Introdução ao Desenvolvimento de Jogos com Unity

Introdução ao Desenvolvimento de Jogos com Unity

Alberto Simões

FCA – Editora de Informática, Lda
www.fca.pt

Edição
FCA – Editora de Informática, Lda.
Av. Praia da Vitória, 14 A – 1000-247 Lisboa
Tel: +351 213 511 448
fca@fca.pt
www.fca.pt

Distribuição
Lidel – Edições Técnicas, Lda.
Rua D. Estefânia, 183, R/C Dto. – 1049-057 Lisboa
Tel: +351 213 511 448
lidel@lidel.pt
www.lidel.pt

Livraria
Av. Praia da Vitória, 14 A – 1000-247 Lisboa
Tel: +351 213 511 448 * Fax: +351 213 522 684
livraria@lidel.pt

ISBN edição impressa: 978-972-722-883-6
1.ª edição impressa: dezembro 2017

Impressão e acabamento: Cafilesa — Soluções Gráficas, Lda — Venda do Pinheiro
Depósito Legal N.º
Capa: José Manuel Ferrão – *Look-Ahead*

A todos os alunos que o sabem, verdadeiramente, ser.

Índice Geral

O Autor

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), em Barcelos, onde é diretor do curso da Licenciatura em Engenharia em Sistemas Informáticos e do Mestrado em Engenharia em Desenvolvimento de Jogos Digitais. Tem lecionado várias unidades curriculares ligadas ao desenvolvimento de jogos digitais, quer no que toca a tecnologias, quer no que diz respeito a técnicas de desenvolvimento de jogos em rede ou à implementação de algoritmos de inteligência artificial.

A sua atividade científica tem focado o processamento de linguagem natural, sendo membro efetivo do Centro Algoritmi da Escola de Engenharia da Universidade do Minho, e membro colaborador do Centro de Estudos Humanísticos, da mesma instituição.

Agradecimentos

Há muito que este livro estava pensado, mas nunca teria avançado se não fosse a motivação do meu grande amigo **Ricardo Queirós** e a paciência da **Sandra Correia** nas minhas "esquisitices" tecnológicas. A ambos, o meu muito obrigado pelo apoio e incentivo.

Ao **Frederico Gonçalves** agradeço a oferta do modelo da formiga, que se tornou a principal personagem do jogo descrito ao longo deste livro. Também ao **José Ferreira**, um obrigado pela sempre competente ajuda na exportação e adaptação de modelos.

À **Cláudia Cruz** um obrigado pela dedicação e profissionalismo, ao **José Manuel Ferrão** um obrigado pela capa deste livro, e ao **Eng.º Frederico Annes** um agradecimento também por tornar possível esta aventura editorial.

Por fim, à minha Mãe e à minha Irmã, um obrigado especial pela paciência e apoio demonstrados ao longo destes meses.

Alberto Simões

Sobre o Livro

Esta pequena secção apresenta alguns pontos que foram tidos em conta durante a escrita deste livro. Sugere-se a sua leitura para perceber de que modo o livro está estruturado e como deverá ser lido para uma melhor compreensão.

PÚBLICO-ALVO

Este livro foi escrito tendo como público-alvo todos aqueles que, com conhecimentos básicos de programação, sejam alunos ou profissionais, pretendem aprender a usar o Unity para o desenvolvimento de jogos.

A linguagem usada é o C#, mas à parte de alguns detalhes, essencialmente no Capítulo 7, os extratos de código usam sintaxe genérica, fácil de compreender a quem tenha conhecimentos de C, C++ ou Java.

EVOLUÇÃO CONSTANTE DO UNITY

Este livro pretende apresentar os conceitos-chave do desenvolvimento de jogos usando a ferramenta Unity. Só por si, este objetivo não é fácil de alcançar, já que o Unity tem evoluído muito nos últimos anos, e cada versão que é colocada no mercado inclui novas funcionalidades, sendo que muitas das funcionalidades existentes são alteradas (e, em alguns casos, até removidas). Assim, a escrita de um livro físico sobre este assunto teve de ser bem pensada, de modo a que continue a ser útil após novas atualizações da ferramenta. Esta é a principal razão pela qual o livro não tem uma abordagem exaustiva, constituindo antes uma espécie de tutorial, em que são apresentados conceitos que são posteriormente colocados em prática num jogo concreto.

Será muito natural que, quando este livro for comprado e lido, alguns dos componentes ou das interfaces apresentados já não sejam idênticos às figuras apresentadas no livro. No entanto, as funcionalidades principais continuarão a existir. Para que o leitor tenha ideia desta evolução, quando os primeiros capítulos foram escritos, em dezembro de 2016, a versão estável do Unity era a 5.5.0. A versão 5.6.0, saída em março de 2017, levou a alguma atualização de figuras e de métodos usados no código apresentado. Em julho foi disponibilizada a versão 2017.1, que é aquela a que este livro se refere. Entretanto a versão 2017.2 já se encontra disponível e a versão 2017.3 encontra-se numa versão *beta*.

ESTRUTURA DO LIVRO

Como referido, a evolução constante do Unity não permite a escrita de um livro em que se explorem todos os detalhes e propriedades disponíveis no Unity. Assim, o livro foi escrito como se se tratasse de um tutorial, que o leitor pode acompanhar e progressivamente apreender os conceitos-chave no desenvolvimento de jogos com esta tecnologia.

À exceção do primeiro capítulo, que faz uma introdução à interface do Unity e a alguma terminologia usada, os restantes capítulos são partes integrantes do tutorial. Cada capítulo é dedicado não a uma parte tecnológica do Unity, mas a um objetivo conceptual do jogo, como se verificará até nos títulos dos capítulos e secções ao longo do livro. No final de cada capítulo o leitor deverá ser capaz de executar o jogo e poderá validar as alterações realizadas durante a leitura desse capítulo.

Por último, o índice remissivo foi preparado de modo a que o leitor possa, mais tarde, usá-lo para consultar alguns comandos e componentes do Unity.

CÓDIGO E RECURSOS

Embora os extratos de código apresentados não estejam sempre comentados, o texto que os acompanha explica o objetivo de cada comando, havendo especial cuidado em salientar aqueles comandos que são introduzidos pela primeira vez.

