início, as folhas são analisadas usando uma função de avaliação. Esta função irá atribuir uma classificação a cada tabuleiro num intervalo entre $-valor$ e $+valor$. Um tabuleiro será avaliado com $+valor$ se a IA tiver ganho o jogo ou com $-valor$ se a IA tiver perdido o jogo. A posição central do intervalo, o valor 0, corresponde ao empate. Um valor positivo, entre 0 e $+valor$, representa o quão próximo está um tabuleiro do empate ou da vitória, enquanto um valor negativo, entre $-valor$ e 0, representa o quão próximo está o tabuleiro da derrota, ou do empate.

Ao avaliar as folhas, colocam-se as avaliações dos respetivos tabuleiros em “bolhas” que vão subindo pela árvore. Em cada nível é aplicada a regra *minimax*: na jogada da IA, é escolhido o ramo com uma avaliação mais promissora; na jogada do utilizador, é escolhido o ramo com a pior avaliação (aquela que mais prejudica a IA). Eventualmente serão obtidos resultados para cada movimento disponível e será possível, simplesmente, escolher o melhor.

3.3.2 MINIMAX: ALGORITMO

O algoritmo aqui apresentado é recursivo. Em cada recursividade calcula-se um novo tabuleiro, a partir do tabuleiro atual e de uma possível jogada. Para o primeiro nível calculam-se os tabuleiros que podem ser obtidos com jogadas válidas da IA. Para o segundo nível calculam-se todos os tabuleiros que podem ser obtidos após o utilizador ter jogado. Este processo recorre até um nível máximo de recursividade ou profundidade (e que pode ser utilizado como mecanismo para controlar o nível de dificuldade).

Nas folhas (quando o algoritmo atinge a profundidade máxima definida) é retornada, simplesmente, a classificação desse tabuleiro (em relação à IA ou ao jogador, dependendo de a profundidade máxima ser ímpar ou par). Nos restantes níveis, o algoritmo escolhe o tabuleiro a partir do qual se obtém uma melhor classificação (quando é a IA a jogar) ou quando se obtém a menor classificação (ou seja, considerando que o utilizador será sempre capaz de escolher a sua melhor jogada possível).

A classe `Minimax` é estática e contém apenas dois métodos. O método `obterMelhorJogada` é um envelope para o segundo método, inicializando o nível de recursividade (a 0) e retornando apenas a melhor jogada calculada.

```
public final class Minimax {
    public static Posicao obterMelhorJogada(Tabuleiro tabuleiro,
                                           int jogador,
                                           int profundidadeMax) {
        Pair<Posicao, Integer> melhorJogadaPontuacao
            = minimax(tabuleiro, jogador, profundidadeMax, 0);
        return melhorJogadaPontuacao.first;
    }
}
```

O método `minimax` (correspondente à dita função recursiva explicada anteriormente) recebe como parâmetros o tabuleiro original, qual o jogador que deverá realizar a próxima jogada, qual o nível de profundidade máxima a que o algoritmo pode recorrer e, finalmente, qual o nível de recursividade atual.

Por sua vez, o resultado da função é um par, composto pela posição que representa a melhor jogada (seja para a IA ou para o jogador) obtida e pela classificação que essa jogada permite obter.

```
private static Pair<Posicao, Integer> minimax(Tabuleiro tabuleiro,
                                             int jogador, int profMax, int profAct) {

    if (tabuleiro.FimDeJogo() || profAct == profMax) {
        int pontuacao = tabuleiro.avalia(jogador);
        return new Pair<Posicao, Integer>(null, pontuacao);
    }

    Posicao melhorJogada = null;
    int melhorPontuacao = 200000; // EXAGERAR

    if (tabuleiro.jogadorAtual() == jogador)
```

```
    melhorPontuacao = -melhorPontuacao;

    ArrayList<Posicao> jogadas
        = tabuleiro.jogadas(tabuleiro.jogadorAtual());

    if (jogadas.size() == 0) {
        int pontuacao = tabuleiro.avalia(jogador);
        return new Pair<Posicao, Integer>(null, pontuacao);
    }

    Pair<Posicao, Integer> atual = null;

    for (Posicao p : jogadas) {
        Tabuleiro novoTabuleiro =
            tabuleiro.realizaJogada(p.x, p.y, tabuleiro.jogadorAtual());
        atual = Minimax.minimax(novoTabuleiro, jogador, profMax,
                                profAct + 1);

        if (tabuleiro.jogadorAtual() == jogador) {
            if (atual.second > melhorPontuacao) {
                melhorPontuacao = atual.second;
                melhorJogada = p;
            }
        } else {
            if (atual.second < melhorPontuacao) {
                melhorPontuacao = atual.second;
                melhorJogada = p;
            }
        }
    }
    return new Pair<Posicao, Integer>(melhorJogada, melhorPontuacao);
}
```

O primeiro passo corresponde a verificar o final de jogo (*game over*) ou o facto de se ter atingido o nível de recursividade máximo. Nesta situação, e tal como descrito anteriormente, é retornado um par com a jogada nula (não foi calculada nenhuma jogada) e a classificação do tabuleiro atual.

De seguida, inicializa-se a melhor jogada com um valor fora do intervalo de valores retornados pela função de avaliação. Se o jogador que realizar a próxima jogada for aquele para o qual queremos calcular a jogada, inicializamos este valor a negativo, de modo a que esse valor possa ser usado como valor inicial no cálculo da classificação máxima. Por outro lado, se o jogador que realizar a próxima jogada for o oponente, então inicializamos essa mesma variável a um valor extremamente alto, de modo a que possa ser usado como valor inicial no cálculo da pior classificação possível.

Para gerar todos os tabuleiros que se podem obter a partir do tabuleiro atual usa-se o método `jogadas`, de modo a obter todas as jogadas possíveis para o próximo jogador. Caso não haja jogadas possíveis para o jogador, retorna-se o resultado de avaliar o tabuleiro atual.



Note-se que no jogo *Othello* é possível “passar a jogada” e, portanto, poder-se-ia ter tido essa situação em consideração. No entanto, como a maior parte dos jogos de tabuleiro não permite que se “passe” a jogada, decidiu-se manter o algoritmo deste modo.

Havendo jogadas possíveis, é necessário iterar sobre todas elas e invocar, recursivamente, o método `minimax`. O ciclo realiza cada uma das jogadas possíveis, obtendo um novo tabuleiro que será enviado na próxima recursividade.



Este passo do algoritmo *Minimax* é a razão pela qual se decidiu implementar o método `realizaJogada` para retornar uma cópia do tabuleiro e não o mesmo tabuleiro alterado. Assim torna-se muito mais simples construir a árvore de tabuleiros.

O resultado desta invocação é comparado com a melhor (ou pior) classificação obtida até ao momento. Se esta jogada for melhor (ou pior), então esta é a escolhida como a melhor (ou pior) jogada encontrada até ao momento. Depois de percorrer todas essas possíveis jogadas, é retornado um par composto pela posição referente à melhor (ou pior) jogada e a respetiva classificação que essa jogada permite obter.

3.3.3 FUNÇÃO DE AVALIAÇÃO

Como foi explicado na secção anterior, o algoritmo *Minimax* baseia-se numa função de avaliação que calcula o quão positivo ou negativo é determinado tabuleiro para determinado jogador. Por exemplo, se existem mais pedras brancas do que pretas há um indício de que o tabuleiro atual é mais propício para o jogador branco.

Existem várias heurísticas que podem ser usadas para avaliar um tabuleiro. A função de avaliação aqui apresentada usa:

- O número de peças que cada jogador tem (supondo que mais peças é sinal de um tabuleiro mais promissor);
- O número de cantos (C) obtidos e cantos perdidos para o adversário (Figura 3.6), supondo que quantos mais cantos um jogador tiver obtido melhor posicionado estará (note-se que os cantos são as únicas posições que, depois de serem tomadas por um jogador, não voltam a mudar de cor);
- O número de quase cantos (QC) obtidos e quase cantos perdidos para o adversário – a jogada num quase canto deve ser evitada, uma vez que permite que o outro jogador tome um canto, que é uma posição forte.

Estas heurísticas não são as únicas que podem ser usadas, nem são necessariamente as melhores.

C	QC					QC	C
QC	QC					QC	QC
QC	QC					QC	QC
C	QC					QC	C

FIGURA 3.6 – Cantos (C) e quase cantos (QC)

Para ser mais fácil o acesso às posições dos cantos e dos quase cantos optou-se por criar duas listas estáticas com as respectivas posições. Assim, no início da classe, juntamente com as outras variáveis de classe, definem-se mais duas, bem como as suas inicializações.

```
private static ArrayList<Posicao> cantos;
private static ArrayList<Posicao> quaseCantos;
static {
    cantos = new ArrayList<Posicao>();
    quaseCantos = new ArrayList<Posicao>();

    cantos.add(new Posicao(0, 0));
    cantos.add(new Posicao(0, COLS - 1));
    cantos.add(new Posicao(LINS - 1, 0));
    cantos.add(new Posicao(LINS - 1, COLS - 1));

    quaseCantos.add(new Posicao(0, 1));
    quaseCantos.add(new Posicao(0, COLS - 2));
    quaseCantos.add(new Posicao(1, 0));
    quaseCantos.add(new Posicao(1, 1));
    quaseCantos.add(new Posicao(1, COLS - 1));
    quaseCantos.add(new Posicao(1, COLS - 2));

    quaseCantos.add(new Posicao(LINS - 2, 0));
    quaseCantos.add(new Posicao(LINS - 2, 1));
    quaseCantos.add(new Posicao(LINS - 2, COLS - 1));
    quaseCantos.add(new Posicao(LINS - 2, COLS - 2));

    quaseCantos.add(new Posicao(LINS - 1, 1));
    quaseCantos.add(new Posicao(LINS - 1, COLS - 2));
}
```

A função de avaliação usa três métodos auxiliares, um que conta o número de pedras de determinado jogador, outro que conta o número de cantos conquistados por determinado jogador, e um outro que conta o número de quase cantos.

```

public int contaPedras(int jogador) {
    int total = 0;
    for (int i = 0; i < LINS; i++) {
        for (int j = 0; j < COLS; j++) {
            if (tabuleiro[i][j] == jogador) total++;
        }
    }
    return total;
}

public int contaCantos(int jogador) {
    int total = 0;
    for (Posicao p : Tabuleiro.cantos) {
        if (tabuleiro[p.x][p.y] == jogador) total++;
    }
    return total;
}

public int contaQuaseCantos(int jogador) {
    int total = 0;
    for (Posicao p : Tabuleiro.quaseCantos) {
        if (tabuleiro[p.x][p.y] == jogador) total++;
    }
    return total;
}

```

Torna-se então possível definir a função de avaliação. Esta função deve retornar o valor zero se os jogadores estiverem empatados (por exemplo, no início do jogo). Deve retornar um determinado valor máximo, positivo, quando o jogador indicado tiver ganho o jogo e retornar esse mesmo valor, mas negativo, quando o jogador indicado tiver perdido o jogo. Quaisquer outros valores nesse intervalo indicam o quão próximo está o jogador de ganhar (ou de perder) o jogo.

```

public int avalia(int jogador) {
    int outro = (jogador == BRANCA) ? PRETA : BRANCA;
    int contagemJog = contaPedras(jogador);
    int contagemAdv = contaPedras(outro);

    // EMPATE FINAL E VITÓRIA/PERDA DE JOGO
    if (FimDeJogo()) {
        if (contagemJog > contagemAdv)
            return 1000;
        else if (contagemAdv > contagemJog)
            return -1000;
        else
            return 0;
    }

    int cantosJog = contaCantos(jogador);
    int cantosAdv = contaCantos(outro);

    int quaseCantosJog = contaQuaseCantos(jogador);
    int quaseCantosAdv = contaQuaseCantos(outro);

```

```
// CANTOS: VALOR ENTRE -500 E 500
int cantos = (cantosJog - cantosAdv) * 125;

// TOTALPEÇAS: ENTRE -100 E 100
int totalPecas = 100 * (contagemJog - contagemAdv) / 64;

// QUASECANTOS: VARIA ENTRE -100 E 100
int quaseCantos = 100 * (quaseCantosJog - quaseCantosAdv) / 12;

// TOTAL VARIA ENTRE -700 E 700
return cantos + totalPecas + quaseCantos;
}
```

Repare-se que este mesmo código pode ser facilmente adaptado para mais heurísticas. Só é necessário garantir que o valor máximo definido (1000) é superior ao total máximo de heurísticas definidas. Também é importante que cada heurística seja balanceada, ou seja, que ela própria resulte num valor nulo quando há empate, ou um mesmo valor positivo ou negativo, conforme essa heurística indique a vitória ou a derrota, respetivamente.

3.4 INTERFACE DE JOGO

A interface de jogo é baseada numa extensão à classe `View`. Esta interface é responsável por invocar métodos sobre a classe `Tabuleiro`, para a sua atualização e invocação do método de IA, animando o processo de jogo.

Para facilitar a implementação será usada uma máquina de estados, que inclui os seguintes estados:

- **Jogador a jogar** – a aplicação está preparada para receber a jogada do utilizador, esperando que toque no ecrã;
- **Jogador a passar** – foi detetado que o jogador não tem qualquer movimento válido e, por isso, deverá passar. Será colocada no ecrã uma mensagem informativa;
- **IA a jogar** – foi terminada a jogada do utilizador e, por isso, poder-se-á iniciar a chamada ao módulo de IA;
- **IA a pensar** – o algoritmo *Minimax* está a correr. Quando terminar, colocará a jogada numa variável predefinida;
- **IA a passar** – o algoritmo *Minimax* terminou sem qualquer jogada possível. Informar o jogador de que o *Othelloid* passa a jogada;
- **Final de jogo** – o jogo terminou, é necessário contar as peças em jogo para apurar o vencedor.

A Figura 3.7 mostra a máquina de estados.

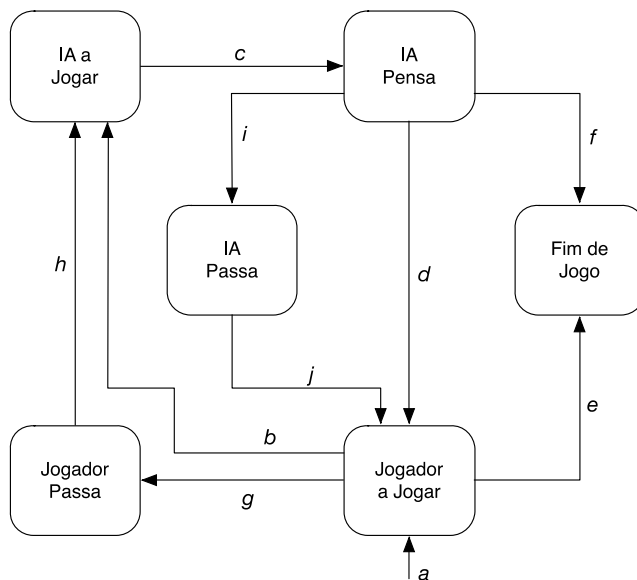


FIGURA 3.7 – Máquina de estados de funcionamento do *Othelloid*

A máquina de estados começa na seta a^{29} . Inicialmente, existem muitas jogadas possíveis, pelo que se aguarda a jogada do utilizador. Quando o utilizador joga, na transição b é colocada a pedra e calculado o novo tabuleiro. O *Othelloid* dá início à sua jogada transitando por c , sendo invocado o algoritmo *Minimax*. Obtida a jogada, a transição d coloca a pedra e calcula o novo tabuleiro. Este ciclo é o habitual durante praticamente todo o jogo.

Eventualmente, o jogador poderá não ter qualquer jogada válida. Neste caso, transita por g e, assim que o utilizador leia a mensagem de que terá de passar a sua jogada, transita por h para a jogada do *Othelloid*.

Do mesmo modo, o algoritmo *Minimax* poderá não retornar nenhuma jogada e, portanto, seguirá a transição i , sendo posteriormente mostrada ao utilizador uma informação de que a jogada foi passada. Assim que o jogador ler essa mensagem, transita-se por j para a jogada do utilizador.

Finalmente, após qualquer jogada, é validado o final de jogo (tabuleiro completamente preenchido), transitando por f ou e para o estado final.

²⁹ Na verdade, mais tarde será possível que seja o módulo de IA a iniciar o jogo. Mas a diferença é mínima e a explicação torna-se mais simples se considerarmos que o jogador é sempre o primeiro.

A classe responsável por esta máquina de estados é definida com as seguintes variáveis de classe:

```
class TabuleiroJogo extends View {
    private Tabuleiro tabuleiro;
    private Activity atividade = null;
    private enum Estado {
        JOGADOR_A_JOGAR, IA_A_JOGAR, IA_A_PENSAR,
        JOGADOR_PASSA, IA_PASSA, FIM_DE_JOGO
    };
    private Estado estado;
    private Paint paint;
    private int dimQuadrado;
    private int nrQuadrados = 8;
    private boolean modoVertical;
    private Posicao posBranco, posPreto;
    private Toast toastAtual = null;
    private ProgressDialog progresso = null;
    private AlertDialog.Builder builder = null;
    private AlertDialog finishDialog = null;
    private Posicao ultimaJogada = null;
    private int profundidade;
    ...
}
```

A primeira variável corresponde ao tabuleiro implementado anteriormente. Segue-se uma variável que guarda uma referência da atividade à qual a *view* está associada. De seguida, definem-se os vários estados possíveis para a máquina de estados, bem como a variável *estado*, que armazenará o estado no qual a máquina se encontra.

O bloco seguinte inclui um conjunto de variáveis úteis para os restantes métodos: informação sobre cor, grossura do traço, tipo e tamanho de letra (variável *paint*); dimensão (largura) de cada quadrado do tabuleiro (calculada dinamicamente de acordo com a resolução do dispositivo); número de quadrados do tabuleiro; se o dispositivo se encontra, ou não, na vertical; posição onde serão colocadas as contagens de peças dos dois jogadores; e quatro variáveis, uma para armazenar a atividade pai que invocou o jogo, outra para armazenar um objeto *Toast* que irá avisar o jogador de tentativas de jogadas inválidas, outra para a janela de progresso, que avisará o jogador do progresso do algoritmo *Minimax*, outra para o construtor de janelas de aviso, para indicar a passagem de jogadas, e finalmente outra para a janela de fim de jogo. Segue-se a variável *ultimaJogada*, onde será colocada a jogada que o algoritmo *Minimax* calculou, e a variável *profundidade*, que calcula o número máximo de níveis da árvore de procura que o algoritmo *Minimax* irá usar.

Para algumas destas variáveis foram criados os respetivos métodos acessores e modificadores.

O construtor desta classe recebe uma referência da atividade à qual a *view* `TabuleiroJogo` está associada.

```
public TabuleiroJogo(Activity atividade) {
    super(atividade);
    this.atividade = atividade;

    // INSTÂNCIA DO FICHEIRO DE PREFERÊNCIAS
    sharedPref = PreferenceManager.getDefaultSharedPreferences(atividade);

    // OBTÉM O NÍVEL DE DIFICULDADE
    profundidade =
        Integer.parseInt(sharedPref.getString("nivel_dificuldade", "1"));

    // OBTÉM A ORDEM DE JOGO
    if (sharedPref.getBoolean("ordem_jogar", true))
        estado = Estado.JOGADOR_A_JOGAR;
    else
        estado = Estado.IA_A_JOGAR;

    tabuleiro = new Tabuleiro();
    paint      = new Paint();

    progresso = new ProgressDialog(context);
    progresso.setMessage("A jogar...");
    progresso.setProgressStyle(ProgressDialog.STYLE_SPINNER);
    progresso.getWindow().setGravity(Gravity.BOTTOM);
    progresso.setIndeterminate(true);
    builder = new AlertDialog.Builder(atividade)
        .setIcon(android.R.drawable.ic_dialog_alert)
        .setTitle("Othelloid")
        .setPositiveButton("OK", null);
}
```

No construtor, invoca-se o construtor da superclasse e são obtidas as informações relativamente ao nível de dificuldade do jogo (que irá corresponder ao número de níveis da árvore de pesquisa do algoritmo *Minimax*) e de quem inicia o jogo. Estas informações são obtidas no ficheiro de preferências configurável através do menu de opções. É também instanciado o `tabuleiro` e o objeto `Paint` usado para desenhar.

Finalmente, as últimas linhas criam uma janela de progresso e um construtor de janelas. A janela de progresso irá usar o sinal de “a processar” do Android. Esta janela será mostrada quando o motor de IA estiver a calcular uma jogada e escondida quando for a vez do utilizador. Define-se a mensagem a mostrar, o estilo da animação, a posição (em baixo) e se o progresso é indeterminado (não é possível calcular uma percentagem atual do estado de processamento). O construtor de janelas será usado para mostrar as mensagens de final de jogo ou de passagem de jogada.

```
@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
    int width  = MeasureSpec.getSize(widthMeasureSpec);
    int height = MeasureSpec.getSize(heightMeasureSpec);
```

```

int d = (width < height) ? width : height;
dimQuadrado = d / nrQuadrados;
modoVertical = height > width;
if (modoVertical) {
    posBranco = new Posicao((int)(dimQuadrado * 0.5),
        ((int)(dimQuadrado * 8.5)));
    posPreto = new Posicao((int)(dimQuadrado * 0.5),
        ((int)(dimQuadrado * 9.5)));
} else {
    posBranco = new Posicao((int)(dimQuadrado * 8.5),
        ((int)(dimQuadrado * 0.5)));
    posPreto = new Posicao((int)(dimQuadrado * 8.5),
        ((int)(dimQuadrado * 1.5)));
}
setMeasuredDimension(width, height);
}

```

O método `onMeasure` é chamado para que cada *view* se possa configurar de acordo com as dimensões do dispositivo. As primeiras duas linhas convertem as medidas obtidas para pixels. Posteriormente, verifica-se qual a dimensão mais pequena (entre a largura e a altura), e com base nesta dimensão calcula-se o tamanho de cada célula do tabuleiro. O resto do método verifica se o dispositivo está na vertical (com altura superior à largura) e calcula posições para a contagem de pedras tendo em conta essa mesma orientação. Finalmente, qualquer definição deste método deve terminar com a invocação ao método `setMeasuredDimension` para definir o tamanho usado.

Para além do método anterior, o uso de uma *view* como *Canvas* obriga à definição do método `onDraw`, invocado sempre que é necessário redesenhar o ecrã. Este método recebe o objeto *Canvas* onde será desenhado o tabuleiro. Este método é o maior desta classe, porque dependendo do estado do jogo (ou seja, de acordo com a máquina de estados apresentada anteriormente) diferentes objetos serão desenhados.

```

protected void onDraw(Canvas canvas) {
    ArrayList<Posicao> jogadasValidas = null;
    /* PARTE 1 */
    if (estado != Estado.IA_A_PENSAR) progresso.hide();

    if (estado == Estado.JOGADOR_A_JOGAR) {
        jogadasValidas = tabuleiro.jogadas(tabuleiro.jogadorAtual());
        if (jogadasValidas.size() == 0)
            estado = Estado.JOGADOR_PASSA;
    }

    /* PARTE 2 */
    paint.setColor(Color.GRAY);
    canvas.drawRect(0, 0, canvas.getWidth(), canvas.getHeight(), paint);

    for (int linha = 0; linha < nrQuadrados; linha++) {
        for (int coluna = 0; coluna < nrQuadrados; coluna++) {
            int a = coluna * dimQuadrado;
            int b = linha * dimQuadrado;
            paint.setColor(Color.WHITE);

```



```

paint.setStrokeWidth(3);
canvas.drawRect(a, b,
                a + dimQuadrado, b + dimQuadrado, paint);
paint.setStrokeWidth(0);
paint.setColor(Color.rgb(123, 167, 123));
canvas.drawRect(a + 3, b + 3,
                a + dimQuadrado - 3, b + dimQuadrado - 3,
                paint );

if (! tabuleiro.Vazia(linha, coluna)){
    paint.setColor( tabuleiro.Branca(linha, coluna) ?
                    Color.WHITE : Color.BLACK);
    canvas.drawCircle(a +(dimQuadrado/2), b +(dimQuadrado/2),
                      dimQuadrado * 0.45f, paint);
}

if (estado == Estado.JOGADOR_PASSA ||
    estado == Estado.JOGADOR_A_JOGAR) {
    Posicao p = new Posicao(linha, coluna);

    if (jogadasValidas != null && jogadasValidas.contains(p)) {
        paint.setARGB(128, 255, 225, 225);
        canvas.drawCircle(a + (dimQuadrado/2),
                          b + (dimQuadrado/2),
                          dimQuadrado * 0.20f, paint);
    }
    if (ultimaJogada != null && p.equals(ultimaJogada)) {
        paint.setColor(Color.RED);
        canvas.drawCircle(a + (dimQuadrado/2),
                          b + (dimQuadrado / 2),
                          dimQuadrado * 0.05f, paint);
    }
}
}
}

/* PARTE 3 */
MostraPontos(canvas);

switch (estado) {
    case JOGADOR_PASSA:  HumanoPassa();      break;
    case IA_PASSA:       ComputadorPassa();  break;
    case IA_A_JOGAR:    ComputadorJoga();    break;
    case FIM_DE_JOGO:   FinalDeJogo();
}
}

```

A função foi dividida em três partes para tornar a explicação mais simples. Na primeira parte começa-se por verificar se a função *Minimax* está a calcular uma jogada. Em caso negativo, a janela de progresso será escondida. Logo de seguida, e se for a vez de o utilizador jogar, são calculados os movimentos possíveis. Caso não exista nenhum, muda-se o estado para JOGADOR_PASSA.

A segunda parte é responsável pelo desenho do tabuleiro. No primeiro passo preenche-se o fundo do `Canvas` a cinzento. Para tal, altera-se o pincel `Paint`, e desenha-se um retângulo com as dimensões do ecrã. De seguida, os dois ciclos aninhados desenharam as 64 casas do tabuleiro, preenchendo-o primeiro a verde e desenhando depois os contornos a branco.

Seguem-se duas estruturas condicionais. A primeira é responsável por desenhar as pedras no tabuleiro: para determinadas coordenadas, se essa posição se encontra ocupada, então desenha a peça na cor respetiva. A segunda trata de duas situações distintas: por um lado, o desenho de pequenos círculos cinzentos nas posições que correspondem a jogadas válidas; por outro, o desenho de um pequeno ponto vermelho sobre a última peça colocada pela IA, de modo a que o jogador tenha noção de qual foi a sua última jogada.

Na terceira parte é invocado um método para o desenho do número de peças de cada jogador. Segue-se uma estrutura que invoca um método para diferentes estados da máquina de estados: passar a jogada do utilizador, passar a jogada da IA, realizar a jogada da IA ou terminar o jogo.

O método que apresenta o número de peças de cada jogador é bastante simples, sendo apenas uma questão de posicionamento de objetos gráficos no `Canvas`.

```
private void MostraPontos(Canvas canvas) {
    paint.setColor(Color.BLACK);
    canvas.drawCircle(posPreto.x, posPreto.y,
        dimensaoQuadrado * 0.3f, paint);

    paint.setColor(Color.WHITE);
    canvas.drawCircle(posBranco.x, posBranco.y,
        dimensaoQuadrado * 0.3f, paint);

    paint.setTextSize(dimensaoQuadrado * 0.5f);

    canvas.drawText(tabuleiro.contaPedras(Tabuleiro.PRETA) + " pedras",
        posPreto.x + dimensaoQuadrado * 0.5f,
        posPreto.y + dimensaoQuadrado * 0.15f, paint);

    canvas.drawText(tabuleiro.contaPedras(Tabuleiro.BRANCA) + " pedras",
        posBranco.x + dimensaoQuadrado * 0.5f,
        posBranco.y + dimensaoQuadrado * 0.15f, paint);
}
```

Outro evento que é necessário tratar é o toque do utilizador no ecrã. É preciso verificar em que posição o jogador tocou e fazer corresponder essa posição a uma posição do tabuleiro; posteriormente, validar se essa posição é, ou não, uma jogada válida. O método responsável pelo tratamento deste evento é denominado `onTouchEvent`.

```
@Override
public boolean onTouchEvent(MotionEvent event) {
    if (estado != Estado.JOGADOR_A_JOGAR) return false;
```

```

if (toastAtual != null) toastAtual.cancel();

int action = event.getAction();

if (action == MotionEvent.ACTION_DOWN) return true;
if (action == MotionEvent.ACTION_UP) {
    if (event.getX() > Tabuleiro.LINS * dimensaoQuadrado ||
        event.getY() > Tabuleiro.COLS * dimensaoQuadrado)
        return false;

    int col = ((int) event.getX()) / dimensaoQuadrado;
    int row = ((int) event.getY()) / dimensaoQuadrado;

    if(!tabuleiro.jogadaValida(row, col, tabuleiro.jogadorAtual())) {
        toastAtual = Toast.makeText(getContext(),
            "Jogada Inválida!", Toast.LENGTH_SHORT);

        toastAtual.show();
    } else {
        ultimaJogada = null;
        tabuleiro = tabuleiro.realizaJogada(row, col,
            tabuleiro.jogadorAtual());

        if (tabuleiro.FimDeJogo()) estado = Estado.FIM_DE_JOGO;
        else estado = Estado.IA_A_JOGAR;

        invalidate();
    }
    return true;
}
return false;
}

```

Este método deve retornar um booleano que indica se o *input* do utilizador foi processado ou não, e recebe o evento de toque, com informação da posição e da fase do toque que despoletou o evento.

O método começa por verificar se a máquina de estados se encontra na fase de o utilizador jogar. Se não estiver nessa fase, então o toque no ecrã deverá ser ignorado. Logo de seguida, é verificado se está a ser mostrado o objeto `Toast` de jogada inválida. Se assim for, é escondido.

O método `getAction` aplicado ao evento recebido permite obter a fase do toque no ecrã: se o dedo foi pousado, arrastado ou levantado. Estas fases correspondem a diferentes valores inteiros que são validados nas estruturas condicionais que se seguem. Neste caso, ignoramos o toque, mas atuamos quando o dedo é levantado.

A atuação começa por validar se o toque foi dentro do tabuleiro. Em caso negativo, não é processado o toque, retornando um valor falso. Em caso positivo, as coordenadas do toque são obtidas usando os métodos `getX` e `getY` e convertidas nas coordenadas da posição do tabuleiro que foi tocada. Estas coordenadas são validadas como jogada válida ou não. Se for uma jogada não válida, é apresentado o objeto `Toast`

com uma mensagem de aviso. Se a jogada for válida, a variável `ultimaJogada` é colocada a `null`, para que a última jogada do motor de IA deixe de estar marcada da próxima vez que o tabuleiro for redesenhado. Segue-se o pedido de realização da jogada à classe responsável pelo tabuleiro. Finalmente, verifica-se se o jogo terminou, alterando o estado do jogo para o valor correspondente.

Para terminar a interface de jogo faltam implementar quatro métodos: a jogada do motor de IA, a situação em que o motor de IA passa o jogo, a situação em que o utilizador passa o jogo e a situação de final de jogo.

As duas passagens de jogo são bastante semelhantes. Correspondem apenas à apresentação de uma janela com essa informação, a mudança de qual o jogador atual (na classe `Tabuleiro`) e a alteração do estado da máquina de estados.

```
private void HumanoPassa() {
    tabuleiro.alternaJogador();
    AlertDialog d = builder.create();
    d.setMessage("Não tem jogadas disponíveis.");
    d.setOnDismissListener(new DialogInterface.OnDismissListener() {
        @Override
        public void onDismiss(DialogInterface dialog) {
            estado = Estado.IA_A_JOGAR;
            postInvalidate();
        }
    });
    d.show();
}

private void ComputadorPassa() {
    AlertDialog d = builder.create();
    d.setMessage("Não posso jogar! Passo!");
    d.setOnDismissListener(new DialogInterface.OnDismissListener() {
        @Override
        public void onDismiss(DialogInterface dialog) {
            estado = Estado.JOGADOR_A_JOGAR;
            postInvalidate();
        }
    });
    d.show();
}
```

Os dois métodos são muito semelhantes. Cada um constrói uma janela, usando o `builder` que foi criado no construtor da classe, com determinado título e determinado texto, e um botão de **OK**. Para além disso, são associados métodos a dois eventos: o evento de aceitar a mensagem clicando em **OK** (que foi definido ao criar a variável `builder`) ou o evento de tocar fora da janela. O método responsável por tratar o evento de o botão ter sido usado é nulo, uma vez que o Android invoca sempre o método `OnDismissListener` quando a janela é fechada e, portanto, pode-se colocar o código a ser executado apenas neste evento.

Estes métodos alteram o estado da máquina de estados e invocam o método `postInvalidate`, responsável por obrigar o ecrã a ser redesenhado. Note que este método tem uma funcionalidade semelhante ao método `invalidate`, com a diferença de que pode ser invocado a partir de uma *thread* que não seja a principal.

No entanto, há uma diferença subtil. O primeiro método, responsável pela situação em que o jogador passa, começa por alterar o jogador atual. Já no segundo método, essa mudança é realizada no método `JogadaComputador` apresentado já de seguida.

```
private void JogadaComputador() {
    estado = Estado.IA_A_PENSAR;
    progresso.show();
    invalidate();
    new Thread(new Runnable() {
        public void run() {
            Posicao jogada = Minimax.obterMelhorJogada(tabuleiro,
                tabuleiro.jogadorAtual(), profundidade);

            if (jogada == null) {
                tabuleiro.alternaJogador();
                estado = Estado.IA_PASSA;
            } else {
                ultimaJogada = new Posicao(jogada);
                tabuleiro = tabuleiro.realizaJogada(jogada.x, jogada.y,
                    tabuleiro.jogadorAtual());

                if (tabuleiro.FimDeJogo()) estado = Estado.FIM_DE_JOGO;
                else estado = Estado.JOGADOR_A_JOGAR;

                postInvalidate();
            }
        }
    }).start();
}
```

A parte interessante deste método é o uso de uma *thread*. Não convém que, enquanto o dispositivo executa o método *minimax*, que é bastante demorado, a aplicação fique bloqueada. Convém, por exemplo, que seja possível abortar o processo para terminar a aplicação. Assim, o processo de cálculo da jogada é feito numa nova *thread*, permitindo que a *thread* principal continue ativa. À parte disso, o método limita-se a invocar o método *minimax* e, se este retornar uma jogada, realiza-la-á, guardando essa informação na variável `ultimaJogada`, para que mais tarde o método `onDraw` seja capaz de representar essa jogada.

Falta definir apenas um método, responsável pelo tratamento do final de jogo, que foi batizado de `FinalDeJogo`, e cujo código é o seguinte:

```
private void FinalDeJogo() {
    progresso.hide();
    int white = tabuleiro.contaPedras(Tabuleiro.BRANCA);
    int black = tabuleiro.contaPedras(Tabuleiro.PRETA);
```

```
String message;
if (white > black) {
    message = "Ganhaste! Parabéns!!";
} else if (white == black) {
    message = "Empate! Foi um bom jogo!";
} else {
    message = "Ganhei! Quando jogamos de novo?";
}
int pts = getPontuacao();
message += "\nPontuação: " + pts;
finishDialog = builder.create();
finishDialog.setMessage(message);
finishDialog.setOnDismissListener(new
    DialogInterface.OnDismissListener() {
        @Override
        public void onDismiss(DialogInterface dialog) {
            getActivity().finish();
        }
    });
finishDialog.show();
}
```

Este método, para além de apresentar qual o jogador vencedor (ou o empate), também calcula a pontuação, que foi definida como uma operação aritmética simples sobre o número de pedras de cada jogador, e o nível de dificuldade (ou profundidade de pesquisa do algoritmo *Minimax*).

```
public int getPontuacao() {
    int pontos_jogador = tabuleiro.contaPedras(Tabuleiro.BRANCA);
    int pontos_ia      = tabuleiro.contaPedras(Tabuleiro.PRETA);
    int dificuldade   = profundidade;
    return dificuldade * (pontos_jogador - pontos_ia);
}
```

Após o final do jogo, é obtida uma referência à atividade associada à *view* `TabuleiroJogo` chamada `Jogo`. A classe `Jogo` tem como principal missão manter o *layout* do jogo (o tabuleiro) e obter a pontuação do mesmo *layout*. Após obter a referência à atividade `Jogo` através do método acesso `getActivity`, invoca-se o seu método `finish`:

```
getActivity().finish();
```

Depois, reescreve-se o método `finish` definindo uma nova *intent*. Esta nova *intent* terá associada a pontuação do jogador obtida através do método `getPontuacao`. A pontuação é incluída na *intent* e devolvida à atividade principal da aplicação (onde será gerido o *leaderboard* dos serviços de jogos da Google).

```
@Override
public void finish() {
    Intent responseIntent = new Intent();
    responseIntent.putExtra("score", tabuleiroJogo.getPontuacao());
    setResult(RESULT_OK, responseIntent);
    super.finish();
}
```

Todas as janelas devem ser fechadas antes de se destruir uma atividade. Sendo assim, no método `onClose` da atividade, deverá fechar a janela final de forma a evitar um *memory leak*.

```
public void onStop() {
    super.onStop();
    AlertDialog dialog = tabuleiroJogo.getDialog();
    if (dialog != null) {
        dialog.dismiss();
        dialog = null;
    }
}
```

3.5 INTERFACE AUXILIAR

A interface auxiliar do jogo inclui todas as atividades (e respetivos *layouts*) que que vão ajudar o utilizador no acesso ao jogo e respetiva configuração. São duas as atividades que serão discutidas nas próximas subsecções: uma atividade que contém um *layout* com o menu do jogo e uma atividade que gere as suas configurações.

3.5.1 ECRÃ INICIAL

Ao iniciar o jogo *Othelloid* é exibido um *layout* (Figura 3.8) com o título do jogo e um menu com itens que permitem:

- **Jogar** – iniciar o jogo com as configurações por omissão;
- **Opções** – definir as configurações do jogo, mais concretamente o nome do jogador, a ordem a jogar e o nível de dificuldade do computador;
- **Ranking** – visualizar a tabela de classificação com as pontuações;
- **Conquistas** – visualizar as conquistas já realizadas.



Os itens dos menus *Ranking* e *Conquistas* serão implementados no Capítulo 6 através dos serviços *leaderboards* e *achievements* que fazem parte da API para jogos da Google, denominada *Google Play Games Services* (GPGS).

O *layout* da atividade localizado em `res/layout/activity_main.xml` é composto por um `LinearLayout` com orientação vertical e tem como imagem de fundo um recurso *drawable* previamente armazenado em `res/drawable-hdpi/background.png`.

O elemento `LinearLayout` é composto por uma `ImageView` com o título do jogo e um novo `LinearLayout` com quatro elementos `ImageView` que representam os quatro itens do menu do jogo. Todos os elementos `ImageView` definem no seu atributo `src` recursos *drawable* do tipo imagem.



FIGURA 3.8 – Ecrã inicial do jogo *Othelloid*

Segue-se o conteúdo do *layout* da atividade inicial do jogo:

```
<LinearLayout ...
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@drawable/background"
    android:gravity="center_vertical" >
    <ImageView
        android:layout_width="330dp"
        android:layout_height="70dp"
        android:id="@+id/imageView"
        android:src="@drawable/Othelloid"
        android:layout_gravity="center_horizontal" />
    <LinearLayout
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="250dp"
        android:gravity="center_vertical">
        <ImageView
            android:layout_gravity="center_horizontal"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
```



```
        android:id="@+id/imgJogar"
        android:src="@drawable/jogar"/>
<ImageView
    android:layout_gravity="center_horizontal"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/imgOpcoes"
    android:src="@drawable/opcoes"
    />
<ImageView
    android:layout_gravity="center_horizontal"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/imgRanking"
    android:src="@drawable/ranking"
    />
<ImageView
    android:layout_gravity="center_horizontal"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/imgConquistas"
    android:src="@drawable/conquistas"
    />
</LinearLayout>
</LinearLayout>
```

O código da atividade principal é bastante simples, já que precisamos apenas de associar o *layout* e definir as ações a serem executadas sempre que se tocar nas opções de jogo. Segue-se o código inteiro da atividade:

```
public class MainActivity extends Activity {
    private static final int REQUEST_CODE = 1;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // CLICAR EM JOGAR
        ImageView imgJ = (ImageView) findViewById(R.id.imgJogar);
        imgJ.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                startActivityForResult(new Intent(MainActivity.this, Jogo.class),
                    REQUEST_CODE);
            }
        });

        // CLICAR EM OPÇÕES
        ImageView imgO = (ImageView) findViewById(R.id.imgOpcoes);
        imgO.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                startActivity(new Intent(MainActivity.this, Opcoes.class));
            }
        });
    }
}
```

```
// CLICAR EM RANKING
ImageView imgR = (ImageView) findViewById(R.id.imgRanking);
imgR.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        //TODO: INVOCAR A API LEADERBOARD A IMPLEMENTAR NO CAPÍTULO 6
    }
});

// CLICAR EM CONQUISTAS
ImageView imgC = (ImageView) findViewById(R.id.imgConquistas);
imgC.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        //TODO: INVOCAR A API ACHIEVEMENT A IMPLEMENTAR NO CAPÍTULO 6
    }
});
}

@Override
protected void onActivityResult(int requestCode,
                                int resultCode,
                                Intent data) {
    if(resultCode==RESULT_OK && requestCode == REQUEST_CODE) {
        if(data.hasExtra("score")) {
            // TODO: CHAMAR AS API DO GPGS PARA ATUALIZAR LEADERBOARDS
        }
    }
}
}
```

A atividade começa por carregar o seu *layout* através do método `setContentView`. Depois, definem-se as ações que vão ser executadas após toque nas opções de jogo. Em primeiro lugar, obtém-se o objeto `View` de cada opção. Depois, define-se um *listener* através do método `setOnClickListener` e implementa-se o respetivo método `onClick`.

Na primeira opção, **Jogar**, é invocado o método `startActivityForResult` com um objeto `Intent` que define explicitamente a atividade a ser chamada, neste caso a atividade `Jogo`, responsável pelo jogo, e um código de pedido. Esse código será útil para, após término do jogo e retorno ao menu principal, identificar a atividade que enviou o resultado alcançado pelo jogador.