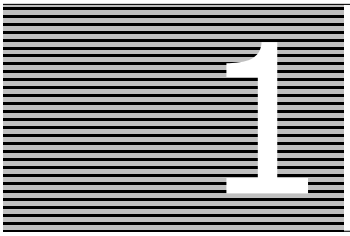
Então, o código aparece, ao longo do livro, de forma progressiva, um método de cada vez e, em várias situações, de forma incremental. Ou seja, é apresentada uma primeira versão do método e, ao longo das páginas seguintes, esse método vai sendo alterado, com a adição de novas funcionalidades. Para não repetir todo o código, e para que facilmente se perceba que se trata de um extrato parcial, recorreu-se ao uso de reticências (`// ...`).

Do mesmo modo, quando são apresentados os inícios das classes, a chaveta final é colocada raramente, já que outros métodos serão adicionados posteriormente. O leitor deverá garantir que essa chaveta existe. Em várias situações, a importação de bibliotecas (comandos `using`) não constam do extrato uma vez que, quer o MonoDevelop quer o Visual Studio, são capazes de sugerir quais as bibliotecas em falta.

Em todo o caso, o leitor poderá sempre consultar o código disponível no sítio da FCA: `http://www.fca.pt/`.

Neste endereço também encontrará uma pasta com alguns recursos usados ao longo do livro, como os modelos das personagens, texturas, sons, entre outros. Destes recursos devo salientar o modelo da formiga, preparado por **Frederico Gonçalves**,[0] que o disponibilizou gentilmente para a escrita deste livro.

[0] O Frederico Gonçalves poderá ser contactado usando o seguinte e-mail: `fredsg_16@hotmail.com`.



INTRODUÇÃO AO UNITY

Este livro pretende ser uma introdução ao Unity, destinando-se a quem tem alguns conhecimentos de programação, em particular, na linguagem C#. Dada a velocidade a que o Unity tem evoluído nos últimos anos, é difícil e inglório o trabalho de escrever um livro que tente explicar cada detalhe, ou cada menu existente na ferramenta. Muito rapidamente, as opções deixam de existir, mudam de nome, ou deixam de fazer sentido. Deste modo, este livro foi pensado de forma a que sirva de introdução à filosofia de desenvolvimento, e não especificamente aos tipos de objetos e componentes existentes no Unity. É certo que não será possível esquecer por completo a descrição de alguns objetos e componentes, descrição essa que nunca será exaustiva.

1.1 O UNITY

O Unity é um motor de jogo. Embora a definição de motor de jogo não seja consensual, existe um conjunto de funcionalidades que estão presentes, como a possibilidade de apresentação gráfica de imagens e modelos tridimensionais, a existência de uma biblioteca de simulação de física ou a reprodução de áudio. Esta lista poderá crescer com outras funcionalidades, como uma abstração para a programação de jogos em rede ou a existência de ferramentas ou algoritmos de inteligência artificial. No entanto, além destas valências, espera-se que um motor de jogo as apresente de forma integrada e que suporte uma linguagem de programação de alto nível para a integração de todas as funcionalidades.

No caso do Unity, podemos dizer sem qualquer dúvida que se define como um motor de jogo. Além de ser uma ferramenta integrada, com um editor de cenas, que permite a manipulação de gráficos 2D e 3D, de integrar um poderoso motor de física e de ser capaz de simular som 3D, inclui recursos para a programação em rede, algoritmos de cálculo de caminhos e sistemas de partículas, de entre um vasto conjunto de funcionalidades. Em termos de programação, o Unity permite que o programador use a linguagem C#,[1] que executa sobre o .net. O facto de o Unity funcionar sobre o .net permite que os jogos desenvolvidos possam ser executados em qualquer arquitetura que disponha de uma implementação .net,

[1] Historicamente também era suportada uma variante de Java/JavaScript denominada recentemente como UnityScript. No entanto, em agosto de 2017 a Unity decidiu descontinuar o suporte a esta linguagem, tal como já tinha feito para a linguagem boo.

seja a da Microsoft, seja o Mono, uma implementação de código aberto do .net. É, pois, natural que o Unity permita a criação de jogos para Windows, macOS ou Linux. Para poder abarcar um maior número de arquiteturas, o Unity também inclui um compilador capaz de transformar código na linguagem de nível intermédio da Microsoft (CIL – *Common Intermediate Language*) para C++, o que lhe permite criar jogos para muitas outras arquiteturas, desde o Android, às consolas da Microsoft e da Sony.

Outro facto que tem levado à adoção em massa do Unity por criadores de jogos independentes (habitualmente designados por *indie game developer*, é a gratuitidade da ferramenta para empresas com lucros baixos. Basta aceder a `http://unity3d.com` e descarregar o instalador. Na secção 1.2 será feita uma breve descrição do processo de instalação.

Neste livro será apresentado um único projeto em Unity, desenvolvido, capítulo a capítulo, adicionando novas funcionalidades. Nem sempre é possível ter capítulos coesos, sem introduzir conceitos um pouco à parte da funcionalidade a ser implementada, pelo que o uso do índice remissivo pode ser útil para encontrar comentários sobre determinados conceitos ao longo do livro.

Para facilitar a leitura do livro, sugere-se a leitura inicial da secção 1.3, que faz uma visita guiada ao Unity, descrevendo conceitos que serão usados nos capítulos seguintes. Essa secção é a única parte do livro que não corresponde a um passo intermédio no desenvolvimento do projeto comum, que só será desvendado no Capítulo 2.

1.2 INSTALAÇÃO DO UNITY

A instalação do Unity em Windows ou macOS é bastante simples. O processo é realizado com base num instalador semelhante ao de tantas outras aplicações. Em primeiro lugar, convém realçar que os instaladores só funcionam com conetividade, ou seja, o instalador propriamente dito é um pequeno executável que é responsável por descarregar todos os ficheiros necessários e instalá-los no devido local, o significa que, infelizmente, não é possível descarregar o programa numa máquina para posteriormente fazer a instalação noutra máquina sem conetividade.

A Figura 1.1 mostra os dois primeiros passos da instalação, que correspondem às boas-vindas ao utilizador do instalador e à aceitação da licença do Unity. De salientar que, embora o Unity seja gratuito para não profissionais, pode passar a ser pago, de acordo com os lucros obtidos pelo seu uso.

Nos dois passos seguintes, apresentados na Figura 1.2, são feitas algumas perguntas relevantes. Na janela da esquerda serão mostrados os vários módulos disponíveis. Dependendo de a instalação estar a ser feita em macOS ou em Windows, as opções disponíveis são diferentes. Mesmo em Windows, as opções serão diferentes se tiver, ou não, o Visual Studio instalado. Caso não o tenha, na lista irá aparecer a opção `Microsoft Visual Stu-`

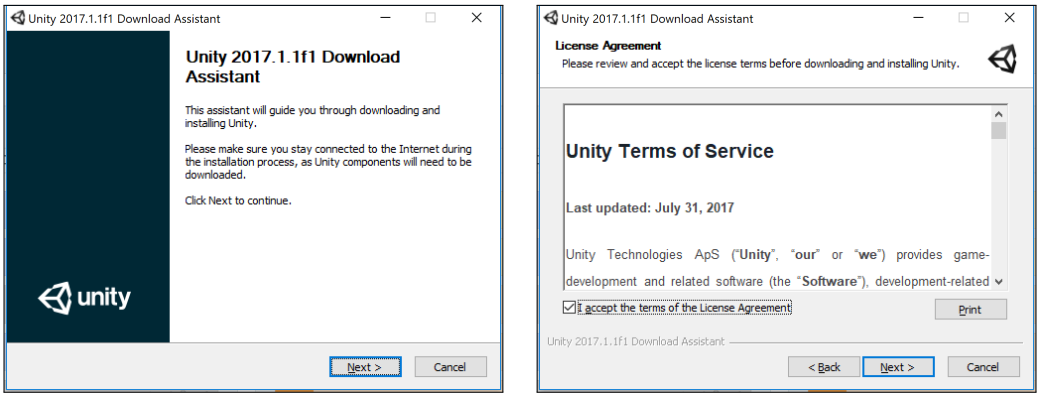


FIGURA 1.1 – 1.º e 2.º passos do processo de instalação

`dio Community` que se sugere que seja escolhida. Embora continue a ser o melhor editor disponível para programar C#, em caso de falta de espaço, é possível instalar e programar com o Unity sem o seu uso, pelo que pode optar por não o selecionar.

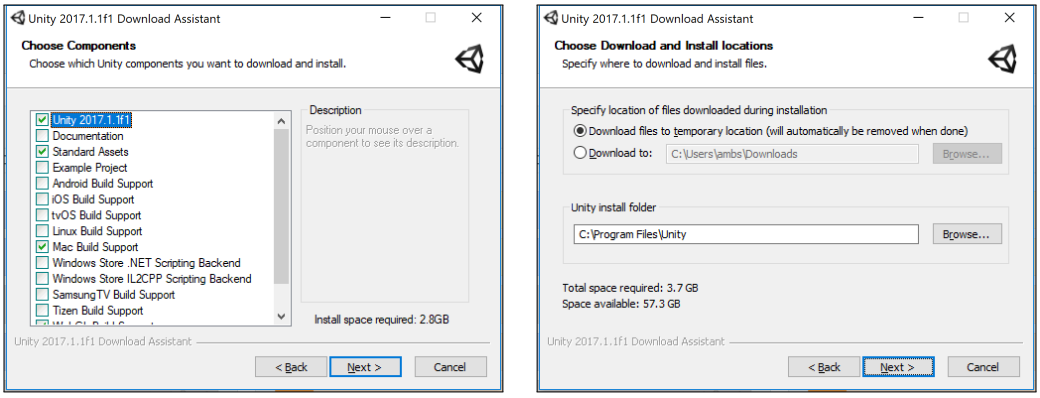


FIGURA 1.2 – 3.º e 4.º passos do processo de instalação

Da lista de módulos apresentada, sugere-se a seleção dos três primeiros itens, que correspondem, respetivamente, ao próprio Unity, à documentação (embora esteja disponível na Internet, por vezes é útil quando não há conetividade) e a um conjunto de pacotes que incluem funcionalidades básicas que podem ser integradas diretamente nos jogos (e que serão usadas ao longo deste livro). A quarta opção corresponde a um projeto de exemplo que pode ser, também, instalado e que é especialmente útil para experimentar e investigar a sua implementação.

As opções restantes correspondem a módulos de suporte para a preparação de jogos para diferentes arquiteturas. Note que a arquitetura em que está a instalar não aparece,

já que é de instalação obrigatória. As restantes são usadas para se poderem criar jogos capazes de funcionar em diferentes arquiteturas e sistemas operativos. Repare também que algumas das opções (como o suporte para Android) podem requerer a instalação de outro software, além do Unity.

Na janela apresentada na imagem do lado direito da Figura 1.2, é solicitada informação sobre onde deve ser o Unity instalado e que pasta deve ser usada para guardar os ficheiros temporários copiados da Internet. A configuração de uma pasta para albergar estes ficheiros é especialmente útil, se tiver problemas de espaço em disco e tiver de jogar com o espaço disponível num disco externo ou memória USB.

Caso tenha solicitado a instalação do Visual Studio, o passo representado na Figura 1.3, à esquerda, irá aparecer, solicitando a confirmação de aceitação da respetiva licença. Durante a instalação também surgirão janelas do Visual Studio, mas não será necessária nenhuma interação.