A segunda opção, **Opções**, invoca o método `startActivity` com um objeto `Intent`, que desta vez inclui uma chamada à atividade `Opcoes` responsável pela configuração do jogo. Esta opção será explicada na secção 3.5.2.

As duas últimas opções, **Ranking** e **Conquistas**, vão permitir ao utilizador ver as pontuações e conquistas de todos os jogadores, tendo como base as API `LeaderBoard` e `Achievement` incluídas no GPGS. A implementação do *ranking* e das conquistas será discutida no Capítulo 6.

Finalmente, dois últimos aspetos relacionados com esta atividade: o modo do ecrã e a saída da aplicação.

O jogo *Othelloid* é exibido no modo de ecrã inteiro. A partir do Android 4.4 é possível definir as aplicações em modo de ecrã inteiro imersivo mesmo quando o utilizador interage com a aplicação. A forma padrão para definir os diferentes estados do ecrã usa o método `getDecorView` da classe `Window`, que devolve um objeto `View` representando a janela de nível de topo. Depois, com o método `setSystemUiVisibility` agrega-se um conjunto de *flags* que vão influenciar o estado do ecrã.

Para aplicações envolventes (por exemplo, jogos, aplicações de desenho), onde se espera que os utilizadores interajam bastante e não precisem de acesso frequente à UI do sistema, usa-se o modo imersivo *sticky* com as *flags* `SYSTEM_UI_FLAG_IMMERSIVE_STICKY`, `SYSTEM_UI_FLAG_FULLSCREEN` e `SYSTEM_UI_FLAG_HIDE_NAVIGATION`.



Modo imersivo *sticky* é um modo de ecrã inteiro no qual as barras do sistema (barra de estado e barra de navegação) aparecem escondidas. O utilizador pode exibir as barras do sistema semitransparentes de forma temporária, voltando as mesmas a esconderem-se automaticamente. O ato de *swiping* não limpa qualquer *flag* ou despoleta qualquer *listener* registado para mudanças de visibilidade, já que a aparência transitória das barras do sistema não é considerada uma mudança de visibilidade da GUI.

Contudo, o leitor deverá ter em conta que, na altura da escrita desta obra, a versão Jelly Bean (Android 4.1-4.3) equipa quase metade dos dispositivos existentes. Sendo assim, para compatibilizar a ação do modo de ecrã inteiro com uma maior gama de dispositivos optou-se por associar um tema à atividade a partir do ficheiro de manifesto:

```
<manifest ...>
  <application ...>
    <activity
      android:theme="@android:style/Theme.NoTitleBar.Fullscreen" ... />
    </application>
  </manifest>
```

Outra questão importante tem a ver com o fecho da aplicação. Note-se que não existe propositadamente essa opção no menu do jogo. Na verdade, as boas práticas no desenho de aplicações Android não aconselham implementar a opção de término de uma aplicação, mas sim deixar que o sistema operativo faça essa gestão.

Neste jogo implementa-se apenas o método `onBackPressed`, que é chamado sempre que o utilizador carrega no botão **Back** do dispositivo. A implementação inclui a chamada de um objeto `AlertDialog` com dois botões.

Caso se pressione o botão **Sim**, é invocado o método `moveTaskToBack`, que move a tarefa contendo a atividade para a última posição da pilha de atividades (*back stack*):

```
@Override
public void onBackPressed() {
    new AlertDialog.Builder(this)
        .setIcon(android.R.drawable.ic_dialog_alert)
        .setTitle("Terminar")
        .setMessage("Tem a certeza?")
        .setPositiveButton("Sim", new DialogInterface.OnClickListener() {
            @Override
            public void onClick(DialogInterface dialog, int which) {
                moveTaskToBack(true);
            }
        })
        .setNegativeButton("Não", null).show();
}
```

A aplicação pode ser retomada através do botão **Recents** do dispositivo, que permite aceder a todas as aplicações em memória.

3.5.2 ECRÃ DE OPÇÕES

O ecrã de opções vai permitir ao utilizador configurar o jogo. As opções de configuração do jogo *Othelloid* ficam a cargo de uma atividade baseada num dos modelos de atividade predefinidos do Android Studio. Para criar a atividade, clique com o botão direito do rato sobre o pacote que contém as classes do jogo e seleccione a opção **New** → **Activity** → **Settings Activity**. De seguida, preencha o formulário da janela ilustrada na Figura 3.9 e clique no botão **Finish**.

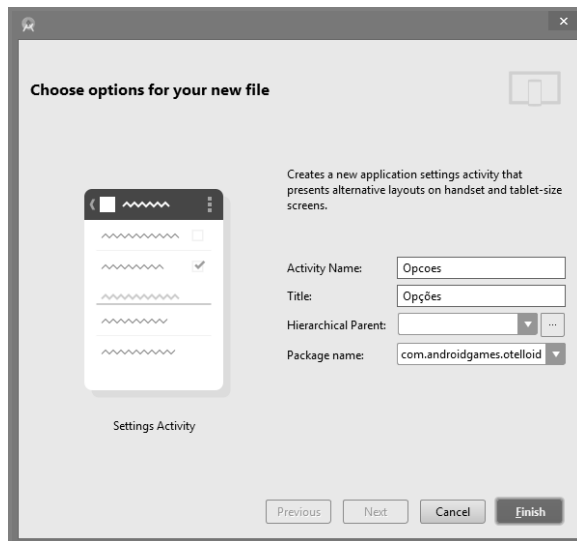


FIGURA 3.9 – Criação de uma atividade do tipo Settings

Por omissão, são gerados vários ficheiros. Para as configurações deste jogo são precisos apenas os seguintes ficheiros:

- ⊙ Uma subclasse de `PreferencesActivity` chamada `Opcoes`;
- ⊙ Dois ficheiros de preferências:
 - `res/xml/pref_geral.xml` – contém as opções de configuração;
 - `res/xml/pref_cabecalhos.xml` – contém os cabeçalhos das categorias das opções (a ser usado em *tablets*).
- ⊙ Um ficheiro de *strings* chamado `res/values/strings_opcoes.xml`.

A classe `Opcoes` carrega o ficheiro de preferências com um conjunto de opções de configuração da aplicação. A classe inclui também métodos que determinam como vão ser apresentadas as opções de configuração mediante estratégias alternativas para a organização dos conteúdos quer para *smartphones*, quer para *tablets*.

Se o dispositivo não tiver um ecrã grande (*tablets*) ou não suportar a classe `PreferenceFragment` (adicionada a partir da API nível 11), as opções de configuração são apresentadas em ecrã inteiro, ou seja, como uma única lista (Figura 3.10) carregada através do ficheiro `pref_geral.xml`.

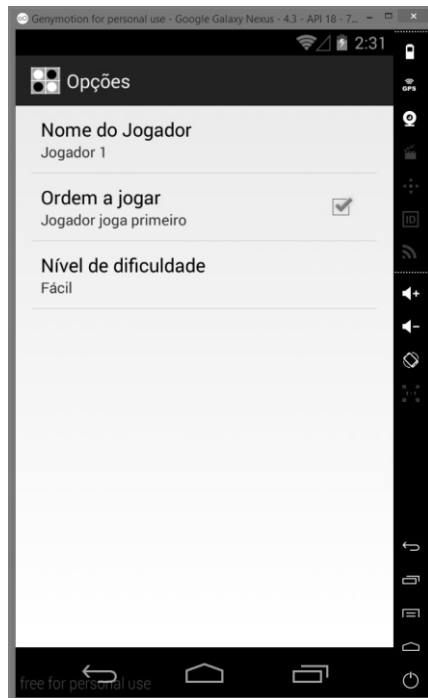


FIGURA 3.10 – Ecrã Opções em *smartphones*

Caso contrário, as opções de configuração são organizadas por categoria e o *layout* é dividido em dois painéis (Figura 3.11): os cabeçalhos das categorias surgem à esquerda e a lista de opções associadas à categoria selecionada surge à direita.

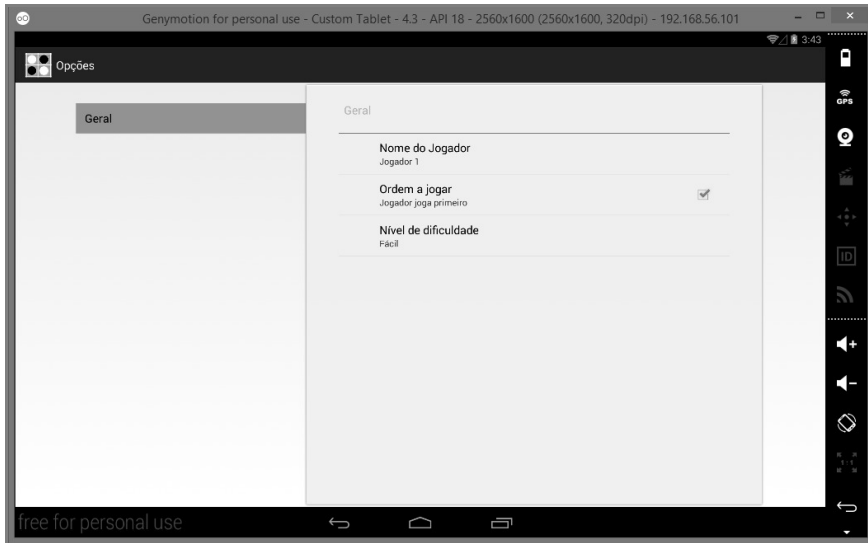


FIGURA 3.11 – Ecrã Opções em *tablets*

Os cabeçalhos das categorias são carregados através do ficheiro `pref_cabecalhos.xml`. Como já foi referido, as opções de configuração são representadas no ficheiro `pref_geral.xml`, que inclui os seguintes objetos:

- ⊙ `EditTextPreference` – para armazenar o nome do jogador;
- ⊙ `CheckBoxPreference` – para definir quem é o primeiro a jogar (se é o jogador ou o computador);
- ⊙ `ListPreference` – para listar os vários níveis de dificuldade do jogo.

O próximo excerto de código mostra todo o conteúdo do ficheiro:

```
<PreferenceScreen>
  <EditTextPreference
    android:key="nome_jogador"
    android:title="@string/pref_titulo_nome_jogador"
    android:defaultValue="@string/pref_desc_nome_jogador"
    ... />
  <CheckBoxPreference
    android:key="ordem_jogar"
    android:title="@string/pref_titulo_ordem_jogar"
    android:summary="@string/pref_desc_ordem_jogar"
    android:defaultValue="true" />
  <ListPreference
    android:key="nivel_dificuldade"
    android:title="@string/pref_titulo_niveis_dificuldade"
```

```

    android:defaultValue="1"
    android:entries="@array/pref_lista_niveis_dificuldade"
    android:entryValues="@array/pref_lista_valores_niveis_dificuldade"
    android:negativeButtonText="@null"
    android:positiveButtonText="@null" />
</PreferenceScreen>

```

O ficheiro de cabeçalhos é bastante simples, incluindo apenas um cabeçalho:

```

<preference-headers>
  <header
    android:fragment=
      "com.androidgames.otelloid.Opcoes$GeneralPreferenceFragment"
    android:title="@string/pref_titulo_cabecalho_geral" />
</preference-headers>

```

Ao seleccionar as opções de configuração surgem janelas de diálogo que vão permitir ao utilizador definir o valor respetivo (Figura 3.12).

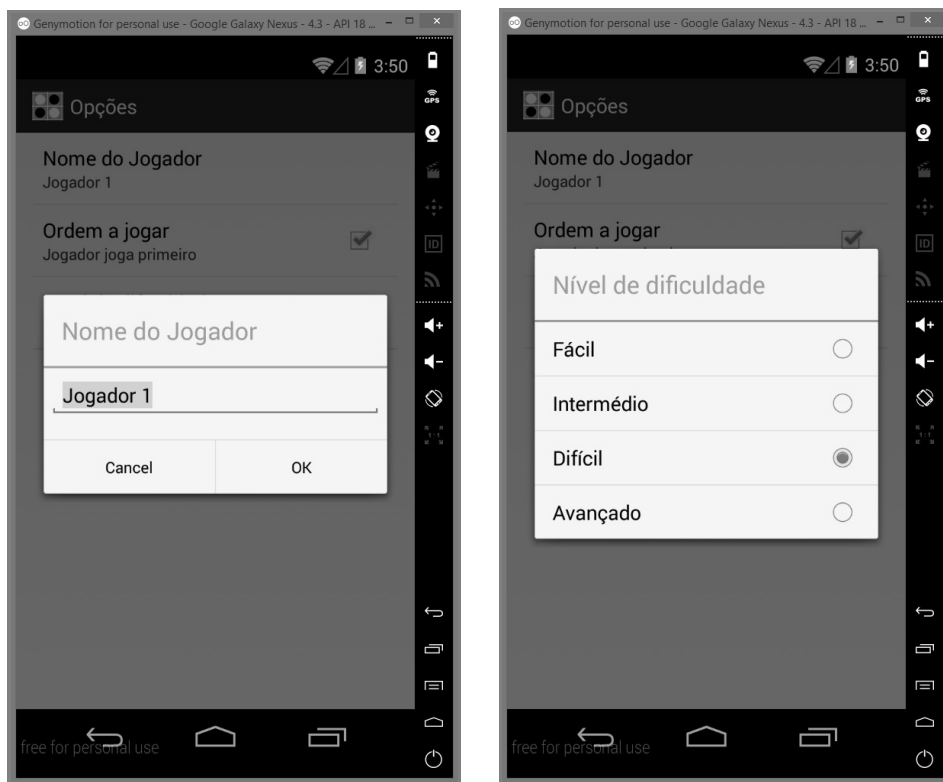


FIGURA 3.12 – Os objetos EditTextPreference e ListPreference

Qualquer ação efetuada no ecrã de configurações vai persistir as alterações no ficheiro de preferências no sistema de ficheiros do dispositivo, mais concretamente na pasta localizada em `/data/data/com.androidgames.Othelloid/shared_prefs` com o nome do ficheiro `com.androidgames.Othelloid_preferences.xml`.

O próximo código mostra o conteúdo do ficheiro **Othelloid_preferences.xml**:

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
  <string name="nivel_dificuldade">4</string>
  <string name="nome_jogador">Ricardo</string>
  <boolean name="ordem_jogar" value="true" />
</map>
```

Em qualquer parte do código da aplicação, é possível obter os valores das opções de configuração. Para obter um objeto `SharedPreferences` usa-se o método `getDefaultSharedPreferences`. Depois, basta usar os métodos `getString` ou `getBoolean`, que recebem como parâmetros os identificadores das opções de configuração (atributo `android:key`) e um valor por omissão. O próximo excerto de código mostra como se afere quem é o primeiro a iniciar o jogo:

```
SharedPreferences sharedPref =
    PreferenceManager.getDefaultSharedPreferences(context);
if (sharedPref.getBoolean("ordem_jogar", true))
    estado = Estado.JOGADOR_A_JOGAR;
else
    estado = Estado.IA_A_JOGAR;
```

4

LIBGDX

Este capítulo introduz a *libgdx*, uma *framework* de desenvolvimento de jogos multi-plataforma. O capítulo é iniciado com uma introdução básica à *framework*, destacando-se a arquitetura de uma aplicação *libgdx* e o seu ciclo de vida. São também explicados os vários módulos da *framework* (*input*, gráficos, ficheiros, áudio e *networking*) que fornecem serviços essenciais para a construção de um videojogo. De seguida, apresentam-se as várias classes do pacote `graphics.g3d` de forma a dar uma visão geral sobre como realizar tarefas básicas com a API 3D desta *framework*. Finalmente, para consolidar todos estes conceitos, o capítulo é finalizado com a implementação de um jogo 3D, o *Balloid*. Para além dos aspetos lúdicos do jogo, o *Balloid* ajuda também a desenvolver a concentração, a destreza e a habilidade motora.

4.1 INTRODUÇÃO À FRAMEWORK LIBGDX

A *libgdx*³⁰ é um motor de jogo gratuito e de código aberto, baseado na linguagem de programação Java, que permite o desenvolvimento de jogos 2D e 3D para Windows, Mac OS X, Linux, Android, iOS, BlackBerry ou HTML5 usando a mesma base de código.

Embora o seu foco principal seja o componente gráfico, inclui também suporte para áudio, gestão de *input*, bibliotecas matemáticas e de simulação física 2D (**Box2D**), bem como alguns outros componentes de utilidade, como o acesso a ficheiros ou a serialização eficiente para os formatos XML e JSON. Para além destes componentes, inclui ainda um editor para a construção de sistemas de partículas, um empacotador de texturas, um gerador de tipos de letra *raster* (*bitmap*) e uma pequena aplicação para a criação de esqueletos de projetos *libgdx* (apresentada na secção 4.1.1).

A *libgdx* também tem integração com o Spine (um animador 2D de esqueletos), o Nextpeer (uma biblioteca para o desenvolvimento de jogos em rede), os otimizadores DexGuard e ProGuard da Saikoa, bem como a biblioteca de simulação física 3D **BulletPhysics**.

³⁰ *Link:* <http://libgdx.badlogicgames.com/>

4.1.1 CRIAÇÃO DE UM PROJETO LIBGDX

A forma mais simples para criar uma aplicação libgdx é usar um gerador de projetos, disponibilizado pela mesma equipa que desenvolve a libgdx. O gerador cria automaticamente um esqueleto base de um projeto libgdx. Os projetos libgdx usam o *Gradle* para várias tarefas: gerir dependências, construir e compilar a aplicação e integrar a mesma num IDE. Esta abordagem permite a independência do IDE e promove o desenvolvimento multidisciplinar.

Para criar um projeto libgdx e o integrar no Android Studio é necessário:

- 1) Aceder à página oficial³¹ da libgdx.
- 2) Clicar na opção **Download Setup App** para descarregar a ferramenta geradora de projetos libgdx (ficheiro JAR).
- 3) Ao fazer duplo clique no ficheiro descarregado surge a janela **Libgdx Project Generator** (Figura 4.1).

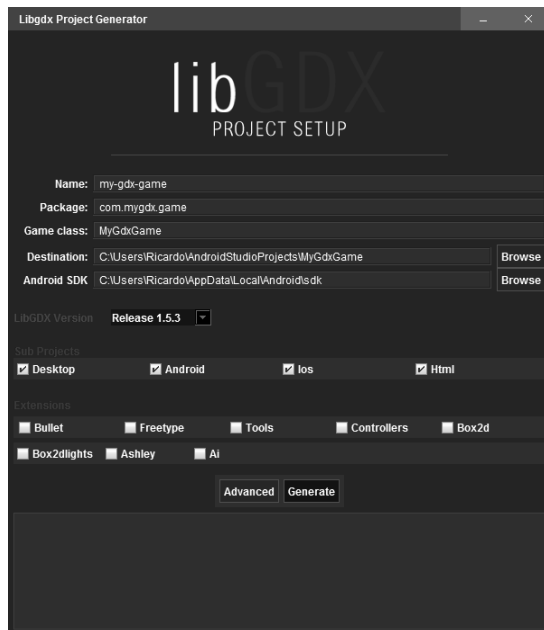


FIGURA 4.1 – Gerador de projetos libgdx



Para executar o ficheiro JAR via linha de comando digite `java -jar gdx-setup.jar`.

³¹ *Link:* <http://libgdx.badlogicgames.com/download.html>

- 4) Especificar o nome da aplicação, o nome do pacote Java, o nome da classe principal para o jogo, a pasta do projeto e o caminho para o SDK Android. De seguida, pode-se selecionar quais as plataformas a suportar. Finalmente, é também possível selecionar extensões para serem incluídas na aplicação. Note que algumas podem não funcionar em todas as plataformas, mas nessa situação aparecerá um aviso. A geração pode ser iniciada usando o botão **Generate**.
- 5) Depois de iniciar o Android Studio, usa-se a opção em **File → Import Project** e seleciona-se o ficheiro **build.gradle** da pasta do projeto gerado.
- 6) Finalmente, a aplicação gerada pode ser compilada e executada num emulador (Figura 4.2).



FIGURA 4.2 – Interface gráfica do projeto libgdx gerado por omissão

O desenvolvimento posterior é realizado diretamente no Android Studio.

4.1.2 ESTRUTURA DE UM PROJETO LIBGDX

Um projeto libgdx é organizado em diferentes subprojetos, em que cada um corresponde a uma das plataformas que o jogo irá suportar. Assim, aparecerá uma pasta para cada uma das plataformas selecionadas, de entre as disponíveis: **Android**, **iOS**, **Desktop** e **Html**. Um quinto subprojeto é criado, chamado **Core**. Este subprojeto contém o código do jogo que é partilhado por todas as plataformas escolhidas. Esta abordagem evita a redundância do código, permitindo que com uma única implementação se consiga disponibilizar o jogo para as diferentes plataformas selecionadas. Assim, é potenciada a modularização e, conseqüentemente, facilitada a manutenção do código.

Segue-se uma visão compacta da estrutura de pastas e ficheiros criada, quando selecionadas as cinco plataformas. Neste exemplo, o nome do projeto criado é `MyGdxGame`.

```
settings.gradle // DEFINIÇÃO DE SUBPROJETOS (CORE, DESKTOP, ANDROID, HTML E iOS)
build.gradle // FICHEIRO GRADLE PRINCIPAL (DEFINE DEPENDÊNCIAS E PLUGINS)
gradlew // SCRIPT DE EXECUÇÃO DO GRADLE EM SISTEMAS UNIX
gradlew.bat // SCRIPT DE EXECUÇÃO DO GRADLE EM SISTEMAS WINDOWS
gradle // WRAPPER DO GRADLE
local.properties // DEFINE A LOCALIZAÇÃO DO SDK DO ANDROID
core/
  build.gradle // FICHEIRO GRADLE DO PROJETO CORE
  src/ // PASTA COM O CÓDIGO-FONTE PARA TODO O CÓDIGO DO JOGO
  src/.../MyGdxGame.java // CLASSE PRINCIPAL DA APLICAÇÃO
android/
  build.gradle // FICHEIRO GRADLE DO SUBPROJETO ANDROID
  AndroidManifest.xml // FICHEIRO DE MANIFESTO
  assets/ // PASTA PARA OS ASSETS (IMAGENS, ÁUDIO, ETC.) DE TODOS OS SUBPROJETOS
  res/ // PASTA PARA OS ÍCONES DA APLICAÇÃO E OUTROS RECURSOS
  src/ // PASTA COM O CÓDIGO-FONTE PARA O SUBPROJETO ANDROID
  src/.../AndroidLauncher.java // CLASSE DE ARRANQUE ANDROID
desktop/...
  src/.../DesktopLauncher.java // CLASSE DE ARRANQUE DESKTOP
html/...
  src/.../HtmlLauncher.java // CLASSE DE ARRANQUE HTML
ios/...
  src/.../IOSLauncher.java // CLASSE DE ARRANQUE iOS
```

A classe principal da aplicação é `MyGdxGame`, na pasta **Core**. É aqui que o programador se vai concentrar na implementação do jogo. No entanto, existe uma classe principal em cada subprojeto. Estas classes preparam a aplicação para a plataforma de destino, e executam, a partir daí, a classe de jogo. A estas classes, específicas para cada uma das plataformas, chamar-se-á a partir de agora **classes de arranque**.



Todos os *assets* da aplicação (como imagens, modelos tridimensionais ou sons) devem estar armazenados na pasta **assets** do subprojeto **Android**. Os subprojetos **Desktop**, **iOS** e **Html** ligam-se diretamente a esta pasta, pelo que não existe necessidade de armazenar múltiplas cópias.

De forma a fornecer o suporte multiplataforma às suas aplicações, a `libgdx` inclui um conjunto de bibliotecas para cada plataforma designado por *backends*. Um *backend* mapeia as invocações a funcionalidades da biblioteca `libgdx` nas bibliotecas nativas da plataforma em causa. A `libgdx` atualmente fornece os seguintes *backends*:

- ⊙ **Android** – usa o Android SDK, via OpenGL ES, como *backend*;
- ⊙ **Lightweight Java Game Library (LWJGL)**³² – é uma biblioteca Java, de código-fonte aberto, para facilitar o desenvolvimento de jogos em termos de

³² Link: <http://www.lwjgl.org>

acesso aos recursos de *hardware* em sistemas *desktop*. Neste contexto, a LWJGL é usada como *backend* para suportar os principais sistemas operativos *desktop* (Windows, Linux e Mac OS X);

- ⊙ **JavaScript/WebGL** – este *backend* usa o *Google Web Toolkit* (GWT)³³ para traduzir o código Java em JavaScript e API relacionadas, como, por exemplo, o *SoundManager 2*³⁴, de forma a funcionar como *backend* combinado para HTML5, WebGL³⁵ e reprodução de áudio;
- ⊙ **iOS/RoboVM** – o projeto de código aberto RoboVM³⁶ permite executar o Java e outras linguagens baseadas na *Java Virtual Machine* em plataformas iOS. O compilador de RoboVM traduz o *bytecode* de Java em código máquina (ARM ou x86) que é executado diretamente na CPU destino, sem ser interpretado.

4.1.3 MÓDULOS

Sendo uma *framework* para o desenvolvimento de jogos, é natural que se espere um conjunto de funcionalidades tipicamente encontradas num videojogo, tais como desenhar **gráficos** no ecrã, reproduzir **áudio**, gerir o *input* do utilizador, ler e gravar dados em **ficheiros** e gerir a comunicação (*networking*).

A *libgdx* disponibiliza um conjunto de **módulos** que cobrem a lista de requisitos acima descrita. Estes módulos fornecem meios comuns, através de uma API uniforme, para interagir com o sistema operativo onde o jogo será executado. Para cada módulo existe uma ou mais **interfaces** Java que são implementadas para cada plataforma. Para o programador é irrelevante a plataforma na qual está a desenvolver, já que aquele apenas interage com as interfaces públicas que cada plataforma implementa.



Todos os módulos da *libgdx*, materializados em interfaces, podem ser acedidos através de campos estáticos da classe *Gdx*.

As próximas secções detalham os módulos que compõem a *framework libgdx*, mais concretamente: *Application*, *Graphics*, *Audio*, *Input*, *Files* e *Network*.

³³ *Link*: <https://developers.google.com/web-toolkit>

³⁴ *Link*: <http://www.schillmania.com/projects/soundmanager2>

³⁵ *Link*: <http://www.khronos.org/webgl>

³⁶ *Link*: <http://www.robvm.com>

4.1.3.1 MÓDULO `Application`

O módulo `Application` fornece métodos para geração de relatórios para *debug* (*logging*), persistência de dados, consulta à plataforma e ao sistema operativo (versão da API Android, tipo de plataforma e uso de memória) e gestão de *threads*.

LOGGING

A `libgdx` suporta a geração de relatórios, ou *logging*. Existem diferentes tipos de mensagens: de erro (que indicam que algo inesperado aconteceu), de informação (que indicam que algo esperado aconteceu) e de *debug* (usadas durante o desenvolvimento para depurar a aplicação). Estas mensagens são geradas com métodos da classe `Gdx.app`, tais como os métodos `log`, `error` ou `debug`. Confira o próximo exemplo:

```
Gdx.app.debug("BOTÃO", "Botão INICIAR pressionado!");
```

Os métodos de *logging* têm dois parâmetros: uma *string* que indica o componente do sistema a partir do qual a mensagem se origina e o texto a ser impresso na consola.



Dependendo da plataforma, a mensagem de *log* é visualizada na consola (**Desktop**), no *LogCat* (**Android**) ou através da classe `GwtApplicationConfiguration` (**Html**).

De modo a que se possa suprimir algumas destas mensagens, sem necessidade de remover manualmente estas instruções, pode-se filtrar os tipos de mensagem a imprimir na consola. Esta filtragem é baseada em níveis de *log* e é realizada através do método `setLogLevel`, como demonstra o próximo código:

```
Gdx.app.setLogLevel(Application.LOG_DEBUG);
```

Os níveis de *log* disponíveis são enumerados na Tabela 4.1.

NÍVEIS DE LOG	DESCRIÇÃO
LOG_NONE	Não imprime <i>logs</i> .
LOG_ERROR	Imprime apenas <i>logs</i> de erro.
LOG_INFO	Imprime <i>logs</i> de erro e informação (nível por omissão).
LOG_DEBUG	Imprime <i>logs</i> de erro, informação e <i>debug</i> .

TABELA 4.1 – Níveis de *logging* na `libgdx`

PERSISTÊNCIA DE DADOS

Para garantir a persistência de dados após o término de execução da aplicação é possível usar a classe `Preferences`. Esta classe providencia um dicionário (*hashmap*) que armazena pares de chave-valor num ficheiro. Ao criar uma instância desta classe, e caso não exista, a `libgdx` cria dinamicamente um novo ficheiro. É possível manter mais do que

um ficheiro de preferências num jogo, bastando para isso usarem-se nomes de ficheiro diferentes. A criação de um novo ficheiro de preferências, ou a abertura de um já existente, é realizada usando o método `getPreferences` e passando o nome de ficheiro como argumento:

```
Preferences prefs = Gdx.app.getPreferences("config.prefs");
```

Para escrever um novo valor é necessário decidir qual o tipo de dados do valor em causa (por exemplo, se corresponde a um valor inteiro ou real) e escolher o nome da chave à qual o valor será associado. Se essa chave já constar desse ficheiro de preferências, o valor será substituído. Não esquecer que, por questões de eficiência, o conteúdo da classe de preferências só é armazenado quando se invoca o método `flush`. Por exemplo, para armazenar um valor inteiro, usa-se o método `putInteger`:

```
prefs.putInteger("volume_som", 100);  
prefs.flush();
```

Para ler um determinado valor a partir de uma classe de preferências é necessário saber a chave à qual o valor está associado, bem como o seu tipo. Se a chave indicada não existir, será usado o valor por omissão do tipo correspondente. Outra hipótese é indicar como segundo argumento do método o valor que deverá ser usado por omissão. Por exemplo, para obter o valor correspondente à chave `volume_som` garantindo que o seu valor será 50, caso não exista outro valor definido nas preferências:

```
int volumeSom = prefs.getInteger("volume_som", 50);
```

QUERYING

O módulo `Application` permite consultar a plataforma em execução, como, por exemplo, saber a versão do sistema operativo, o tipo de plataforma ou o uso de memória.

Para obter a versão do sistema operativo usa-se o método `getVersion`. Para a plataforma Android, é devolvida a versão da API (correspondente à versão do sistema operativo Android). Este valor permite que o utilizador trate de forma especial as versões do Android que não incluam algumas das funcionalidades desejadas. Para outras plataformas, que não o Android, o valor devolvido por este método é 0.

Também é possível escrever código diferente para cada plataforma, no sentido de a suportar devidamente. Para isso usa-se o método `getType`, que devolve um valor que corresponde à plataforma em execução. O exemplo seguinte mostra o seu uso:

```
switch (Gdx.app.getType()) {  
    // CÓDIGO PARA A APLICAÇÃO DESKTOP  
    case Desktop: break;  
    // CÓDIGO PARA A APLICAÇÃO ANDROID  
    case Android: break;  
    // CÓDIGO PARA A APLICAÇÃO WebGL  
    case WebGL: break;
```

```
// CÓDIGO PARA A APLICAÇÃO IOS
case ios: break;
// CÓDIGO PARA OUTRO TIPO DE APLICAÇÕES
default: break;
}
```

Finalmente, é possível obter detalhes sobre o uso atual de memória, de forma a que o programador possa detetar alocações de memória excessivas, ou que possa libertar alguns recursos caso o dispositivo esteja sobrecarregado. Segue-se um exemplo de como obter a quantidade de memória (em *bytes*) que está em uso pelo *heap* correspondente:

```
long memUsageJavaHeap = Gdx.app.getJavaHeap();
long memUsageNativeHeap = Gdx.app.getNativeHeap();
```

MULTITHREADING

Quando o jogo é criado, a *libgdx* cria uma *thread* separada denominada *main loop thread* e o contexto do OpenGL é a ela ligado. Todo o processamento de eventos ou de renderização acontece nesta *thread* e não na *thread* da interface.

É possível executar código específico na *thread* de renderização a partir de uma outra *thread* usando-se o método `Application.postRunnable`. O código será executado na *thread* de renderização na próxima *frame*. A estrutura habitual é mostrada no próximo código:

```
Gdx.app.postRunnable(new Runnable() {
    @Override
    public void run() {
        // CÓDIGO A SER EXECUTADO
    }
});
```

4.1.3.2 MÓDULO Graphics

A interface `Graphics` abstrai a complexidade inerente à programação gráfica usando a API OpenGL ES. Mais concretamente, a interface abstrai a comunicação com a unidade de processamento gráfico (GPU), fornecendo métodos de conveniência para obter instâncias de *wrappers* OpenGL ES, independentes da plataforma.

Esta interface inclui também um conjunto de métodos para obter informações sobre o ecrã, como a resolução, a densidade (número de pixels por polegada), a orientação, as implementações OpenGL disponíveis e o tempo de renderização entre *frames*.

A Tabela 4.2 apresenta alguns dos métodos mais usados da interface.

MÉTODOS	DESCRIÇÃO
<code>getDeltaTime</code>	Obtém o intervalo de tempo entre a atual e a última <i>frames</i> , em segundos.
<code>getWidth/getHeight</code>	Obtém o tamanho do ecrã do dispositivo em pixels.
<code>getFramesPerSecond</code>	Obtém o número médio de <i>frames</i> por segundo.
<code>getGL20/getGL30</code>	Obtém uma instância OpenGL 2.0/3.0.
<code>setTitle(String title)</code>	Define o título da janela.
<code>isFullScreen</code>	Verifica se a aplicação está em ecrã inteiro ou não.

TABELA 4.2 – Métodos da interface `Graphics`

Por exemplo, para obter uma instância OpenGL 2.0 usa-se o seguinte código:

```
GL20 gl = Gdx.graphics.getGL20();
```

O método devolve uma instância que pode ser usada para desenhar no ecrã sem que o programador precise de saber detalhes relativos aos diferentes *backends* suportados pela `libgdx`. O trecho de código seguinte pinta o ecrã de vermelho:

```
gl.glClearColor(1f, 0.0f, 0.0f, 1);
gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
```

4.1.3.3 MÓDULO `Audio`

A interface `Audio` facilita a criação e reprodução de ficheiros de áudio. São suportados dois tipos de som – *music* e *sound* – que são usados em situações distintas. Os tipos de ficheiros suportados são WAV, MP3 e OGG.

As **instâncias de som** (*sound*) são carregadas em memória e podem ser reproduzidas em qualquer altura. São ideais para efeitos de som que se repetem várias vezes durante um jogo, como, por exemplo, explosões ou tiros.

As **instâncias de música** (*music*), por outro lado, são fluxos de dados (*streams*) de ficheiros no disco (ou cartão SD). Cada vez que um ficheiro é executado, é feita a transmissão (*streaming*) do ficheiro para o dispositivo de áudio. Isto permite que uma música longa possa ser reproduzida sem que seja necessário carregar todo o seu conteúdo para memória.

Para que se possam carregar sons ou músicas para reprodução usam-se os métodos `Gdx.audio.newSound` e `Gdx.audio.newMusic`, respetivamente. Ambos os métodos recebem como argumento um objeto `FileHandle` ligado ao ficheiro que inclui o som ou música a ser reproduzido.

O trecho de código seguinte executa repetidamente, a partir do disco, um ficheiro de som chamado **minhaMusica.mp3**, usando um volume médio:

```
Music musica = Gdx.audio.newMusic(  
    Gdx.files.getFileHandle("data/minhaMusica.mp3", FileType.Internal));  
musica.setVolume(0.5f);  
musica.play();  
musica.setLooping(true);
```



Existe um limite máximo de 1 MB para os dados de áudio descodificados (no caso dos sons). Daí que a funcionalidade para reprodução de sons deva ser usada apenas em pequenos trechos de som, de modo a que a limitação de tamanho não seja um problema.

Para maior controlo sobre o dispositivo de áudio existem as classes `AudioDevice` e `AudioRecorder`, que podem ser obtidas através da interface `audio`. Estas classes permitem executar amostras de som *Pulse-Code Modulation* (PCM) no dispositivo de áudio, bem como gravar amostras a partir de um microfone.

4.1.3.4 MÓDULO `Input`

As várias plataformas suportadas pela `libgdx` têm diferentes métodos de *input*. Em sistemas *desktop*, os utilizadores interagem com a aplicação através do teclado ou do rato. No Android, o rato³⁷ é substituído por um ecrã sensível ao toque e o teclado físico é muitas vezes inexistente. Os dispositivos Android incluem ainda um conjunto de sensores, como sejam o acelerómetro e uma bússola (sensor de campo magnético).

A interface `Input` abstrai todos estes métodos de entrada. Por exemplo, os eventos associados ao rato e ao toque em ecrãs *touchscreen* são tratados como sendo a mesma coisa.

Para receber e lidar com o *input* corretamente é necessário criar uma classe que implemente a interface `InputProcessor`. Um `InputProcessor` é utilizado para receber eventos de entrada do teclado e toque em ecrã tátil (rato na plataforma *desktop*). Para isso, uma instância dessa classe tem que ser registada, usando-se o método `Input.setInputProcessor(InputProcessor)`. O `InputProcessor` será executado em cada *frame* antes da invocação do método `ApplicationListener.render`. Um exemplo de implementação de um `InputProcessor` será apresentado na implementação do jogo deste capítulo, o *Balloid*.

³⁷ Note que alguns dispositivos, como os Samsung Note, que incluem um *stillus*, o toque deste no ecrã é considerado, pelo sistema Android, como sendo um rato, pelo que muitas vezes é necessário programar o comportamento quer para o toque, quer para o rato. Um exemplo concreto será apresentado no Capítulo 5.

A Tabela 4.3 apresenta alguns dos métodos mais usados da interface.

MÉTODOS	DESCRIÇÃO
<code>isTouched</code>	Verifica se o ecrã é tocado por um dedo ou com o rato.
<code>isButtonPressed</code>	Verifica se o botão do rato é pressionado.
<code>isKeyPressed</code>	Verifica se uma tecla do teclado é pressionada.
<code>getAccelerometerX/Y/Z</code>	Obtém o valor do acelerómetro no eixo do x, y, z.
<code>Vibrate/cancelVibrate</code>	Ativa/cancela a vibração do dispositivo.
<code>setCatchBackKey(boolean)</code> <code>setCatchMenuKey(boolean)</code>	Lida com as teclas virtuais (<i>back/menu</i>) do Android.

TABELA 4.3 – Métodos da interface Input

4.1.3.5 MÓDULO Files

A interface `Files` fornece uma forma genérica para aceder a ficheiros, independentemente da plataforma, facilitando o processo de leitura/escrita de ficheiros. Um caso típico é o carregamento de *assets* de um jogo (texturas, ficheiros de som) para memória a partir de uma pasta. O próximo exemplo demonstra a criação de uma instância de uma textura, lendo-a de um ficheiro:

```
Texture myTexture = new
    Texture(Gdx.files.internal("assets/texture/brick.png"));
```

Existem dois tipos de armazenamento: **interno** ou **externo** ao dispositivo.

Cada aplicação instalada tem uma pasta dedicada para o **armazenamento interno**. O acesso a esta pasta é limitado a essa mesma aplicação. Pode-se pensar neste tipo de armazenamento como uma área de trabalho privada. Para se obter um identificador de um ficheiro armazenado nesta área privada usa-se o método `Gdx.files.internal(String path)`, que retorna um `FileHandle`. Um ficheiro interno é relativo à pasta **assets** nas plataformas Android e WebGL. Na plataforma *desktop* é relativo à pasta raiz da aplicação.

Por sua vez, o **armazenamento externo** (por exemplo, num cartão SD) pode não estar sempre disponível porque, por exemplo, o utilizador poderá remover o dispositivo de memória externo. Nesse sentido, os ficheiros para armazenamento externo devem ser considerados voláteis. Para obter um identificador de um ficheiro de um dispositivo externo usa-se o método `Gdx.files.external(String path)`, que devolve um `FileHandle`. Um ficheiro externo é relativo ao cartão SD para a plataforma Android. Na plataforma *desktop* é relativo à pasta **home** do utilizador.

De seguida apresentam-se alguns exemplos de tipos de ficheiros e qual o tipo de armazenamento aconselhável:

- **Ficheiros internos** – todos os *assets* (imagens, ficheiros de áudio, etc.) que são fornecidos com a aplicação são ficheiros internos. Ao usar o gerador de projetos libgdx, basta armazená-los na pasta **assets** do projeto;
- **Ficheiros locais** – se necessitar de escrever ficheiros pequenos, por exemplo, gravar um estado do jogo, use ficheiros locais. Estes são, em geral, privados para a aplicação. Por outro lado, se quiser um armazenamento do tipo chave-valor, opte por usar ficheiros de preferências;
- **Ficheiros externos** – se necessitar de gravar ficheiros grandes, por exemplo, *screenshots*, ou ficheiros descarregados a partir da Web, estes devem ser gravados no armazenamento externo. Note-se, mais uma vez, que o armazenamento externo é volátil, podendo um utilizador removê-lo ou apagar os ficheiros em causa.

4.1.3.6 MÓDULO `Network`

A interface `Net` fornece métodos para executar operações de rede, como, por exemplo, pedidos HTTP (GET, POST) e comunicação cliente/servidor TCP através de *sockets*. A interface é acedida através do método `Gdx.getNet` ou com a variável `Gdx.net`.

Para executar um pedido HTTP use o método `sendHttpRequest (HttpRequest, HttpResponseListener)`. O primeiro argumento é o endereço usado no pedido, e o segundo argumento uma instância de `HttpResponseListener` responsável por processar o resultado desse pedido. Os resultados dos pedidos HTTP são do tipo `HttpResponse` e incluem a informação retornada e o código HTTP de erro ou sucesso. A Tabela 4.4 apresenta alguns dos métodos mais usados desta interface.

MÉTODOS	DESCRIÇÃO
<code>sendHttpRequest (HttpRequest req, HttpResponseListener listener)</code>	Faz um pedido HTTP, isto é, processa o <code>HttpRequest</code> e reporta o <code>HttpResponse</code> para o <code>HttpResponseListener</code> .
<code>cancelHttpRequest (HttpRequest req)</code>	Cancela um pedido HTTP.
<code>newClientSocket (Protocol protocol, String host, int port, SocketHints hints)</code>	Cria um novo <i>socket</i> cliente TCP que se conecta ao <i>host</i> e porta respetiva.
<code>newServerSocket (Protocol protocol, int port, ServerSocketHints hints)</code>	Cria um novo <i>socket</i> servidor na porta respetiva, usando o <code>Protocol</code> dado, e espera por conexões de entrada.
<code>openURI (String URI)</code>	Inicia o navegador Web para exibir uma URI.

TABELA 4.4 – Métodos da interface `Net`

O uso de *sockets* TCP, para comunicar com um servidor remoto, é feito através da invocação do método `newClientSocket`. O *socket* devolvido disponibiliza dois canais de comunicação, o `InputStream` e o `OutputStream`.

Para criar um *socket* servidor TCP, que espere por conexões de entrada, invoca-se o método `newServerSocket`. O `ServerSocket` devolvido oferece o método `accept`, que espera por uma conexão de entrada.

Os detalhes de implementação de clientes ou servidores TCP escapam ao âmbito deste livro, pelo que sugerimos a leitura da documentação da `libgdx` e do Java.

4.1.4 CICLO DE VIDA DE UMA APLICAÇÃO LIBGDX

Habitualmente, as *frameworks* para desenvolvimento de jogos disponibilizam um ciclo principal para controlar o jogo. Numa aplicação `libgdx` emprega-se o conceito baseado em eventos. Sendo assim, a lógica da aplicação é implementada na interface `ApplicationListener`, que tem métodos que são automaticamente invocados quando a aplicação é criada, pausada, retomada, renderizada ou destruída.

O próximo trecho de código mostra o esqueleto de código da classe `Jogo` que implementa a interface `ApplicationListener`:

```
public class Jogo implements ApplicationListener {
    @Override
    public void create() {...}
    @Override
    public void render() {...}
    @Override
    public void resize(int width, int height) {...}
    @Override
    public void pause() {...}
    @Override
    public void resume() {...}
    @Override
    public void dispose() {...}
}
```

Tudo o que é preciso fazer é implementar estes métodos na classe principal do subprojeto **Core** (projeto partilhado do jogo). A `libgdx` chamará cada um destes métodos quando necessários. A Figura 4.3 apresenta o ciclo de vida de uma aplicação `libgdx`.



Uma aplicação `libgdx` é, por natureza, baseada em eventos. Não existe um ciclo principal explícito, no entanto, o método `ApplicationListener.render()` pode ser considerado o corpo desse ciclo principal.

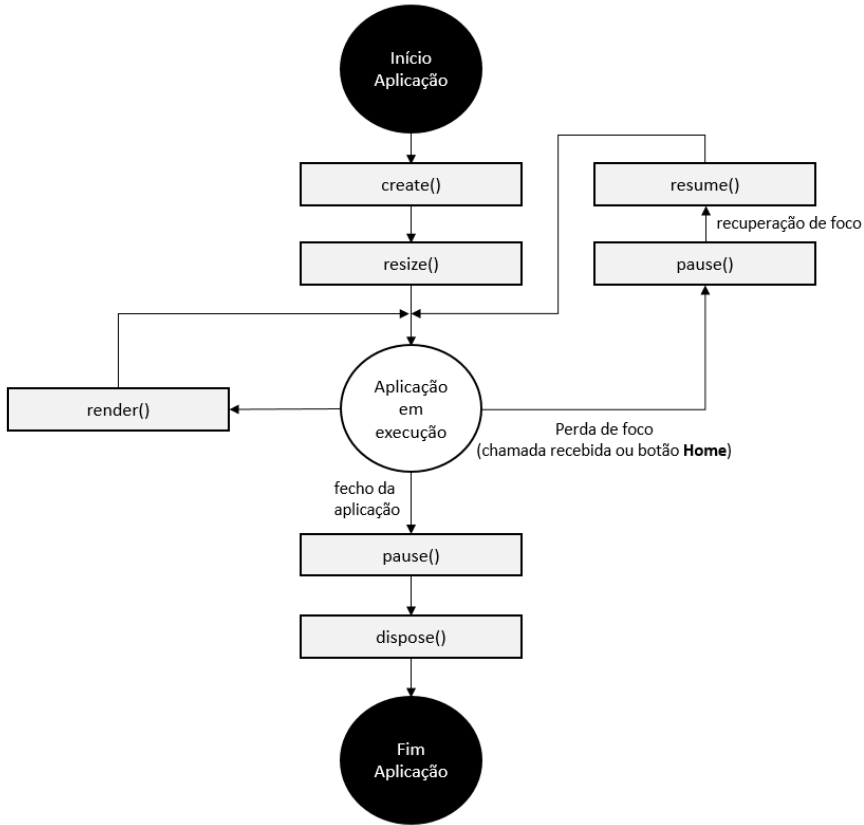


FIGURA 4.3 – Ciclo de vida de uma aplicação libgdx

O método `create` é chamado quando a aplicação é iniciada, sendo o local ideal para criar os objetos utilizados na aplicação. Logo a seguir é chamado o método `resize`. Este método é invocado sempre que o ecrã de jogo é redimensionado e o jogo não está no estado de pausa. Os parâmetros são a nova largura e altura do ecrã. Enquanto a aplicação é executada, a libgdx vai chamar regularmente o método `render` (a cada *frame*), que é o ciclo principal da aplicação e onde todo o processo de renderização é feito. As atualizações à lógica do jogo são geralmente executadas neste método.

O método `pause` é chamado sempre que o botão **Home** é pressionado ou que uma chamada é recebida. Neste método deve-se guardar o estado do jogo. A partir do estado de pausa, se a aplicação é retomada, é chamado o método `resume`. Finalmente, o método `dispose` é chamado quando a aplicação é destruída, sendo onde todos os objetos criados devem ser destruídos. A invocação deste método é precedida por uma chamada ao método `pause`.

4.1.5 CLASSES DE ARRANQUE

Para cada plataforma (Android, iOS, *desktop*, etc.) é necessário criar uma classe denominada **classe de arranque**.



Por omissão, o gerador de projetos libgdx já faz esse trabalho por nós.

As classes de arranque definem o ponto de entrada de uma aplicação libgdx. Para cada plataforma-alvo, um pedaço de código irá instanciar uma implementação concreta da interface `Application`, fornecido pelo *backend* da plataforma.



Um objeto `Application` pode ser uma instância de uma aplicação *desktop*, Android, iOS ou HTML5. Cada classe `Application` tem os seus próprios métodos de arranque e inicialização.

Por exemplo, para sistemas *desktop* usa-se o *backend* LWJGL do seguinte modo:

```
public class DesktopStarter {
    public static void main(String[] argv) {
        LwjglApplicationConfiguration cfg = new LwjglApplicationConfiguration();
        new LwjglApplication(new MyGame(), cfg);
    }
}
```

Para a plataforma Android, a classe de arranque correspondente é a seguinte:

```
public class AndroidStarter extends AndroidApplication {
    public void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        AndroidApplicationConfiguration cfg = new
            AndroidApplicationConfiguration();
        initialize(new MyGame(), cfg);
    }
}
```

Estas duas classes são automaticamente criadas em subprojetos separados (**Desktop** e **Android**) do projeto libgdx.

Na secção anterior foi criada a classe `Jogo` que implementava a interface `ApplicationListener`. Uma instância da classe `Jogo` é passada para os respetivos métodos de inicialização. Assim, a libgdx permite que uma única implementação possa ser usada pelas diferentes plataformas. O próximo diagrama UML (Figura 4.4) apresenta a interface `Application` e as suas relações com as outras classes.

O `ApplicationListener` pode ser fornecido para qualquer implementação de `Application`. Isso significa que só é necessário escrever a lógica do programa uma vez e executá-la em diferentes plataformas, passando-a para uma implementação concreta de um `Application`.

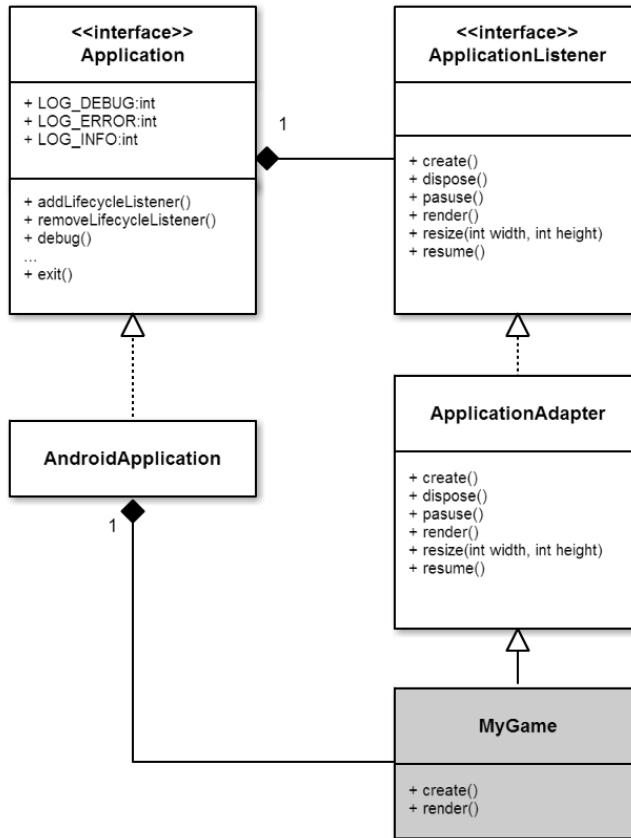


FIGURA 4.4 – Diagrama de classes UML representando a interface `Application`



No caso de nem todos os métodos serem relevantes pode-se derivar da classe `ApplicationAdapter`.

4.1.6 API 3D

Esta secção dá uma visão geral sobre a API 3D da `libgdx`. A API inclui um conjunto de classes e interfaces agrupadas no pacote `com.badlogic.gdx.graphics.g3d`.

4.1.6.1 CLASSES `Model` e `ModelInstance`

A classe `Model` representa um modelo tridimensional composto por uma hierarquia de nós, em que cada nó é uma combinação de uma geometria (malha) e um material. Uma instância desta classe não se destina a ser diretamente renderizada. Em vez disso, criam-se uma ou mais instâncias do modelo que serão utilizadas para a

renderização. Isto permite que se possam renderizar vários objetos usando uma única representação em memória do modelo tridimensional.

A classe `ModelInstance` representa uma instância de um modelo. Através de uma instância desta classe é possível definir transformações sobre o objeto e modificar os materiais. Várias instâncias podem ser criadas a partir do mesmo modelo, todos partilhando as malhas e texturas originais.

É possível realizar transformações de translação, rotação ou escala a uma instância. Esta transformação pode ser realizada durante o carregamento do modelo para memória, ou dinamicamente.

4.1.6.2 CLASSE `ModelBatch`

A classe `ModelBatch` faz a gestão da renderização das instâncias dos modelos. A classe abstrai todo o código típico associado à renderização, permitindo que o programador se concentre na lógica do jogo. Tipicamente, instancia-se um objeto `ModelBatch` no método `create` da aplicação através do seu construtor:

```
ModelBatch modelBatch;
...
@Override
public void create () {
    modelBatch = new ModelBatch();
    ...
}
@Override
public void render () {
    ...
    modelBatch.begin(camera);
    modelBatch.render(...);
    ... // ADICIONAR OUTRAS CHAMADAS PARA RENDERIZAÇÃO
    modelBatch.end();
    ...
}
```

Depois, usa-se o objeto `modelBatch` para iniciar o processo de renderização. A renderização deve ser feita em cada *frame* através do método `render`. Para começar a renderização usa-se o método `begin` passando uma câmara de projeção. De seguida, usa-se o método `render` passando a(s) instância(s) dos modelos a ser(em) desenhada(s). Quando todos os objetos tiverem sido desenhados com invocações sucessivas a este método, invoca-se o método `end`. Só então será feita, realmente, a renderização. Isto permite que a `libgdx` possa realizar otimizações.

Se, por alguma razão, for necessário forçar a renderização antes da invocação do método `end`, então deve-se usar o método `flush`. Por fim, é necessário destruir o objeto através do seu método `dispose` no método `dispose` da aplicação:

```
@Override
public void dispose () {
    modelBatch.dispose();
    ...
}
```

4.1.6.3 CLASSE `ModelBuilder`

A classe `ModelBuilder` cria um ou mais modelos a partir do código. Permite que se criem modelos com base num conjunto de modelos simples. Para iniciar a criação usa-se o método `begin`; quando terminar usa-se o método `end`, que devolve o modelo construído. No entanto, para criar modelos predefinidos sugere-se o uso dos métodos do género `create[forma]`. O próximo exemplo (Figura 4.5) mostra como criar um cubo através do método `createBox`:

```
ModelBuilder modelBuilder = new ModelBuilder();
model = modelBuilder.createBox(5f, 5f, 5f,
    new Material(ColorAttribute.createDiffuse(Color.GREEN)),
    Usage.Position | Usage.Normal);
```

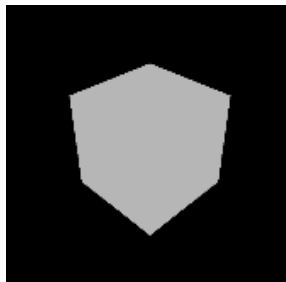


FIGURA 4.5 – Criação de um cubo através da classe `ModelBuilder`

O exemplo anterior começa por instanciar um `ModelBuilder` e usa o seu método `createBox` para criar um modelo (objeto `Model`) com um único nó contendo a forma de um cubo. Este método tem a seguinte assinatura:

```
Model createBox(float width, float height, float depth,
    Material material, long attributes)
```

Os primeiros parâmetros definem a largura, a altura e a profundidade do cubo (neste caso, 5x5x5). Depois, é adicionado um material com uma cor difusa³⁸ verde. Adicionam-se ainda atributos, sendo necessário, pelo menos, o atributo `Usage.Position`. O atributo `Usage.Normal` acrescenta normais ao cubo, por isso, e por exemplo, a iluminação funcionará corretamente. A classe `Usage` é subclasse de `VertexAttributes`.

³⁸ O material difuso indica que a luz deverá ser refletida de forma difusa, como um objeto comum.

4.1.6.4 CLASSE Camera

A classe `Camera` funciona como uma câmara no mundo real. O seu uso é bastante simples, sendo possível, por exemplo, mover, rodar ou fazer *zoom in/out* em torno de um jogo, sem que seja necessário manipular diretamente as matrizes de projeção que estão escondidas na implementação.

A classe `Camera` é a classe base das classes `OrthographicCamera` e `PerspectiveCamera`. A primeira tem uma projeção ortográfica, enquanto a segunda projeta mediante uma perspetiva baseada no campo de visão e do tamanho da janela. A principal diferença (Figura 4.6) é que a câmara de perspetiva usa ativamente a terceira dimensão para criar o efeito de profundidade, enquanto a câmara ortográfica projeta tudo no ecrã numa linha reta. A terceira dimensão pode ser usada para mover as coisas umas para trás das outras, mas não há nenhum efeito de profundidade. Este tipo de técnica é usado para desenvolver jogos a duas dimensões com objetos tridimensionais. Por exemplo, dois objetos podem estar separados por 500 pixels no eixo Z e mesmo assim ter o mesmo tamanho no ecrã.

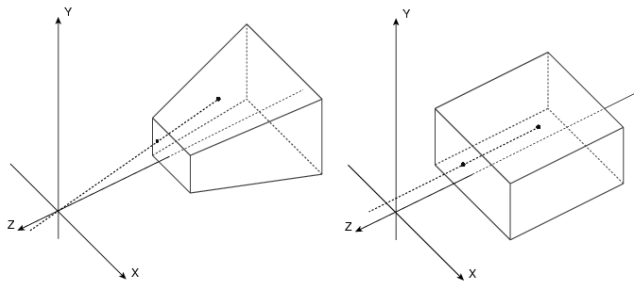


FIGURA 4.6 – Projeção por perspetiva e ortográfica

Segue-se um exemplo do uso de uma câmara com projeção por perspetiva:

```
PerspectiveCamera cam = new PerspectiveCamera(67,
    Gdx.graphics.getWidth(), Gdx.graphics.getHeight());
cam.position.set(10f, 10f, 10f);
cam.lookAt(0, 0, 0);
cam.near = 1f;
cam.far = 300f;
cam.update();
```

Começa-se por instanciar um objeto `PerspectiveCamera` com um campo de visão de 67 graus (valor comum) e o tamanho da janela com a largura e a altura do ecrã do dispositivo. De seguida, define-se a posição da câmara e põe-se esta a apontar para as coordenadas (0,0,0). Definem-se também as propriedades `far` e `near`. Estas duas dimensões indicam que os objetos que estejam mais distantes do que 300 unidades da câmara não serão desenhados, e que os objetos que estejam mais próximos do que 1 unidade da câmara também não o serão. Finalmente, atualiza-se a câmara com todas as alterações feitas.

4.1.6.5 A MINHA PRIMEIRA APLICAÇÃO 3D EM LIBGDX

Nesta secção apresenta-se uma aplicação básica libgdx, que consiste no desenho de um cubo e onde se tratam alguns aspetos relacionados com a sua renderização e perspetivas do modelo através da rotação interativa de uma câmara (Figura 4.7).

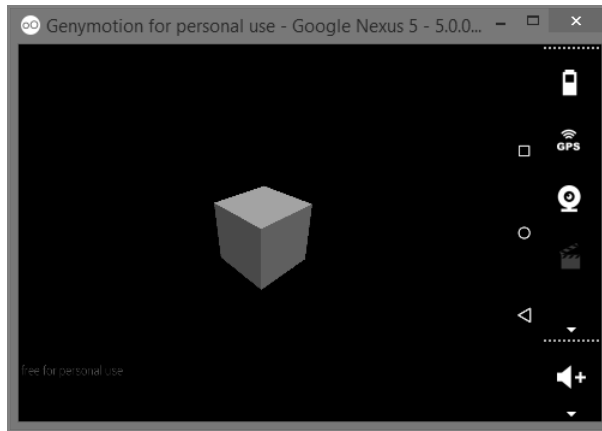


FIGURA 4.7 – Uma aplicação libgdx básica

Depois de se criar um novo projeto através do gerador de projetos libgdx, usando o nome da aplicação **MeuCubo**, importa-se o projeto no Android Studio. De seguida, inclui-se o código base da aplicação no ficheiro **MeuCubo.java** da pasta **core/src**:

```
public class MeuCubo implements ApplicationListener {

    public Environment environment;
    public PerspectiveCamera cam;
    public CameraInputController camController;
    public ModelBatch modelBatch;
    public Model model;
    public ModelInstance instance;

    @Override
    public void create() {
        // CRIAÇÃO DO MODEL BATCH QUE FARÁ A GESTÃO DE RENDERIZAÇÃO DOS MODELOS
        modelBatch = new ModelBatch();

        // CRIAÇÃO DE CÂMARA E POSICIONAMENTO TENDO EM CONTA O MODELO
        cam = new PerspectiveCamera(67,
            Gdx.graphics.getWidth(), Gdx.graphics.getHeight());
        cam.position.set(10f, 10f, 10f);
        cam.lookAt(0,0,0);
        cam.near = 1f;
        cam.far = 300f;
        cam.update();

        // CRIAÇÃO DO CONTROLADOR DE ENTRADA DE CÂMARA GENÉRICO PARA TORNAR A APLICAÇÃO INTERATIVA
        camController = new CameraInputController(cam);
    }
}
```

```
Gdx.input.setInputProcessor(camController);

// CRIAÇÃO DO MODELO (CUBO) E GERAÇÃO DE UMA INSTÂNCIA DESSE MODELO
ModelBuilder modelBuilder = new ModelBuilder();
model = modelBuilder.createBox(5f, 5f, 5f,
    new Material(ColorAttribute.createDiffuse(Color.GREEN)),
    Usage.Position | Usage.Normal);
instance = new ModelInstance(model);

// CONFIGURAÇÃO DO AMBIENTE COM ILUMINAÇÃO SIMPLES
environment = new Environment();
environment.set(new ColorAttribute(
    ColorAttribute.AmbientLight, 0.4f, 0.4f, 0.4f, 1f));
environment.add(
    new DirectionalLight().set(0.8f, 0.8f, 0.8f, -1f, -0.8f, -0.2f));
}

@Override
public void render() {
    // RESPOSTA A EVENTOS DO UTILIZADOR E ATUALIZAÇÃO DA CÂMARA
    camController.update();

    // PREENCHIMENTO DO ECRÃ A PRETO
    Gdx.gl.glViewport(0, 0,
        Gdx.graphics.getWidth(),
        Gdx.graphics.getHeight());
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT | GL20.GL_DEPTH_BUFFER_BIT);

    // DESENHO DAS INSTÂNCIAS DO MODELO USANDO A CÂMARA
    modelBatch.begin(cam);
    modelBatch.render(instance, environment);
    modelBatch.end();
}

@Override
public void dispose() {
    // LIMPEZA DOS RECURSOS UTILIZADOS
    modelBatch.dispose();
    model.dispose();
}

@Override
public void resize(int width, int height) {}

@Override
public void pause() {}

@Override
public void resume() {}
}
```

No método `create` começa-se por instanciar um objeto `ModelBatch` para fazer a gestão da renderização dos modelos. Depois, adiciona-se uma câmara que vai permitir visualizar a cena 3D a partir de um determinado ponto da cena, em perspetiva.

É também instanciado um objeto `CameraInputController`, passando o objeto `PerspectiveCamera` como argumento. Este objeto permite ao utilizador movimentar a posição da câmara usando o toque. Depois, através do método `setInputProcessor` da interface `input`, define-se qual o processador de entrada que vai lidar com todos os eventos de toque. Neste caso, a rotação da câmara será controlada através do toque e arrastamento sob o cubo.

Ainda no método `create`, cria-se o objeto 3D através da classe `ModelBuilder`. Neste caso, cria-se um cubo com um tamanho de 5x5x5. O método `createBox` devolve um objeto `Model` que representa um modelo. Um modelo contém a informação geral sobre o objeto que representa, mas não contém informações sobre onde renderizar o objeto. Sendo assim, cria-se um `ModelInstance`. O objeto `ModelInstance` contém informações sobre a localização, rotação e escala do objeto. Por omissão, a localização é nas coordenadas (0,0,0). Quando a aplicação terminar, o modelo deverá ser destruído no método `dispose`.

Para terminar, falta adicionar iluminação à cena. Para tal, instancia-se a classe `Environment` definindo-se a luz direcional (como um foco de luz), indicando o ponto de origem e a sua direção sob o objeto (Figura 4.8).

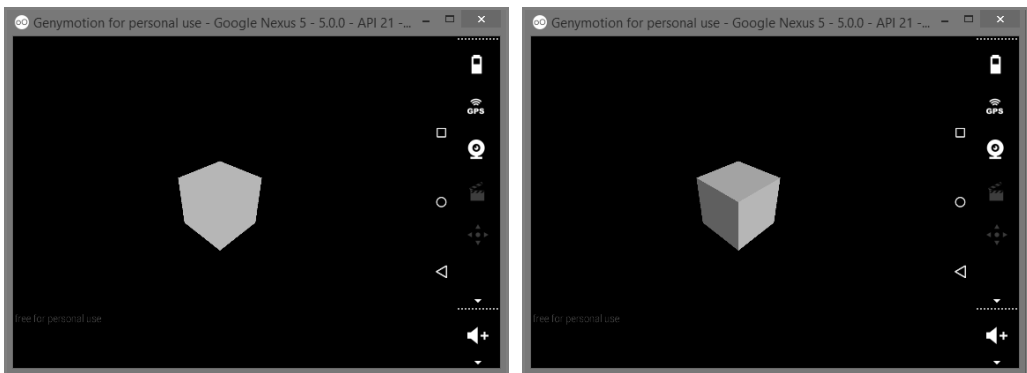


FIGURA 4.8 – Cubo sem e com luz direcional

No método `render` começa-se por definir a janela de exibição e por preenchê-la a preto. Depois, usa-se o objeto `ModelBatch` para iniciar o processo de renderização.

A renderização deve ser feita a cada *frame* através do método `render` da classe `MeuCubo`. Para começar a renderização usa-se o método `begin` sobre o objeto `modelBatch`, passando-lhe a câmara de projeção. De seguida, usa-se o método `render` passando a instância do modelo a ser desenhado e, neste caso específico, o respetivo ambiente (configuração de iluminação). Finalmente, ao terminar a adição de objetos de renderização, invoca-se o método `end`. É nessa altura que o `libgdx` realiza a renderização.

4.2 JOGO BALLOID

De forma a aplicar os conceitos teóricos da secção 4.1, apresenta-se agora o desenho e a implementação de um jogo 3D chamado *Balloid*³⁹.

O jogo tem como objetivo controlar um objeto (bola) e movê-lo em direção a outro objeto (anel, ou *donut*). Para isso será usada a inclinação do dispositivo indicando para que lado a bola se deve mover, como que sob a força da gravidade. Sempre que a bola toca no anel, este desaparece, reaparecendo aleatoriamente noutra local. Ao mesmo tempo, a bola perde atrito, tornando-se mais difícil de controlar. Cada vez que o anel é atingido, a pontuação do jogador aumenta.

Seguindo a filosofia dos jogos dos anos 1980, não é possível “ganhar” o jogo. Este simplesmente se torna cada vez mais difícil. No entanto, é possível “perder”, quando o jogador é incapaz de controlar a bola e esta atinge os limites do ecrã. A interface gráfica do jogo é apresentada na Figura 4.9.

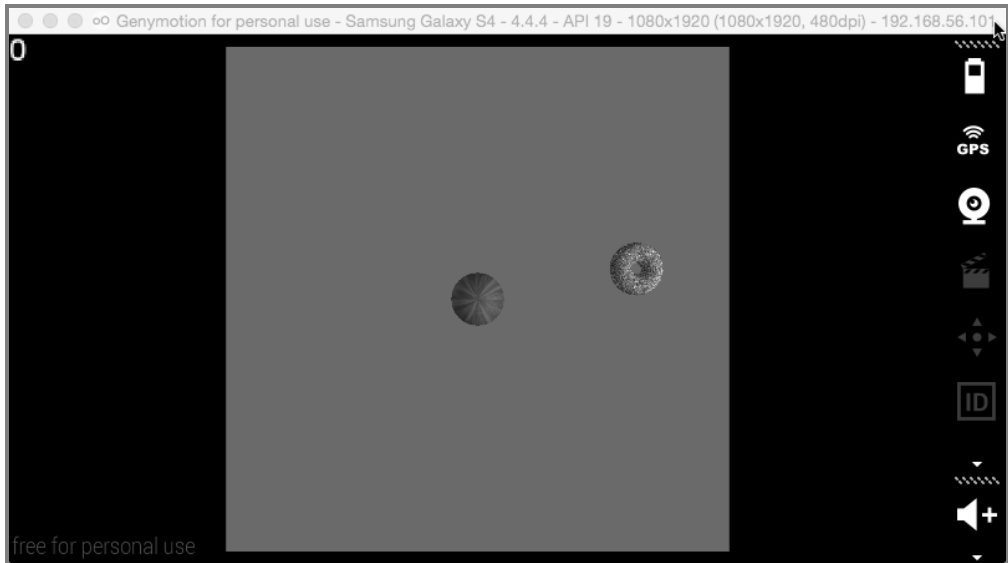


FIGURA 4.9 – A interface gráfica do jogo *Balloid*

Comece por criar um projeto através da ferramenta geradora de projetos libgdx. Defina o nome do projeto como *Balloid*. Depois, no Android Studio importe o projeto.

³⁹ Note-se que este capítulo se centra na implementação lógica do jogo e não introduz quaisquer cuidados gráficos ou de design.

4.2.1 ARQUITETURA BASE DO JOGO

Nesta secção descreve-se a estrutura base do jogo *Balloid*. A Figura 4.10 apresenta o diagrama de classes do jogo.

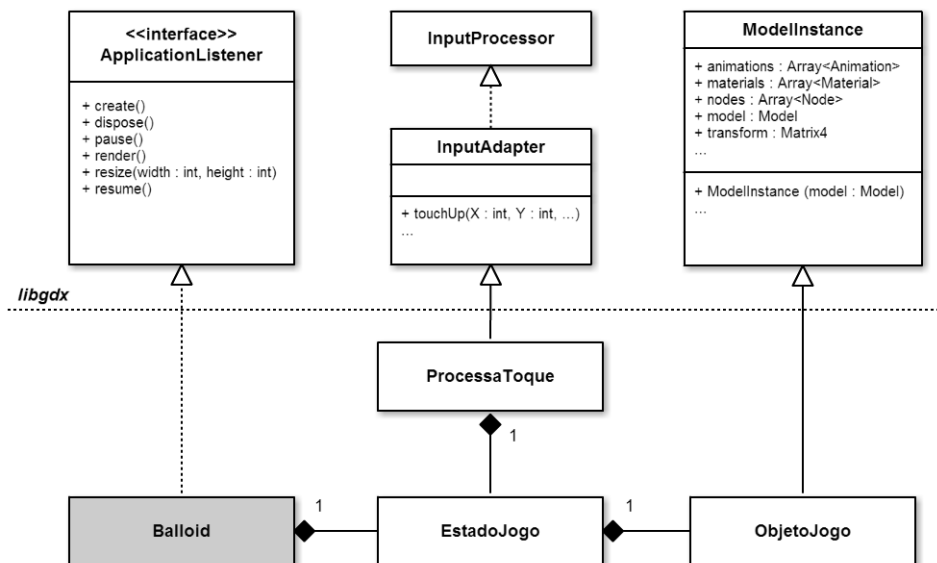


FIGURA 4.10 – Diagrama de classes do jogo *Balloid*

A classe *Balloid* é a classe principal do jogo. A classe implementa a interface *ApplicationListener*, que tem métodos que são automaticamente invocados quando a aplicação é criada, pausada, retomada, renderizada ou destruída.

A classe *Balloid* mantém o estado do jogo através da classe *EstadoJogo*. A classe guarda informações sobre os modelos 3D do jogo (bola e anel), bem como da pontuação atual do jogador e outros valores úteis, como o valor da recompensa do próximo anel a atingir.

Um desses modelos 3D é a bola. Esta vai estar em constante movimento durante o jogo, sofrendo várias alterações a nível da sua posição e rotação. A classe *ObjetoJogo* condensa essas alterações, permitindo ter um código mais compacto e funcional. Embora na implementação do *Balloid* não seja necessário, esta classe também armazena a escala do objeto. Sugerimos que se use uma classe semelhante a esta para todos os objetos que se desloquem, rodem ou mudem de tamanho num jogo.

Por fim, a classe *ProcessaToque* vai lidar com o *input* do utilizador. Neste caso, esta classe só irá ser utilizada no fim do jogo (*game over*). Nesta altura, e após toque no ecrã, o jogo reinicia-se.

As próximas secções apresentam e explicam algumas das tarefas associadas ao jogo que envolvem não só a codificação de alguns métodos do ciclo de vida da aplicação, bem como a criação de modelos 3D a serem usados durante o jogo. As tarefas são as seguintes: modelação 3D, criação de objetos e renderização.

4.2.2 CRIAÇÃO DE MODELOS 3D

Ainda antes de começar a programar, será necessário criar os dois modelos tridimensionais para o jogo. Nesta secção apresenta-se a criação dos modelos 3D usados. Embora não seja esse o foco principal deste livro, parece-nos relevante apresentar algumas das alternativas para criar modelos tridimensionais.

Para o desenho de modelos 3D, como a bola de jogo, usa-se um *software* de modelação gráfica 3D. Existem vários disponíveis no mercado, alguns dos quais gratuitos. Alguns exemplos são o Blender (gratuito), o 3ds Max e o Maya, entre outros. De modo a simplificar-se, optou-se por usar o Clara.io⁴⁰, um *software* de modelação gráfica 3D baseado na Web (Figura 4.11).

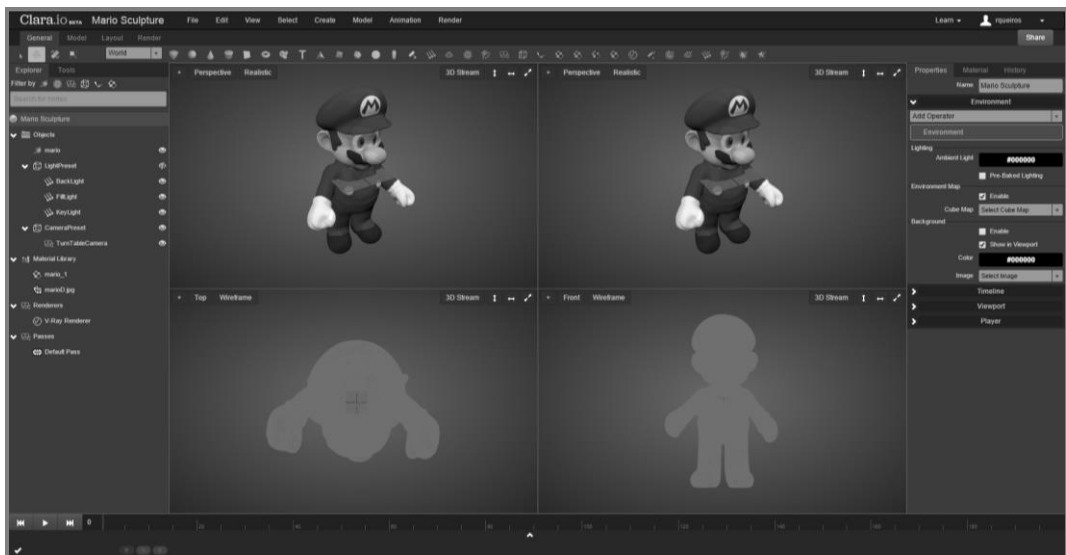


FIGURA 4.11 – O *software* de modelação gráfica 3D Clara.io

O jogo terá dois modelos: a bola e o anel. Como os processos de criação e de modelação dos objetos são similares, detalhar-se-á apenas o processo de criação de uma bola.

⁴⁰ Link: <https://clara.io/>

Depois de criado e autenticado um utilizador, cria-se uma nova cena através da opção **File → New → Empty Scene**. Neste caso, a cena foi batizada de **Sphere**. Depois, adiciona-se uma esfera clicando no ícone **Sphere** da barra de ferramentas. A esfera é posicionada no centro do plano (Figura 4.12). Não altere a escala do objeto, mantendo o raio com tamanho de 0.5 unidades.

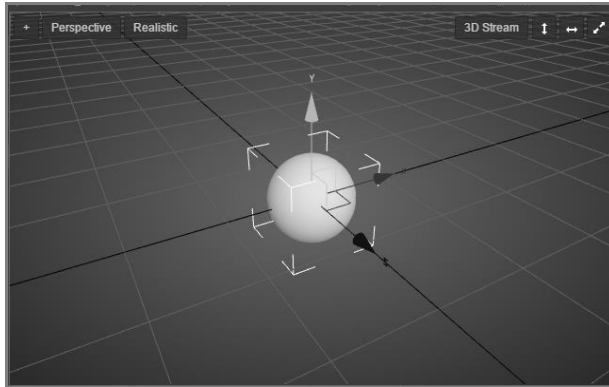


FIGURA 4.12 – Desenho de uma esfera na Clara.io

De seguida, aplica-se um material ao objeto através da opção **Render → Material Browser...** Na janela **Import Material** (Figura 4.13) seleciona-se o material; por exemplo, **WebGL Wood**.



FIGURA 4.13 – A janela Import Material

A esfera fica agora com um aspeto similar a uma bola de madeira (Figura 4.14). Para terminar exporta-se a cena através da opção **File → Export All → OBJ**.

Ao exportar a cena, é descarregado no computador um ficheiro ZIP com os seguintes ficheiros:

- ⊗ **Sphere.obj** – ficheiro do modelo a carregar;
- ⊗ **Sphere.mtl** – ficheiro do material que o modelo usa;
- ⊗ ***.png** – ficheiro(s) com a(s) textura(s) que o material usa.

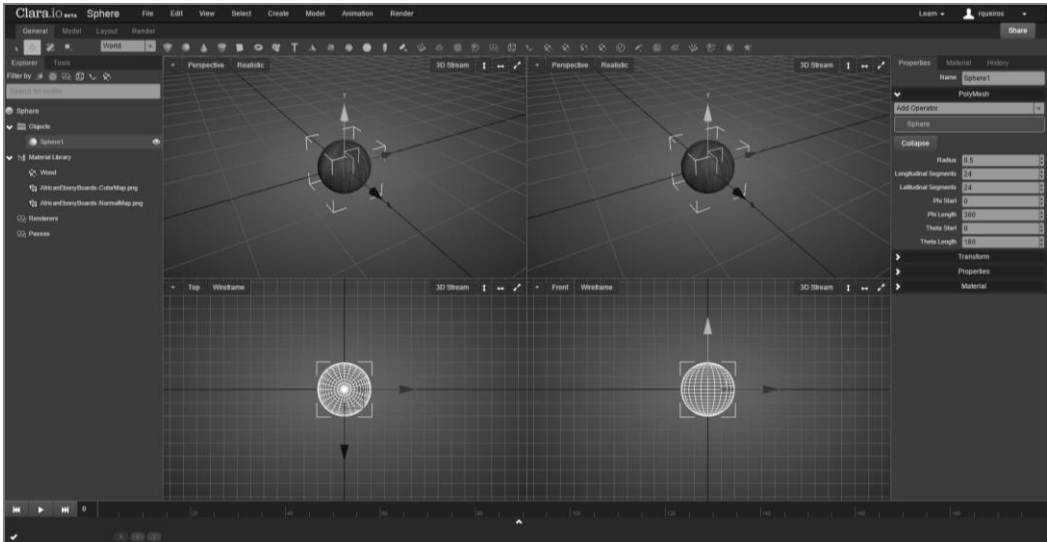


FIGURA 4.14 – A esfera de madeira

O uso de um ficheiro OBJ é suficiente numa fase de teste da aplicação, mas não é adequado para uso numa aplicação real, já que o formato de ficheiro não inclui informações suficientes para a renderização de modelos complexos. De forma a contornar este problema, a libgdx inclui a ferramenta **fbx-conv**, que converte os modelos exportados de *software* de modelação 3D num formato adequado para renderização em aplicações libgdx.

Ao contrário do que o nome possa sugerir, a ferramenta **fbx-conv** é adequada para a conversão de vários formatos de ficheiros (incluindo OBJ), embora FBX seja o formato de ficheiro preferido, pois quase todas as aplicações de modelação 3D o suportam.

Existem dois formatos de ficheiro suportados pela libgdx: **g3dj** (formato de texto – JSON – para facilitar o *debugging*) e **g3db** (formato binário, a usar no modo *release*, porque é mais compacto e, portanto, mais rápido para carregar). Para converter a esfera para o formato **g3db** usa-se na linha de comando a seguinte instrução:

```
fbx-conv Sphere.obj Sphere.g3db
```

O resultado é um novo ficheiro chamado **Sphere.g3db**. Este ficheiro, juntamente com os ficheiros PNG, devem ser copiados para a pasta **assets** do subprojeto **Android**.

Finalmente, reproduza toda a sequência anterior para o segundo modelo tridimensional a criar, o anel. Crie uma nova cena e adicione um anel clicando no ícone **Torus** da barra de ferramentas. Depois, associe o material **Grass**. Por fim, exporte o modelo 3D para o formato OBJ e use novamente a ferramenta **fbx-conv** para gerar o ficheiro **Ring.g3db**.

4.2.3 ESTADO DO JOGO

Em primeiro lugar será criada uma nova classe que corresponde ao estado do jogo. Esta classe deve ser criada na pasta **core/src/com/balloid/game**. No caso do *Balloid* esta classe vai funcionar quase como uma estrutura, armazenando só alguns dados. A definição da classe é apresentada de seguida:

```
public class EstadoJogo {
    // INFORMAÇÃO SOBRE SE O JOGO TERMINOU
    public boolean gameOver = false;
    // PONTUAÇÃO ATUAL
    public int pontuacao = 0;
    // PRÉMIO PELO PRÓXIMO ANEL
    public int premio = 2;
    // ATRITO ATUAL
    public float atrito = 0.9f;
    // VETOR COM VELOCIDADE ATUAL
    public Vector3 velocidade = new Vector3(0f, 0f, 0f);
    // OBJETO COM INFORMAÇÃO DA BOLA
    public ObjetoJogo bola;
    // OBJETO COM INFORMAÇÃO DO ANEL ATUAL
    public ModelInstance anel;
}
```

Embora tivesse sido mais correto criar aqui um construtor com a inicialização destes valores, optou-se por realizar a inicialização manualmente na altura necessária, o que tornará mais fácil de compreender o funcionamento da aplicação.

4.2.4 OBJETO DE JOGO

Tal como indicado previamente, será usada uma classe, de nome *ObjetoJogo*, para armazenar informação sobre a bola. Esta classe, que será criada na mesma pasta que a anterior, coloca em evidência a posição, a rotação e a escala do objeto. Isto torna o manuseamento do objeto mais fácil do que a manipulação direta do objeto de tipo *Matrix4* que é usado para armazenar toda esta informação de forma compacta:

```
public class ObjetoJogo extends ModelInstance {
    public final Vector3 posicao = new Vector3();
    public final Quaternion rotacao = new Quaternion();
    public final Vector3 escala = new Vector3(1,1,1);

    public ObjetoJogo(Model modelo) {
        super(modelo);
    }

    public void atualiza() {
        this.transform.set(posicao, rotacao, escala);
    }

    public void rodar(Vector3 eixo, float angulo) {
        rotacao.mulLeft(new Quaternion(eixo, angulo));
    }

    public void rodarRad(Vector3 eixo, float angulo) {
        this.rodar(eixo, (float) Math.toDegrees(angulo));
    }
}
```

Esta classe estende a classe `ModelInstance`, o que significa que vai representar a instância de um modelo tridimensional. Para além dos atributos herdados, a classe define três atributos públicos: a rotação, a posição e a escala. A posição e a escala são descritas por vetores tridimensionais. A rotação é descrita por um objeto denominado quatérnio (`Quaternion`). Este objeto é capaz de representar a rotação de um objeto num mundo tridimensional. No entanto, não iremos entrar em detalhe sobre isto.

O construtor desta classe é semelhante ao construtor de um `ModelInstance`, recebe o `Model` a ser instanciado e usa o construtor da superclasse.

O método `atualiza` é usado quando algum dos três atributos definidos é alterado, para que a classe atualize o modelo da superclasse, colocando-o na posição certa, com a rotação e as escalas desejadas.

Os dois métodos `rodar` e `rodarRad` são responsáveis por rodar o objeto. Recebem o eixo sobre o qual será executada a rotação e o ângulo da rotação. Para o primeiro, o ângulo deverá ser indicado em graus, enquanto o segundo método permite o uso de radianos. A rotação é obtida pela multiplicação do quatérnio que representa a rotação desejada com o quatérnio que representa a rotação atual.

4.2.5 LÓGICA DE JOGO

A lógica de jogo será implementada na classe principal do projeto, ou seja, na classe `Balloid`. Esta secção irá apresentar progressivamente a lógica do jogo, alterando os métodos aos poucos.



Tentaremos que o código seja apresentado de forma clara, permitindo ao leitor acompanhar todo o desenvolvimento. No entanto, caso se perca, deverá aceder ao projeto na página do livro em www.fca.pt.

O primeiro passo será a criação da cena base: definição da luz ambiente, colocação de um plano onde a bola deslizará e definição da posição da câmara. Posteriormente, será colocada a bola em movimento. Seguir-se-á a colocação do anel e a deteção de colisão. Finalmente, serão adicionados detalhes como a contagem da pontuação, a deteção de final de jogo e, ainda, a adição de som.

4.2.5.1 DEFINIÇÃO DA CENA

Para a definição da cena começa-se por adicionar a luz ambiente. Para isso, será criada uma instância da classe `Environment`:

```
private Environment ambiente;

@Override
public void create () {
    ambiente = new Environment();
    ambiente.set(
        new ColorAttribute(
            ColorAttribute.AmbientLight, 0.8f, 0.8f, 0.8f, 1f));
    ambiente.add(
        new DirectionalLight().set(0.8f, 0.8f, 0.8f, -1f, -0.8f, -0.2f));
    ...
}
```

De seguida, será adicionado o plano onde a bola vai deslizar. O plano vai estar centrado na posição (0,0,0) e será quadrado, com 10 unidades de lado. A sua criação vai recorrer ao `ModelBuilder`, usando uma forma predefinida.

Em primeiro lugar definiram-se as variáveis `Xmax`, `Xmin`, `Zmax` e `Zmin`, que representam os limites do plano. Para além destas, também foi criada a variável `mPlane`, que irá armazenar o modelo do plano. Para armazenar a instância do plano optou-se por usar um vetor de instâncias. Assim, será possível colocar neste vetor todos os objetos que se pretendem desenhar, e passá-lo diretamente ao método `render` do `BatchRenderer`:

```
private Model mPlano;
private final float Xmax= 5f, Zmax=5f;
private final float Xmin = - Xmax, Zmin = - Zmax;
private Vector<ModelInstance> instancias = new Vector<ModelInstance>();
@Override
public void create () {
    ...
    ModelBuilder modelBuilder = new ModelBuilder();
    mPlano = modelBuilder.createRect(Xmin, 0, Zmax,
                                    Xmax, 0, Zmax,
                                    Xmax, 0, Zmin,
```

```

Xmin, 0, Zmin, 0, 1, 0,
new Material(
    ColorAttribute.createDiffuse(
        new Color(0.8f, 0.2f, 0.3f, 0.0f)),
        VertexAttributes.Usage.Position|
        VertexAttributes.Usage.Normal);
instancias.add(new ModelInstance(mPlano));
...
}

```

A criação do plano recebe os quatro pontos que correspondem aos quatro vértices. Segue-se um vetor que indica qual a direção normal ao plano (basicamente, a direção para a qual o plano deve ser renderizado), que neste caso é um vetor vertical. Finalmente, são indicados o material e os atributos, tal como demonstrado anteriormente com a criação de um cubo.

Embora não seja nossa intenção, iremos precisar de complicar um pouco a matemática para a definição da câmara. Para que seja possível considerar que o jogador perde quando a bola desliza para fora do plano e, ao mesmo tempo, que o mesmo acontece quando a bola desaparece do ecrã, é importante que a câmara esteja configurada de modo a que a margem do plano coincida com a margem do dispositivo (na dimensão mais pequena). Para isso, o ângulo de visão da câmara será calculado com recurso a alguma trigonometria, de acordo com a Figura 4.15.

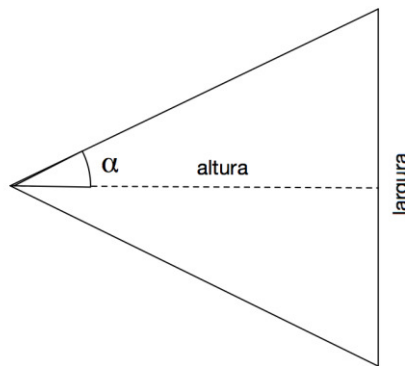


FIGURA 4.15 – Cálculo do ângulo de visão

Nesta imagem considere-se a câmara no vértice esquerdo do triângulo. Considere-se também que a aresta oposta a esse vértice corresponde ao plano que, como se sabe, tem 10 unidades de largura. Supondo que se coloca a câmara também a uma altura de 10 unidades em relação ao plano, é possível calcular o ângulo de visão β que corresponde ao dobro do ângulo α , do triângulo retângulo composto pela altura (10 unidades) e por metade da largura do plano (cinco unidades). Este ângulo pode ser obtido usando a função matemática *arcotangente*:

$$\beta = 2 \times \alpha = 2 \times \arctan \frac{0.5 \times largura}{altura} = 2 \times \arctan2(0.5 \times largura, altura)$$

A função *arctan2* é usada para remover a necessidade da divisão, permitindo à biblioteca matemática realizar algum tipo de otimização. Assim, o código necessário para a criação e posicionamento da câmara é:

```
...
private PerspectiveCamera cam;
@Override
public void create () {
    ...
    float alturaCamera = 10f;
    float anguloCamera = (float) Math.toDegrees(2.0 *
        Math.atan2(Xmax, alturaCamera));
    cam = new PerspectiveCamera(anguloCamera,
        Gdx.graphics.getWidth(), Gdx.graphics.getHeight());
    cam.position.set(0f, alturaCamera + 0.5f, 0f);
    cam.lookAt(0,0,0);
    cam.near = 1f;
    cam.far = 50f;
    cam.update();
    ...
}
```

Note-se que a câmara foi colocada ligeiramente acima do definido anteriormente para que a margem do plano não fique demasiado resvés ao limite do dispositivo.

4.2.5.2 MOVIMENTAÇÃO DA BOLA

Esta secção é dedicada à criação da bola e à programação do seu movimento de acordo com o movimento da bússola do dispositivo.

Em primeiro lugar será carregado o modelo da bola e, aproveitando a boleia, será também carregado o modelo do anel. De seguida, define-se a simulação física da velocidade e atrito e, finalmente, a aceleração de acordo com o movimento da bússola.

Existem duas formas de carregar modelos gráficos: uma delas é síncrona e a outra assíncrona. Na primeira abordagem, a execução do programa para à espera que os modelos sejam carregados. Na segunda abordagem, é criada uma nova *thread* que irá carregar as texturas ao mesmo tempo que o programa segue a sua execução normal. Será utilizada a segunda abordagem, dado ser a mais usual. O carregamento dos modelos é controlado por um objeto do tipo *AssetManager*.



Outra característica do *AssetManager* é o carregamento único de recursos partilhados, economizando memória. Se, por exemplo, dois dos seus modelos utilizarem a mesma textura, a textura será carregada apenas uma vez. A textura é mantida em memória enquanto pelo menos um dos modelos estiver carregado e eliminada automaticamente quando não estiverem mais em uso.

O carregamento da textura é, então, realizado usando o seguinte código:

```
private Model mEsfera, mPlano, mAnel;
private AssetManager assets;
private boolean aCarregar;

@Override
public void create () {
    ...
    assets = new AssetManager();
    assets.load("data/Sphere.g3db", Model.class);
    assets.load("data/Grass.g3db", Model.class);
    aCarregar = true;
}

@Override
public void render () {
    if (aCarregar) {
        if (assets.update()) carregamentoTerminado();
        return;
    }
}

private void carregamentoTerminado() {
    mEsfera = assets.get("data/Sphere.g3db", Model.class);
    estado.bola = new ObjetoJogo(mEsfera);
    estado.bola.posicao.set(0f, 0.5f, 0f);
    instancias.add(estado.bola);
    mAnel = assets.get("data/Grass.g3db", Model.class);
    aCarregar = false;
}
```

Ainda no método `create`, é criado um `AssetManager` e indicado que devem ser carregados os dois modelos, da bola e do anel. Coloca-se também a variável `aCarregar` como `true` para que se saiba quando o carregamento dos modelos terminou.

No método `render`, que é invocado em cada *frame* do jogo, deve-se verificar se os modelos já foram carregados. Se já tiverem sido carregados, então o método `update` da variável `assets` irá retornar um valor verdadeiro. Se ainda não tiverem sido carregados, a função retorna. Quando o carregamento terminar, o método `carregamentoTerminado` é invocado.

Neste método, os modelos são obtidos a partir do `AssetManager` e a bola é instanciada usando a classe `ObjetoJogo` definida anteriormente. A instância é colocada no centro da cena, a uma altura de 0.5, que corresponde ao raio da esfera. Finalmente, a variável `aCarregar` passa a `false`.

Para simular⁴¹ o movimento da bola será usado um vetor tridimensional já declarado na classe `ObjetoJogo`. Esse vetor será inicializado no método `create`:

```
private EstadoJogo estado;
@Override
public void create () {
    estado = new EstadoJogo();
    ...
    estado.velocidade = new Vector3(0f, 0f, 0f);
}
```

Para realizar o movimento da bola, bem como a rotação, de acordo com o vetor velocidade, é usado o código que se segue:

```
@Override
public void render() {
    ...
    float deltaTempo = Gdx.graphics.getRawDeltaTime();

    Vector3 posicao = estado.bola.posicao.cpy();
    estado.velocidade.scl(1 - (estado.atrito * deltaTempo));
    posicao.add(estado.velocidade);

    float distancia = estado.velocidade.len();
    Vector3 normal = new Vector3();
    normal.set(estado.velocidade.cpy().nor()).crs(Vector3.Y);
    estado.bola.rodarRad(normal, -2 * distancia);
    estado.bola.posicao.set(posicao);
    estado.bola.atualiza();
}
```

Mais uma vez, este código usa alguma matemática mais complicada. Em primeiro lugar, é usado o método `getRawDeltaTime` para obter o número de segundos que decorreram desde a última vez que o método `render` foi invocado. Este valor é útil para que se consiga um movimento suave da bola. Caso contrário, como o método `render` não é invocado em intervalos de tempo constantes, iria parecer que a bola se desloca aos solúços.

De seguida, é criada uma cópia da posição atual da bola. Note-se a necessidade do uso do método `cpy`. Caso contrário, é criada uma referência para o mesmo vetor e a alteração de valores num dos vetores afetará o outro vetor. Segue-se a atualização da velocidade de acordo com o atrito. O valor de atrito é multiplicado pelo delta temporal de modo a homogeneizar o seu valor. Este valor de atrito deve ser removido à

⁴¹ A forma correta de implementar o jogo seria tirar partido de uma biblioteca de simulação física. Neste caso, com a `libgdx`, a escolha natural seria o **BulletPhysics**. No entanto, a manipulação direta do objeto parece-nos mais interessante e educativa, já que pode ser generalizada para muitas outras situações.

velocidade. A forma mais simples de o realizar é multiplicar⁴² o vetor de velocidade pelo complemento do atrito (assim, se o atrito for, por exemplo, 10%, multiplica-se o vetor de velocidade por 90%, diminuindo-o em exatamente 10%). Finalmente, este valor de velocidade é adicionado à posição atual da bola, calculando a posição de destino onde a bola deve aparecer na próxima *frame*.

O resto do código corresponde à movimentação da bola para a posição calculada e a sua rotação para que pareça que rodou. Para melhor compreender este processo de rotação a Figura 4.16 esquematiza o que se pretende.

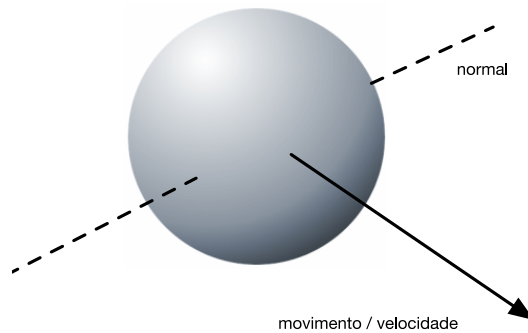


FIGURA 4.16 – Movimento de rotação da bola

A bola deve rodar na direção indicada pelo vetor da imagem. Isso significa que o seu eixo de rotação será perpendicular a esse mesmo vetor e que atravessará a bola pelo seu centro, tal como é indicado na figura, pelo tracejado. Este vetor é designado por **normal**.

O cálculo deste **normal** é realizado com base no **produto externo** do vetor de velocidade normalizado por um outro vetor que se quer "**normal**" ao vetor de velocidade e ao vetor que se pretende obter. Assim, uma cópia do vetor de velocidade é normalizada⁴³ (usando-se o método `nor`) e o seu **produto externo** (método `crs`) calculado com o vetor vertical, definido por $(0,1,0)$, que está definido na `libgdx` como `Vector3.Y`.

Sabendo a distância da deslocação (que corresponde ao comprimento do vetor de velocidade), é fácil obter a rotação necessária. Como a esfera tem diâmetro de 1 unidade, o seu perímetro é π , que corresponde ao número de radianos de um ângulo raso. Ou seja,

⁴² A multiplicação de um vetor por um escalar também é designada por **escalar o valor**, daí o método usado se chamar `scal`, abreviatura de *scale*.

⁴³ A normalização de um vetor corresponde a transformá-lo num vetor unitário, ou seja, num vetor que tenha a mesma direção do vetor original, mas cujo comprimento seja 1.

a deslocação de uma distância de π obriga a uma rotação completa (2π). Logo, é possível usar o dobro da distância percorrida como ângulo de rotação.

Depois de realizar a rotação da esfera, a sua posição é alterada para a posição de destino. Finalmente, é invocado o método `update` para que a classe `ObjetoJogo` atualize a posição do modelo tridimensional.

O cálculo da velocidade vai ser realizado de acordo com a inclinação do dispositivo, usando o sensor de bússola. Para uma aplicação usar este sensor deve, em primeiro lugar, indicá-lo no ficheiro de manifesto da aplicação. No caso da `libgdx`, este ficheiro está em `android/manifests/AndroidManifest.xml`, e deve ser adicionada a seguinte linha:

```
<manifest ...
  <uses-feature android:name="android.hardware.sensor.compass"
    android:required="true"/>
  ...
</manifest>
```

A Figura 4.17 mostra como atuam os sensores de bússola. No caso concreto do *Balloid*, com o dispositivo deitado sobre uma mesa, a bola não se deve mexer. Quando o jogador inclinar o dispositivo para si, ou a partir de si (movimento *roll*), deverá ser mudada a posição *Z* da bola. Por sua vez, ao inclinar para um lado ou para o outro (movimento *pinch*), deverá ser alterada a posição *X* da bola. O valor destes dois sensores pode ser obtido através dos métodos `Gdx.input.getRoll()` e `Gdx.input.getPitch()`, respetivamente.

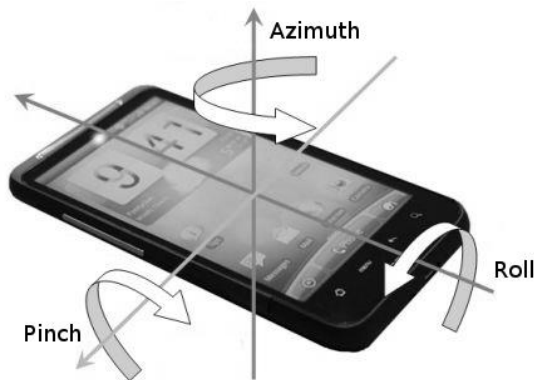


FIGURA 4.17 – Sensores de bússola (*compass*)

Os métodos de consulta aos valores dos sensores retornam a inclinação do dispositivo em graus. Este valor será convertido em radianos, e posteriormente usada a função *seno* para obter uma noção de inclinação, que será usada como um fator de aceleração na bola. Mais uma vez note-se que este modelo não respeita as leis da Física.

Logo abaixo do código apresentado anteriormente, segue-se o bloco que calcula a nova velocidade:

```
float racio = 1000f;
float pitch = Gdx.input.getPitch();
float roll = Gdx.input.getRoll();

double alturaX = -5 * Math.sin(Math.toRadians(pitch));
double alturaZ = -5 * Math.sin(Math.toRadians(roll));

estado.velocidade.add(new Vector3((float) alturaX / racio, 0f,
                                   (float) alturaZ / racio));
```

Depois de obtidos os ângulos dos sensores (*pitch* e *roll*) em graus, os valores são convertidos para radianos, e é usado o *seno* para obter uma distância de movimento. A multiplicação por 5, e posterior divisão por 1000, foi obtida após algumas experiências, de modo a que a velocidade da bola não seja excessiva. A velocidade calculada é adicionada à velocidade atual (o que permite que o jogador possa compensar um movimento brusco para um lado com outro movimento brusco para o outro).

Logo de seguida, será desenhada a cena que corresponde a limpar o ecrã e a desenhar os objetos do vetor *instancias*, usando o *modelBatch*:

```
Gdx.gl.glViewport(0, 0, Gdx.graphics.getWidth(),
                  Gdx.graphics.getHeight());
Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT | GL20.GL_DEPTH_BUFFER_BIT);

modelBatch.begin(cam);
modelBatch.render(instancias, ambiente);
modelBatch.end();
```

4.2.5.3 ANÉIS E COLISÕES

O passo seguinte no desenvolvimento do jogo será a criação de um anel numa posição aleatória e, sempre que a bola lhe tocar, fazê-lo desaparecer e criar um novo. A leitura do modelo já foi feita, juntamente com o carregamento da esfera.

A criação do anel será feita no método *render* sempre que não exista um anel em jogo. Assim, e logo depois da linha que obtém a posição atual da bola (em negrito no código seguinte), adiciona-se o restante código:

```
...
Vector3 posicao = estado.bola.posicao.cpy();
if (estado.anel == null) {
    estado.anel = new ModelInstance(mAnel);

    float x = (float) Math.random() * (Xmax - Xmin) - Xmax;
    float z = (float) Math.random() * (Zmax - Zmin) - Zmax;

    estado.anel.transform.setToTranslation(x, 0.5f, z);
```

```
instancias.add(estado.anel);
}
...
```

Este bloco verifica se o anel guardado no objeto `EstadoJogo` é nulo. Se assim for, é criada uma nova instância e calculada uma posição aleatória dentro dos limites do plano. Finalmente, o anel é colocado na posição certa e adicionado ao vetor de instâncias a desenhar.

O próximo passo corresponde a detetar a colisão da bola com o anel. Mais uma vez, a solução usada será a mais simples possível. Como ambos os objetos são circulares e têm o mesmo raio, bastará verificar se a distância entre a posição da bola e a posição do anel é menor do que o diâmetro dos mesmos. O código que trata dessa deteção é apresentado de seguida e será colocado antes da estrutura condicional apresentada anteriormente.

```
...
Vector3 posicao = estado.bola.posicao.cpy();
Vector3 aPosicao = new Vector3();
if (estado.anel != null) {
    estado.anel.transform.getTranslation(aPosicao);
    if (aPosicao.dst(posicao) < 1f) {
        instancias.remove(estado.anel);
        estado.anel = null;
        estado.pontuacao += estado.premio;
        estado.premio *= 2;
        estado.atrito *= 0.9;
    }
}
if (estado.anel == null) {
    ...
}
```

Neste bloco de texto começa-se por declarar uma variável onde será guardada a posição do anel (`aPosicao`). Se o anel existir, a sua posição é obtida e armazenada. Depois, é calculada a distância dessa posição com a posição da bola. Se a distância for menor do que 1 unidade (soma do raio da bola com o raio do anel), então houve colisão. Nesse caso, o anel é removido do vetor de instâncias e o seu valor passa a nulo. Posteriormente, incrementa-se a pontuação, aumenta-se o prémio pelo próximo anel e diminui-se o atrito.

Finalmente, será preciso tratar da situação de final de jogo. Para isso será usada uma variável booleana, presente no `EstadoJogo`, de nome `gameOver`. No início, a variável deve ter o valor `false`:

```
@Override
public void create () {
    estado = new EstadoJogo();
    estado.gameOver = false;
    ...
}
```

Já no método `render`, deverá ser verificada a posição da bola e, caso esta ultrapasse os limites definidos, indicado o fim de jogo e removido o anel atual (para o caso de o jogador reiniciar o jogo). Este código pode ser adicionado logo após a obtenção da posição atual da bola:

```
Vector3 posicao = estado.bola.posicao.cpy();
if (posicao.x < Xmin || posicao.x > Xmax ||
    posicao.z < Zmin || posicao.z > Zmax) {
    estado.gameOver = true;
    instancias.remove(estado.anel);
}
```

4.2.5.4 INTERFACE E INTERAÇÃO COM O UTILIZADOR

Nesta secção será adicionada informação sobre a pontuação atual, bem como a mensagem de *game over* caso o jogador perca.

Para adicionar informação escrita a solução será usar um tipo de letra *TrueType* ou então um tipo de letra *bitmap* (*raster*). A vantagem do primeiro é a possibilidade de modificar o tamanho de letra sem que se perca qualidade. Já o segundo, sendo *bitmap*, gera texto pixelizado quando aumentado em demasia.

A `libgdx` inclui ferramentas para a geração de tipos de letra *bitmap* a partir de um tipo *TrueType* e um tamanho fixo de letra. Na implementação do *Balloid*, e para simplificar, será usado o tipo e tamanho de letra por omissão da `libgdx`.

Para manusear o tipo de letra será necessária uma instância da classe `BitmapFont` e, para a renderizar, será necessária uma instância da classe `SpriteBatch`. Esta última é semelhante à `ModelBatch`, mas para a renderização de objetos bidimensionais. Para além destas duas variáveis, iremos também criar duas variáveis inteiras para guardar a posição onde a pontuação deverá ser colocada:

```
private SpriteBatch spriteBatch;
private BitmapFont bitmapFont;
private int scoreX, scoreY;
```

A inicialização destas variáveis será feita no método `create`. Pode adicionar o bloco de código seguinte em qualquer sítio desse método:

```
spriteBatch = new SpriteBatch();
bitmapFont = new BitmapFont();
bitmapFont.setColor(Color.WHITE);
bitmapFont.setScale(4f);
scoreX = 5;
scoreY = Gdx.graphics.getHeight() - 5;
```

Depois de criado o objeto para guardar o tipo de letra, é alterada a cor com que esta será impressa, bem como a sua escala (neste caso, multiplicando-a por quatro⁴⁴). A posição onde a pontuação é impressa será no topo esquerdo. Assim, deixa-se uma margem de 5 pixels à esquerda, e de 5 pixels em cima. Como o ponto (0,0) é no fundo do ecrã, usa-se a altura do ecrã para obter a posição certa onde o texto deverá ser apresentado.

No final do método `render`, após a renderização dos objetos tridimensionais, é renderizada a pontuação:

```
...
modelBatch.end();

spriteBatch.begin();
bitmapFont.draw(spriteBatch,
                Integer.toString(estado.pontuacao),
                scoreX, scoreY);
spriteBatch.end();
}
```

Para tratar a situação de *game over*, e logo após verificar se os modelos já foram carregados, será verificado o estado da variável `gameOver`:

```
@Override
public void render () {
    if (aCarregar) {
        ...
    }

    if (estado.gameOver) {
        spriteBatch.begin();
        String message = "GAME OVER!";
        BitmapFont.TextBounds bounds = bitmapFont.getBounds(message);
        bitmapFont.draw(spriteBatch, message,
                        Gdx.graphics.getWidth()/2 - bounds.width/2,
                        Gdx.graphics.getHeight()/2 + bounds.height/2 );
        spriteBatch.end();
        return;
    }
}
```

Este bloco apenas desenha a mensagem de *game over*. No entanto, e para centrar a mensagem no ecrã, é necessário ter noção dos tamanhos do ecrã e da mensagem que será impressa. Para isso é usado o método `getBounds`, que retorna uma estrutura com a largura e a altura do texto em pixels. A estrutura condicional termina com o retorno da função para garantir que o jogo não continua (e, portanto, o ambiente de jogo não é desenhado).

⁴⁴ Note que esta multiplicação deveria ser em função do tamanho do ecrã, mas optou-se por manter o código simples.

Por fim, é necessário permitir ao utilizador voltar ao jogo, tocando no ecrã de *game over*. Isso será feito com recurso à classe `ProcessaToque`, uma extensão da classe `InputAdapter`. Será, então, criado um novo ficheiro para esta classe:

```
public class ProcessaToque extends InputAdapter {
    private EstadoJogo status;

    public ProcessaToque(EstadoJogo estado) {
        super();
        status = estado;
    }

    @Override
    public boolean touchUp(int x, int y, int pointer, int button) {
        if (status.gameOver) {
            status.gameOver = false;
            status.pontuacao = 0;
            status.premio = 2;
            status.atrito = 0.9f;
            status.velocidade = new Vector3(0f, 0f, 0f);
            status.bola.posicao.set(0f, 0.5f, 0f);
            status.bola.rotacao.set(new Quaternion());
        }
        return false;
    }
}
```

Esta classe tem um construtor que armazena o estado de jogo e implementa o método que trata o evento de toque no dispositivo. Este toque, se for feito durante um *game over*, despoleta uma reinicialização do estado⁴⁵.

Para que esta classe seja usada bastará alterar o método `create` para a associar à aplicação. Estas linhas podem ser adicionadas em qualquer ponto do método, desde que depois da inicialização do estado:

```
ProcessaToque inputProcessor = new ProcessaToque(estado);
Gdx.input.setInputProcessor(inputProcessor);
```

4.2.5.5 SOM

Para terminar serão adicionados dois sons: um quando um anel é destruído e o outro quando o jogador perde o jogo. Em primeiro lugar são definidas duas variáveis do tipo `Sound`:

```
private Sound plim, fail;
```

⁴⁵ A reinicialização do estado deveria fazer parte de um método da classe `EstadoJogo`, de modo a que este código não fosse aqui repetido.

No método `create` os sons devem ser carregados a partir de ficheiros. Sugere-se que se procure, em sítios Web de amostras gratuitas, dois sons para estas duas situações, e que estes sejam guardados na mesma pasta dos modelos tridimensionais com os nomes `plim.mp3` e `fail.mp3`. O carregamento dos ficheiros é obtido com:

```
plim = Gdx.audio.newSound(Gdx.files.internal("data/plim.mp3"));
fail = Gdx.audio.newSound(Gdx.files.internal("data/fail.mp3"));
```

A reprodução será feita dentro das estruturas condicionais nas quais se detetam a colisão com o anel e o final de jogo:

```
if (posicao.x < Xmin || posicao.x > Xmax ||
    posicao.z < Zmin || posicao.z > Zmax) {
    ...
    fail.play();
}

if (estado.anel != null) {
    estado.anel.transform.getTranslation(aPosicao);
    if (aPosicao.dst(posicao) < 1f) {
        ...
        plim.play();
    }
}
```

4.2.5.6 LIMPEZAS FINAIS

Grande parte dos objetos que foram usados ao longo das secções anteriores deve ser libertada. Essa tarefa deve ser realizada, exceto em situações específicas, no método `dispose`. Assim, podemos libertar a memória usada com:

```
@Override
public void dispose () {
    modelBatch.dispose();
    spriteBatch.dispose();
    mEsfera.dispose();
    mAnel.dispose();
    mPlano.dispose();
    bitmapFont.dispose();
    plim.dispose();
    fail.dispose();
}
```

5

UNITY 3D PARA ANDROID

O Unity 3D é um motor de jogo bastante popular, que inclui uma versão gratuita que pode ser usada livremente. Um dos seus pontos fortes é a possibilidade de preparar jogos para várias plataformas ao mesmo tempo, entre as quais diferentes plataformas móveis, como sejam o Android ou o iOS. Neste capítulo é feita uma introdução básica ao Unity 3D, aos vários conceitos que envolvem a construção de um jogo nesta ferramenta e à escrita de *scripts* numa variante do Java denominada JavaScript. Esta introdução será guiada pelo desenvolvimento de um jogo específico – uma variação do conhecido jogo *Arkanoïd*⁴⁶ numa versão a três dimensões: o *Arkadroid 3D*.

5.1 UNITY 3D

O Unity 3D⁴⁷ é constituído por um conjunto de aplicações que inclui um editor de ambientes 2D/3D, um editor de código, compiladores e um potente motor para o desenvolvimento de jogos. O motor é baseado em vários módulos, como sejam:

- Um motor gráfico, com suporte nativo de Windows DirectX 11, OpenGL e OpenGL ES, com renderização de iluminação de forma diferida e mais de uma centena de sombreadores (*shaders*), entre muitas outras funcionalidades;
- Um motor áudio, baseado na biblioteca FMOD, uma das bibliotecas mais usadas para a criação e reprodução de áudio interativo, que inclui filtros como distorção, coro, eco ou *reverb*;
- Um motor de simulação de propriedades físicas 3D, baseado no motor NVIDIA PhysX, que inclui funcionalidades como o uso de gravidade, vários tipos de juntas, tecido interativo, bonecos de trapos ou tração automóvel.

⁴⁶ Um videojogo desenvolvido pela Taito em 1986.

⁴⁷ O Unity 3D pode ser obtido de forma gratuita em <http://unity3d.com/>.

O editor interativo de ambientes 2D/3D permite a importação de modelos de diferentes origens, como o Blender, o Autodesk Maya ou o Cinema 4D, e inclui um editor integrado de terrenos, que permite a criação interativa de terrenos, incluindo a colocação de vegetação de forma procedimental.

A programação em Unity 3D é feita em C# ou JScript (uma variante de Java da Microsoft). É possível que diferentes partes do jogos sejam implementadas em linguagens diferentes, embora em certas situações de dependências cruzadas o uso das duas linguagens seja desaconselhado. A possibilidade de uso destas duas linguagens de programação existe graças ao uso da plataforma *Mono* (uma plataforma *open-source* semelhante ao *.net* da Microsoft).



Há algo importante a realçar, que corresponde ao facto de a linguagem de *scripting* acima referida não corresponder de forma alguma à conhecida linguagem JavaScript usada para a programação Web. A linguagem usada pelo Unity 3D tem algumas semelhanças com a linguagem usada na Web, já que ambas se baseiam na linguagem Java. O leitor é aconselhado a procurar documentação específica da Microsoft ou do Unity 3D e nunca relativa à programação Web.

A instalação do Unity 3D é trivial, e corresponde ao processo típico de instalação de aplicações em Windows ou Mac OS X. Após a instalação, e na primeira vez em que se executa o Unity 3D, será aberto, de forma automática, um projeto de demonstração de um jogo. Poderá usar a tecla **play** (reproduzir) para o experimentar.



Neste capítulo serão apresentadas imagens do editor Unity 3D, na sua versão 5.0, para Mac OS X. Este facto, aliado às possibilidades de configuração dos vários separadores do editor, pode fazer com que pareçam bastante diferentes da interface obtida com a instalação de uma nova versão de Unity 3D. Note, também, que o desenvolvimento do Unity 3D está atualmente em grande velocidade, o que pode levar a que algumas das funcionalidades aqui descritas tenham mudado de sítio.

Nas próximas secções será implementado o jogo *Arkadroid 3D*. É um jogo em ambiente tridimensional, em que o jogador controla um pequeno bloco (a que chamaremos *pad*) para fazer uma bola saltitar, destruindo pequenos cubos. O jogo termina se o jogador perder a bola um número máximo de vezes (a que chamaremos “vidas”) ou se conseguir destruir com sucesso todos os blocos.



O Unity 3D é uma ferramenta demasiado complexa para ser abordada num único capítulo, ou mesmo num único livro. Por essa razão, este capítulo demonstra a construção de um jogo para Android, numa espécie de tutorial, sem entrar em detalhes concretos em relação a determinados aspetos não centrais para os objetivos deste livro.

5.2 PROJETO UNITY 3D PARA ANDROID

O primeiro passo a realizar consiste na criação de um novo projeto. Assim que se inicia o Unity3D é mostrada a janela da Figura 5.1, que permite aceder a projetos já desenvolvidos ou criar novos projetos. Permite também aceder a outros conteúdos disponibilizados para a aprendizagem de Unity.

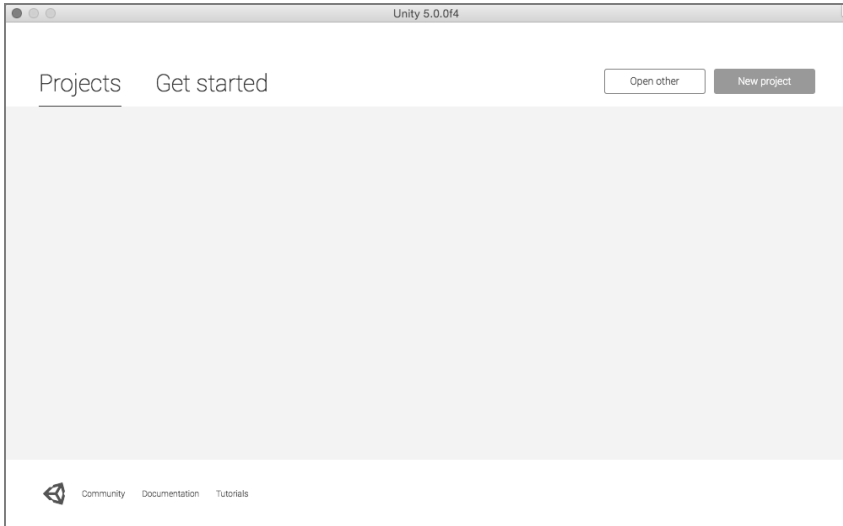


FIGURA 5.1 – Janela de início do Unity 5.0

Depois de escolhida a opção para criar um novo projeto (**New Project**), surgirá o assistente para a criação de um novo projeto, tal como apresentado na Figura 5.2.

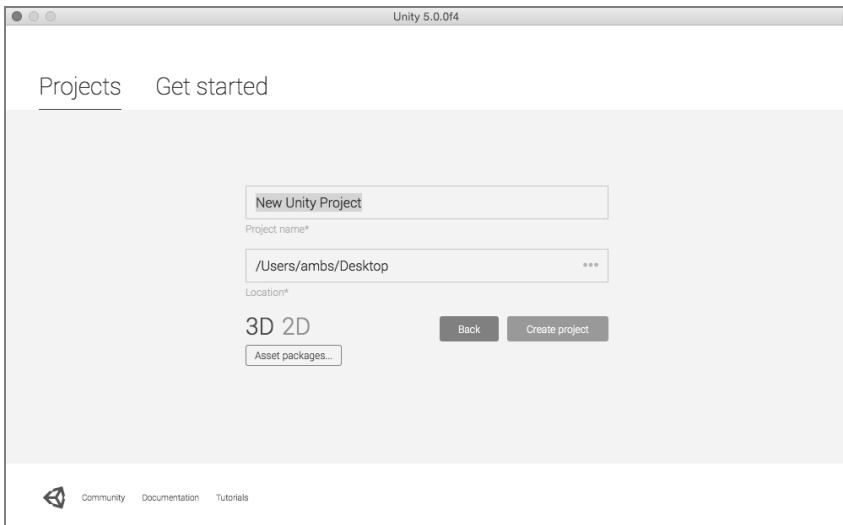


FIGURA 5.2 – Assistente de criação de projeto

Nesta janela serão indicados o nome do projeto e a pasta onde esse projeto deve ser criado. Note que, tal como noutras aplicações de desenvolvimento, um projeto Unity corresponde a uma pasta e não a um único ficheiro.

Também é possível selecionar se o jogo será baseado em 2D ou 3D. A verdade é que é possível iniciar um jogo 3D usando a opção 2D, e vice-versa. No entanto, esta escolha inicial permite que o Unity selecione um conjunto de opções por omissão que fazem mais sentido para esse tipo de jogos. No caso do *Arkadroid*, será usada a opção para 3D.

O botão **Asset packages...** permite escolher que pacotes extra (de funcionalidades disponibilizadas como extras, por exemplo, da loja de *Assets* do Unity) serão precisos durante o desenvolvimento do jogo. Embora seja possível escolher neste momento um grande conjunto de pacotes e, durante o desenvolvimento do jogo, usá-los ou não, é preferível ir adicionando os pacotes necessários ao longo do desenvolvimento, à medida que estes vão sendo precisos. Seguindo esta filosofia, não será selecionado nenhum pacote. Deverá ser indicada, apenas, a pasta na qual o projeto será criado.



Tal como na criação de um projeto no Android Studio, deverá ser indicada uma pasta não existente, que será criada e na qual o Unity 3D colocará todos os ficheiros relevantes. O uso de uma pasta e não de um ficheiro permite que o Unity 3D vá verificando, regularmente, se foram adicionados ou alterados ficheiros nessa pasta e, em caso afirmativo, que os vá importando de forma automática.

Ao criar o projeto, usando a opção **Create Project**, aparecerá uma interface semelhante à da Figura 5.3.

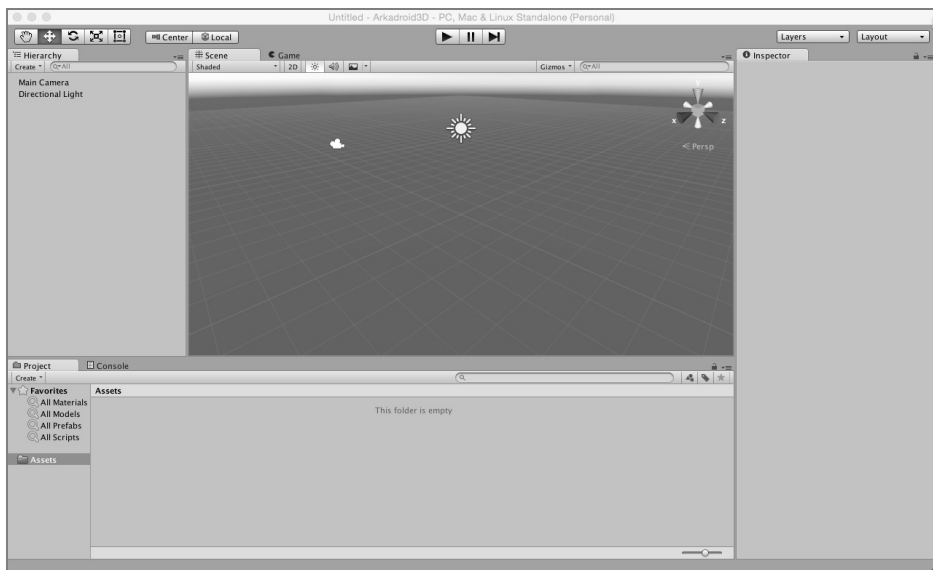


FIGURA 5.3 – Interface Unity 3D num novo projeto

A interface do Unity 3D é composta por vários separadores, que podem ser colocados em qualquer posição, tornando o desenvolvimento mais agradável e prático. Por exemplo, para quem desenvolve num computador com dois monitores é possível distribuir o ambiente por ambos.

Os principais separadores no ambiente Unity 3D são:

- **Scene** (cena) – apresenta a cena (ambiente 3D ou 2D) a ser editada, numa interface típica de modelador de objetos tridimensionais. Aqui é possível movimentar os objetos arrastando-os ou alterando o seu tamanho, a sua orientação ou a sua rotação, usando os cinco botões da barra de ferramentas para alternar entre as diferentes funcionalidades. Estes quatro botões podem ser acionados usando o teclado, através das teclas **Q**, **W**, **E**, **R** e **S**, respetivamente;
- **Game** (jogo) – mostra a cena atual renderizada. Ao usar a tecla de reprodução (botão **play**) este separador será selecionado automaticamente, para que o jogo seja ali executado;
- **Project** (projeto) – é semelhante à interface do explorador de ficheiros do Windows ou Mac OS X, e lista os vários ficheiros e pastas existentes no projeto Unity;
- **Hierarchy** (hierarquia) – apresenta, de forma hierárquica, os vários objetos presentes na cena atual. Permite que estes possam ser selecionados de forma eficiente, e que esta hierarquia possa ser alterada rapidamente, movendo os objetos com o rato. Este separador permite que se selecione um objeto, se mova o rato para o separador **Scene** e se possa premir a tecla **f**. Ao realizar isto, o Unity irá focar automaticamente o objeto, tornando a procura de elementos gráficos muito mais rápida;
- **Inspector** (inspetor) – semelhante aos inspetores de outras ferramentas, como o Microsoft Word ou o Microsoft Powerpoint, mostra os diferentes componentes e propriedades do objeto atualmente selecionado, permitindo que estes sejam alterados. Detalhes sobre o significado de componentes ou propriedades serão apresentados mais adiante;
- **Console** (consola) – apresenta mensagens de erro, de aviso ou de informação, sejam estas geradas pelo editor, pelo compilador ou pelo código desenvolvido.

Todos estes separadores podem ser movimentados para diferentes posições. No entanto, o Unity inclui um conjunto de modelos que poderão ser usados para rapidamente mudar a posição destes separadores. Estes modelos estão disponíveis em

Window → Layouts. Este menu também permite que se grave um novo modelo de acordo com o gosto pessoal do utilizador.



Cada uma das próximas secções deste capítulo corresponde a uma parte isolada na construção do *Arkadroid*. Isto significa que poderá e deverá ser usada a tecla de reprodução (vulgo botão **play**, que se encontra na zona de botões, ao centro) para verificar se todas as funcionalidades implementadas na respetiva secção estão presentes e funcionais. Por exemplo, neste momento o projeto já poderá ser executado. O resultado será uma cena vazia, com fundo azul.

5.3 CENA BASE

O jogo *Arkadroid* terá uma estrutura semelhante ao esquema da Figura 5.4. Existe um bloco (denominado *pad* para não se confundir com os outros blocos existentes na cena) que será controlado pelo utilizador usando o dedo ou o rato. O objetivo é driblar uma bola de modo a destruir todos os blocos que se encontram no fundo. Cada um destes blocos dá uma pontuação e autodestrói-se depois de um número variável de colisões com a bola. O jogador perderá “vidas” sempre que deixe escapar a bola.

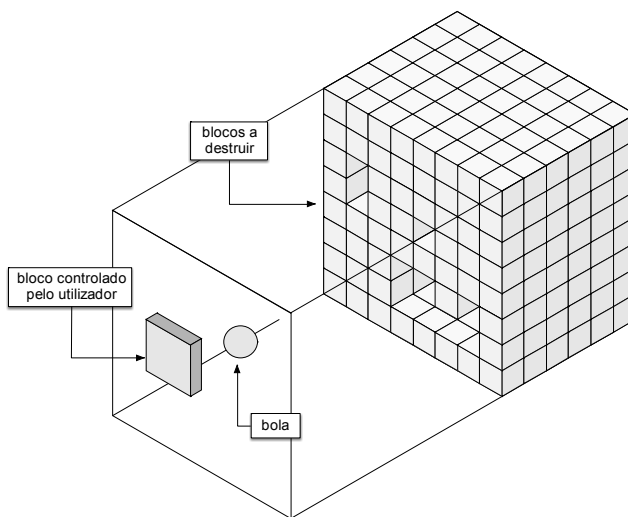


FIGURA 5.4 – Esquema isométrico do ambiente de jogo

Ao iniciar um projeto, a cena de jogo está vazia, apenas com uma câmara, que não pode ser apagada⁴⁸ e uma fonte de luz. A câmara, denominada *Main Camera*, permite indicar qual será a perspetiva que o jogador terá em relação ao mundo tridimensional. Por sua vez, a fonte de luz permite que o mundo esteja iluminado (*Directional Light*).

⁴⁸ Na verdade, que não **deve** ser apagada, exceto em situações específicas.

Nesta secção constrói-se a estrutura base do jogo. Embora no diagrama as paredes sejam transparentes, no jogo elas serão opacas, colocando o jogador dentro de uma caixa. Para a construção dessa caixa vão ser usados cinco planos, um para cada uma das quatro paredes e um último para o fundo da caixa.

Embora seja possível colocar objetos no mundo tridimensional de forma manual, arrastando-os e rodando-os, nesta situação serão apresentadas as posições, os ângulos de rotação e a escala que cada um dos objetos deve ter. Este conjunto de informação (posição, ângulos de rotação e escala) está presente em todo e qualquer objeto Unity 3D. Por exemplo, ao selecionar a câmara principal na hierarquia (*Main Camera*), no inspetor verá, entre outros, um primeiro componente (*Transform*) em que esta informação está presente (Figura 5.5).

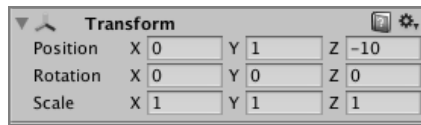


FIGURA 5.5 – Componente *Transform*

Para criar as paredes iremos adicionar planos, um de cada vez, usando a opção **GameObject → 3D Object → Plane**. Depois de criado, o plano deverá aparecer no separador **Scene**. Ao seleccioná-lo, aparecerá um sistema cartesiano que permite que este se possa arrastar. Na hierarquia também aparece um novo objeto, de nome *Plane*. Para mudar o nome do objeto basta seleccioná-lo na hierarquia e depois clicar uma vez com o rato. Serão necessários cinco planos, com as propriedades descritas na Tabela 5.1. Esta mesma tabela também indica a posição e a rotação a aplicar à câmara principal.

NOME DO OBJETO	POSIÇÃO	ROTAÇÃO	ESCALA
ParedeBaixo	0, 0, -5	90, 0, 0	2, 2, 10
ParedeEsquerda	-5, 0, 0	90, 90, 0	2, 2, 10
ParedeDireita	5, 0, 0	90, 270, 0	2, 2, 10
ParedeTopo	0, 0, 5	90, 180, 0	2, 2, 10
Fundo	0, -30, 0	0, 0, 0	2, 2, 2
MainCamera	0, 10, 0	90, 0, 0	1, 1, 1
Holofote	0, 15, 0	90, 0, 0	1, 1, 1

TABELA 5.1 – Posição, rotação e escala dos cinco planos, da câmara principal e da fonte de luz

Os valores apresentados podem ser calculados (tendo em conta uma caixa de largura 10 com profundidade 30), mas o mais natural é que os objetos sejam colocados de forma interativa até obter a estrutura desejada. No entanto, para que este tutorial seja fácil de seguir, é mais simples a colocação direta dos vários valores.

Ao colocar os valores na *Main Camera* aparecerá uma pequena janela com a vista a partir da câmara. Note que esta vista será muito escura por ainda não ter sido adicionada qualquer fonte de luz. Em todo o caso, pode-se alinhar a câmara usada no editor de cena com a *Main Camera*. Para isso, seleciona-se esta última e usa-se o menu **GameObject → Align View to Selected**.

Para terminar a criação da cena base é necessário alterar a fonte de luz. Para esta cena não será usada a iluminação por omissão, pelo que a fonte de luz já existente deverá ser apagada (depois de selecionada, usar a tecla **Delete**). Para o *Arkadroid* será usado um holofote (*spotlight*), uma fonte de luz forte e centrada. Para criar o holofote use a opção **GameObject → Light → Spotlight**.

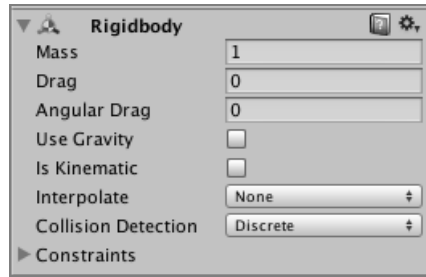
O holofote será colocado ligeiramente atrás da câmara, de acordo com as posições indicadas na Tabela 5.1. As fontes de luz no inspetor incluem, para além do componente *Transform*, um componente *Light* (luz) onde se pode alterar diferentes propriedades referentes à forma como a fonte de luz irá funcionar. Para o holofote do *Arkadroid* será usada uma amplitude (**Range**) de 25, com um ângulo (**Angle**) de 90, cor (**Color**) branca e intensidade (**Intensity**) 6. As restantes propriedades serão mantidas com os valores por omissão.

5.4 CORPOS RÍGIDOS

O próximo passo será a criação da bola. Mais uma vez vai ser usada uma primitiva 3D nativa do Unity: uma esfera. A criação da esfera é feita usando **GameObject → 3D Object → Sphere**. A esfera vai iniciar junto ao *pad* (que ainda não foi criado), na posição 0, 5.3, 0 e com um tamanho um pouco menor do que o original, usando uma escala de 0.6 para todos os eixos.

Para que a bola se movimente, e reaja a colisões, de acordo com o motor de física do Unity, é necessário adicionar à bola, para além dos componentes criados por omissão, o componente *Rigidbody* (corpo rígido). Isto é feito selecionando a bola e usando o menu **Component → Physics → Rigidbody**.

O componente de corpo rígido, apresentado na Figura 5.6, permite definir algumas propriedades, como sejam a massa do corpo ou a sua resistência (linear ou angular). No caso do *Arkadroid*, interessa que a bola não tenha gravidade, caso contrário irá cair no fundo da caixa e não mais mexer-se. Assim, para que funcione devidamente no contexto deste jogo, o uso da gravidade será desligado. Ao desligar a gravidade, a bola deixa de se mexer, o que também não é o objetivo. Pretende-se que ela se desloque para o fundo do caixote, mas saltite, e regresse até ao *pad*, subindo e descendo durante o jogo.

FIGURA 5.6 – Componente *Rigidbody*

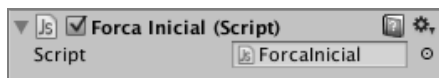
Para obter o comportamento descrito será necessário, por um lado, aplicar uma força inicial à bola para que ela se movimente e, por outro, alterar o seu material para que funcione como uma bola de borracha.

A força inicial terá de ser programada. A programação no Unity é feita em componentes de nome *script*, que são associados a um ou mais objetos. Para criar uma *script* seleciona-se a esfera e, por exemplo, a opção do menu **Component** → **Add...** Surgirá uma janela semelhante à da Figura 5.7. Nesta janela aparecem os diferentes tipos de componentes disponíveis, bem como uma opção para criar uma nova *script*: **New Script...**



FIGURA 5.7 – Adição manual de um componente

Após a seleção da opção para criar a *script*, o Unity solicita o seu nome. A *script* será chamada de *ForcaInicial*. A mesma janela pergunta qual a linguagem a usar. Como já foi referido neste capítulo, iremos usar JavaScript, pelo que esta opção deverá ser selecionada. A Figura 5.8 mostra o componente *Forca Inicial*.

FIGURA 5.8 – Componente *Forca Inicial*

Fazendo duplo clique sobre o nome da *script*, será aberto um editor externo (por omissão, o *MonoDevelop*). Neste momento, o código será apenas o seguinte:

```
public var forcaInicial : double = 15;

function Start () {
    var rigidbody = GetComponent.<Rigidbody>();
    rigidbody.AddForce(Vector3.down * forcaInicial, ForceMode.Impulse);
}
```

A primeira linha declara uma variável pública: uma variável que é editável por outras classes, ou pelo próprio utilizador usando a forma gráfica do componente (Figura 5.9). As variáveis são declaradas com o comando `var` seguido do nome da variável. Segue-se um carácter de dois pontos, o tipo de dados da variável (neste caso, um real de precisão dupla) e a atribuição inicial.

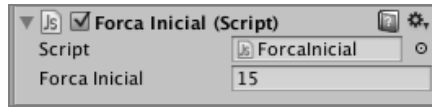


FIGURA 5.2 – Componente *Forca Inicial* após criação da *script*

A seguir é apresentado o método `Start`. Este método é invocado pelo Unity quando o componente em causa é inicializado na cena, no momento em que o jogo inicia a sua execução. Neste método acede-se ao corpo rígido do objeto que está associado a este componente (a bola) e adiciona-se-lhe uma força. O método `GetComponent` permite obter um determinado componente associado a um objeto. Por sua vez, o método `AddForce` permite adicionar-lhe a força, indicando que esta será no sentido descendente (usando para isso um vetor unitário disponível na classe `Vector3`) e com a força inicial atribuída à variável. Por sua vez, o segundo argumento indica que esta força não será constante, mas um impulso: será aplicada apenas uma vez ao objeto.

Repare-se, na Figura 5.9, que a variável `forcaInicial` ficou disponível no componente respetivo. Isto é possível porque a variável foi declarada como pública.



Depois de gravar a *script* no *MonoDevelop*, esta será compilada pelo Unity de forma automática. Passados alguns segundos aparecerão possíveis erros sintáticos na barra de estado (barra inferior) da janela do Unity. Torna-se assim bastante eficiente o processo de criação de uma *script* e da sua validação sintática. Do mesmo modo, o campo **Forca Inicial** mostrado anteriormente poderá demorar alguns segundos até aparecer.

O segundo passo corresponde à associação de um **material físico** à bola. Os materiais físicos definem como se devem compor as colisões entre diferentes objetos. Por exemplo, um material físico pode definir que um objeto se comporta como gelo, fazendo com que outros objetos deslizem sobre o primeiro. Ou, como no caso da bola, indicando que esta se deve comportar como se de borracha se tratasse.

A criação do material físico é feita usando o menu **Assets** → **Create** → **Physic Material**. Sugere-se a alteração do nome do material para **Borracha**.

Foge ao âmbito deste capítulo descrever os vários campos referentes à criação de um material. No caso deste material, reduz-se a fricção (*Friction*), e aumenta-se a elasticidade (*Bounciness*). A Figura 5.10 mostra os vários parâmetros que podem ser usados para configurar o comportamento de material físico.

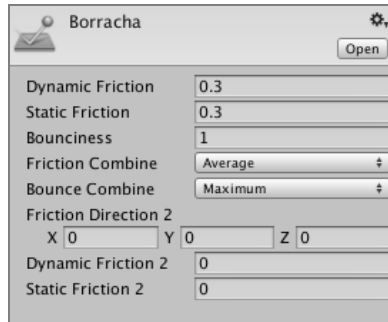


FIGURA 5.10 – Detalhes do material físico Borracha

Depois de criado o material, seleciona-se a bola e, no componente *Sphere Collider* (responsável por calcular colisões numa esfera), altera-se o campo **Material** para **Borracha**, usando o pequeno ponto à direita daquele campo. Ao clicar nesse ponto, abrir-se-á uma janela com diferentes materiais disponíveis. Deverá ser selecionado o material **Borracha**. O estado do componente *Sphere Collider* é apresentado na Figura 5.11.



FIGURA 5.11 – Componente *Sphere Collider* após seleção do material

5.5 INTERAÇÃO COM O UTILIZADOR

Nesta secção será criado o *pad*, que permitirá ao utilizador jogar. Para além disso, também será implementada a funcionalidade de permitir que o utilizador possa terminar a execução do jogo, saindo da aplicação.

O *pad* não é mais do que um cubo, desproporcionado. Para criá-lo utiliza-se o menu habitual: **GameObject** → **3D Object** → **Cube**. Este cubo será colocado na posição 3.5, 6, 0, de modo a ficar encostado a uma das paredes e a uma distância da câmara suficiente para permitir que seja controlado com o dedo do utilizador. A escala do cubo

será 3, 0.2, 3. Mais tarde, este cubo terá de ficar translúcido, mas isso será discutido posteriormente, quando se proceder à associação de texturas aos vários objetos em cena.

Embora para os objetos criados anteriormente isso não tenha sido relevante, neste caso é importante que se altere o nome do objeto (na hierarquia) para *pad*. Isto é importante porque algumas *scripts* irão referir-se a este objeto por este nome.

Para o movimento do *pad*, há que fazer uma pequena chamada de atenção: embora a maior parte dos dispositivos Android seja usado com o dedo ou uma caneta genérica, alguns dispositivos usam canetas específicas, que são interpretadas pelo Unity como sendo um movimento do rato, e não do dedo. Assim, para que o jogo seja jogável nos vários dispositivos será necessário ter em consideração ambas as situações.

O movimento do rato pode ser controlado usando métodos da classe `Input`. O método `GetMouseButtonDown` retorna um valor verdadeiro quando determinado botão do rato é premido e o `GetMouseButtonUp` quando o botão do rato é largado.

Para além destes dois métodos, existe uma variável de classe com tipo `Vector3` que permite obter a posição do rato: `Input.mousePosition`.

De forma semelhante à apresentada na secção 5.4, será criada uma *script* de nome *mover*, associada ao *pad*, que irá verificar se o rato foi usado.



Esta e outras *scripts* serão construídas de forma progressiva. Será sempre indicado em que posição o código apresentado deve ser introduzido. Em caso de dúvida deverá consultar o código disponibilizado na página do livro em www.fca.pt, até este se esgotar ou ser publicada nova edição atualizada ou com alterações.

O código seguinte mostra a primeira etapa da *script*. As duas linhas iniciais definem duas variáveis privadas, o que significa que não são visíveis no inspetor, nem alteráveis por qualquer outro código. A primeira, booleana, será usada para saber se o rato se encontra premido e no sítio correto (sobre o *pad*). A segunda irá armazenar a posição do rato a cada momento. O método `Update` é invocado pelo Unity de forma automática sempre que seja necessário renderizar uma *frame* (termo usado para referir cada imagem que é produzida e enviada para o monitor). Um jogo típico produz entre 30 a 60 *frames* por segundo, sendo que quantas mais conseguir produzir mais natural será o movimento quando observado por um ser humano. Assim sendo, ao escrever código em métodos `Update` deve-se tentar reduzir o peso computacional ao máximo:

```
private var ratoAtivo : boolean = false;
private var ratoUltimaPosicao : Vector3;
function Update () {
    if (Input.GetMouseButtonDown(0)) {
        ratoUltimaPosicao = Input.mousePosition;
        if (noPad(ratoUltimaPosicao)) ratoAtivo = true;
    }
}
```

```
if (ratoAtivo) {
    var delta = Input.mousePosition - ratoUltimaPosicao;
    ratoUltimaPosicao = Input.mousePosition;
    moverPad(delta);
}

if (Input.GetMouseButtonUp(0)) ratoAtivo = false;
}
```

Neste momento, o método é composto por três partes distintas, cada uma protegida por uma instrução condicional.

A primeira estrutura condicional é ativada quando o utilizador pressiona o botão esquerdo do rato (ou usa a caneta no dispositivo móvel). O valor 0 corresponde ao primeiro botão. Quanto isto acontece é guardada a posição atual do rato e é invocado o método `noPad`, que iremos definir de seguida, que valida se o clique foi feito sobre o *pad*. Se assim for, a variável `ratoAtivo` ficará com um valor verdadeiro.

A segunda estrutura condicional é executada quando o botão do rato se encontra premido. Neste caso, é obtida a posição atual do rato e comparada com a posição que estava guardada, obtendo um vetor de movimento (variável `delta`). De seguida, é atualizado o valor da posição do rato e é invocado o método `moverPad`, que é responsável por calcular a nova posição para o *pad* com base no vetor calculado. Este método será implementado mais tarde.

Finalmente, a terceira estrutura condicional é executada quando o botão do rato é largado, pelo que se deve atualizar a variável, indicando que o rato já não se encontra ativo.

Antes de implementarem os dois métodos em falta, será implementado o movimento pelo dedo, que também será usado pelos dois métodos em falta. Para isso, será necessário alterar o código anterior. Em primeiro lugar, adiciona-se⁴⁹ uma variável booleana que controla se o dedo foi usado:

```
private var dedoAtivo : boolean = false;
```

Para além disso, será necessário adicionar outros três blocos condicionais no método `Update`, por exemplo, antes do fecho da última chaveta. Estes três blocos condicionais são apresentados no código seguinte. Mais uma vez, os três blocos correspondem aos três movimentos: colocar o dedo no ecrã, mover o dedo ou levantar o dedo. Para controlar o movimento do dedo usa-se a variável `touchCount` da classe `Input`, que indica quantos dedos são detetados no ecrã, e o método `GetTouch`, que retorna informação sobre esses mesmos dedos.

⁴⁹ A variável deve ser definida no início da *script*, juntamente com as outras variáveis já definidas.

A estrutura retornada por este método inclui a posição e a fase de movimento para cada um dos dedos detetados:

```
if (Input.touchCount > 0 &&
    Input.GetTouch(0).phase == TouchPhase.Began) {
    var infoToque = Input.GetTouch(0);
    if (noPad(new Vector3(infoToque.position.x,
                        infoToque.position.y, 0)))
        dedoAtivo = true;
}
if (dedoAtivo && Input.GetTouch(0).phase == TouchPhase.Moved) {
    infoToque = Input.GetTouch(0);
    moverPad(new Vector3(infoToque.deltaPosition.x,
                        infoToque.deltaPosition.y, 0));
}
if (Input.touchCount > 0 &&
    Input.GetTouch(0).phase == TouchPhase.Ended)
    dedoAtivo = false;
```

A primeira estrutura condicional verifica se existe pelo menos um dedo sobre o ecrã e, se assim for, obtém a informação sobre o primeiro desses dedos, ou seja, o dedo número 0. Se a fase do movimento corresponder à posição do dedo no ecrã, então será obtida a informação desse dedo, guardando-a na variável `infoToque`, que será usada na condição seguinte para obter a posição atual do dedo. Ao contrário da posição do rato, que é um vetor tridimensional, a posição do dedo é um vetor bidimensional. Isto obriga a que, para invocar o método `moverPad`, tenha de se criar um vetor tridimensional com as duas coordenadas da posição do dedo e o valor 0, de modo a completar o vetor. Ou seja, o que se está a validar é se o utilizador colocou o primeiro dedo sobre o *pad* e, nessa situação, a ativar a variável `dedoAtivo`.

O segundo bloco é equivalente ao bloco que deteta o movimento do rato. Neste caso, verifica-se se o dedo se encontra ativo e se foi movido (fase de movimento). No entanto, neste caso não é necessário guardar a posição do dedo, já que o Unity calcula isso automaticamente. O vetor de movimento pode ser obtido a partir da informação do dedo usando a variável `deltaPosition`.

Finalmente, o terceiro bloco deteta se o utilizador levantou o dedo e, nessa situação, desativa o movimento do bloco.

Segue-se a definição do método `noPad`, que verifica se o dedo foi colocado sobre o *pad*. Este método usa uma técnica denominada *ray casting*, que poderá ser traduzida por “lançamento de raios”. Corresponde a criar uma linha virtual a partir de um ponto (neste caso, um ponto do ecrã) e em determinada direção (neste caso, uma direção ortogonal ao plano do ecrã), e verificar se essa linha intersesta algum objeto (neste caso, o *pad*):

```
private function noPad(v : Vector3) {
    v.x /= Screen.width;
    v.y /= Screen.height;
    var r : Ray = Camera.main.ViewportPointToRay(v);
```



```

var h : RaycastHit;
return Physics.Raycast(r, h) && h.collider.gameObject.name == "pad";
}

```

Este método é definido como *private*, o que indica que não pode ser invocado por outros objetos. O único argumento deste método é um vetor tridimensional que indica a posição do rato ou do dedo sobre o ecrã. Note-se que o ecrã é bidimensional, pelo que a terceira posição do vetor pode ser ignorada.

A posição obtida no método `Update` é um par de coordenadas em pixels. Como o número de pixels varia de acordo com o dispositivo a ser utilizado, é necessário converter estas coordenadas de modo a que sejam relativas, no intervalo entre 0 a 1, em que a posição 0 corresponde à margem esquerda, ou fundo, e a posição 1 à margem direita, ou topo. Esta conversão é simples, bastando para isso dividir o valor recebido pelo tamanho respetivo da largura ou altura do ecrã.

De seguida, calcula-se uma linha perpendicular à câmara, com início no ponto em que o dedo ou o rato iniciou o movimento. Este raio virtual sai da câmara e irá validar se interseta, ou não, o *pad*. Em caso positivo, o dedo encontra-se sobre o *pad*. Para obter o raio usa-se o método `ViewportPointToRay` aplicado à câmara principal.

A variável `h` do tipo `RaycastHit` irá armazenar informação sobre a primeira interseção do raio com um objeto da cena. O método `Physics.Raycast` é responsável por “lançar” o raio. Se este tiver intersetado um objeto, o método retorna um valor verdadeiro. Assim, se houver interseção, é necessário verificar qual o objeto que o raio intersetou. É possível obter o nome do objeto usando uma série de indireções. Do objeto `h` obtém-se o *collider* intersetado pelo raio; deste, obtém-se o objeto que contém esse *collider*; e, finalmente, obtém-se o nome desse mesmo objeto. Se esse nome for *pad*, é porque o dedo foi colocado sobre o *pad* e, portanto, o método deverá retornar um valor verdadeiro.

Por sua vez, o método `moverPad` é apresentado no código seguinte. Este método recebe o vetor de movimento do dedo (ou do rato) e aplica-o ao *pad*. Antes de alterar a posição do objeto (que está presente no componente *Transform*), são definidos dois fatores usados para controlar a relação do movimento do dedo com o movimento do *pad*. O valor definido é uma aproximação a estes fatores, embora diferentes utilizadores possam preferir valores ligeiramente diferentes.

```

private function moverPad(d : Vector3) {
    var Xfactor : double = 0.03;
    var Yfactor : double = 0.03;
    transform.position.x += d.x * Xfactor;
    transform.position.z += d.y * Yfactor;
}

```

Com a adição do código anterior, o *pad* já se movimenta, mas não para quando se encosta a uma parede. Há muitas formas de resolver esse problema, mas uma delas é usar o método `LateUpdate`, que é semelhante ao `Update`, sendo também executado a

cada *frame*, mas com a garantia de que só é executado após todos os métodos `Update` de todos os objetos terem já sido executados.

```
function LateUpdate() {
    this.transform.position.z =
        Mathf.Clamp(this.transform.position.z, -3.5, 3.5);
    this.transform.position.x =
        Mathf.Clamp(this.transform.position.x, -3.5, 3.5);
}
```

O código anterior trata desse problema, usando a função `Clamp`. Esta função recebe um valor e um intervalo, e garante que esse valor se encontra dentro do intervalo. Ou seja, para os exemplos do código, se o valor se encontrar entre -3.5 e 3.5, então mantém-se inalterado. Mas se for superior a 3.5, a função retorna 3.5, e caso seja inferior a -3.5, a função retorna -3.5. É uma forma prática de garantir que determinado valor se encontra, sempre, em determinado intervalo.

Portanto, na situação em causa, quando o *pad* se encosta à parede terá a sua posição *z* ou *x* com valores 3.5 ou -3.5 e, nessa situação, os valores não poderão subir, ou descer, respetivamente.

Antes de terminar esta secção, será implementada uma *script* muito simples para permitir ao utilizador terminar o jogo. Será uma nova *script*, de nome *sair*, a ser criada no objeto *pad*.

```
function Update () {
    if (Input.GetKeyDown(KeyCode.Escape)) {
        Application.Quit();
    }
}
```

Esta *script* apenas verifica que determinada tecla foi premida. Neste caso, valida se a tecla que foi premida corresponde à tecla **escape** e, em caso afirmativo, invoca o método para sair da aplicação. Note-se que esta funcionalidade só irá funcionar quando o jogo estiver a ser executado na sua plataforma final, já que o método `Quit` não produz qualquer efeito quando o jogo é executado dentro do Unity.

5.6 INTERFACE COM O UTILIZADOR

Neste momento, quando o jogador perde uma bola não acontece nada. No entanto, é suposto que o jogador tenha um número máximo de “vidas” e que, sempre que perca uma “vida”, a bola seja recolocada em jogo.

Ao adicionar “vidas” a um jogo é necessário informar o utilizador do número de “vidas” ainda restantes. Existem várias formas para o fazer. Alguns jogos mostram vários ícones, em que cada um corresponde a uma “vida”, outros apenas um valor de “vidas” restantes. Será escolhida a segunda solução, por uma questão de simplicidade.

Para apresentar o número de vidas será usado um *GUIText*, um objeto que permite a colocação de texto sobre a imagem 3D renderizada. A criação deste tipo de objeto mudou nas versões recentes do Unity e é provável que volte a mudar em breve. Em todo o caso, a sequência de comandos **GameObject** → **Create Empty** e **Component** → **Rendering** → **GUI Text** deverá funcionar em qualquer versão. Para que mais tarde seja fácil referir este objeto, será batizado de *vidas*.

Este objeto tem características muito próprias, e distintas dos restantes objetos criados até agora. Por exemplo, a posição indicada no componente *Transform* não corresponde a uma posição na cena, mas a um valor relativo ao ecrã, indicado no intervalo entre 0 e 1. Além disso, o objeto não é corretamente visível no separador **Scene**.

O texto com as “vidas” será colocado no canto superior esquerdo. O nome do *GUIText* deve ser alterado para *vidas*, e a sua posição no componente *Transform* será 0, 1, 0, sendo que o primeiro elemento indica a posição no eixo horizontal (à esquerda) e o segundo elemento a posição no eixo vertical (no topo). O componente específico referente ao *GUIText* é apresentado na Figura 5.12.



FIGURA 5.12 – Componente *GUIText*

Este componente inclui vários campos. O texto (**Text**) corresponde ao texto que deve ser apresentado no ecrã. No caso do *Arkadroid*, esse texto será dinâmico, pelo que pode ficar vazio.

Por sua vez, a âncora (**Anchor**) corresponde à posição do *GUIText* que será usada para o colocar na posição indicada (no componente *Transform*). Ou seja, ao colocar texto no canto superior esquerdo do ecrã, é normal que se escolha a âncora **upper left**. Ao colocar alguns alinhada à margem direita, possivelmente **middle right** seria suficiente. Neste caso, será usado **upper left** (Figura 5.13).

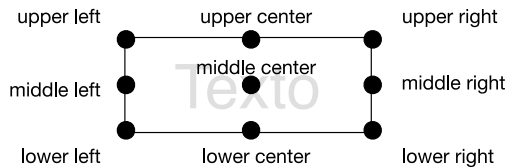


FIGURA 5.13 – Diferentes posições das âncoras num *GUI Text*

Segue-se o alinhamento do texto (**Alignment**) equivalente ao alinhamento de texto num editor de texto comum. Neste caso, será mantido o alinhamento à esquerda.

O campo seguinte, **Pixel Offset**, permite gerar alguma margem, em pixels, à volta do texto. Note-se que fazê-lo de forma proporcional ao ecrã, usando a posição no componente *Transform*, pode não ser suficientemente flexível.

De seguida, é possível especificar o espaçamento entre linhas (**Line Spacing**), desnecessário quando o texto tem apenas uma linha, e a quantos espaços corresponde um carácter de tabulação (**Tab Size**). Mais relevantes são os parâmetros seguintes, onde se pode escolher o tipo de letra (**Font**), um material a ser usado na renderização do texto (**Material**, mas que não será usado neste caso), o tamanho de letra (**Font Size**), o estilo (**Font Style**) e a cor (**Color**) que será usada para o texto. Os campos seguintes têm que ver com o tratamento do texto (**Antialiasing**) e o suporte, ou não, de formatação através de comandos *RichText*.

No caso do *Arkadroid*, será importante adicionar o **Pixel Offset** de **4** e **-4**, escolher o tipo de letra por omissão (**Arial**) e o tamanho (por exemplo, **15**) e mudar a cor a gosto.

Neste objeto *GUIText* será criada uma nova *script* responsável por atualizar, no ecrã, o número de “vidas” restantes. Essa *script* terá o nome `contarVidas`, com o seguinte conteúdo:

```
public var vidas: int = 4;
function Update () {
    GetComponent.<GUIText>().text = vidas +
        ( vidas > 1 ? " Vidas" : " Vida");
}
```

O código desta *script*, que ainda vai ser alterado, é bastante simples. É definida uma variável pública (que poderá ser atualizada por outros objetos da cena) que contará o número de “vidas” disponíveis (neste caso, 4). No método `Update` é apenas atualizado o texto apresentado no *GUIText*: o número de “vidas” seguido da palavra **vidas** ou **vida**, consoante exista mais do que uma “vida”, ou apenas uma “vida” disponível. Para isso é usado o método `GetComponent`, que permite obter um componente associado ao objeto atual.

Para contar quantas “vidas” o utilizador já perdeu será necessário contar o número de vezes que o utilizador não conseguiu bater na bola com o *pad*. A melhor forma de o fazer é criar um objeto virtual, que não é renderizado, mas que é sensível a colisões

ou a gatilhos (*triggers*). Um *trigger* permite detetar se realmente existe uma colisão, mas o efeito da colisão (a bola fazer ricochete) não é produzido. Ou seja, apenas temos informação se a bola passou, ou não, por **dentro** de determinado objeto. Neste caso, será um objeto vazio, apenas com um *collider* associado (um detetor de colisões).

O objeto vazio é criado usando o menu **GameObject → Create Empty**. Note-se que um objeto vazio não é mais do que um ponto (transparente) na cena. Em termos de componentes, só contém o *Transform*. A adição do *collider* é realizada depois de seleccionar o objeto vazio usando o menu **Component → Physics → Box Collider**. Em termos de configuração, este objeto será colocado na posição $0, 13, 0$, que corresponde a uma posição atrás do *pad*. Em termos de tamanho, será necessário que cubra toda a área por onde a bola possa passar, pelo que será usada a escala $10, 0.1, 10$. Neste caso colocamos um *collider* pouco espesso, já que corresponde, praticamente, a um plano que ativará um método quando algum objeto o atravessar. Iremos denominar este objeto baliza.

Ao olhar com atenção para as posições dos vários objetos, pode-se ver que este *collider* ficou entre a câmara e o *pad*. Ora, ao controlar a interação com o utilizador estamos a usar um raio, que sai da câmara e que, esperamos, colida com o *pad*. Ao colocar um objeto no meio, tudo fica estragado. Felizmente, o Unity permite que se indique que determinado objeto não deve ser considerado para efeitos de *ray casting*. Esta opção é ativada seleccionando o objeto e, no topo do inspetor, mudar a camada (**Layer**) para ignorar raios (**Ignore Raycast**) – Figura 5.14.

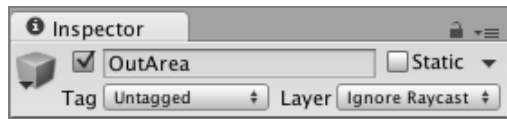


FIGURA 5.14 – Topo do inspetor com a camada Ignore Raycast ativa

Antes de adicionar algum código a este objeto, falta apenas indicar ao *collider* que se deve comportar como um *trigger*, ligando a opção **Is Trigger** ao inspetor do objeto. Neste objeto vamos ainda adicionar uma *script*, chamada *foraDeJogo*, que será responsável por processar uma bola perdida pelo utilizador. O código completo desta *script* é o seguinte:

```
public var contador : contarVidas;
public var bola : ForcaInicial;

function OnTriggerEnter(other: Collider) {
    contador.vidas--;
    Handheld.Vibrate();
    bola.restart();
}
```

Inicialmente, são declaradas duas variáveis: uma referente ao *GUIText* que apresenta o número de “vidas” restantes e outra referente à bola. Neste caso, estamos a declarar essas duas variáveis, não com o tipo de dados mais natural, mas com o nome de duas *scripts* que associamos a esses mesmos objetos. Isto permite que do nosso código possamos aceder diretamente a variáveis ou métodos públicos dessas *scripts*.

O método `OnTriggerEnter` é invocado pelo Unity quando um objeto entra na área de um *collider* configurado para funcionar como *trigger*. O parâmetro do método é o objeto que colidiu. No caso presente sabe-se que o único objeto que pode ter colidido é a bola e, portanto, não será necessário garantir que foi esse o objeto a colidir.

O conteúdo do método corresponde, apenas, a descontar uma “vida” no total de “vidas” disponíveis, fazer o dispositivo vibrar e invocar o método `restart` na bola (método que ainda não foi definido). Este método é responsável por atualizar a posição da bola, de modo a iniciar novo jogo (caso contrário a bola não voltaria para a área de jogo). De seguida, mostra-se o código reescrito da *script* `ForcaInicial` que está associada à bola:

```
public var forcaInicial : double = 15;
function Start () {
    restart();
}
public function restart () {
    transform.position = Vector3(0, 5.3, 0);
    var rigidbody = GetComponent.<Rigidbody>();
    rigidbody.velocity = Vector3.zero;
    rigidbody.angularVelocity = Vector3.zero;
    rigidbody.AddForce(Vector3.down * forcaInicial, ForceMode.Impulse);
}
```

Até ao momento, esta *script* tinha apenas o método `Start`, que adicionava uma força à bola. Agora, sempre que o jogador perde a bola, ela deverá ser recolocada em jogo, e a força inicial deve ser, também, adicionada. Assim, foi criado um método público, `restart`, que altera a posição do objeto para a posição inicial que tinha sido definida na secção 5.4. Para além disso, é removida qualquer aceleração que o objeto tenha nesse momento (seja velocidade linear ou angular) e, finalmente, aplica-se a mesma força inicial que já era aplicada na versão anterior da *script*.

Antes de terminar esta secção, é necessário fazer a associação da bola e do *GUIText* à *script* `foraDeJogo`. Para isso, deve-se selecionar o objeto `baliza`. No inspetor da *script* existem dois campos, denominados **Bola** e **Contador**. Deve arrastar, do separador **Hierarchy**, a esfera para o campo **Bola** e o objeto `vidas` para o campo **Contador**.

5.7 USO DE CENAS DISTINTAS

Nesta secção serão adicionadas, para além da cena atual (que corresponde à parte jogável) duas novas cenas: uma para servir de título do jogo e outra para servir de aviso quando o jogador perde todas as “vidas” (*game over*).

O primeiro passo corresponde a gravar a cena atual, usando o menu **File → Save Scene**, e o nome `jogo`. De seguida, será criada uma nova cena com **File → New Scene**. Note-se que, quando se muda de cena ativa, os objetos apresentados na hierarquia são atualizados para aqueles que se encontram na cena atual.

Nesta cena será usada uma imagem externa para servir de logótipo do jogo. A sua criação poderá ser feita numa aplicação como o Gimp ou o Adobe Photoshop. Para dar um toque mais profissional usou-se um tipo de letra inspirado no jogo *Arkanoid* original, que pode ser descarregado⁵⁰ livremente. A imagem usada é apresentada na Figura 5.15. Sugere-se o uso de uma imagem com fundo transparente, armazenada no formato *Portable Network Graphic* (PNG).



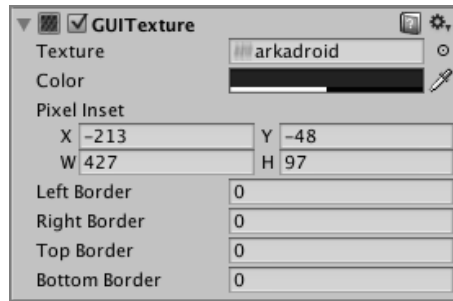
FIGURA 5.15 – Imagem criada numa aplicação externa

A importação da imagem no Unity não pode ser mais simples. Tanto pode ser usado o explorador do Windows (ou do Mac OS X) para colocar o ficheiro de imagem na pasta **assets** dentro da pasta do projeto Unity, como se pode arrastar a imagem diretamente para o separador **Project**.

A colocação da imagem na cena será conseguida graças ao uso de uma *GUITexture*. A ideia é semelhante à de uma *GUIText*, mas em que será usada uma imagem em vez de um texto. A *GUITexture* cria-se usando **GameObject → Create Empty**, seguido de **Components → Rendering → GUI Texture**. Tal como no caso da *GUIText*, a posição deste objeto é relativa ao ecrã. Assim, para centrar a imagem na área de jogo será usado o par de coordenadas `0.5, 0.5`. Também será necessário alterar a escala do objeto, colocando todas as coordenadas a zero.

A Figura 5.16 mostra o componente *GUITexture* do objeto acabado de criar. O primeiro campo (**Texture**) corresponde à imagem a ser usada. Depois de ter sido importada, é possível selecionar a imagem arrastando-a para o campo em causa ou usando o pequeno círculo à direita do campo. Neste caso, a cor é irrelevante.

⁵⁰ Disponível em: <http://www.fontspace.com/freaky-fonts/arkanoid>

FIGURA 5.16 – Componente *GUITexture*

Os quatro campos seguintes (**Pixel Inset**) correspondem a uma forma de indicar qual a posição relativa da imagem em relação à posição indicada no componente *Transform*. Em primeiro lugar, os campos **W** e **H** correspondem ao tamanho, largura e altura, em pixels, desejados para a imagem. Note-se que o tamanho indicado poderá ser diferente do tamanho original da imagem (maior ou mais pequeno). Também se deve ter cuidado em usar uma largura e altura que mantenham o aspeto da imagem original. Os campos **X** e **Y** indicam a posição relativa do pixel do canto superior esquerdo da imagem em relação à posição indicada. Ou seja, supondo que se deseja centrar a imagem (como é o caso), o valor de **X** deve ser um valor negativo correspondente a metade da largura da imagem, e o valor de **Y** um valor negativo correspondente a metade da altura da imagem.

Finalmente, os últimos quatro campos permitem definir uma margem, em pixels, à volta da imagem usada.

Para terminar a cena basta escolher a cor de fundo. Para isso, deve ser selecionada a câmara principal na hierarquia (*Main Camera*), alterado o campo **Clear Flags** para **Solid Color**, de modo a que o fundo se transforme numa cor única, e alterado o campo **Background** para a cor desejada.

Nesta altura já é possível experimentar a cena, executando-a. Deverá aparecer a imagem importada sobre um fundo da cor escolhida. Falta, no entanto, que seja possível clicar em algum sítio da imagem para se passar ao jogo. Para isso, será associada uma *script* de nome `temporizador` a um dos elementos da cena, por exemplo, ao *GUITexture*. Esta *script* irá mudar automaticamente para a cena de jogo passados alguns segundos, ou então após o utilizador ter clicado no ecrã (com o rato ou com um dedo).

De seguida, mostra-se o código completo da *script* criada. O temporizador é criado usando uma variável de precisão dupla (`double`) que, em cada *frame* (ou seja, sempre que o método `Update` é invocado), é incrementada com o número de segundos que decorreram desde que este mesmo método foi invocado pela última vez. Este valor é obtido usando o método `deltaTime` da classe `Time`.

```
private var temporizador : double = 0;
function Update () {
```



```
temporizador += Time.deltaTime;

if (temporizador >= 10 ||
    Input.GetMouseButtonDown(0) || Input.touchCount > 0)
    Application.LoadLevel("jogo");

if (Input.GetKeyDown(KeyCode.Escape))
    Application.Quit();
}
```

A primeira estrutura condicional verifica se o temporizador chegou aos 10 segundos, ou seja, se o utilizador clicou no rato ou se existe pelo menos um dedo a tocar no ecrã. Se qualquer uma destas condições for verdadeira, então será carregada a cena de nome `jogo`, que gravamos no início desta secção.

Por sua vez, a segunda estrutura condicional é usada para terminar a aplicação caso o jogador use a tecla **escape**.

Com isto fica terminada a cena de início do jogo, que pode ser gravada usando um qualquer nome, como, por exemplo, `inicio`.



Embora o código já devesse funcionar, existe um detalhe final necessário para que o Unity seja capaz de, dinamicamente, carregar cenas. Essa explicação aparecerá logo após a criação da cena de fim de jogo (*game over*).

A cena de *game over* será pouco diferente. Para a criar usa-se a mesma opção usada anteriormente e, mais uma vez, obtém-se uma cena vazia. Aqui, em vez de se usar uma imagem, será usado diretamente texto, com um *GUIText*. Após a criação do objeto vazio e a adição do componente *GUIText*, define-se o seu texto como *Game Over*, define-se um tamanho de letra e cor a gosto e coloca-se o texto centrado, usando as coordenadas 0.5, 0.5 no componente *Transform*, definindo a âncora no componente *GUIText* como sendo **middle center** e escolhendo o alinhamento centrado.

De seguida, e tal como foi feito na cena de início, será seleccionada a câmara e alterada a forma de preencher o fundo, bem como a cor a usar. Aqui será usado um tom avermelhado, já que corresponde à perda do jogo.

Será também necessário permitir que o jogador possa voltar ao jogo (clicando no ecrã) ou sair da aplicação. Mais uma vez, este comportamento vai ser obtido usando uma *script* (de nome `voltar`) muito semelhante à usada na cena inicial, tal como apresentada de seguida. Esta *script* pode ser associada a qualquer um dos objetos da cena.