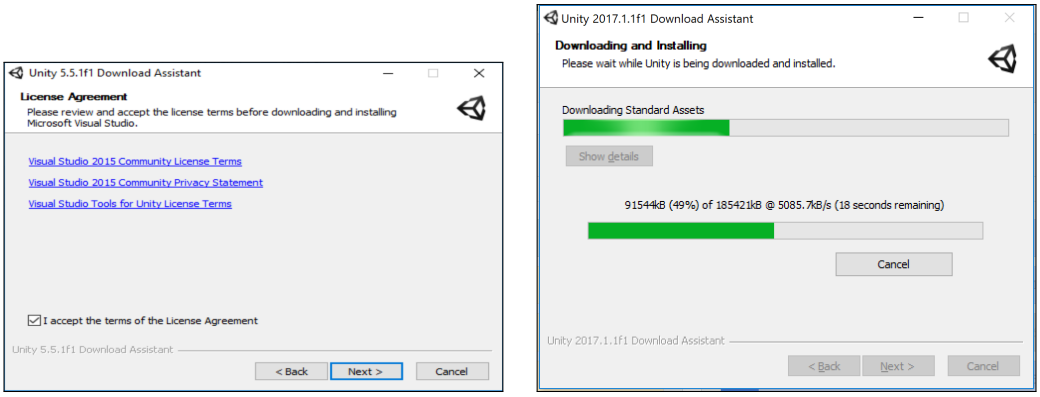


Figura 1.3 – 5.º e 6.º sexto passos do processo de instalação

O processo de instalação pode ser demorado, dependendo quer da velocidade da ligação à Internet, quer da velocidade da própria máquina onde a instalação está a ser realizada. A imagem da direita da Figura 1.3 mostra o processo de instalação[2]. Terminada a instalação, será apresentada a imagem da Figura 1.4, onde poderá selecionar se o Unity deve ser iniciado após o término da instalação.

Embora o Unity seja gratuito, é recomendado o registo do utilizador, bastando para isso aceder ao sítio do Unity e solicitar a opção de registo, como é demonstrado na Figura 1.5.

Finalmente, de realçar que, no momento em que este livro está a ser escrito, não existe uma versão oficial do Unity para Linux, embora seja possível encontrar algumas versões *beta*.

[2] Note que, durante a instalação, podem surgir novas janelas do instalador, como as já referidas, do Visual Studio, ou a duplicação da janela de instalação do Unity.

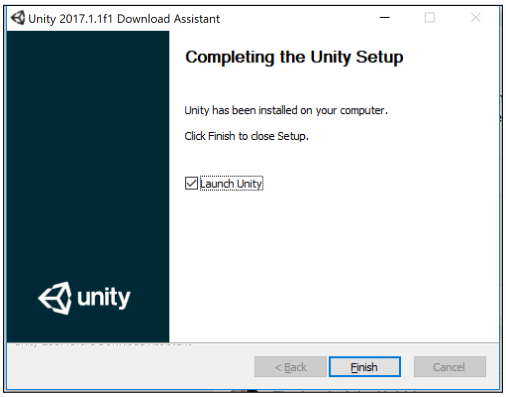


FIGURA 1.4 – Último passo do processo de instalação

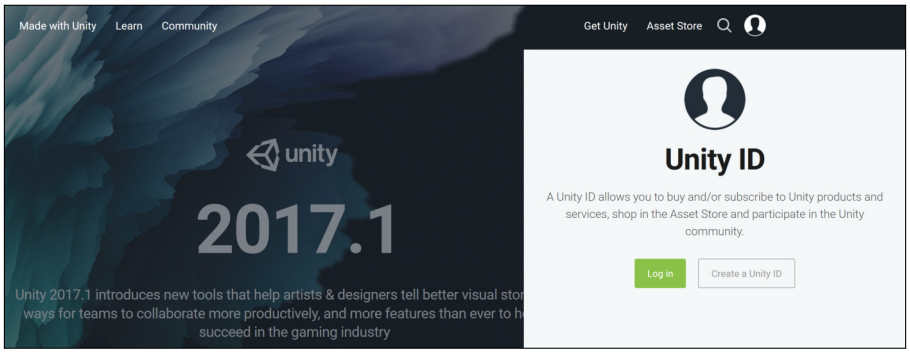


FIGURA 1.5 – Opção para a criação de um perfil no sítio do Unity

1.3 VISITA GUIADA

Esta secção descreve o processo de execução do Unity pela primeira vez e apresenta algumas das suas funcionalidades com a criação de um pequeno projeto de demonstração. Note-se que embora vários conceitos sejam abordados nesta secção, o seu objetivo não é detalhar a sua utilização, mas apenas demonstrar o seu uso.

Na primeira execução, será necessário configurar o Unity com o perfil criado (ou utilizando um perfil de uma rede social).

A Figura 1.6 é apresentada durante a primeira execução do Unity. Se tiver conetividade, será possível autenticar-se com o seu utilizador. Caso contrário, poderá executar o Unity sem autenticação.

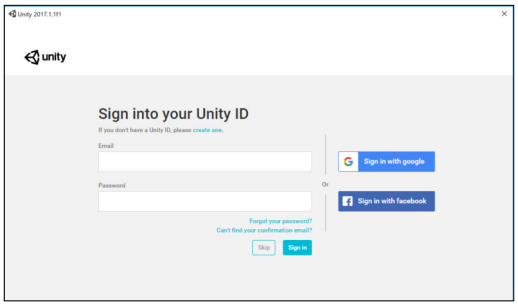


FIGURA 1.6 – Autenticação no Unity

1.3.1 CRIAÇÃO DE UM PROJETO

Sempre que o Unity é iniciado, a janela que surge à esquerda na Figura 1.7 é apresentada. Aí é possível a criação de novos projetos, ou o acesso a projetos já existentes (gravados na própria máquina, ou num dos serviços do Unity). Usando a opção `new`, para criação de um novo projeto, é apresentada a janela da direita da Figura 1.7, onde são definidas as propriedades básicas do projeto, como a pasta onde será criado e o nome da pasta para o projeto. Repare que, ao contrário de outras aplicações, como o Word ou Excel, os projetos Unity não correspondem a um único ficheiro, mas a uma pasta, que irá crescer bastante, em espaço ocupado, ao longo do desenvolvimento do jogo.

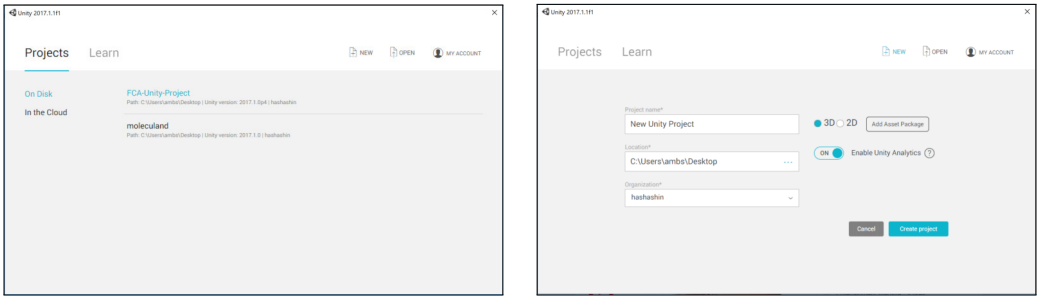


FIGURA 1.7 – Janela inicial do Unity e janela de criação de projeto

Também poderá escolher a criação de um projeto 3D ou 2D (note que esta é apenas uma opção inicial). Neste caso, o Unity irá criar uma cena adaptada ao tipo de jogo, colocando uma câmara ortogonal ou de perspetiva. O menu `Add Asset Package` permite a escolha de pacotes de funcionalidades para serem importados diretamente no projeto. Tal pode ser feito posteriormente, pelo que não será necessário saber, com antecedência, os pacotes que serão usados. Finalmente, a opção `Enable Unity Analytics` permite o uso de ferramentas do Unity para analisar o projeto em desenvolvimento. Depois de usar a opção `Create projet`, irá aparecer o editor do Unity, tal como apresentado na Figura 1.8.

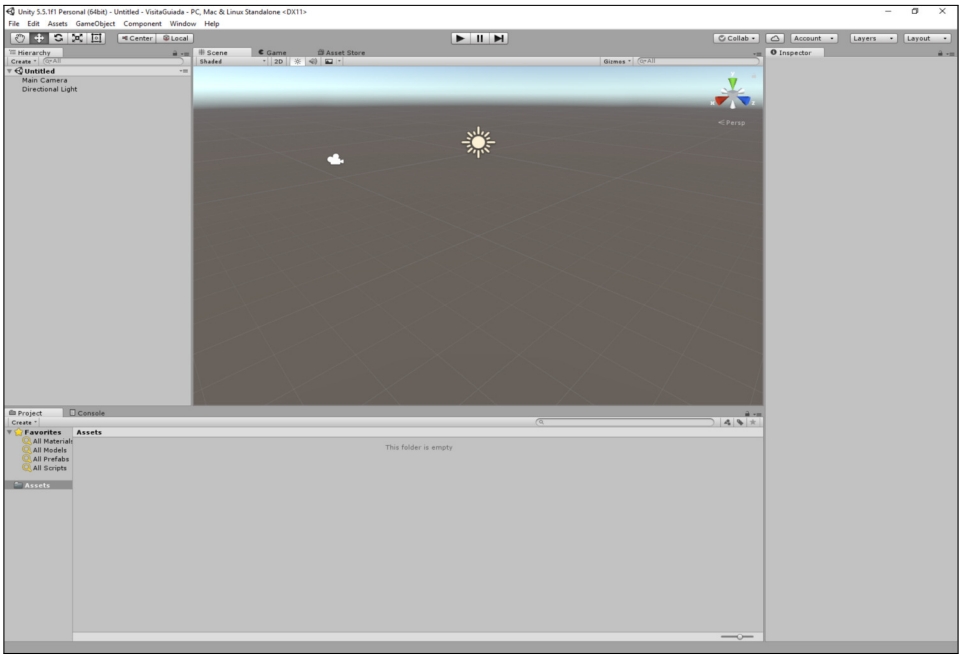


FIGURA 1.8 – Editor (*Integrated Development Environment*, IDE) do Unity

A forma como os vários separadores são apresentados poderá mudar entre versões do Unity. No entanto, existe um conjunto que se deverá manter estável, nomeadamente:

- *Game* (*Jogo*) – Apresenta a tela de jogo, tal como será apresentado ao jogador. É útil para ter noção dos objetos capturados pela câmara e testar o jogo.
- *Scene* (*Cena*) – Apresenta a cena do jogo, seja uma cena tridimensional ou bidimensional. É aqui que serão colocados e manipulados os objetos da cena.
- *Hierarchy* (*Hierarquia*) – Estrutura textual, hierárquica, dos objetos existentes na cena atual.
- *Project* (*Projeto*) – Pasta do projeto, com acesso aos vários ficheiros existentes (semelhante ao explorador típico dos sistemas operativos).
- *Inspector* (*Inspetor*) – Análise e alteração das propriedades dos componentes do objeto selecionado.
- *Console* (*Consola*) – Apresentação de erros, avisos e outras mensagens.

Há um conjunto de outros separadores que podem ser acionados, mas que serão introduzidos a seu tempo, sempre que seja necessário.

Além dos separadores, a interface do Unity inclui uma barra de ferramentas, sob o menu, onde se pode aceder a um conjunto de botões (alguns dos quais serão referidos na secção 1.3.2), e uma barra de estado, no fundo, onde é apresentada a última mensagem do separador *Consola*.

1.3.2 MANIPULAÇÃO DE OBJETOS

Para se perceber o funcionamento de cada um dos separadores do Unity adiciona-se um objeto na cena. Para isso, usa-se o menu `GameObject` → `3D Object` → `Cube`.

A Figura 1.9 mostra o separador *Cena* com a cena alterada. Note que o cubo aparece selecionado e um sistema de eixos é sobreposto ao objeto.

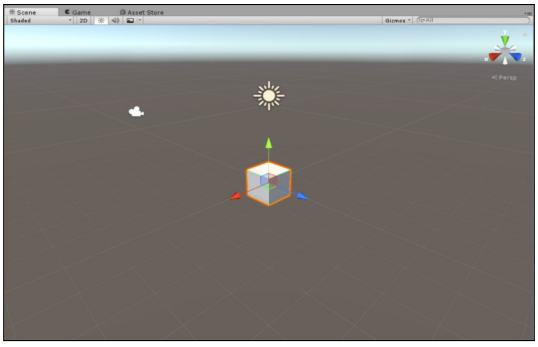


FIGURA 1.9 – Cena com o novo cubo

No canto superior deste separador é apresentado um pequeno diagrama que representa a direção de cada eixo (Figura 1.10). No Unity, o eixo y é vertical e o eixo z aponta, em profundidade, na direção oposta da posição inicial da câmara. Este diagrama é sempre útil para perceber a direção em que se movimentam os objetos.