```
function Update () {
    if (Input.GetKeyDown(KeyCode.Escape))
        Application.Quit();

    if (Input.GetMouseButtonDown(0) || Input.touchCount > 0)
        Application.LoadLevel("jogo");
}
```

Para terminar falta um pequeno detalhe. Durante o jogo, quando o número de “vidas” é esgotado, será necessário apresentar a cena de *game over*. Para isso, grava-se a cena atual como *gameover* e abre-se novamente a cena *jogo*. A forma mais simples de abrir uma cena é usar o separador **Project** e escolher a cena em causa.

A contagem de “vidas” está a ser feita no objeto de nome *vidas*, na *script* de nome *contarVidas*. Agora terá de ser alterada para que quando o número de “vidas” for esgotado seja carregada a nova cena:

```
public var vidas: int = 4;
function Update () {
    GetComponent.<GUIText>().text = vidas +
    ( vidas > 1 ? " Vidas" : " Vida");
    if (vidas <= 0)
        Application.LoadLevel ("gameover");
}
```

Se neste momento o jogo for experimentado, quando o Unity tiver de carregar uma nova cena irá dar erro. Para que isso não aconteça é necessário indicar quais as cenas que devem ser preparadas para carregamento e qual a ordem (tipicamente apenas a primeira é relevante, já que é a cena que é arrancada no início da execução do jogo). Essa configuração é feita no menu **File → Build Settings...** (Figura 5.17).



FIGURA 5.17 – Opções de construção do executável

Nesta janela devem ser colocadas as três cenas que criámos, sendo que a primeira deverá ser a *inicio*. Para isso pode-se arrastar as cenas do separador **Project**. Depois de colocar as três cenas, poderá fechar a janela e experimentar o jogo. Note que, embora se

tenha indicado que a cena `inicio` é aquela que deve ser executada em primeiro lugar, o Unity irá executar a cena que está carregada naquele momento (de modo a não obrigar o programador a percorrer todo o jogo para experimentar uma determinada cena que só aparece no final). Para experimentar a funcionalidade completa deve ser aberta a cena `inicio` e executado o jogo.

5.8 PREFABRICADOS

O *Arkadroid* está praticamente funcional. Fica a faltar pouco mais do que um conjunto de blocos que se possam destruir usando a bola. Para introduzir alguns conceitos novos os blocos serão construídos de forma completamente automática, de modo a que cada jogo seja um jogo possivelmente diferente.

Em primeiro lugar, será criado um cubo específico, na cena de jogo. Será adicionado comportamento a esse cubo através de uma *script*. Depois, esse cubo será promovido a um modelo, que no Unity é designado por *Prefab*, uma abreviatura de prefabricado. O cubo será apagado da cena, e depois será uma outra *script* que irá criar novas instâncias deste modelo.

Em paralelo, será criado um novo *GUIText*, a colocar no canto inferior direito do ecrã de jogo, com o total de pontos obtidos até ao momento.

Este será o primeiro passo: criar um *GUIText*, usando a sequência de comandos já apresentada, e batizá-lo de `pontos`. A sua posição, no componente *Transform*, é `1, 0, 0`, o que corresponde ao canto inferior direito. Para que esta posição funcione corretamente será necessário alterar a âncora para **lower right** e o alinhamento para **right**. Sugere-se o uso de uma cor e um tamanho de letra semelhantes aos usados para o *GUIText* com as “vidas”.

De seguida, apresenta-se o código da *script* `contarPontos`, associada a este *GUIText*. Tem uma variável privada, de nome `pontos`, que irá armazenar os pontos obtidos pelo jogador. O método `Update`, executado a cada *frame*, limita-se a atualizar o texto mostrado no ecrã. Por sua vez, o método `incrementar` recebe um número de pontos e incrementa esse valor ao contador de pontos.

```
private var pontos: int = 0;

function Update () {
    GetComponent.<GUIText>().text = "Pontuação: " + pontos;
}

public function incrementar(pts: int) {
    pontos += pts;
}
```

O próximo passo será a criação de um cubo que será, mais tarde, convertido num *Prefab*. Começa-se por criar o cubo. A sua posição é irrelevante, já que irá depois ser destruído e instanciado em várias posições pré-determinadas. A escala também não será alterada. No entanto, será adicionada uma *script* que permitirá que o cubo seja destruído após um determinado número de colisões. A *script*, de nome *destrutor*, é apresentada no código seguinte:

```
public var toquesNecessarios : int = 1;
public var pontos: int = 1;
private var nrToques: int = 0;
private var pontuacao: contarPontos;

function Start() {
    pontuacao = GameObject.Find("pontos").GetComponent.<contarPontos>();
}

function OnCollisionExit(collisionInfo : Collision) {
    nrToques++;
    if (nrToques == toquesNecessarios) {
        GameObject.Destroy(gameObject);
        pontuacao.incrementar(pontos);
    }
}
```

Esta *script* introduz alguns novos conceitos de forma propositada, uma vez que algumas das abordagens já utilizadas podiam ser aqui reutilizadas.

Existem quatro variáveis, duas públicas e duas privadas. As públicas correspondem ao número de toques que a bola terá de dar ao cubo para que este seja destruído e ao número de pontos atribuídos quando o cubo for destruído. Embora estes valores sejam usados quase como constantes, permitem que, mais tarde, se possa aumentar a complexidade do jogo com menor esforço.

As duas variáveis privadas correspondem ao número de toques que o cubo já sofreu e a uma referência à *script* *contarPontos*, associada ao *GUIText* que apresenta no ecrã o total de pontos atual.

Até agora, quando foram usadas referências entre objetos, usou-se o inspetor para associar o objeto externo à *script* em causa. Neste caso, essa associação é feita programaticamente no método *Start*. Usa-se o método *Find* da classe *GameObject* para encontrar um objeto na cena atual com determinado nome. Segue-se o uso de um método para obter o componente responsável por contar os pontos (*script* *contarPontos*).

O método *OnCollisionExit* é invocado quando um objeto deixa de estar em colisão com outro objeto. Existe um método equivalente para quando a colisão se inicia. No entanto, se destruíssemos o objeto no instante do início da colisão, o motor de física não calcularia o movimento de ricochete. Como o único objeto em movimento no jogo é a bola, não é necessário verificar qual o objeto que colidiu. Assim, incrementa-se o número

de toques que o bloco sofreu e, se esse número for igual ao número necessário para o bloco ser destruído, então faz-se a sua destruição com o método `Destroy` da classe `GameObject`. Posteriormente, incrementa-se o total de pontos usando o método acabado de implementar na *script* `contarPontos`. Este cubo irá servir de modelo (*Prefab*) para os cubos todos a destruir. Para criar o *Prefab* usa-se o menu **Assets** → **Create** → **Prefab**. Será criado um objeto vazio no separador **Project** a que se deve associar um nome. Será usado o nome `bloco`. Este `bloco` encontra-se, neste momento, vazio, pelo que será necessário arrastar o cubo do separador **Hierarchy** para dentro do *Prefab* acabado de criar. Este é um processo simples, mas muito útil. Depois de criado o *Prefab* o cubo deve ser apagado do separador **Hierarchy**.

Finalmente, serão criados vários cubos, com base no *Prefab*, e colocados de forma automática na área de jogo. Segue-se a *script* `popular` que deverá ser colocada num objeto qualquer da cena, por exemplo, no *pad*. Esta é capaz de ser a *script* mais complicada, mas nem por isso a mais longa. Em primeiro lugar, é criada uma variável pública e, portanto, visível no inspetor, onde será guardado o *Prefab* do cubo:

```
public var modelo : GameObject;
function Start () {
    var n : int = 40;
    var matriz = new int[9,9,9];
    while (n > 0) {
        var x = Random.Range(-4, 4);
        var z = Random.Range(-4, 4);
        var y = Random.Range(1, 9);
        if (matriz[x+4,z+4,y] == 0) {
            n--;
            matriz[x+4,z+4,y] = 1;
            GameObject.Instantiate(modelo,Vector3(x,-y,z), Quaternion.identity);
        }
    }
}
```

O resto do código encontra-se todo no método `Start`, invocado sempre que a cena de jogo se inicia. Aqui, é criada uma matriz tridimensional que corresponde às 9x9x9 posições onde podem existir cubos. A variável `n` corresponde ao número de blocos a serem criados. O ciclo que se segue irá criar um cubo enquanto essa variável contiver um valor positivo. Para cada cubo são calculadas, de forma aleatória, as coordenadas em que esse cubo será colocado. O método `Range` da classe `Random` retorna um valor aleatório num determinado intervalo.

A condição que se segue verifica se a posição da matriz já se encontra ocupada. Se estiver ocupada, o valor `n` não é alterado e, portanto, são calculadas três novas coordenadas. Se não estiver ocupada, então o valor `n` é decrementado, a posição da matriz alterada, para indicar que as coordenadas estão ocupadas, e o método `Instantiate` da classe `GameObject` usado para criar um clone do *Prefab* na cena atual. Este método recebe o *Prefab* que será clonado, a posição onde será colocado (na forma de

um vetor tridimensional) e qual a sua rotação. No caso concreto, não se deseja qualquer rotação, pelo que se usa o método `Identify` da classe `Quaternion`⁵¹ para obter a rotação identidade (`identity`). Finalmente, para que tudo funcione, é necessário ir ao inspetor desta *script* e associar ao campo **Modelo** o *Prefab* que foi criado.

5.9 MATERIAIS

Neste momento o jogo está pouco cativante, dada a falta de cor. A associação de texturas a objetos é conhecida pela criação de materiais. Para associar uma textura, por exemplo, às paredes do jogo começa-se por obter uma imagem para servir de padrão.



As texturas aqui usadas foram encontradas na Internet recorrendo a um motor de busca tradicional, como o Google ou o Yahoo. Para as paredes usou-se a pesquisa *brick texture tileable*, para a bola *golf ball texture*, para a *pad glass tileable texture* e, finalmente, para os cubos procurou-se por *concrete texture tileable*.

Depois de encontradas as imagens ideais, estas devem ser importadas através da técnica habitual, já usada na importação do logótipo do *Arkadroid*: colocar as imagens no separador **Project**, ou usando o gestor de ficheiros, na pasta **assets** do projeto. Para criar o material basta arrastar a imagem para o objeto no qual se deseja aplicar o padrão. O Unity criará automaticamente um material. Por exemplo, arrastando a imagem que escolhemos para uma das paredes obteremos o material da Figura 5.18.

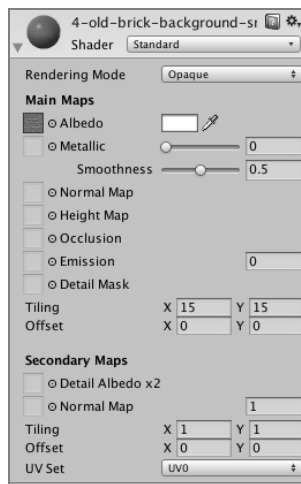


FIGURA 5.18 – Componente relativo ao material

⁵¹ `Quaternion` é um tipo de dados usado para representar a rotação de um objeto num mundo tridimensional. Os seus detalhes fogem ao objetivo introdutório deste livro.

Neste componente é possível alterar um grande número de opções, das quais as mais importantes para a situação atual são o tipo de sombreador (para dispositivos móveis sugere-se o uso dos sombreadores para **Mobile**) e o número de vezes que esta imagem se irá repetir (opção **Tiling**).

Para as paredes foi escolhido o sombreador **Mobile/Diffuse**. Note que depois de criar o material para uma das paredes pode reutilizá-lo para as restantes paredes. O material, depois de configurado para uma das paredes, aparecerá na pasta **Materials**. Este material pode ser posteriormente arrastado para cada uma das paredes, por exemplo, no separador **Hierarchy**. Este mesmo processo (obter imagem, importar imagem, criar material) deve ser repetido para a bola e para o *pad*. Note-se que em relação ao *pad* é importante que este seja transparente. Para isso, é necessário usar uma imagem com canal alfa⁵² (com transparência). Depois, para além de escolher o sombreador **Mobile/Diffuse**, deverá ser alterado o **RenderingMode** para **Transparent**.

Finalmente, para os cubos é necessário ter algum cuidado, já que se trata de um *Prefab*. É necessário arrastar o *Prefab* para a cena (uma qualquer posição), e aí associar-lhe a textura. Depois, voltar a arrastar o cubo alterado (com a textura associada) a partir do separador **Hierarchy** de novo para o *Prefab*, no separador **Project**. Termina-se removendo o cubo da cena. A Figura 5.19 mostra o *Arkadroid* terminado, já com texturas associadas.

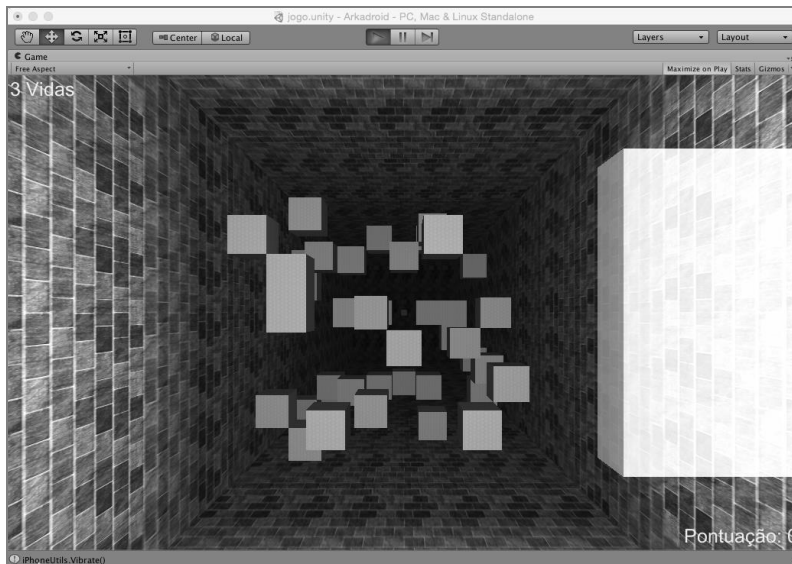


FIGURA 5.3 – *Arkadroid* com texturas

⁵² Explicar o conceito de canal alfa (*alpha channel*) e como criar uma imagem transparente foge ao âmbito deste livro. No entanto, qualquer editor, como o Gimp ou o Photoshop, permite fazê-lo de forma simples.

5.10 SOM

Finalmente, falta-nos adicionar som ao jogo. Vamos adicioná-lo em diferentes fases: uma música para o início do jogo, outra durante o jogo, um som quando a bola bate no *pad* ou num bloco, um som quando se perde uma vida e, finalmente, um som quando se perde o jogo. Mais uma vez, usamos a Internet para obter estes sons⁵³:

- Para as músicas – usamos *Arkanoid main theme* para encontrar a música original do jogo *Arkanoid*, e para a música durante o jogo optamos por procurar por *background music free for game*;
- Para os sons – usamos *bouncy ball sound effect* para a bola a bater nos objetos, *laugh sound effect free* para quando o jogador perde uma vida e *failure sound effects free* para quando o jogador termina o jogo.

Tal como para as imagens, os sons podem ser importados no projeto Unity simplesmente arrastando-os para as pastas. Para o uso de som são necessários dois componentes distintos, um componente que recebe (escuta) o som e outro que produz o som. O componente que recebe o som chama-se *Audio Listener*. Habitualmente, este componente está associado a uma câmara. Para o componente de produção de som pode ser usada uma fonte de som (**Audio Source**) ou uma zona de ressonância (**Reverb Zone**).

A inclusão do som na cena inicial de jogo é relativamente simples. Basta abrir a cena em causa e criar um objeto `AudioSource` usando o menu **GameObject** → **Audio** → **Audio Source**. Este comando cria um objeto vazio com o componente *Audio Source* associado. A Figura 5.20 mostra as opções do inspetor deste componente.

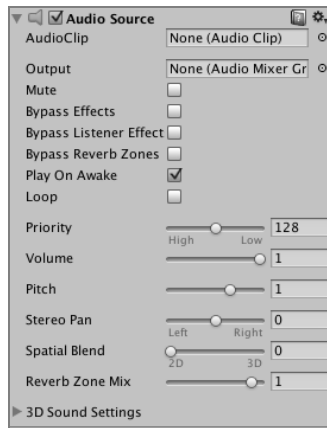


FIGURA 5.20 – Componente *Audio Source*

⁵³ Quer em relação às imagens, quer aos sons procurados e encontrados na Internet, é importante ter cuidado para que os seus direitos de autor permitam o seu uso!

Para a música inicial basta arrastar o ficheiro importado com o som desejado para o campo **Audio Clip**. Para além disso, deve-se garantir que a opção **Play on Awake** está ativa (para que o som inicie mal a cena seja executada) e, caso a música tenha tempo inferior aos 10 segundos da cena inicial, escolher a opção **Loop**.

Os restantes sons serão adicionados na cena `jogo`. Em primeiro lugar, a música de fundo pode ser criada usando este mesmo modo, criando um `AudioSource`, definindo a música em causa e ativando a opção **Loop**, para que a música não pare durante o jogo. Como os sons (habitualmente designados por FX) do jogo devem ser mais fortes do que a música, o volume desta pode ser diminuído, também neste inspetor, para **0.5**.

Ficam a faltar os efeitos sonoros. O som de quando a bola bate num objeto deverá, logicamente, ser produzido pela própria bola. Para isso será selecionada a esfera e adicionado o componente *Audio Listener*: **Component** → **Audio** → **Audio Source**. Deverá ser associado o som em causa e desativada a opção **Play on Awake**, já que o som só será executado quando a bola bater em algum objeto. Para que isto seja possível será criada uma *script* na esfera, de nome `batimento`, com o código que se segue:

```
function OnCollisionEnter () {  
    GetComponent.<AudioSource>().Play();  
}
```

Neste código usamos o método `GetComponent` para obter o `AudioSource` associado ao objeto (ou ao primeiro, caso exista mais do que um). O método `Play` permite reproduzir o som em causa.

O som de perda de jogo vai ser colocado de forma semelhante à música das duas cenas anteriores. Na verdade, será feita uma pequena batota. Na cena `gameover` será criado um `AudioSource` (objeto) e associado o som escolhido para o final de jogo. Mantém-se o **Play on Awake** e tudo deverá funcionar bem.

Para o som de perda de bola, este terá de ser adicionado na cena `jogo`. Será criado um `AudioSource` no objeto `baliza`. Ao `AudioSource` será associado o som de perda de bola e removida a opção **Play on Awake**. Tal como no caso anterior, a reprodução será controlada através de uma *script*, já criada, de nome `foraDeJogo`. O código seguinte mostra o método `OnTriggerEnter` dessa *script* devidamente alterado:

```
function OnTriggerEnter(other: Collider) {  
    contador.vidas--;  
    Handheld.Vibrate();  
    GetComponent.<AudioSource>().Play();  
    bola.restart();  
}
```

5.11 DISPONIBILIZAÇÃO

Antes de criar um pacote APK para instalação no Android é necessário configurar a aplicação. Para isso, seleciona-se o menu **Edit** → **Project Settings** → **Player**; surgirá um separador de configuração. Neste separador é necessário configurar alguns aspetos. Será necessário indicar o nome da empresa e o do jogo/produto. Posteriormente, podem ser adicionados ícones e, caso seja relevante, qual a imagem a usar para o cursor (e a posição do cursor que corresponde à ponta do cursor). Note que, tal como qualquer outra imagem, deve ser colocada na pasta do projeto para que seja importada.

Mais abaixo, seleciona-se o separador correspondente ao Android. Neste quadro é importante preencher, essencialmente, o **Bundle Identifier**, que tem de ser uma *string* iniciada com `com.` seguida de um identificador da empresa e do produto. Por exemplo: `com.fcagames.arkadroid`. Também se deverá indicar a versão da aplicação e a versão do Android com que este jogo será compatível. Depois de tudo configurado, usa-se o menu **File** → **Build Settings** para indicar qual a arquitetura de destino, tal como se mostra na Figura 5.21. Nesta janela escolhe-se a arquitetura de destino (Android).



FIGURA 5.21 – Preparação da aplicação Android

Finalmente, o processo de criação do pacote APK é concluído com a ativação do botão **Build**. Note-se que na primeira vez que este botão é usado, o Unity irá solicitar alguma informação, como, por exemplo, qual o sítio onde se encontra instalado o *Android Development Tools* (ADT). Também poderá ser usado um dispositivo Android conectado ao PC, mas é um passo opcional, podendo ser cancelado. Finalmente, será pedido o nome para o ficheiro e o sítio onde este deve ser colocado.

6

GOOGLE PLAY, SERVIÇOS E PUBLICAÇÃO

Este capítulo começa por introduzir os serviços do Google Play Services que são a base para o desenvolvimento de aplicações mais ricas dentro do universo dos produtos Google. De seguida, dá-se ênfase a um dos serviços mais importantes, que é o Google Play Games Services. Este serviço melhora as experiências dos jogos ao disponibilizar um conjunto de funcionalidades, tais como conquistas, tabelas de classificação, a gravação dos dados do jogo na *cloud* e o modo multijogador em tempo real. Para consolidar estes conceitos, integram-se algumas destas funcionalidades no jogo *Othelloid*, detalhado no Capítulo 3. Finalmente, enumeram-se os passos necessários para a publicação do jogo na Google Play, a loja *online* da Google.

6.1 GOOGLE PLAY SERVICES

O Google Play Services (internamente chamado *Google Mobile Services* – GMS) é uma biblioteca de serviços para dispositivos Android. Quando foi introduzido pela primeira vez, em 2012, fornecia acesso simples à API do Google+. Desde então tem vindo a expandir-se para cobrir uma grande variedade de necessidades, permitindo que as aplicações comuniquem facilmente com os serviços de forma *standard*. A Tabela 6.1 enumera as principais API incluídas no Google Play Services⁵⁴.

Os serviços são automaticamente atualizados através da Google Play em dispositivos Android (versão 2.3 ou mais recente) com a aplicação Google Play Store instalada. O processo de atualização resume-se à distribuição automática de um ficheiro APK, aumentando a rapidez e tornando mais fácil para os utilizadores receberem as mais recentes atualizações e poderem assim integrá-las nas suas aplicações. Esta abordagem permite também oferecer novas funcionalidades para dispositivos antigos sem ter que depender dos *Original Equipment Manufacturers* (OEM), evitando assim a fragmentação da plataforma.

⁵⁴ A versão atual é a 7 (2 de março de 2015).

PRODUTO	DESCRIÇÃO
Google+	Disponibiliza <i>login</i> único (<i>single sign-on</i>) aos produtos Google providenciando uma experiência mais rica e personalizada.
Location	Abstrai o uso de tecnologias de localização, como o <i>geofencing</i> e o reconhecimento de atividade.
Maps	Permite a inclusão de mapas Google (mapas, marcadores, câmara) e <i>Street View</i> sem necessidade de abrir uma aplicação separada.
Places	Permite criar aplicações baseadas na localização, que respondem contextualmente às empresas locais e outros lugares perto do dispositivo.
Drive	Disponibiliza um local de armazenamento único com ferramentas de procura e sincronização sofisticadas.
Fit	Permite rastrear as atividades dos utilizadores, definir metas de <i>fitness</i> e oferecer uma visão abrangente de <i>fitness</i> ao utilizador.
Games	Permite adicionar às aplicações uma experiência mais competitiva e social através do uso de tabelas de classificação (<i>leaderboards</i>), conquistas e sessões multijogador.

TABELA 6.1 – Principais produtos do Google Play Services

6.1.1 CONFIGURAÇÃO DO GOOGLE PLAY SERVICES

Para desenvolver e testar aplicações Android usando as API do Google Play Services, é necessário uma configuração prévia do projeto Android. Comece por verificar se tem instalado o pacote Google Play Services, conforme descrito na secção 1.3 (Capítulo 1) deste livro. Caso não o tenha, adicione-o através do SDK Manager. Depois, para o testar, tem duas alternativas: usar um dispositivo Android (versão 2.3 ou superior) com a Google Play Store ou usar um emulador Android (versão 4.2.2 ou superior) com as API da Google. De seguida, é necessário preparar a aplicação Android para poder usar as API do Google Play Services. Execute a próxima sequência de passos no Android Studio:

- 1) Abra o ficheiro **build.gradle** incluído na pasta do módulo da aplicação e adicione uma nova regra de construção sobre as dependências para a versão mais recente do Google Play Services:

```
apply plugin: 'com.android.application'
...
dependencies {
    compile 'com.android.support:appcompat-v7:21.0.3'
    compile 'com.google.android.gms:play-services:7.0.0'
}
```



Certifique-se de que altera este número de versão sempre que o Google Play Services é atualizado.

- 2) Guarde as alterações e clique na opção **Sync Project with Gradle Files** da barra de ferramentas.

Nas versões do Google Play Services (antes da 6.5), tinha de se compilar todo o pacote da API na aplicação. Em certos casos, tornava-se mais difícil manter o número de métodos da aplicação abaixo do limite de 65 536 (16 *bits*). A partir da versão 6.5, pode-se compilar seletivamente as API do Google Play Services⁵⁵. Por exemplo, para incluir apenas a API do Google Maps troque a próxima instrução de compilação:

```
compile 'com.google.android.gms:play-services:7.0.0'
```

Por esta linha:

```
compile 'com.google.android.gms:play-services-maps:7.0.0'
```

A partir daqui, pode-se começar a usar as API do Google Play Services.

6.1.2 CLASSE `GoogleApiClient`

Para fazer uma conexão às API da Google incluídas na biblioteca **Google Play Services** é necessário criar uma instância da classe `GoogleApiClient`. Esta classe fornece um ponto de entrada comum a todos os serviços do Google Play e gere a conexão de rede entre o dispositivo do utilizador e cada serviço do Google. A Figura 6.1 mostra como a classe fornece uma interface para conectar e invocar qualquer um dos serviços disponíveis do Google Play, como, por exemplo, o Google Play Games e o Google Drive.

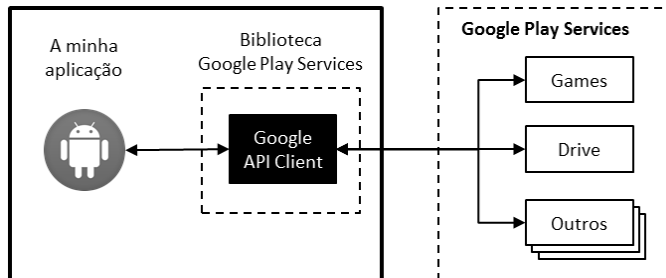


FIGURA 6.1 – Classe `GoogleApiClient`

Comece por criar uma instância da classe `GoogleApiClient` usando as API `GoogleApiClient.Builder` no método `onCreate` da atividade `Android`. A classe fornece métodos que permitem especificar as API do Google a usar. O próximo excerto de código exemplifica o uso de um objeto `GoogleApiClient` para se conectar ao Google Drive:

```
GoogleApiClient mGoogleApiClient = new GoogleApiClient.Builder(this)
    .addApi(Drive.API)
    .addScope(Drive.SCOPE_FILE)
    .build();
```

⁵⁵ *Link:* <https://developer.android.com/google/play-services/setup.html#split>

Podem-se adicionar várias API e escopos para o mesmo objeto `GoogleApiClient` através dos métodos `addApi` e `addScope`. Antes de começar uma conexão chamando o método `connect` no objeto `GoogleApiClient`, deve-se especificar uma implementação para as interfaces `ConnectionCallbacks` e `OnConnectionFailedListener`. Estas interfaces recebem respostas do método assíncrono `connect` quando a conexão com o Google Play Services for bem-sucedida, falhar ou ficar suspensa. Segue-se um exemplo de uma atividade que implementa as interfaces:

```
public class MyActivity extends FragmentActivity
    implements ConnectionCallbacks, OnConnectionFailedListener {
    private GoogleApiClient mGoogleApiClient;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // CRIA UMA INSTÂNCIA GOOGLEAPICLIENT
        mGoogleApiClient = new GoogleApiClient.Builder(this)
            .addApi(Drive.API)
            .addScope(Drive.SCOPE_FILE)
            .addConnectionCallbacks(this)
            .addOnConnectionFailedListener(this)
            .build();
    }
    @Override
    public void onConnected(Bundle connectionHint) {
        // CONECTADO AO GOOGLE PLAY SERVICES!
    }
    @Override
    public void onConnectionSuspended(int cause) {
        // A CONEXÃO FOI INTERROMPIDA. DESATIVE QUALQUER COMPONENTE GUI QUE DEPENDA
        // DAS API DA GOOGLE ATÉ QUE O MÉTODO ONCONNECTED() SEJA CHAMADO
    }
    @Override
    public void onConnectionFailed(ConnectionResult result) {
        // ESTE CALLBACK É IMPORTANTE PARA LIDAR COM ERROS DE CONEXÃO
    }
}
```

De seguida, devem-se redefinir os métodos `onStart` e `onStop` da atividade de forma a invocar os métodos `connect` e `disconnect` do objeto `GoogleApiClient`, respetivamente. Por exemplo:

```
@Override
protected void onStart() {
    super.onStart();
    mGoogleApiClient.connect();
}
@Override
protected void onStop() {
    mGoogleApiClient.disconnect();
    super.onStop();
}
```

6.1.3 GOOGLE PLAY GAMES SERVICES

Segundo a Google, “três em quatro utilizadores Android jogam jogos”⁵⁶. Com mais de mil milhões de utilizadores Android ativos, os jogos assumem hoje em dia um papel fundamental. Nesta linha de pensamento, em maio de 2013, na sua conferência anual (Google I/O), a Google apresentou uma biblioteca de serviços focada nos jogos digitais – o **Google Play Games Services (GPGS)** –, definida como uma ferramenta multiplataforma que sincroniza o progresso, as conquistas e outros dados de jogos no Android, na Web, e em dispositivos iOS. Para materializar este serviço e disponibilizá-lo a todos, a Google apresentou a aplicação **Google Play Games**⁵⁷ (Figura 6.2).

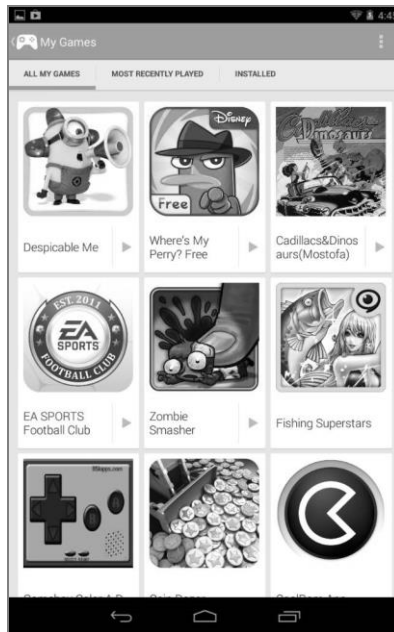


FIGURA 6.2 – A aplicação Google Play Games

A aplicação Google Play Games funciona como um *hub* social, sendo a forma mais fácil de descobrir novos jogos, de controlar as conquistas e as pontuações e de jogar com amigos em todo o mundo. Em suma, reúne todos os seus jogos do Google Play numa experiência Android unificada. As principais funcionalidades são: descobrir novos jogos, jogar com amigos e ver o que estes estão a jogar, participar em jogos de vários jogadores, controlar as suas conquistas e comparar pontuações com outros jogadores.

⁵⁶ Greg Hartrell – gestor de projetos do Google Play – em entrevista à *VentureBeat* (dezembro de 2013).

⁵⁷ Link: <https://play.google.com/store/apps/details?id=com.google.android.play.games>

Apesar de ser uma aplicação interessante para todos os que gostam de jogar e de desafiar os seus amigos, é agora necessário que os jogos sejam adaptados e preparados para serem usados. De acordo com a Google⁵⁸, o Google Play Games recebeu mais de 100 milhões de novos utilizadores no primeiro semestre de 2014, tornando-se na rede de jogos móveis com o mais rápido crescimento de sempre. Com os serviços disponibilizados pelo GPGS, é possível incorporar nas aplicações Android alguns conceitos tipicamente encontrados nos jogos tradicionais, conforme enumera a Tabela 6.2.






ÍCONE	NOME DO SERVIÇO	DESCRIÇÃO
	<i>Achievements</i> (conquistas)	Definição de desafios que, se forem atingidos, vão enriquecer o <i>status</i> do jogador ou desbloquear o acesso a outros níveis.
	<i>Leaderboards</i> (tabelas de classificação)	<i>Ranking</i> com as classificações dos jogadores, permitindo comparar os seus resultados com os de outros jogadores, com base, por exemplo, nos círculos de amigos do Google+.
	<i>Saved Games</i> (jogos gravados)	Armazenamento remoto (<i>cloud</i>) das configurações, dos estados e do progresso dos jogadores no jogo.
	<i>Multiplayer</i> (multijogador)	Modo de jogo que possibilita que vários jogadores, em tempo real, possam cooperar ou competir num jogo.
	<i>Quests</i> (missões)	Introdução de desafios periódicos no jogo que os jogadores possam completar para ganharem recompensas. Desta forma, os programadores podem lançar desafios periódicos para a sua comunidade de jogadores.

TABELA 6.2 – Visão geral do Google Play Games Services

6.2 INTEGRAÇÃO DO GPGS NO JOGO *OTHELLOID*

O GPGS inclui um conjunto de serviços multiplataforma de funcionalidades populares de jogos. Nesta secção explicam-se os passos necessários para a integração de duas dessas funcionalidades no jogo *Othelloid* – as conquistas e as tabelas de classificação.

6.2.1 CONFIGURAÇÃO NA GOOGLE PLAY DEVELOPER CONSOLE

Esta secção explica como usar a Google Play Developer Console (GPDC) para configurar os serviços de jogos a serem incluídos no jogo *Othelloid*.

⁵⁸ Link: <http://goo.gl/ORD6B1>

Para tal, execute a seguinte sequência de passos:

1) Iniciar sessão na GPDC:

- Aceda à Google Play Developer Console⁵⁹;
- Associe uma conta Gmail;
- Caso seja a primeira vez, deve ler e aceitar o acordo de distribuição e fazer o pagamento de uma taxa de inscrição (25 dólares), através do sistema de pagamento móvel da Google, chamado *Google Wallet*;
- Complete os dados da conta.

2) Adicionar o jogo à GPDC:

- Selecione, no menu lateral esquerdo, a opção **Game services**;
- Clique no botão **Add new game**;
- Na janela **Set up Google Play Game Services for an app**, especifique se o jogo usa já as API da Google. Se está a criar um jogo desde o início ou nunca lhe associou as API da Google, mantenha ativa a opção **I don't use any Google API in my game yet**. Digite o nome do jogo, atribua-lhe uma categoria e, de seguida, clique no botão **Continue** (Figura 6.3);

SET UP GOOGLE PLAY GAME SERVICES FOR AN APP

Do you already use Google APIs in your app?

I don't use any Google APIs in my game yet I already use Google APIs in my game

What is the name of your game?

8 of 30 characters

This is the name that will be displayed to users in Google Play game services.

What kind of game is it?

The category helps users browse interesting games.

Google Play game services use the following APIs: Google Play App State, Google+ API, Google Play Game Services and Google Play Game Management
We will automatically create a project for your game on the Google Developers Console and enable the necessary APIs for you.

FIGURA 6.3 – Configuração do Google Play Games Services para o jogo

⁵⁹ Link: <https://play.google.com/apps/publish>

- No formulário **Game Details**, pode completar os detalhes do jogo adicionando uma descrição e elementos gráficos⁶⁰. Apenas o campo **Name** é necessário na fase de teste do jogo, devendo os outros campos ser preenchidos antes de poder publicar o seu jogo;
- Clique no botão **Save** para criar o jogo na GPDC.

3) Gerar um *ID OAuth 2.0* do cliente:

O jogo deve ter um *client ID OAuth 2.0*, a fim de ser autenticado e autorizado a chamar os serviços do Google Play. Para configurar a associação entre o *client ID* e o jogo é necessário gerar o identificador e vinculá-lo ao jogo.

→ Passo 1 – Criar uma *linked app* (uma para cada plataforma a suportar):

- Abra a página *Linked apps*;
- O acesso ao GPGS pode ser feito através de um SDK para Android, de um SDK nativo para iOS (para jogos em iPhone e iPad) ou mesmo para a Web através de API REST usando bibliotecas para JavaScript, Java, Python, PHP, entre outras. Neste caso, clique no botão *Android*;
- Adicione o pacote do jogo: `com.androidgames.othelloid`;
- Defina se quer ativar o modo multijogador e antipirataria (impede os utilizadores que não tenham instalado o seu jogo através do Google Play de aceder ao *GMS*) e clique no botão *Save and Continue*.

→ Passo 2 – Criar o *ID* do cliente:

- Clique no botão *Authorise your app now* para iniciar o processo de criação de um *client ID OAuth 2.0*;
- Digite o nome de projeto e um logótipo do produto (opcional). Clique no botão **Continue**;
- Especifique as configurações do *ID* do cliente, devendo fornecer o nome do pacote do projeto Android do jogo e a impressão digital (*fingerprint*) SHA1 do certificado;

⁶⁰ *Link*: <https://support.google.com/googleplay/android-developer/answer/1078870?hl=en>



O Android requer que todas as aplicações sejam assinadas digitalmente com um **certificado** para que possam ser instaladas. O Android usa esse certificado para identificar o autor de uma aplicação, sendo que o certificado não precisa de ser assinado por uma autoridade certificadora. As *apps* Android costumam usar certificados autoassinados. O programador da *app* detém a chave privada do certificado. É possível assinar uma aplicação no modo *debug* ou *release*. Deve-se assinar a aplicação no modo *debug* durante o desenvolvimento da aplicação e no modo *release* quando a aplicação estiver pronta para ser distribuída. O Android gera automaticamente um certificado para assinar aplicações em modo *debug*. Para assinar aplicações em modo *release* é necessário gerar o seu próprio certificado.

Segue-se um exemplo de como obter uma impressão digital de um certificado no modo *debug*. Abra um terminal e execute a ferramenta **keytool** para obter a impressão digital SHA1 do certificado:

```
keytool -exportcert -alias androiddebugkey
-keystore c:\Users\Ricardo\.android\debug.keystore -list -v
```

A ferramenta gera a impressão digital para o terminal. Por exemplo:

```
SHA1: C4:38:EF:A0:9B:45:C6:10:77:47:EA:9A:C9:BA:AE:C0:C2:11:46:B7
```



No Android Studio, a *keystore* (modo *debug*) está localizada em `C:\Users<user>\.android\` (Windows 7 ou superior). A ferramenta **keytool** solicita uma senha para a *keystore*. A senha padrão no modo *debug* é **android**.

→ Copie o output anterior no campo Signing certificate fingerprint (SHA1) e clique no botão Create client. É exibida uma janela (Figura 6.4) com um identificador da aplicação (application ID).

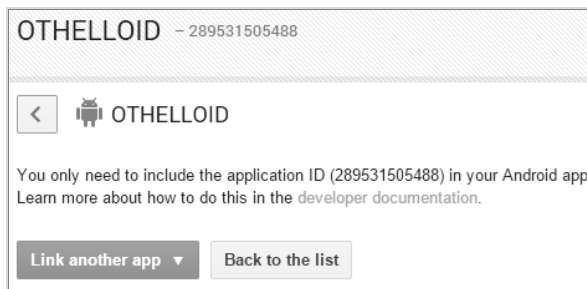


FIGURA 6.4 – Geração da aplicação e do respetivo identificador

O *application ID* é uma *string* com 12 ou mais dígitos que, posteriormente, deverá fazer referência no ficheiro de manifesto do projeto Android *Othelloid*. Note que, para as aplicações Android, o *client ID* é obtido automaticamente a partir do *application ID*.



Deverá criar dois *client ID*: um no modo *debug* e outro no modo *release*. Certifique-se de que usa o mesmo nome do pacote para ambos. Esta abordagem permite que o Google Play Services reconheça chamadas dos APK assinados com qualquer um dos certificados.

Poderá sempre encontrar esta informação mais tarde, revisitando a página **Linked Apps** (Figura 6.5).

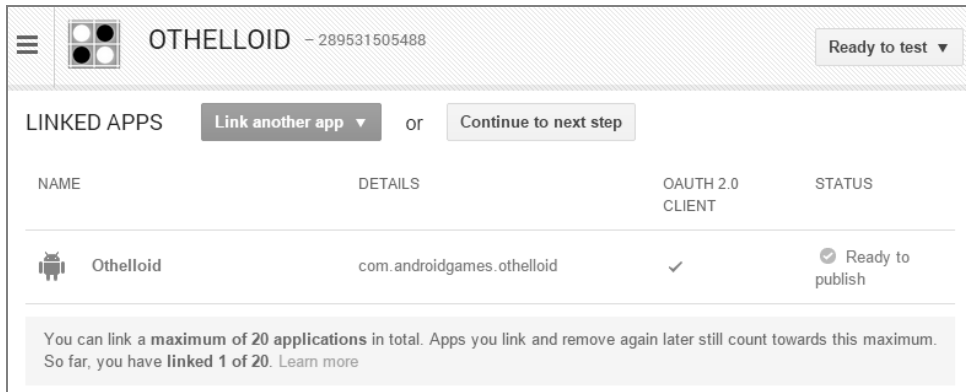


FIGURA 6.5 – Lista de aplicações associadas ao jogo *Othelloid*

6.2.2 IMPLEMENTAÇÃO

Nesta secção descrevem-se os passos necessários para o acesso às API do GPGS e como integrar os seus serviços, tais como as tabelas de classificação e as conquistas. Antes de aceder programaticamente às API do GPGS, associe o identificador da aplicação gerado na secção 6.2.1 e adicione os metadados sobre o jogo no ficheiro de manifesto. Mais concretamente, execute a seguinte sequência de passos:

- 1) Grave o identificador da aplicação num ficheiro de recurso chamado **games_ids.xml** na pasta **res/values** do projeto Android:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_id">289531505488</string>
</resources>
```

- 2) Edite o ficheiro **AndroidManifest.xml** e inclua dois elementos `<meta-data>` no elemento `<application>`:

```
<meta-data android:name="com.google.android.gms.games.APP_ID"
  android:value="@string/app_id" />

<meta-data android:name="com.google.android.gms.version"
  android:value="@integer/google_play_services_version"/>
```

6.2.2.1 BIBLIOTECA BASEGAMEUTILS

De forma a simplificar a codificação desta secção, usa-se a biblioteca de código **BaseGameUtils**⁶¹. Descarregue a biblioteca e associe-a ao projeto do jogo executando os seguintes passos:

- 1) Abra o projeto no Android Studio.
- 2) Clique em **File → Import Module** e seleccione a biblioteca **BaseGameUtils**.
- 3) Modifique as dependências do ficheiro **build.gradle** para o módulo que vai usar a **BaseGameUtils**:

```
dependencies {
    compile project(':BaseGameUtils')
    // ...
}
```

- 4) Compile o projeto seleccionando a opção **Build → Make Project**.