FIGURA 1.10 – Eixos e vista no separador *Cena*

Além disso, sob os eixos, é apresentada informação sobre o tipo de câmara em uso, que varia entre uma câmara com perspetiva (por omissão) e uma câmara isométrica.


De realçar que cada objeto tem um sistema de eixos local que poderá não corresponder ao sistema de eixos global. Ao selecionar um objeto na *Cena*, será apresentado, sobre o objeto, o sistema de eixos local.

Por sua vez, o separador *Hierarquia* apresenta uma nova entrada correspondente ao cubo, como é visível na Figura 1.11, onde se vê, além da entrada correspondente ao cubo, uma entrada correspondente a uma câmara e uma outra correspondente a uma fonte de luz. Repare que ao visualizar a cena também já apareciam dois ícones correspondentes à fonte de luz (sol) e à câmara.

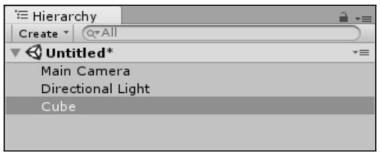


FIGURA 1.11 – Hierarquia apresentando o cubo criado

Finalmente, a Figura 1.12 apresenta o *Inspetor*[3], com os vários componentes que estão atribuídos ao cubo e que serão analisados na secção 1.3.3: *Transform*, *Cube* (ou *Mesh Filter*), o *Box Collider* e o *Mesh Renderer*. Além destes componentes, o *Inspetor* também mostra o material atualmente em uso no objeto selecionado.

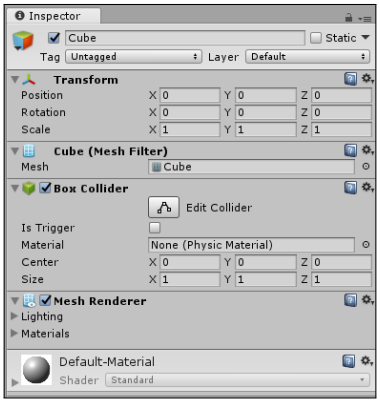


FIGURA 1.12 – *Inspetor* do cubo

É possível usar a *Hierarquia* ou a *Cena* para selecionar objetos e, posteriormente, sincronizar a seleção realizada num destes separadores com a seleção do separador complementar. Para isso, basta selecionar um objeto na *Hierarquia*, passar o rato por cima da

[3] Por uma questão de facilidade na leitura, sempre que o texto não for ambíguo, será apenas utilizado o nome do separador em itálico.

Cena e pressionar a tecla **f**. O objeto selecionado na *Hierarquia* será colocado em foco na *Cena*. O mesmo pode ser feito a partir da *Cena* para a *Hierarquia*.

Voltando à *Cena*, o cubo pode ser manipulado usando o rato, arrastando o sistema de eixos em qualquer das seis direções possíveis. Ao fazê-lo, o componente *Transform* irá atualizar as coordenadas da posição do objeto. Mas, além da posição, o editor permite outras operações, que podem ser controladas através dos cinco botões disponíveis na barra de ferramentas:

Estas cinco ferramentas podem ser acedidas usando as teclas **q**, **w**, **e**, **r** e **t**, respetivamente e só funcionam quando o separador *Cena* está ativo. A primeira permite a manipulação da vista sobre a cena, alterando a forma como ela é apresentada. É útil para navegar sobre a cena, acedendo aos objetos que se pretendem manipular. Note que, ao usar esta ferramenta com a tecla **Alt** pressionada, poderá rodar a cena, sendo que com a *scroll wheel* do rato poderá controlar o nível de *zoom*. A segunda ferramenta permite alterar a posição dos objetos selecionados, como já foi referido. A terceira é usada para rodar o objeto sobre um dos três eixos. Ao fazê-lo, a rotação apresentada no componente *Transform* será atualizada. As duas ferramentas finais permitem alterar a escala ou o tamanho dos objetos. A quarta altera o tamanho do objeto mantendo o seu centro. Assim, ao alargar o objeto numa direção, o lado oposto irá também aumentar. Por sua vez, a última ferramenta permite selecionar faces do objeto e manipular diretamente o seu tamanho, garantindo que todas as outras faces se mantêm inalteradas. Ao usar estas duas ferramentas, a informação de escala e de posição do componente *Transform* irá sendo atualizada.

Existem outros botões disponíveis na barra de ferramentas do Unity. Para testar o jogo executando o projeto, usa-se o primeiro botão (*Play*) do seguinte grupo de botões:

Deste grupo, o segundo botão permite parar (pausar) temporariamente o jogo e o terceiro permite iterar, um passo de cada vez (útil para depuração). Ao experimentar o botão *Play*, o Unity irá automaticamente colocar o separador *Jogo* visível e apresentará o cubo.

Repare que o ângulo a partir do qual o cubo é mostrado não é necessariamente o mesmo ângulo apresentado no separador *Cena*, o que significa que a câmara de jogo e a câmara do editor são independentes. Uma forma de validar essa distinção é selecionar a câmara na *Hierarquia* e ativar o separador *Cena*. Será apresentada a vista atual do editor, e num pequeno retângulo, a forma como a câmara apresenta a cena (Figura 1.13).

Por vezes, é mais fácil manusear a vista da *Cena* do que manipular a posição e ajustar a configuração de visibilidade da câmara de jogo. Nesse sentido, o Unity permite

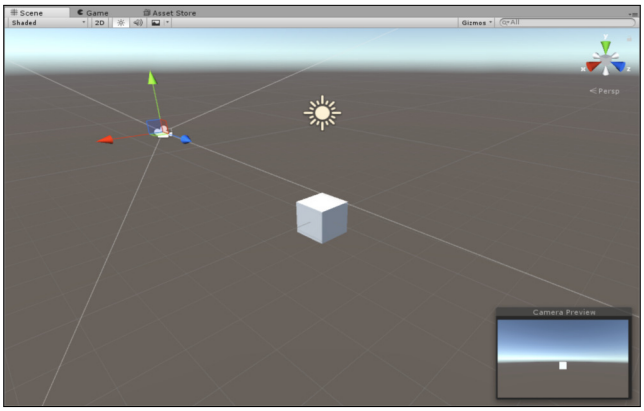


FIGURA 1.13 – Vista do editor *versus* vista da câmara

que o utilizador possa manipular a vista da *Cena*, ajustando o ângulo e nível de *zoom* que mais lhe agradem, e posteriormente, aplicar essa configuração à câmara de jogo. Para tal, basta selecionar a câmara na *Hierarquia* e usar a opção do menu `Game Object` → `Align with View`.

1.3.3 COMPONENTES

Anteriormente, foi referido que cada objeto tem um conjunto de componentes associados (Figura 1.12), que são apresentados no *Inspetor*. No topo, existem algumas propriedades do objeto, como o seu nome, se está ativo, a que camada (*layer*) pertence, se tem alguma etiqueta atribuída (*tag*), entre outras.

Surgem, depois, os componentes, que correspondem a comportamentos que se associam aos objetos. Existem muitos tipos de componentes e é também possível criar novos. Cada componente tem um conjunto de campos, ou propriedades, que podem ser alterados, de modo a configurarem o comportamento desejado.


É possível alterar os valores do *Inspetor* durante a execução do jogo, permitindo a validação das configurações de cada componente em tempo real. No entanto, assim que a execução terminar, os valores originais serão repostos.

Para melhor se perceber em que consistem os componentes, segue-se uma pequena descrição dos componentes associados ao cubo:

- *Transform* — Existe em todos os objetos que são colocados na cena de jogo. A sua função é representar a posição, rotação e escala dos objetos.

- *Cube (Mesh Filter)* — Para representar o cubo, é necessário um modelo tridimensional base. Um modelo tridimensional é designado por *mesh*, e este *Mesh Filter* consiste na representação específica do cubo.
- *Box Collider* — No Unity há um conjunto de componentes capazes de detetar colisões entre objetos, designados por *colliders*. Neste caso, trata-se de um *collider* com forma de cubo.
- *Mesh Renderer* — É responsável por transformar a *mesh* (armazenada no *Mesh Filter*) numa representação gráfica (fazer a renderização do modelo).

Novos componentes podem ser adicionados usando o botão `Add Component`, na parte inferior do *Inspetor*, ou o menu `Component`. Para transformar o cubo num corpo rígido (com características de um corpo rígido, como estar sujeito à gravidade), seleciona-se o objeto em causa, e usa-se o botão `Add Component`, adicionando-se o componente *Rigidbody* (Figura 1.14).

FIGURA 1.14 – Componente *Rigidbody*

Repare que a opção que permite que o objeto seja sujeito à gravidade está ativa e, portanto, se o projeto for executado, o cubo irá mover-se, caindo no vazio. Alguns componentes interagem entre si, o que faz com que, por vezes, ao adicionar-se um componente, sejam adicionados outros.

A Figura 1.15 mostra os componentes associados à fonte de luz e à câmara:

- *Camera* — Responsável pela configuração da câmara, guarda informação sobre a sua abertura, tipo de projeção, campo de visão, entre outras propriedades.
- *GUI Layer* — Responsável pela renderização de componentes de interação com o utilizador (*Graphical User Interface*, GUI) sobre a vista do jogo.
- *Flare Layer* — Faz a renderização de luzes e reflexos na câmara.
- *Audio Listener* — Recolhe sons que sejam emitidos na cena de jogo e reproduz para o utilizador.
- *Light* — Responsável pela iluminação omnipresente, permitindo a configuração de propriedades como a cor, intensidade e a direção, entre outras.

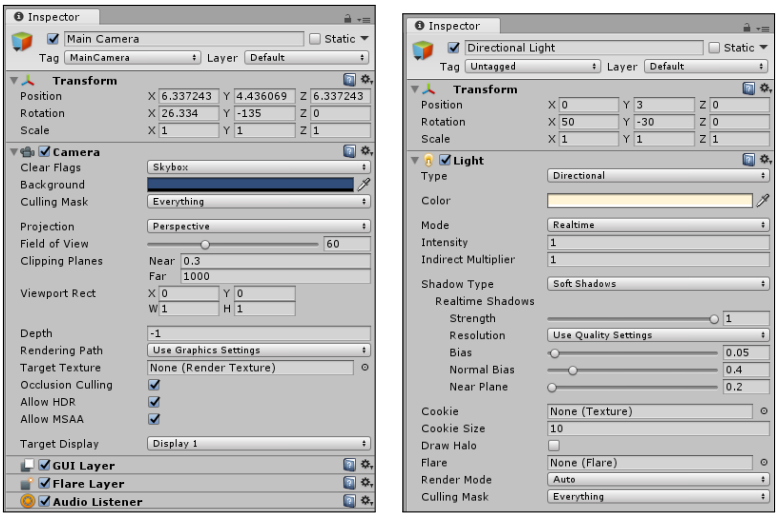


FIGURA 1.15 – Componentes da câmara e da fonte de luz

1.3.4 MATERIAIS

Ainda no *Inspetor* do cubo, apresentado na Figura 1.12, existem duas zonas colapsadas, referentes à iluminação (`Lighting`) e aos materiais (`Materials`). A primeira permite definir de que modo o objeto irá reagir às luzes e, a segunda, um conjunto de cores ou texturas a serem aplicadas ao objeto.

Um dos métodos para criar um material é, no separador *Projeto*, usar o botão `Create`, que apresenta a opção `Material`. Os materiais são guardados como ficheiros independentes, com a extensão `.mat`. A Figura 1.16 mostra o *Inspetor* de um material criado com o nome `Red`.

No topo do *Inspetor* aparece a possibilidade de alterar o *shader* que será usado para desenhar a textura. Um *shader* não é mais do que um algoritmo que indica como as cores e as texturas devem ser processadas antes de serem aplicadas aos objetos e posteriormente renderizadas.

Dependendo do *shader* selecionado, as propriedades do material podem variar. Neste caso, são apresentadas diversas propriedades desde a cor ou textura de albedo (correspondente à zona iluminada), propriedades metálicas, mapas de normais e de alturas, entre outras opções. Como exemplo, a cor de albedo pode ser alterada clicando na cor branca, no *Inspetor*, e escolhendo uma cor. Posteriormente, para atribuir o material acabado de criar ao objeto, pode-se, simplesmente, arrastá-lo desde o separador *Projeto* para o objeto em causa, apresentado na *Cena*. Depois de o fazer, o cubo deverá ter mudado de cor.