A partir daqui, está pronto para começar a usar os serviços de jogos da Google.

6.2.2.2 ACESSO ÀS API

Antes de o jogo poder fazer chamadas aos serviços do jogo, deve primeiro estabelecer uma conexão assíncrona com os servidores da Google Play através do objeto `GoogleApiClient` e autenticar o utilizador com os serviços do jogo.

Na atividade principal do projeto *Othelloid*, comece por importar o pacote `com.google.android.gms` e a classe `BaseGameUtils`. Depois, implemente na atividade principal as interfaces `ConnectionCallbacks` e `OnConnectionFailedListener` da classe `GoogleApiClient`:

```
import com.google.android.gms.*;
import com.google.example.games.basegameutils.BaseGameUtils;
public class MainActivity extends Activity implements
    GoogleApiClient.ConnectionCallbacks,
    GoogleApiClient.OnConnectionFailedListener { ... }
```

De seguida, crie um objeto `GoogleApiClient` no método `onCreate` da atividade usando a classe `GoogleApiClient.Builder`:

```
private GoogleApiClient myGoogleApiClient;
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```

⁶¹ Disponível no GitHub em: <https://github.com/playgameservices/android-basic-samples>

```
// CRIA A GOOGLE API CLIENT COM ACESSO AOS SERVIÇOS PLUS E GAMES
myGoogleApiClient = new GoogleApiClient.Builder(this)
    .addConnectionCallbacks(this)
    .addOnConnectionFailedListener(this)
    .addApi(Plus.API).addScope(Plus.SCOPE_PLUS_LOGIN)
    .addApi(Games.API).addScope(Games.SCOPE_GAMES)
    // ADICIONE AQUI OUTRAS API SE NECESSÁRIO
    .build();
// ...
}
```

Depois, use o método `connect` no método `onStart` da atividade para que o `GoogleApiClient` inicie conexão de forma automática. No método `onStop` é chamado o método `disconnect` para fechar a conexão ao Google Play Services:

```
@Override
protected void onStart() {
    super.onStart();
    myGoogleApiClient.connect();
}

@Override
protected void onStop() {
    super.onStop();
    myGoogleApiClient.disconnect();
}
```

Se o utilizador é autenticado com sucesso ou se já se conectou com êxito e ainda não fez *sign-out*, o sistema chamará o método `OnConnected`:

```
@Override
public void onConnected(Bundle connectionHint) {
    // O JOGADOR FEZ O SIGN-IN COM SUCESSO!
    // ATUALIZAR GUI (ESCONDER O BOTÃO SIGN IN, MOSTRAR O BOTÃO SIGN OUT)
    // ATIVAR FUNCIONALIDADES QUE DEPENDEM DO SIGN-IN
    ...
}
```

Se o *sign-in* falhar, é chamado o método `onConnectionFailed`. Se a falha não puder ser resolvida, o jogo deve mostrar um botão de *sign-in* para permitir aos utilizadores autenticarem-se novamente:

```
private static int RC_SIGN_IN = 9001;

private boolean myResolvingConnectionFailure = false;
private boolean myAutoStartSignInFlow = true;
private boolean mySignInClicked = false;

@Override
public void onConnectionFailed(ConnectionResult connectionResult) {
    if (myResolvingConnectionFailure) {
        // A RESOLVER
        return;
    }
}
```

```

// SE O BOTÃO SIGN IN FOI CLICADO OU SE O SIGN-IN AUTOMÁTICO ESTIVER ATIVO
if (mySignInClicked || myAutoStartSignInFlow) {
    myAutoStartSignInFlow = false;
    mySignInClicked = false;
    myResolvingConnectionFailure = true;
    // TENTAR RESOLVER A FALHA DE CONEXÃO USANDO A BASEGAMEUTILS
    // O VALOR DE R.STRING.SIGNIN_OTHER_ERROR DEVE REFERENCIAR UMA DESCRIÇÃO DE ERRO
    // GENÉRICA (STRING) NO FICHEIRO STRINGS.XML, COMO, POR EXEMPLO, "ERRO NO SIGN-IN,
    // TENTE MAIS TARDE."
    if (!BaseGameUtils.resolveConnectionFailure(this,
        myGoogleApiClient, connectionResult,
        RC_SIGN_IN, getString(R.string.signin_other_error))) {
        myResolvingConnectionFailure = false;
    }
}
// CÓDIGO PARA MOSTRAR O BOTÃO SIGN IN
}

@Override
public void onConnectionSuspended(int i) {
    // TENTATIVA DE RECONECTAR
    myGoogleApiClient.connect();
}

```

De seguida, implemente o método `onActivityResult` para lidar com o resultado da resolução da conexão:

```

protected void onActivityResult(int requestCode, int resultCode,
    Intent intent) {
    if (requestCode == RC_SIGN_IN) {
        mySignInClicked = false;
        myResolvingConnectionFailure = false;
        if (resultCode == RESULT_OK) {
            myGoogleApiClient.connect();
        } else {
            // EXIBE UMA CAIXA DE DIÁLOGO ALERTANDO O UTILIZADOR DE QUE O SIGN-IN FALHOU.
            // O R.STRING.SIGNIN_FAILURE DEVE REFERENCIAR UMA STRING DE ERRO NO FICHEIRO STRINGS.XML
            // INDICANDO AO UTILIZADOR ESSA FALHA, COMO, POR EXEMPLO, "UNABLE TO SIGN IN."
            BaseGameUtils.showActivityResultError(this,
                requestCode, resultCode, R.string.signin_failure);
        }
    }
}

```

6.2.2.3 SIGN-IN AUTOMÁTICO

De forma a utilizar os serviços de jogos, é necessário que o jogo implemente a autenticação do utilizador com o GPGS. Se o utilizador não estiver autenticado, o jogo irá encontrar erros ao fazer chamadas às API dos serviços. A **autenticação** é feita através de um processo denominado *sign-in*. Recomenda-se que este processo seja automático para que sempre que o jogador inicia o jogo, não necessite de intervir manualmente para se

identificar. Esta secção descreve algumas técnicas de implementação de *sign-in* que o jogo deve usar para proporcionar uma experiência de utilização rica e transparente, como recomendado pelas boas práticas de *sign-in*⁶².

Comece por associar um `View.OnClickListener` à atividade, de forma a permitir que o jogo detete quando o utilizador clicar nos botões **Sign in** e **Sign out**:

```
public class MainActivity extends Activity implements
    View.OnClickListener,
    GoogleApiClient.ConnectionCallbacks,
    GoogleApiClient.OnConnectionFailedListener { ... }
```

De seguida, adicione os botões **Sign in** e **Sign out** ao *layout* da atividade do jogo (Figura 6.6). O botão **Sign in** é representado graficamente pela classe `SignInButton` que a Google disponibiliza para autenticação do utilizador. O botão **Sign out** é um botão básico que deve ficar invisível através da definição do valor `gone` no atributo `android:visibility` do elemento `Button`. O jogo só deve colocar o botão **Sign out** visível após uma autenticação bem-sucedida por parte do utilizador:

```
<!-- BOTÃO SIGN IN -->
<com.google.android.gms.common.SignInButton
    android:id="@+id/sign_in_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

<!-- BOTÃO SIGN OUT -->
<Button
    android:id="@+id/sign_out_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Sign Out"
    android:visibility="gone" />
```



No código anterior adicionou-se um botão *standard* para o *sign-in*. Uma alternativa seria ter um botão personalizado. Para personalizar objetos gráficos e demais iconografia, relacionados com o acesso ao GPGS, siga as diretrizes da marca que se encontram disponíveis em <https://developers.google.com/games/services/branding-guidelines>.

Defina no método `onCreate` os *callbacks* para detetar se o utilizador clicou nos botões **Sign in** e **Sign out**:

```
findViewById(R.id.sign_in_button).setOnClickListener(this);
findViewById(R.id.sign_out_button).setOnClickListener(this);
```

⁶² Link: <https://developers.google.com/games/services/checklist>

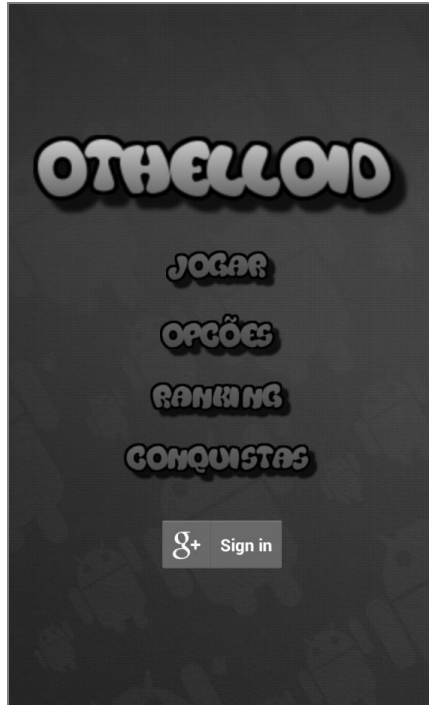


FIGURA 6.6 – Tela inicial do jogo *Othelloid* com o botão Sign in ativo

Para iniciar o fluxo de *login* quando o utilizador clica no botão **Sign in**, substitua o método `onClick`:

```
@Override
public void onClick(View view) {
    if (view.getId() == R.id.sign_in_button) {
        // INICIA O FLUXO ASSÍNCRONO DO SIGN-IN
        mySignInClicked = true;
        myGoogleApiClient.connect();
    } else if (view.getId() == R.id.sign_out_button) {
        // SIGN-OUT
        mySignInClicked = false;
        Games.signOut(myGoogleApiClient);

        // MOSTRA O BOTÃO SIGN IN E ESCONDE O BOTÃO SIGN OUT
        findViewById(R.id.sign_in_button).setVisibility(View.VISIBLE);
        findViewById(R.id.sign_out_button).setVisibility(View.GONE);
    }
}
```

O GPGS suporta *sign-in* assíncrono. Para que o sistema notifique sempre que o utilizador faz com sucesso o *sign-in*, altere o método `onConnected` da seguinte forma:

```
@Override
public void onConnected(Bundle connectionHint) {
    // O JOGADOR FEZ O SIGN-IN COM SUCESSO!
```

```
// ESCONDER O BOTÃO SIGN IN, MOSTRAR O BOTÃO SIGN OUT
findViewById(R.id.sign_in_button).setVisibility(View.GONE);
findViewById(R.id.sign_out_button).setVisibility(View.VISIBLE);
// ATUALIZAR GUI, ATIVAR FUNCIONALIDADES QUE DEPENDEM DO SIGN-IN, ETC.
...
}
```

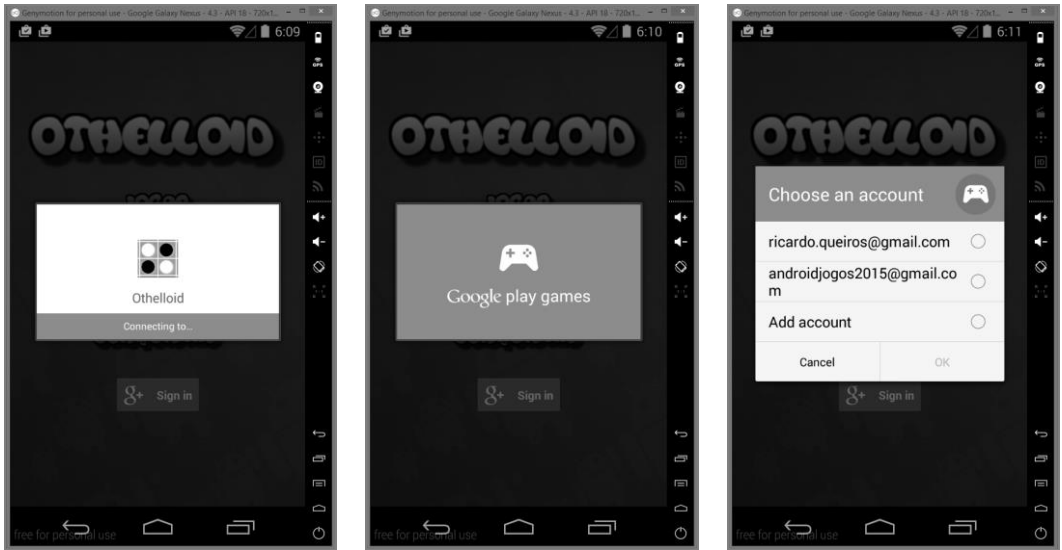
Depois de os utilizadores se autenticarem com êxito pela primeira vez, o jogo não deve pedir as credenciais sempre que os utilizadores iniciem o jogo novamente, só quando explicitamente fechem a sessão (*sign-out*). Para adicionar um *sign-in* automático, altere o método `onStart` de acordo com o próximo excerto de código. Para verificar se o utilizador está conectado, usa-se o método `isConnected`.

```
boolean myExplicitSignOut = false;
// DEFINIDO COMO TRUE QUANDO ESTÁ A MEIO DO FLUXO DE SIGN-IN,
// PARA SABER QUE NÃO PODE TENTAR CONECTAR-SE NO ONSTART
boolean myInSignInFlow = false;
// INICIALIZADO NO ONCREATE
GoogleApiClient myGoogleApiClient;

@Override
protected void onStart() {
    super.onStart();
    if (!myInSignInFlow && !myExplicitSignOut) {
        // SIGN-IN AUTOMÁTICO
        myGoogleApiClient.connect();
    }
}

@Override
public void onClick (View view) {
    if (view.getId() == R.id.sign_out_button) {
        // UTILIZADOR EXPLICITAMENTE FEZ SIGN-OUT, SENDO ASSIM DESATIVA-SE O SIGN-IN AUTOMÁTICO
        myExplicitSignOut = true;
        if (myGoogleApiClient != null && myGoogleApiClient.isConnected())
        {
            Games.signOut(myGoogleApiClient);
            myGoogleApiClient.disconnect();
        }
    }
    ...
}
```

Finalmente, execute o jogo no emulador. Na primeira vez é necessário que o jogador pressione o botão **Sign in** e selecione a conta de e-mail a usar para se conectar ao GPGS (Figura 6.7).

FIGURA 6.7 – Autenticação no jogo *Othelloid*

Após uma autenticação bem-sucedida é exibida uma mensagem de boas-vindas ao utilizador Google (Figura 6.8).

FIGURA 6.8 – *Sign-in* bem-sucedido e boas-vindas ao jogador

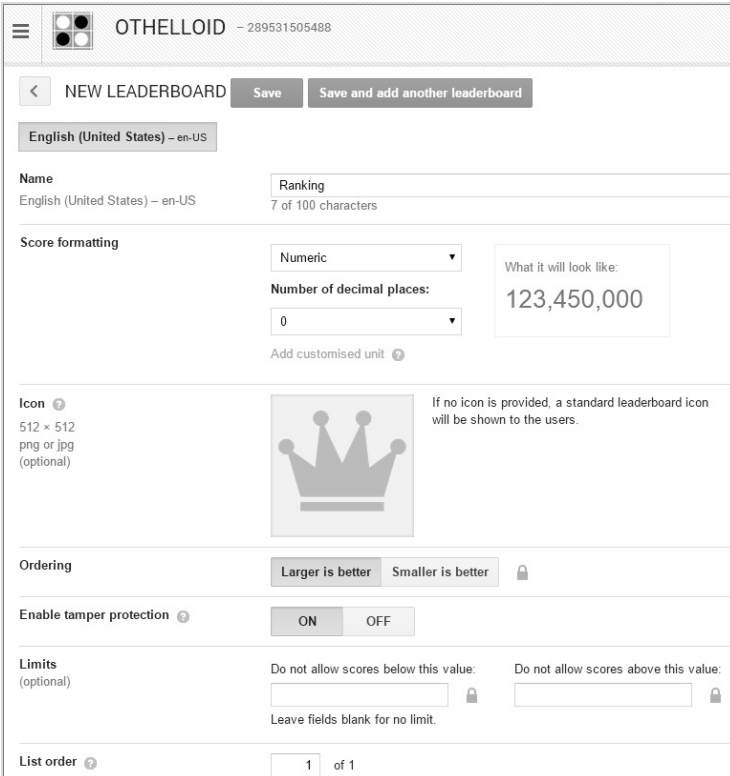
A partir daqui, podem ser implementadas e adicionadas novas funcionalidades ao jogo. As próximas secções explicam como adicionar *leaderboards* (tabelas de classificação) e *achievements* (conquistas) ao jogo *Othelloid*.

6.2.2.4 LEADERBOARDS

Os *leaderboards* (ou tabelas de classificação) permitem visualizar uma determinada classificação dos jogadores num jogo. Esta secção indica como usar a API `Leaderboard` para submeter a pontuação de um jogador e mostrar a tabela de classificação respetiva. Para implementar um *leaderboard* num jogo Android são necessárias duas tarefas:

- 1) Criar um *leaderboard* na GPDC.
- 2) Codificar a submissão da pontuação do jogador e a exibição do *leaderboard*.

Para criar um *leaderboard* para o jogo *Othelloid*, aceda à consola, selecione o jogo, depois o separador **Leaderboards** e, de seguida, clique no botão **Add leaderboard**. Surge uma página com um formulário para a criação do *leaderboard* (Figura 6.9).



The screenshot displays the 'NEW LEADERBOARD' configuration screen for the 'OTHELLOID' app. The interface includes a navigation bar at the top with a hamburger menu, the app name 'OTHELLOID', and a unique identifier '-289531505488'. Below the navigation bar, there are two buttons: 'Save' and 'Save and add another leaderboard'. The main form is organized into several sections:

- Name:** A text input field containing 'Ranking' with a character count of '7 of 100 characters'.
- Score formatting:** A dropdown menu set to 'Numeric' and a 'Number of decimal places' dropdown set to '0'. A preview box shows 'What it will look like: 123,450,000'. There is also an 'Add customised unit' link.
- Icon:** An optional field for a 512x512 pixel icon (png or jpg). A crown icon is shown as a placeholder. A note states: 'If no icon is provided, a standard leaderboard icon will be shown to the users.'
- Ordering:** Two radio buttons: 'Larger is better' (selected) and 'Smaller is better'. A lock icon is present next to the 'Smaller is better' option.
- Enable tamper protection:** A toggle switch currently set to 'ON'.
- Limits (optional):** Two input fields for 'Do not allow scores below this value' and 'Do not allow scores above this value'. A note below says 'Leave fields blank for no limit.' Lock icons are present next to the input fields.
- List order:** A dropdown menu set to '1 of 1'.

FIGURA 6.9 – Criação de um *leaderboard*

No campo **Name** insira um nome curto (até 100 caracteres) para o *leaderboard*.

No campo **Score formatting** deve definir o formato da pontuação. Apesar de todas as pontuações submetidas serem armazenadas internamente como inteiros longos, o serviço pode apresentá-las ao utilizador em vários formatos:

- ⊙ **Numérico** – número com ou sem casas decimais;
- ⊙ **Tempo** – submissão em milissegundos (por exemplo, 66 032 representa 1:06:03);
- ⊙ **Moeda** – submissão como 1/1 000 000 da unidade da moeda principal (por exemplo, 19,95 milhões de euros seria interpretado como US \$19.95, supondo que especificou a moeda como USD).



O formato numérico também suporta unidades personalizadas. Por exemplo, se o jogo mede pontuações em metros, pode-se especificar “metros” como a unidade padrão para o *leaderboard*. Para tal, basta clicar no **link Add customised unit** e acrescentar o texto “metros”.

No campo **Icon**⁶³ pode definir uma imagem (512x512 PNG ou JPEG) que será mostrada aos utilizadores para representar o *leaderboard*.

No campo **Ordering** define-se um de dois tipos de ordenação:

- ⊙ **Larger is better** – ordenação padrão. Tipicamente, é a opção desejada em jogos nos quais os jogadores amealham pontos;
- ⊙ **Smaller is better** – são usados, ocasionalmente, nos casos em que uma pontuação menor seria melhor. Os exemplos mais comuns deste tipo de *leaderboard* estão em jogos de corrida, nos quais a pontuação representa o tempo do jogador para terminar a corrida.

O campo **Enable tamper protection** permite ativar a proteção contra pontuações suspeitas, adulterando assim a classificação apresentada no *leaderboard*. A proteção inibe a visualização dessas pontuações.

O campo opcional **Limits** permite definir valores para os limites inferior e superior da pontuação que são permitidos na tabela de classificação. Isso pode ajudá-lo a descartar submissões de pontuação que são claramente fraudulentas.

O campo **List Order** define a ordem pela qual os *leaderboards* são exibidos aos utilizadores. Os jogos podem ter várias tabelas de classificação (até um máximo de 70). Por exemplo, um jogo multinível pode fornecer um *leaderboard* diferente para cada nível ou um jogo de corrida ter um *ranking* separado para cada pista.

Crie um *leaderboard* para armazenar a pontuação numérica do jogo. Após o preenchimento do formulário, clique no botão **Save** e guarde o seu identificador. Do lado da consola, precisa-se apenas de obter o identificador do *leaderboard*. Na janela com a lista

⁶³ Pode descarregar uma galeria de ícones para o Google Play Games Services através do *link*: <https://developers.google.com/games/services/downloads/games-branding-icons.zip>

de *leaderboards* (Figura 6.10) clique no *link* **Get resources** e copie o elemento `string` que contém o identificador do *leaderboard* para o ficheiro de recurso `res/values/games-ids.xml` do projeto Android.

```
<string name="leaderboard_ranking">CgkI0Nawy7YIEAIQAQ</string>
```

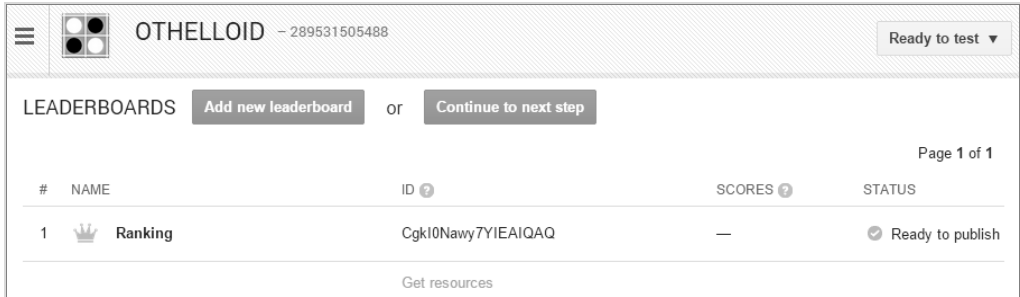


FIGURA 6.10 – Listagem de *leaderboards*

De seguida, passa-se para a codificação do jogo *Othelloid* de forma a integrar o *leaderboard*. As classes e interfaces que lidam com os *leaderboards* estão incluídas no pacote `com.google.android.gms.games.leaderboard`.

A interface `Leaderboards` é o ponto de entrada para grande parte das funcionalidades associadas aos *leaderboards*. A interface inclui vários métodos, destacando-se os apresentados na Tabela 6.3.

MÉTODO	DESCRIÇÃO
<code>getLeaderboardIntent(GoogleApiClient client, String lbId)</code>	Obtém um objeto <code>Intent</code> com um <i>leaderboard</i> de um jogo.
<code>loadPlayerCenteredScores(GoogleApiClient client, String lbId, int span, int lbCollection, int maxResults)</code>	Carrega assincronamente a página da pontuação centrada no jogador para um determinado <i>leaderboard</i> . Se o jogador não tiver uma pontuação, é devolvida a página de topo. O carregamento é feito mediante um determinado espaço temporal (<code>span</code> pode ser <code>TIME_SPAN_DAILY</code> , <code>TIME_SPAN_WEEKLY</code> ou <code>TIME_SPAN_ALL_TIME</code>) para um determinado âmbito (<code>lbCollection</code> pode ser <code>COLLECTION_PUBLIC</code> ou <code>COLLECTION_SOCIAL</code>) e com um número máximo de pontuações (<code>maxResults</code>) por página (entre 1 e 25). Devolve um objeto <code>PendingResult</code> para aceder aos dados, quando disponíveis.
<code>loadTopScores(GoogleApiClient client, String lbId, int span, int lbCollection, int maxResults)</code>	Carrega assincronamente a página de topo da pontuação para um determinado <i>leaderboard</i> . Os restantes parâmetros são similares ao método <code>loadPlayerCenteredScores</code> . Devolve um objeto <code>PendingResult</code> para aceder aos dados, quando disponíveis.

MÉTODO	DESCRIÇÃO
<pre>submitScore(GoogleApiClient client, String lbId, long score)</pre>	<p>Envia uma pontuação para um determinado <i>leaderboard</i> para o jogador conectado no momento. A pontuação é ignorada se for pior do que a pontuação anteriormente apresentada para o mesmo jogador. Este método é do tipo <i>fire-and-forget</i>, ou seja, use-o se não precisar de ser notificado dos resultados da submissão da pontuação, embora note que a atualização pode não ser enviada para o servidor até à sincronização seguinte. O significado do valor <i>score</i> depende da formatação do <i>leaderboard</i> estabelecido na consola.</p>
<pre>submitScoreImmediate(GoogleApiClient client, String lbId, long score)</pre>	<p>Método semelhante ao anterior com exceção de que tentará imediatamente submeter a pontuação para o servidor e devolverá um objeto <code>PendingResult</code> com informações sobre a submissão.</p>

TABELA 6.3 – Métodos da interface `Leaderboards`

SUBMISSÃO DE PONTUAÇÃO

Após iniciar o jogo *Othelloid* e se conectar com sucesso ao GPGS, o jogador pode começar a jogar selecionando a opção **Jogar** do menu principal. Ao clicar nessa opção, é iniciada uma nova atividade chamada `Jogo` que associa o *layout* do tabuleiro materializado na classe `TabuleiroJogo`. Quando o jogador termina a sua *performance* no jogo, é necessário atualizar a sua pontuação na tabela de classificação (Figura 6.11).

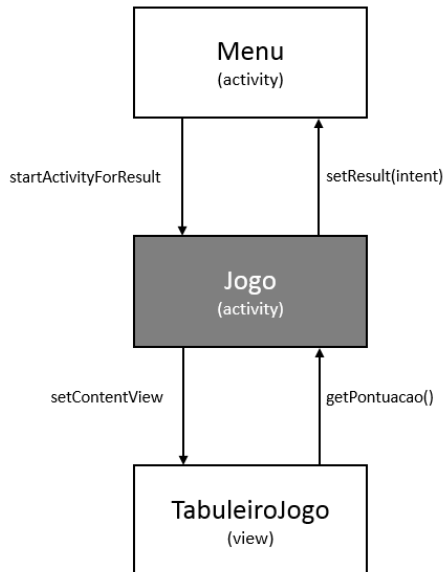


FIGURA 6.11 – Listagem de *leaderboards*

Como já foi referido no Capítulo 3, na atividade `Jogo` é implementado o método `finish` definindo uma nova *intent*. Nesta nova *intent* é associada a pontuação do jogador obtida através do método `getPontuacao` da classe `TabuleiroJogo` e enviada a *intent* para a atividade que iniciou a chamada através do método `setResult`:

```
@Override
public void finish() {
    Intent responseIntent = new Intent();
    responseIntent.putExtra("score", tabuleiroJogo.getPontuacao());
    setResult(RESULT_OK, responseIntent);
    super.finish();
}
```

Como a atividade `Jogo` foi iniciada com o método `startActivityForResult`, a *intent* anterior é recebida na atividade principal no método `onActivityResult`:

```
@Override
protected void onActivityResult(int requestCode,
    int resultCode, Intent data) {
    if(resultCode==RESULT_OK && requestCode == REQUEST_PLAY) {
        if(data.hasExtra("score")) {
            Bundle b = data.getExtras();
            score = b.getInt("score");
            fullBoard = b.getBoolean("fullboard");
            myScore = true;
            myGoogleApiClient.connect();
        }
    }
}
```

Usa-se o método `getExtras` do objeto `Intent` para obter um mapa de dados da *intent* previamente adicionados através do método `putExtra`. Depois, usa-se o método `getInt` para recuperar a pontuação do jogador como um valor inteiro. Finalmente, faz-se novamente a conexão ao GPGS, que entretanto tinha sido perdida ao chamar a atividade `Jogo`. Se a conexão for bem-sucedida, é invocado o método `onConnected`:

```
@Override
public void onConnected(Bundle bundle) {
    // SE HÁ RESULTADOS PARA SUBMETER
    if(myScore) {
        // SUBMETE SCORE USANDO A API LEADERBOARD
        Games.Leaderboards.submitScore(
            myGoogleApiClient,
            getString(R.string.leaderboard_ranking),
            score);
        myScore = false;
    }
}
```

A submissão da pontuação é feita através do método `submitScore`, que recebe três parâmetros: um objeto `GoogleApiClient`, o identificador do *leaderboard* e o valor da pontuação.

EXIBIÇÃO DA TABELA DE CLASSIFICAÇÃO

A exibição do *leaderboard* é feita após o utilizador clicar na opção *Ranking* do menu principal. Para tal, inclua o método `startActivityForResult` passando-lhe a invocação do método `getLeaderboardIntent`, que devolve um objeto `Intent` responsável por exibir a GUI padrão do *leaderboard*:

```

ImageView imgRanking = (ImageView) findViewById(R.id.imgRanking);
imgRanking.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        if (myGoogleApiClient != null && myGoogleApiClient.isConnected())
        {
            startActivityForResult(
                Games.Leaderboards.getLeaderboardIntent(
                    myGoogleApiClient,
                    getString(R.string.leaderboard_ranking)),
                    REQUEST_LEADERBOARD);
        }
    } else {
        // PARA ACEDER AO LEADERBOARD É NECESSÁRIO ESTAR AUTENTICADO
    }
});

```

Os próximos ecrãs exibem os resultados desta invocação (Figura 6.12).

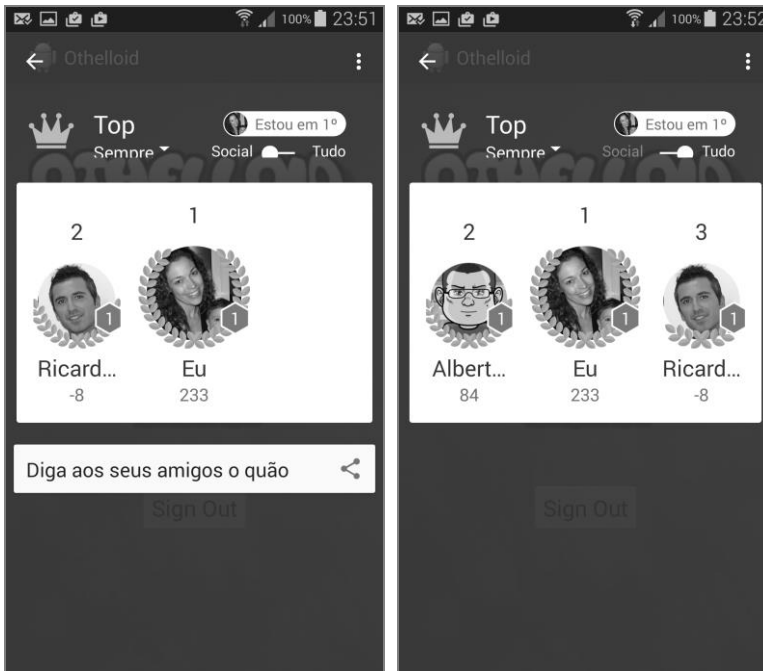


FIGURA 6.12 – *Leaderboards* sociais e públicos

O SDK do Play Games cria automaticamente *leaderboards* **diários, semanais e desde sempre**. Sendo assim, não há nenhuma necessidade de criar *leaderboards* separados para cada período de tempo.

Os *leaderboards* podem assumir dois âmbitos: **sociais** e **públicos**. A classificação social é um *leaderboard* composto por pessoas nos círculos do utilizador (ou, mais precisamente, os membros dos círculos que o utilizador escolheu para partilhar a sua aplicação) que decidiram partilhar a sua atividade de jogo com o utilizador.

O *leaderboard* público é composto por jogadores que optaram por partilhar a sua atividade publicamente. Se o jogador não tiver escolhido partilhar a sua atividade publicamente, não vai aparecer neste *ranking*.

6.2.2.5 ACHIEVEMENTS

Os *achievements* (ou conquistas) são desafios estabelecidos no jogo que, ao serem atingidos/desbloqueados, premeiam o jogador. Uma conquista pode ser, por exemplo, obter uma determinada pontuação ou concluir um determinado nível. Ao concretizar uma conquista, é exibido no ecrã do jogo uma *pop-up* que mostra o que já foi desbloqueado pelo jogador. Além disso, as conquistas ajudam o jogador a ganhar pontos de experiência no seu perfil do Play Games. As conquistas também podem ser uma maneira divertida e motivadora para os jogadores, pois estes podem comparar o seu progresso uns com os outros, potenciando a competitividade.

Esta secção mostra como usar a API *Achievement* para desbloquear conquistas no jogo. A API encontra-se no pacote `com.google.android.gms.games.achievements`. Para implementar um *achievement* num jogo Android são necessárias duas tarefas:

- 1) Criar um *achievement* na GPDC.
- 2) Codificar o desbloqueio da conquista e a exibição do *achievement*.

Para criar um *achievement* para o jogo *Othelloid*, aceda à consola, selecione o jogo, depois o separador **Achievements** e, de seguida, clique no botão **Add achievement**. Surge uma página com um formulário para a criação do *achievement* (Figura 6.13).

No campo **Name** insira um nome curto (até 100 caracteres) para o *achievement*.

O campo **Description** é uma descrição concisa sobre o *achievement*, informando o jogador como ganhar a conquista (por exemplo, “Apanhe 20 moedas nos primeiros 60 segundos”). O campo tem 500 caracteres como limite.

The screenshot shows the 'NEW ACHIEVEMENT' form in the Google Play Console. At the top, there's a navigation bar with a hamburger menu, the app name 'OTHELLOID', and a user ID '289531505488'. A 'Ready to test' button is in the top right. Below the navigation bar, there are two buttons: 'Save' and 'Save and add another achievement'. The form is for an achievement in 'English (United States) - en-US'. The 'Name' field contains 'Nivel FÁCIL' (11 of 100 characters). The 'Description' field contains 'Este desafio é conquistado quando o jogador ganha um jogo ao computador no nível fácil.' (87 of 500 characters). The 'Icon' field shows a placeholder image of a gear with a ribbon. The 'Incremental achievements' field has a checkbox that is unchecked. The 'Initial state' field has two buttons: 'Revealed' (selected) and 'Hidden'. The 'Points' field contains '200' (200 of 1,000 achievement points distributed). The 'List order' field contains '1' (1 of 1).

FIGURA 6.13 – Criação de um *achievement*

No campo **Icon** pode definir uma imagem (512x512 PNG ou JPEG) que será mostrada aos utilizadores para representar o *achievement*.

No campo **Incremental Achievements** deve indicar se a conquista é incremental ou não. As conquistas são tipificadas como padrão ou incrementais. Geralmente, uma conquista incremental envolve um jogador a fazer um progresso gradual para ganhar uma conquista por um longo período de tempo. O serviço GPGS mantém o controlo das informações de progresso, alerta o jogo quando o jogador atingiu os critérios necessários para desbloquear a conquista e diz ao jogador o quão longe está de alcançar a conquista.



As conquistas incrementais são cumulativas entre sessões de jogo, e o progresso não pode ser removido ou redefinido durante o jogo. Ao criar uma conquista incremental, deve-se definir o número total de passos necessários para desbloqueá-la (entre 2 e 10 000). À medida que o utilizador faz progressos no sentido de desbloquear a conquista, deve-se informar o número de passos adicionais que o utilizador fez para o Google Play Services. A partir do momento em que o número total de passos atinge o valor de desbloqueio, a conquista é desbloqueada.

No campo **Initial state** define-se o estado da conquista. As conquistas podem estar num de três estados diferentes:

- ⊙ **Escondido** – neste estado, os detalhes sobre a conquista estão escondidos do jogador. O Google Play Services fornece uma descrição genérica e um ícone reservado para a conquista enquanto está no estado oculto. É recomendável fazer uma conquista oculta se contiver um *spoiler* sobre o jogo que não se quer revelar muito cedo (por exemplo, “Descubra que foi sempre um fantasma todo o tempo!”);
- ⊙ **Revelado** – significa que o jogador tem conhecimento sobre a conquista, mas que ainda não a alcançou. A maioria das conquistas começa neste estado;
- ⊙ **Desbloqueado** – significa que o jogador alcançou a conquista com sucesso. Uma conquista pode ser desbloqueada *offline*. Quando o jogo fica *online*, ele sincroniza com o Google Play Services para atualizar o estado desbloqueado da conquista.

No campo **Points** define-se o número de pontos associados a uma conquista. Este valor representa o quão difícil é alcançar a conquista. Recomenda-se valores múltiplos de 5 pontos para cada conquista, num total máximo de 200 pontos. Pode-se distribuir um máximo de 1000 pontos para todas as conquistas combinadas num jogo.

O campo **List order** define a ordem pela qual as conquistas são mostradas aos jogadores.



Um jogo deve ter pelo menos cinco conquistas antes de ser publicado. Pode-se testar com menos de cinco conquistas, mas são necessárias cinco conquistas criadas antes de publicar o jogo.

Crie cinco conquistas: quatro conquistas associadas à vitória do jogador em cada nível e uma última que é conquistada quando o jogador termina um jogo em qualquer nível, deixando o tabuleiro apenas com pedras suas (*fullboard*). Após o preenchimento de cada formulário para as cinco conquistas, clique no botão **Save**. No fim, é exibida a janela com a lista de *achievements* (Figura 6.14).

Do lado da consola, precisa-se apenas de obter os identificadores dos *achievements*. Para tal, clique no *link* **Get resources** e copie-os para o ficheiro de recurso **games-ids.xml** do projeto Android:

```
<string name="achievement_1">CgkI0Nawy7YIEAIQAg</string>
<string name="achievement_2">CgkI0Nawy7YIEAIQAw</string>
<string name="achievement_3">CgkI0Nawy7YIEAIQBA</string>
<string name="achievement_4">CgkI0Nawy7YIEAIQBQ</string>
<string name="achievement_5">CgkI0Nawy7YIEAIQBg</string>
```

#	NAME	ID	POINTS	UNLOCKED % TOTAL # / TIME	STATUS
1	Nível FÁCIL	CgkI0Navy7YIEAIQAg	200	—	Ready to publish
2	Nível INTERMÉDIO	CgkI0Navy7YIEAIQAw	200	—	Ready to publish
3	Nível DIFÍCIL	CgkI0Navy7YIEAIQBA	200	—	Ready to publish
4	Nível AVANÇADO	CgkI0Navy7YIEAIQBQ	200	—	Ready to publish
5	FULL BOARD	CgkI0Navy7YIEAIQBg	200	—	Ready to publish

Get resources Total points: 1,000

FIGURA 6.14 – Listagem de *achievements*

De seguida, passa-se para a codificação do jogo *Othelloid* de forma a integrar os *achievements*. As classes e interfaces que lidam com os *achievements* estão incluídas no pacote `com.google.android.gms.games.achievement`. Neste pacote está incluída a interface `Achievements`, que é o ponto de entrada para grande parte das funcionalidades associadas aos *achievements*. A interface inclui vários métodos, destacando-se os seguintes apresentados na Tabela 6.4.

MÉTODO	DESCRIÇÃO
<code>getAchievementsIntent (GoogleApiClient client)</code>	Obtém um objeto <code>Intent</code> para mostrar a lista de conquistas de um jogo.
<code>increment (GoogleApiClient apiClient, String id, int numSteps)</code>	Incrementa uma determinada conquista incremental (<code>id</code>) por um dado número de etapas (<code>numSteps</code>). Uma vez atingida a conquista com, pelo menos, o número máximo de passos, esta será automaticamente desbloqueada. Quaisquer outros incrementos serão ignorados. Use este método se não precisar de saber imediatamente o <i>status</i> da operação. Para a maioria das aplicações, este será o método preferido de usar, embora tenha em atenção que a atualização só é enviada para o servidor na sincronização seguinte. Para obter uma atualização e informação imediatas sobre como correu a operação, use o método <code>incrementImmediate</code> , que devolverá um objeto <code>GamesPendingResult</code> .
<code>reveal (GoogleApiClient apiClient, String id)</code>	Revela uma conquista escondida ao jogador. Se a conquista já foi desbloqueada, este método não terá qualquer efeito. O método <code>revealImmediate</code> tentará atualizar de imediato a conquista do utilizador no servidor e devolverá um objeto a <code>GamesPendingResult</code> , que pode ser usado para recuperar o resultado.

MÉTODO	DESCRIÇÃO
<pre>setSteps(GoogleApiClient apiClient, String id, int numSteps)</pre>	<p>Define uma conquista para ter, pelo menos, um dado número de etapas concluídas. Ao chamar esse método, enquanto a conquista tiver mais passos do que o valor fornecido, esse método é ignorado. Uma vez que a conquista atinge o número máximo de passos, a conquista será automaticamente desbloqueada e quaisquer outras operações de mutação serão ignoradas. O método <code>setStepsImmediate</code> tentará atualizar de imediato a conquista do utilizador no servidor e devolverá um objeto a <code>GamesPendingResult</code>, que pode ser usado para recuperar o resultado.</p>
<pre>unlock(GoogleApiClient apiClient, String id)</pre>	<p>Desbloqueia uma conquista para o jogador atual. Se a conquista estiver escondida, vai revelá-la ao jogador. O método <code>unlockImmediate</code> tentará atualizar de imediato a conquista do utilizador no servidor e devolverá um objeto a <code>GamesPendingResult</code>, que pode ser usado para recuperar o resultado.</p>

TABELA 6.4 – Métodos da interface `Achievements`

DESBLOQUEIO DE CONQUISTA

Após iniciar o jogo *Othelloid* e se conectar com sucesso ao GPGS, o jogador pode começar a jogar. Após terminar um jogo, a informação relativamente à pontuação segue o fluxo já explicado na secção anterior. Depois, no método `onConnected`, a grande diferença em relação ao *leaderboard* é verificar-se se o jogador ganhou ao computador (pontuação maior do que 0) e, baseado no nível, desbloquear-se a conquista respetiva. Para desbloquear uma conquista invoca-se o método `unlock`, que recebe dois parâmetros: um objeto `GoogleApiClient` e o identificador do *achievement*:

```
public void onConnected(Bundle bundle) {
    ...

    // SE HÁ RESULTADOS PARA SUBMETER
    if(myScore) {
        ...

        // SUBMETE CONQUISTA BASEADA NO NÍVEL CASO TENHA GANHO (PONTUAÇÃO SUPERIOR A 0)
        if(score>0) {
            switch (nivel) {
                case 1: Games.Achievements.unlock(myGoogleApiClient,
                    getString(R.string.achievement_1)); break;
                case 2: Games.Achievements.unlock(myGoogleApiClient,
                    getString(R.string.achievement_2)); break;
                case 3: Games.Achievements.unlock(myGoogleApiClient,
                    getString(R.string.achievement_3)); break;
            }
        }
    }
}
```

```
case 4: Games.Achievements.unlock(myGoogleApiClient,
    getString(R.string.achievement_4)); break;
}

// SUBMETE CONQUISTA EM CASO DE FULLBOARD
if(fullBoard) {
    Games.Achievements.unlock(myGoogleApiClient,
        getString(R.string.achievement_5));
    fullBoard = false;
}
}
myScore = false;
}
```

Um elemento visual flutuante surge no topo do ecrã, informando o jogador sobre a conquista (Figura 6.15).



FIGURA 6.15 – Conquista de um *achievement*

Se a conquista for do tipo incremental (ou seja, são necessárias várias etapas para desbloqueá-la), use o método `increment`. Não é necessário escrever código adicional para desbloquear uma conquista. O GPGS desbloqueia automaticamente a conquista, uma vez que atinge o número necessário de passos:

```
Games.Achievements.increment(
    myGoogleApiClient,
    "my_incremental_achievement_id",
    1);
```

EXIBIÇÃO DA LISTA DE CONQUISTAS

A exibição da lista de conquistas é feita quando o utilizador clica na opção **Conquistas** do menu principal. Para tal, use o método `startActivityForResult` passando-lhe a invocação do método `getAchievementsIntent`, que devolve um objeto `Intent` responsável por exibir a GUI padrão da lista de *achievements*. O objeto `Intent` é o primeiro parâmetro do método `startActivityForResult`. No próximo excerto de código, o `REQUEST_ACHIEVEMENTS` é um inteiro arbitrário usado como código de pedido:

```
ImageView imgConquistas = (ImageView) findViewById(R.id.imgConquistas);
imgConquistas.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        if (myGoogleApiClient != null && myGoogleApiClient.isConnected())
        {
            startActivityForResult(
                Games.Achievements.getAchievementsIntent(
                    myGoogleApiClient), REQUEST_ACHIEVEMENTS);
        }
    } else {
        // PARA ACEDER À LISTA DE CONQUISTAS É NECESSÁRIO ESTAR AUTENTICADO
    }
});
```

Os próximos ecrãs exibem os resultados desta invocação (Figura 6.16).



FIGURA 6.16 – Listagem de *achievements* conquistados durante o jogo

6.2.3 TESTE E PUBLICAÇÃO DOS SERVIÇOS

Para garantir que os serviços estão a funcionar corretamente no jogo, convém testar a aplicação antes de publicá-la. Para tal, deve adicionar contas de utilizadores para fins de testes. Antes de a *app* ser publicada, apenas as contas de teste listadas na GPDC podem ser autenticadas. No entanto, mal a aplicação seja publicada, qualquer conta está autorizada a aceder aos serviços. Para adicionar contas de utilizadores para testes execute os seguintes passos:

- 1) Abra a página **Testing** do jogo na GPDC e clique no botão **Add testers**.
- 2) Na caixa de diálogo que aparece (Figura 6.17) digite os endereços de e-mail das contas da Google que deseja adicionar como *testers* (um por linha).

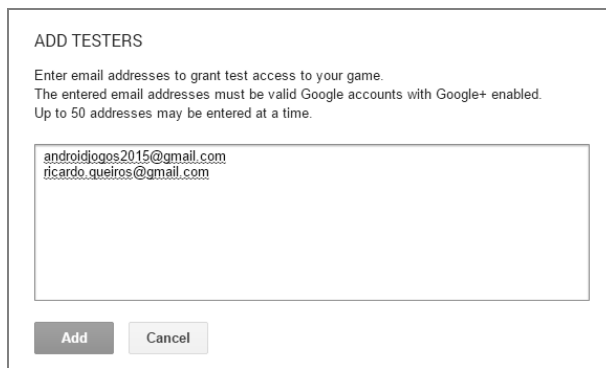


FIGURA 6.17 – Adição de endereços de e-mail para testar o jogo

- 3) Clique no botão **Add** para gravar as contas. A partir daqui, as contas adicionadas já acedem aos serviços de jogo (Figura 6.18).

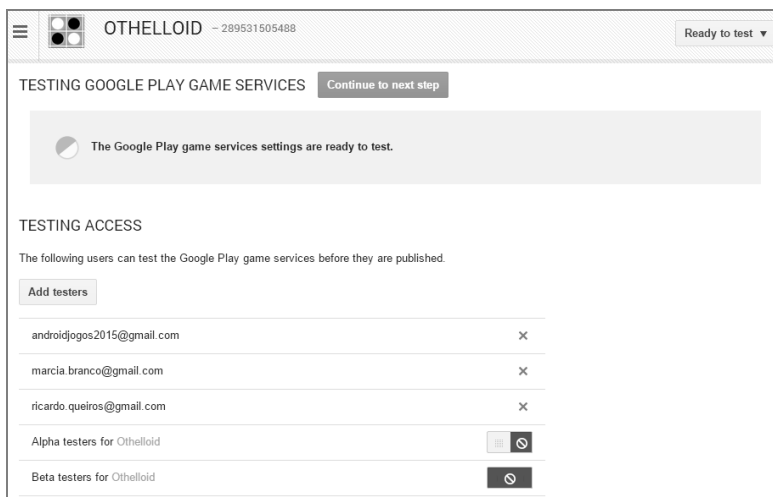


FIGURA 6.18 – Contas de *testers* para o jogo *Othelloid*



Recomenda-se que teste a aplicação num dispositivo físico. Contudo, se não tiver um dispositivo físico, é possível testá-la num emulador. Para tal, deve descarregar uma imagem do sistema que inclua o Google Play Services, na versão Android 4.2.2, a partir do SDK Manager.

Para executar o jogo num dispositivo físico, a fim de o testar:

- 1) Verifique se possui uma conta listada nas contas de teste autorizadas para testar o jogo.
- 2) Exporte o APK e assine-o com o mesmo certificado que usou para criar o projeto na GPDC. Para exportar um APK assinado no Android Studio clique em **Build** → **Generate Signed APK**.
- 3) Instale o APK assinado no dispositivo (usando a ferramenta **adb** ou partilhando o APK na *cloud*, por exemplo, através de uma conta *dropbox*).



Quando executa a aplicação diretamente a partir do Android Studio, este irá assinar a aplicação, por omissão, com o seu certificado de *debug*. Se não usar esse certificado *debug* ao configurar a aplicação na GPDC, isso irá causar erros. Certifique-se de que executa um APK que exportou e assinou com um certificado que corresponde a um dos certificados utilizados durante a instalação da aplicação na GPDC.

Após o teste da aplicação, está na hora de publicar os serviços do jogo de forma a que estes fiquem disponíveis para todos os utilizadores. Para publicar as alterações do jogo acesse à página **Publishing** na GPDC (Figura 6.19). Se faltarem itens (ou estiverem mal configurados) que impeçam a publicação do jogo, a página informa quais são esses itens para que possa corrigi-los. Caso contrário, deve apenas clicar no botão **Publish your game**.

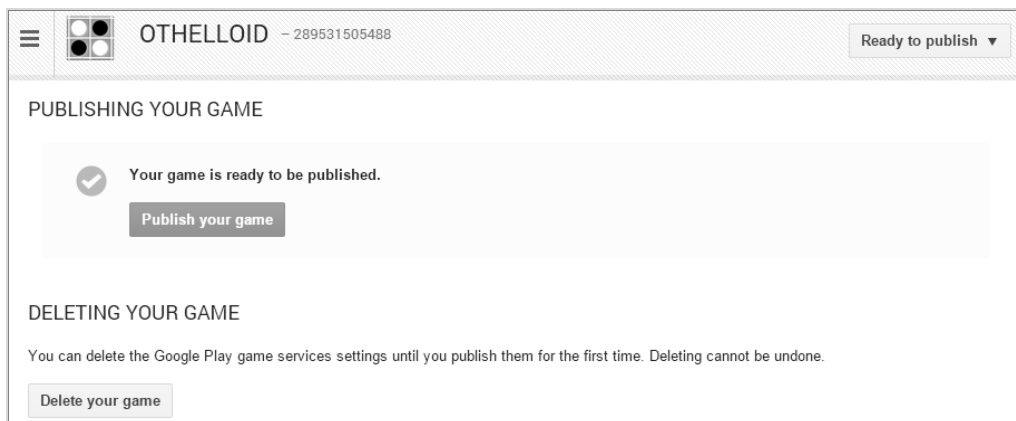


FIGURA 6.19 – Publicação dos serviços do jogo *Othelloid*

6.3 PUBLICAÇÃO DO JOGO NA GOOGLE PLAY

Esta secção resume algumas das tarefas que precisa de completar antes de distribuir a sua aplicação por todos os utilizadores.

6.3.1 VISÃO GERAL DA PUBLICAÇÃO

A publicação (*publishing*) é o processo geral que faz com que as suas aplicações Android fiquem disponíveis para os utilizadores. A Figura 6.20 mostra como o processo de publicação se encaixa no processo global de desenvolvimento de aplicações Android. O processo de publicação é normalmente realizado após terminar de testar a sua aplicação num ambiente de depuração. Quando se pretende publicar uma aplicação Android é necessário executar duas tarefas principais:

- 1) Preparar a aplicação para *release* – antes de publicar uma aplicação e de a distribuir pelos utilizadores, é necessário preparar a aplicação (testar e gerar uma versão final) e os seus materiais promocionais.
- 2) Distribuir a aplicação pelos utilizadores – durante esta etapa deve-se publicar, divulgar e vender a versão da aplicação junto dos utilizadores.

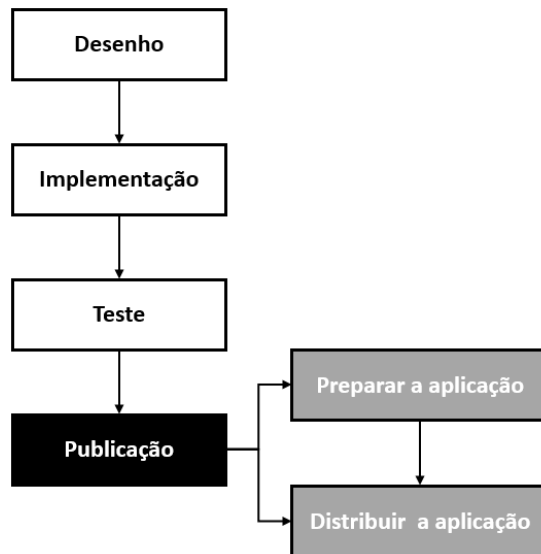


FIGURA 6.20 – Processo de desenvolvimento de aplicações Android

Nas próximas secções detalham-se as várias etapas necessárias à preparação da aplicação e à sua consequente distribuição por todos os utilizadores.

6.3.2 PREPARAÇÃO DA APLICAÇÃO

Esta secção resume as principais tarefas que precisa de executar para preparar a sua aplicação com vista à fase final de distribuição por todos os utilizadores. Para preparar a aplicação para distribuição pública é necessário executar quatro tarefas principais:

- 1) Reunir materiais e recursos.
- 2) Configurar a aplicação.
- 3) Construir a versão final da aplicação em modo *release*.
- 4) Testar a versão final da aplicação.

6.3.2.1 REUNIR MATERIAIS E RECURSOS

Para começar a preparar a sua aplicação para distribuição é necessário reunir vários itens:

- ⊗ **Ícone de aplicação** – inclua um ícone de aplicação que obedeça às diretrizes de ícone recomendadas. Um ícone da aplicação ajuda os utilizadores a identificarem a sua aplicação no ecrã inicial do dispositivo, no *launcher*, no gestor de aplicações, etc. Para além disso, os serviços de publicação, tais como o Google Play, exibem o ícone para os utilizadores;
- ⊗ **Contrato de Licença de Utilizador Final (EULA)** – o EULA pode ajudar a proteger a sua pessoa, organização e propriedade intelectual, pelo que se recomenda que forneça um com a sua aplicação;
- ⊗ **Outros recursos** – materiais promocionais e de *marketing* para divulgar a sua aplicação. Por exemplo, se está a pensar usar o Google Play como meio de distribuição, vai precisar de preparar um texto promocional e de criar *screenshots* da sua aplicação.

6.3.2.2 CONFIGURAR A APLICAÇÃO

Depois de reunir todos os materiais de apoio, pode começar a configurar a aplicação para distribuição. Esta secção apresenta um resumo das alterações de configuração recomendadas, a serem feitas no código-fonte, ficheiros de recursos e ficheiro de manifesto, antes de distribuir a aplicação. Embora a maioria delas seja opcional, são consideradas boas práticas de codificação e o programador é encorajado a implementá-las.

Em alguns casos, é possível que já tenha feito essas alterações como parte de processo de desenvolvimento. As configurações são as seguintes:

- ⊙ **Escolher um bom nome para o pacote** – certifique-se de que escolhe um nome de pacote que seja adequado ao longo da vida da aplicação, já que não pode mudar o nome do pacote depois de distribuir a aplicação pelos utilizadores;
- ⊙ **Desligar o registo (*logging*) e *debug*** – certifique-se de que desativa o *log* e a opção de depuração antes de compilar a versão final da aplicação. A primeira ação pode ser feita removendo as chamadas aos métodos da classe `Log` no código-fonte. Para desativar a depuração remova o atributo `android:debuggable` do elemento `<application>` no ficheiro de manifesto, ou definindo o valor do atributo para falso. Para além disso, remova quaisquer ficheiros de *log* ou de teste criados no seu projeto;
- ⊙ **Limpar as pastas dos projetos** – limpe o projeto e certifique-se de que está em conformidade com a estrutura de diretórios recomendadas pela Google. Deixando ficheiros órfãos no projeto pode impedir que a aplicação compile e que se comporte de forma imprevisível. No mínimo, deve rever o conteúdo das pastas `lib`, `jni`, `res` e `src` e procurar ficheiros que já não sejam usados para os excluir;
- ⊙ **Rever e atualizar o ficheiro de manifesto e definições do *Gradle*** – verifique se os itens seguintes estão definidos corretamente:
 - Elemento `<uses-permission>` – especifique somente as permissões que são relevantes e necessárias para a sua aplicação;
 - Atributos `android:icon` e `android:label` – especifique valores para estes atributos, que estão localizados no elemento `<application>`;
 - Atributos `android:versionCode` e `android:versionName` – especifique valores para estes atributos, que estão localizados no elemento `<manifest>`.



Para definir as informações de versão para a aplicação, dois atributos estão disponíveis no ficheiro de manifesto e devem sempre ser preenchidos: `android:versionCode`, um valor inteiro que representa a versão do código da aplicação em relação a outras versões; e `android:versionName`, um valor *string* que representa a versão do código da aplicação e que é mostrada a todos os utilizadores. O valor é uma *string* que possa descrever a versão da aplicação como `<major>.<minor>.<point>` ou como qualquer outro tipo de identificador de versão absoluta ou relativa.

- ⊗ **Resolução de problemas de compatibilidade** – o Android fornece várias ferramentas e técnicas para tornar a sua aplicação compatível com uma ampla gama de dispositivos. Para tornar a aplicação disponível para o maior número de utilizadores considere fazer o seguinte:
 - Adicionar suporte para múltiplas configurações do ecrã;
 - Otimizar a aplicação para *tablets* Android;
 - Considere o uso da biblioteca de suporte (**Support Library**) para garantir compatibilidade com versões antigas do Android.

6.3.2.3 CONSTRUIR A VERSÃO FINAL DA APLICAÇÃO EM MODO *RELEASE*

Após terminar de configurar a aplicação, pode construir um ficheiro APK assinado e otimizado. Com o Android Studio pode automatizar todo o processo de construção.

O sistema Android exige que cada aplicação instalada seja assinada digitalmente com um certificado que é da propriedade do programador da aplicação (ou seja, um certificado em que o programador detém a chave privada). O sistema Android usa o certificado como meio de identificar o autor de uma aplicação e estabelecer relações de confiança entre as aplicações. O certificado que usa para a assinatura não precisa de ser assinado por uma autoridade certificadora; o sistema Android permite que assine as suas aplicações com um certificado autoassinado.

Pode-se assinar uma aplicação no modo *debug* ou *release*. Assina-se a aplicação no modo *debug* durante o desenvolvimento e no modo *release* quando está pronto para distribuir a sua *app*. O SDK Android gera um certificado para assinar aplicações em modo *debug*. Para assinar aplicações em modo *release* precisa de gerar o seu próprio certificado.

Para assinar a aplicação no modo *release* no Android Studio siga estes passos:

- 1) Inicie o Android Studio.
- 2) Selecione a opção do menu **Build** → **Generate Signed APK**.
- 3) Na janela **Generate Signed APK Wizard** clique no botão **Create new** para gerar uma nova *keystore*.
- 4) Na janela **New Key Store** providencie a informação necessária, conforme demonstra a Figura 6.21.
- 5) De volta à janela anterior clique no botão **Next**. Na nova janela selecione o caminho para o ficheiro APK a ser gerado e o modo de construção (*debug* ou *release*) e clique no botão **Finish** (Figura 6.22).

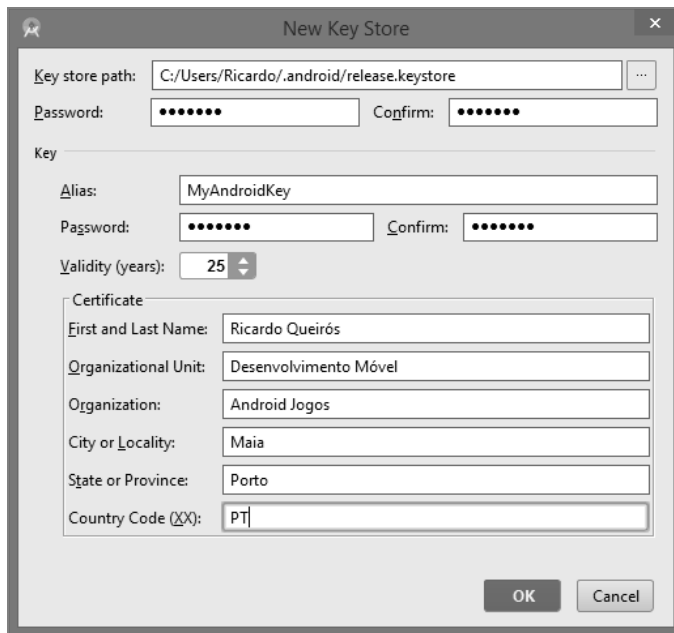
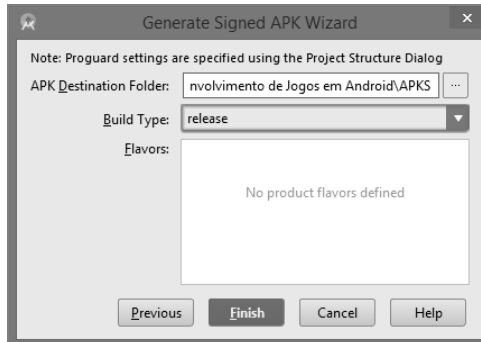
FIGURA 6.21 – Criação de uma nova *keystore* no Android Studio

FIGURA 6.22 – Geração de um APK assinado no Android Studio

Após a execução das etapas com sucesso, acaba de gerar um APK assinado no modo *release*. No Android Studio pode-se ainda configurar o projeto para assinar automaticamente o APK durante o processo de compilação. Siga os próximos passos:

- 1) No Android Studio clique com o botão direito na *app* e selecione **Open Module Settings**.
- 2) Na janela **Project Structure** selecione o módulo da aplicação e clique no separador **Signing**.

- 3) Adicione uma nova configuração (Figura 6.23) e preencha as informações relacionadas com a *keystore* criada anteriormente.

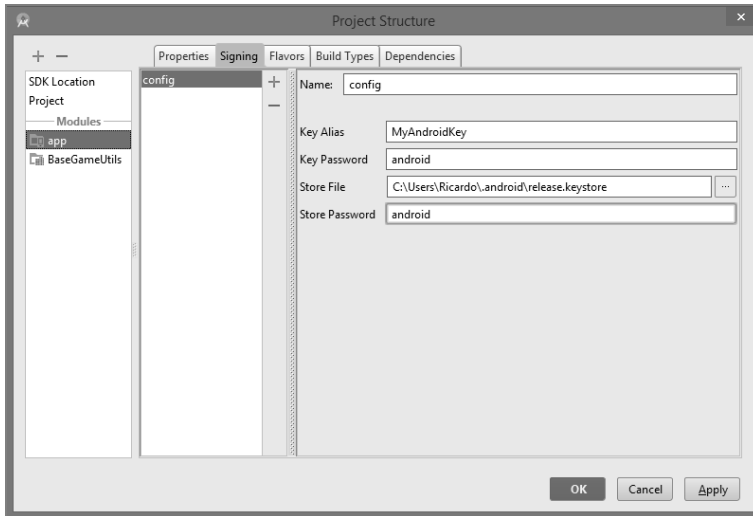


FIGURA 6.23 – Criação de uma configuração para assinatura automática no Android Studio

- 4) Clique no separador **Build Types** e selecione **Release**.
- 5) No campo **Signing Config** selecione a configuração criada (Figura 6.24).

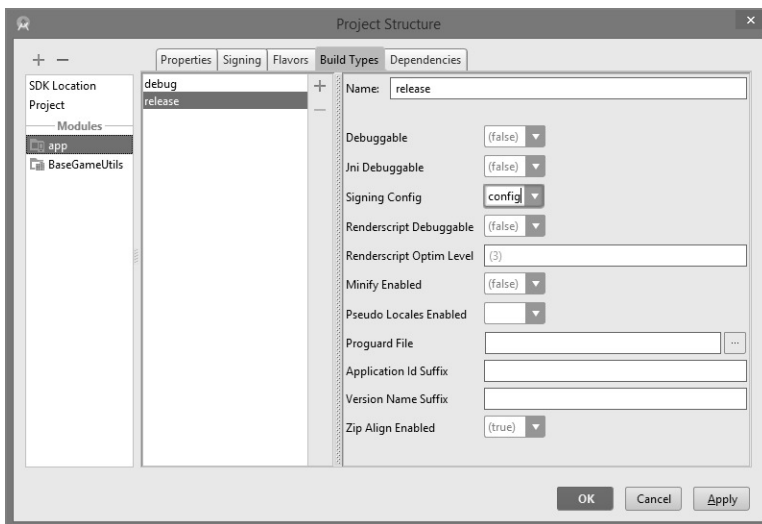


FIGURA 6.24 – Seleção de uma configuração para assinatura automática no Android Studio

- 6) Finalmente, clique no botão **OK**.

Também é possível especificar as configurações de assinatura da aplicação nos ficheiros *Gradle*.

6.3.2.4 TESTAR A VERSÃO FINAL DA APLICAÇÃO

Testar a versão *release* da sua aplicação ajuda a garantir que esta é executada corretamente no dispositivo e em condições de rede realistas. Idealmente, deve testar a sua aplicação em, pelo menos, um dispositivo *smartphone* e um *tablet* para verificar se os elementos de interface gráfica são dimensionados corretamente e se o desempenho da aplicação e eficiência da bateria são aceitáveis. Algumas ideias para teste são: alterar a orientação do ecrã, alterar a configuração do dispositivo (por exemplo, a linguagem do sistema ou a disponibilidade do teclado), verificar tempo de vida da bateria, verificar dependência com recursos externos (por exemplo, se a aplicação depende de acesso à rede, *Bluetooth*, GPS, deve testar o que acontece quando recursos, não estão disponíveis).

6.3.3 DISTRIBUIÇÃO DA APLICAÇÃO

A distribuição de uma aplicação Android pode ser feita de várias formas. Tipicamente, é feita através de uma loja *online* como a Google Play, mas pode-se também distribuir as aplicações através de um *site*, ou diretamente ao utilizador através do e-mail. Contudo, se pretender distribuir as suas aplicações para uma maior audiência, a solução ideal é a Google Play.

6.3.3.1 GOOGLE PLAY

A Google Play é a loja *online* da Google para distribuição de aplicações, jogos, filmes, música e livros (Figura 6.25). Quando publicar na Google Play, tenha a noção de que está a colocar a sua aplicação em frente a mil milhões de utilizadores Android ativos, em mais de 190 países em todo o mundo.

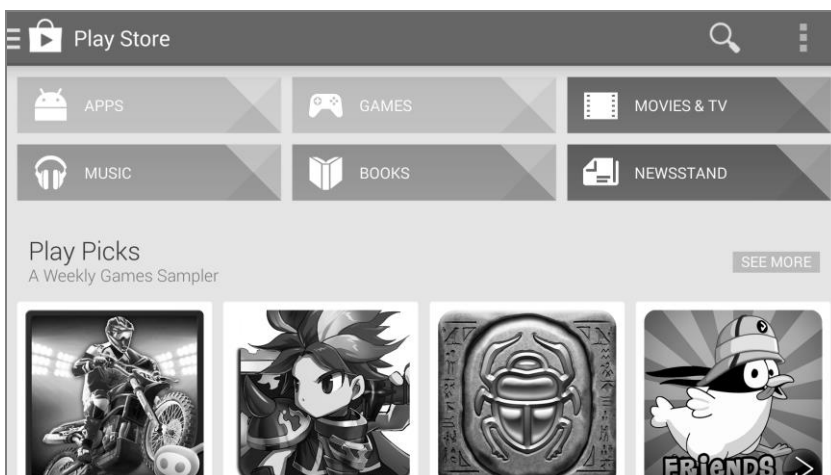


FIGURA 6.25 – Acesso à Google Play através da *app* Play Store

Seguem-se algumas razões⁶⁴ pelas quais deve priorizar o Android e a Google Play para a distribuição da sua aplicação:

- ⊙ **Popularidade** – com mais de mil milhões de utilizadores Android ativos, 84% dos *smartphones* mundiais e 66% dos *tablets* a nível mundial usam Android;
- ⊙ **Poderoso** – 86% dos dispositivos Android correm as versões Android 4.0 ou superior;
- ⊙ **Heterogeneidade** – Android Wear e Android TV providenciam mais oportunidades para cativar os utilizadores;
- ⊙ **Rentabilidade** – entre junho de 2013 e junho de 2014, a Google Play pagou cinco mil milhões de dólares aos programadores de aplicações. Ano após ano, os pagamentos a programadores cresceram mais de 250%;
- ⊙ **Crescimento** – entre janeiro e junho de 2014, com 100 milhões de novos utilizadores, o Google Play Games tornou-se na rede de jogos móveis com o mais rápido crescimento de sempre;
- ⊙ **Oferta** – mais de 50 biliões de aplicações foram descarregadas da Google Play.

6.3.3.2 PUBLICAÇÃO NA GOOGLE PLAY

O processo de publicação de uma aplicação na Google Play é composto por três etapas básicas:

- 1) **Preparação de materiais promocionais** – para aproveitar totalmente os recursos de *marketing* e publicidade da Google Play, é recomendado que crie materiais promocionais para a sua aplicação, tais como *screenshots*, vídeos, gráficos e texto promocional.
- 2) **Configuração de opções e *upload* de recursos** – a Google Play permite direcionar a sua aplicação para vários utilizadores e dispositivos. Ao configurar várias definições da Google Play, pode-se escolher os países que se quer alcançar, as línguas que se deseja usar e o preço que se pretende cobrar em cada país. Também pode configurar mais detalhes, como o tipo de aplicação, a categoria e a classificação de conteúdos. Quando terminar de

⁶⁴ Fontes: IDC, Smartphone OS Market Share (Q2 2014), Strategy Analytics Tablet & Touchscreen Strategies (TTS) service (abril de 2014) e o *link* developer.android.com/about/dashboards

configurar as opções, pode enviar os materiais promocionais e a aplicação como uma *draft app* (não publicada).

- 3) **Publicação da versão *release* da aplicação** – se as configurações de publicação estão corretas e a sua aplicação carregada está pronta para ser lançada ao público, pode simplesmente clicar em **Publish** na GPDC e em poucos minutos a aplicação ficará disponível para *download* em todo o mundo.

De seguida, apresentam-se os passos necessários a executar na GPDC para publicar o jogo *Othelloid* na Google Play.

- 1) Aceda à GPDC.
- 2) Selecione **All applications** no menu lateral esquerdo (Figura 6.26).

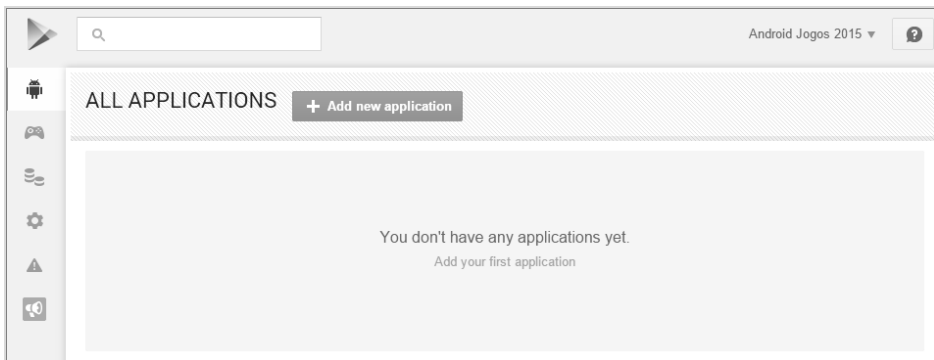


FIGURA 6.26 – Página All applications na Google Play Developer Console

- 3) Clique no botão **Add new application**. Na nova janela defina a língua por omissão e o título da aplicação. De seguida, pode optar por clicar no botão **Upload APK** ou **Prepare Store Listing** (Figura 6.27).

FIGURA 6.27 – Janela Add New Application na Google Play Developer Console

- 4) Ao clicar no botão **Store Listing** surge a página **Store Listing** (Figura 6.28), na qual deve preencher o formulário com informações sobre o jogo (título, descrição curta e longa, ícones, gráficos e vídeos promocionais, categoria, classificação de conteúdos e dados de contacto).

The screenshot shows the 'STORE LISTING' page for the game 'Othelloid'. At the top, there is a 'Draft' button. Below that, the 'PRODUCT DETAILS' section is visible. The language is set to 'Portuguese (Portugal) - pt-PT'. The form includes fields for 'Title*', 'Short description*', and 'Full description*'. The title is 'Othelloid' (9 of 30 characters). The short description is 'Jogo de tabuleiro para dois jogadores inspirado no famoso Reversi (ou Othello)' (78 of 80 characters). The full description is a detailed paragraph about the game. At the bottom, there is a link to tips on creating policy-compliant app descriptions.

FIGURA 6.28 – Página Store Listing

- 5) Adicione alguns *screenshots* do jogo (Figura 6.29). Tenha a noção de que estas serão as primeiras imagens que vão aparecer na Google Play quando os utilizadores selecionarem o jogo *Othelloid*.

The screenshot shows the 'Screenshots' section. It includes instructions: 'Default - Portuguese (Portugal) - pt-PT', 'JPEG or 24-bit PNG (no alpha). Min length for any side: 320px. Max length for any side: 3840px. At least 2 screenshots are required overall. Max 8 screenshots per type. Drag to reorder or to move between types.' Below this, there is a note about showcasing the app in the 'Designed for tablets' list. The main area shows four phone screenshots of the game and a large grey box with a plus sign and the text 'Add screenshot' and 'Drop image here'.

FIGURA 6.29 – Adicionar *screenshots* do jogo

- 6) Na página **Pricing & Distribution** (Figura 6.30) deve definir se a aplicação vai ser gratuita ou paga. Para publicar aplicações pagas é necessário configurar uma conta de comerciante. Para além desta decisão, deve identificar quais os países que poderão descarregar a aplicação e se a mesma vai estar disponível para outras plataformas (Android Wear e Android TV).

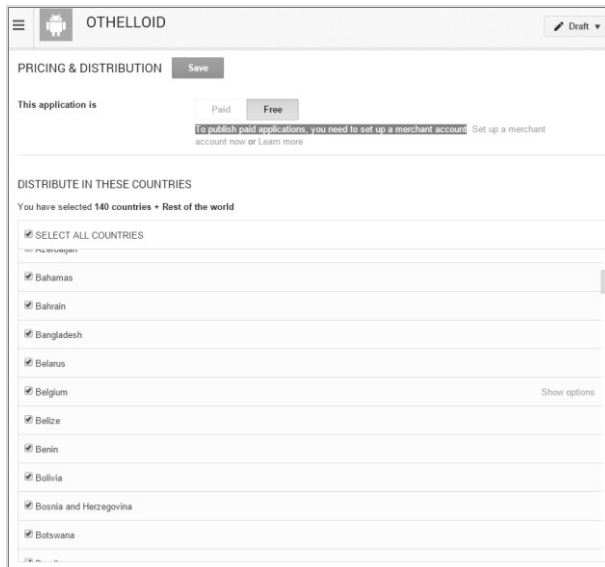


FIGURA 6.30 – Página Pricing & Distribution

- 7) Na página **APK** (Figuras 6.31 e 6.32) faça *upload* do APK para poder publicar a aplicação na Google Play, clicando no botão **Upload your first APK to Production** e adicionando o APK da versão final da aplicação no modo *release*.

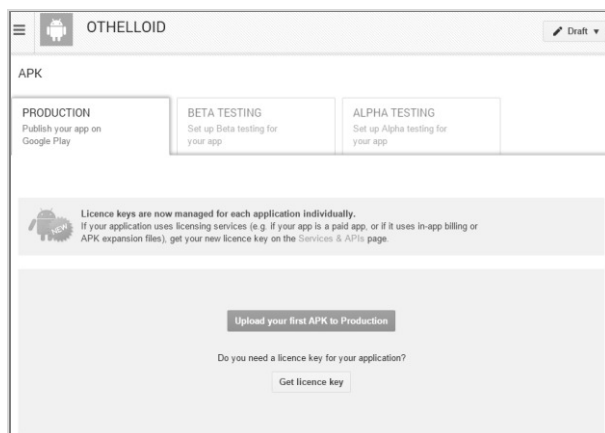


FIGURA 6.31 – Página APK

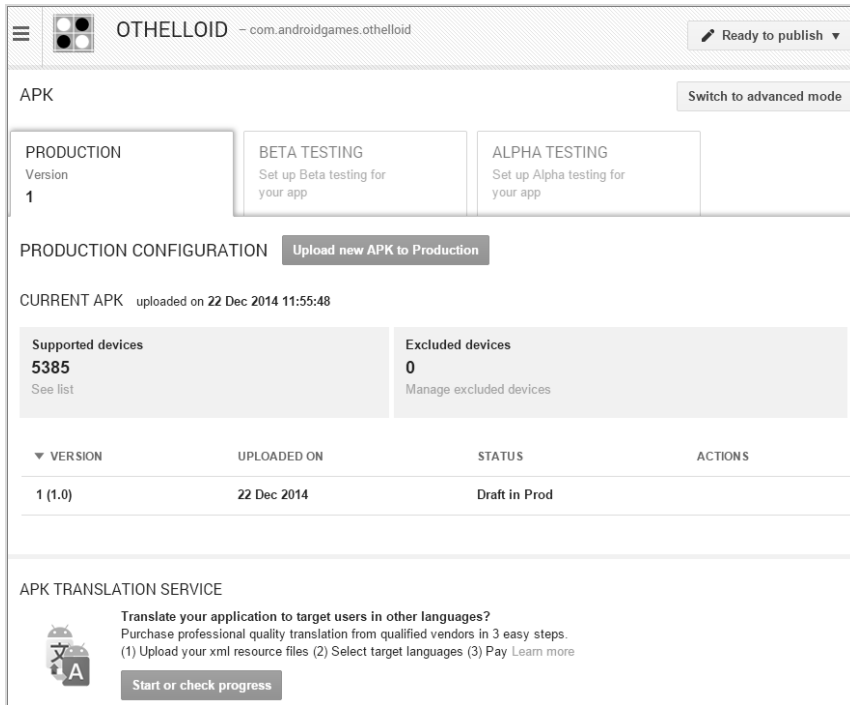
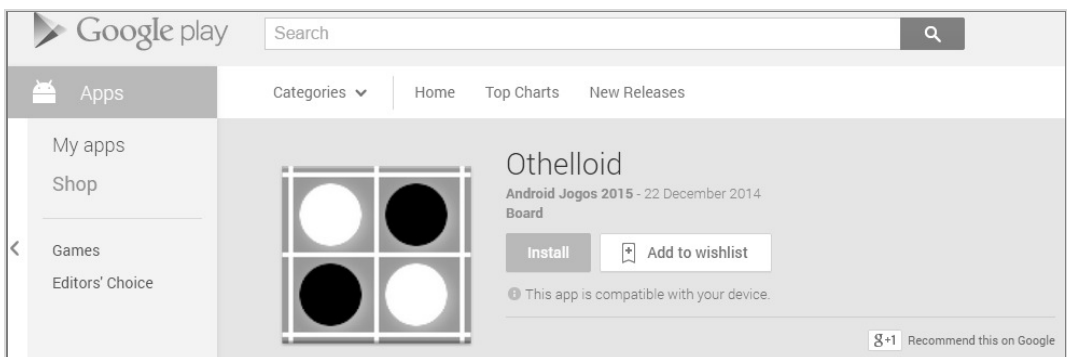


FIGURA 6.32 – Upload do APK

- 8) Falta apenas publicar a aplicação. Vá ao canto superior direito da consola e publique a *app*, seleccionando a opção **Publish this app**.

A partir daqui, já pode aceder à Google Play e descarregar o jogo *Othelloid* conforme mostra a Figura 6.33.

FIGURA 6.33 – O jogo *Othelloid* na Google Play

GLOSSÁRIO DE TERMOS PORTUGUÊS EUROPEU⁶⁵ / PORTUGUÊS DO BRASIL

Para facilitar a utilização deste livro pelos leitores brasileiros, incluímos este glossário de termos.

PORTUGAL	BRASIL
Aceder	Acessar
Aplicação	Aplicativo
Arranque	Inicialização
Barra de estado	Barra de status
Base de dados	Banco de dados
Diretório	Diretoria
Ecrã	Tela
Ficheiro	Arquivo
Fornecedor	Provedor
Gestão	Gerenciamento
Guardar/Gravar	Salvar
Ligação	Conexão
Ligar	Conectar
Modelo	Padrão
Monitorizar	Monitorar
Normalização	Estandardização, padronização
Palavra-passe	Senha
Personalizar	Customizar
Rato	Mouse
Registo	Registro
Separador	Aba
Sistema operativo	Sistema operacional
Utilizador	Usuário

⁶⁵ Designa-se por Português Europeu a variante da língua falada em Angola, Cabo Verde, Guiné-Bissau, Moçambique, Portugal, São Tomé e Príncipe e Timor-Leste.

ÍNDICE REMISSIVO

A

acelerómetro	78
<i>achievements</i>	254
<i>Activity</i>	29
<i>Android Backup Service</i>	93
Android Device Monitor	27
Android Studio	6
<i>Android Virtual Device</i>	19
App Game Kit	115
<i>Audio Source</i>	227
AVD Manager	19

B

BaseGameUtils	240
BitmapDrawable	56
Box2D	119
BroadcastReceiver	35
BulletPhysics	119

C

Camera	174
Canvas	52
Citrus Game Engine	108
Clara.io	180
ClipDrawable	50
Cocos2d-x	107
<i>Collider</i>	208
Construct 2	117
ContentProvider	40
Corona SDK	118
Cry Engine	106

D

DocumentsProvider	102
Drawable	45

G

Game Salad	112
------------------	-----

GameMaker Studio	114
Genymotion	24
GestureDetector	69
GLSurfaceView	62
Google Play	269
Google Play Developer Console	236
Google Play Games Services	234
Google Play Services	230
<i>Google Wallet</i>	236
GoogleApiClient	232
<i>Gradle</i>	17
<i>GUIText</i>	214

I

inteligência artificial	128
intent	30
IntentService	33

L

LayerDrawable	51
<i>leaderboards</i>	247
libgdx	110, 156
LogCat	37
Lollipop	2

M

Marmalade	111
Material Design	2
MediaPlayer	84
<i>Minimax</i>	128
Model	171
ModelBatch	172
ModelBuilder	173
ModelInstance	172
Modo imersivo sticky	150
MotionEvent	67
<i>Multiplayer</i>	235

N

<i>NinePatch</i>	47
------------------------	----

O		Sensores 78	
OAuth 2.0 237		Service 32	
onDraw 54		SHA1 238	
onKeyUp 76		SharedPreferences 88	
onTouchEvent 67		ShiVa 113	
OpenGL ES 61		sign-in 243	
P		SoundPool 83	
Paint 54		StateListDrawable 51	
PerspectiveCamera 174		Storage Access Framework 96	
postInvalidate 55		SurfaceView 57	
publishing 263		T	
Q		Transform 204	
Quests 235		U	
R		Unity 198	
Reach 3DX 116		Unity 3D 105	
Rigidbody 205		V	
S		View 53	
Saved Games 235		Vision Engine 109	