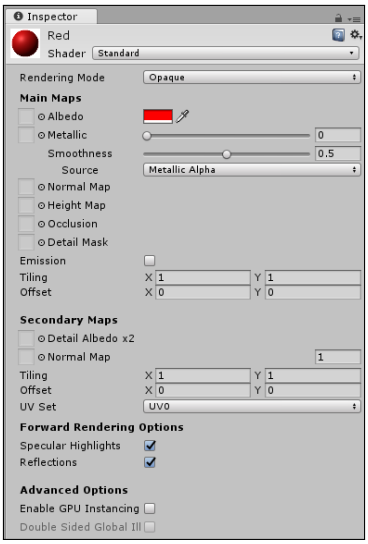


FIGURA 1.16 – *Inspetor* referente ao material `Red`


A associação dos materiais aos objetos pode ser vista como uma referência. Isto significa que, se no inspetor de um objeto que use determinado material se alterar a cor ou a textura, então, todos os objetos que usam esse material irão sofrer alterações. Assim, se houver necessidade de que vários objetos tenham controlo total sobre o seu material, independente de todos os outros, será preciso criar um material para cada um desses objetos.

1.3.5 PRÉ-FABRICADOS

Supondo a necessidade de criar 20 cubos semelhantes ao que existe na *Cena*, uma abordagem possível seria a seleção de um destes objetos seguida do uso das operações de *Copiar*/*Colar* ou de *Duplicate* (**Ctrl+d**) no menu `Edit`. Cada cópia seria colocada no sítio desejado, e o resultado o esperado. No entanto, se mais tarde fosse necessária a alteração do tamanho de cada um desses cubos, seria preciso fazê-lo um a um.

Uma alternativa é a criação de um objeto especial, não na *Cena* mas no *Projeto*, denominado pré-fabricado ou *prefab*. Este tipo de objetos é uma espécie de modelo, a partir do qual o Unity cria cópias, ou clones.

Tal como os materiais, os *prefabs* também são ficheiros independentes armazenados na pasta do projeto. Para a sua criação, pode ser usado o menu `create` do separador *Projeto*. A Figura 1.17 mostra, à esquerda, um *prefab* vazio (de nome `Cubo`).

Para associar um objeto a um *prefab* basta arrastá-lo da *Hierarquia* para o *prefab* vazio. A Figura 1.17 apresenta, à direita, o *prefab* depois de se lhe associar um objeto.

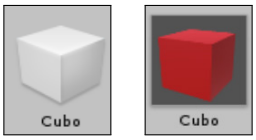


FIGURA 1.17 – *Prefab* vazio e *prefab* com um modelo associado

 Todos os *prefabs* vazios são representados por um cubo. Os *prefabs* preenchidos tanto podem ser representados por um cubo azul, como por uma miniatura do objeto, dependendo do nível de *zoom* definido no separador *Projeto*.

 Também é possível criar um *prefab* arrastando diretamente um objeto da *Hierarquia* para o separador *Projeto*, sem ser necessário criar o *prefab* vazio.

Depois de criar o *prefab*, o objeto que lhe deu origem deve ser removido da *Cena*. Isto é importante, pois o Unity não guarda qualquer referência entre este objeto e o *prefab* que foi criado. Depois de apagado o cubo da *Cena*, podem-se criar instâncias arrastando o *prefab* do separador *Projeto* para a *Cena* (Figura 1.18). Neste processo o Unity guarda informação de referência, o que permite que, ao alterar o *prefab*, as alterações sejam repercutidas nas instâncias.

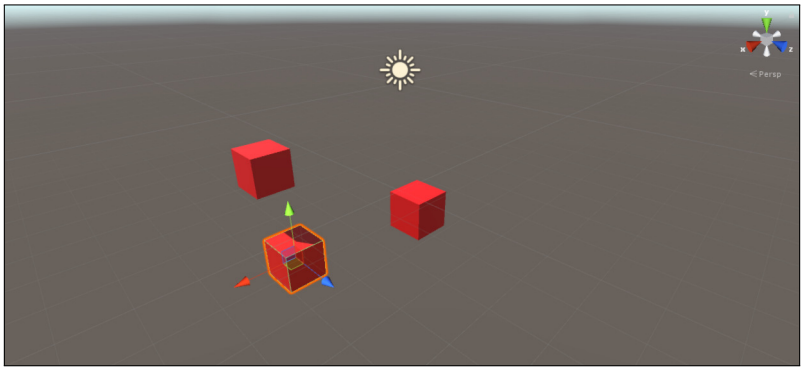


FIGURA 1.18 – *Cena* com três cubos criados a partir de um *prefab*

A Figura 1.19 mostra o inspetor para uma instância de um *prefab*. Nestes objetos, o inspetor mostra três novos botões que permitem, respetivamente, selecionar o modelo (e

fazer alterações diretamente sobre este), reverter o objeto atual (perdendo qualquer alteração que tenha sido feita em relação ao modelo) e aplicar ao modelo (copiando as alterações feitas na instância para o modelo, atualizando, portanto, todos os modelos).

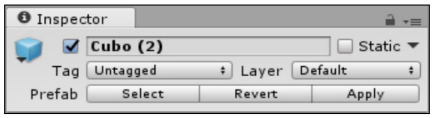


FIGURA 1.19 – *Inspetor* de uma instância de um *prefab*

1.3.6 *SCRIPTING*

A programação de jogos em Unity corresponde, na sua maioria, à criação de novos componentes. Estes são ficheiros de código (C#) armazenados na pasta de projeto.

De seguida, será criada uma *script* que irá fazer com que cada cubo rode sobre si mesmo. Como se pretende que todos os cubos se movam do mesmo modo, a *script* deve ser adicionada diretamente ao *prefab*.


Também é possível adicionar componentes numa das instâncias e, posteriormente, usar o botão `Apply` no topo do *Inspetor*.

Para criar a *script* usa-se o botão `Add Component` presente no *Inspetor* do *prefab*. Ao usar este botão e digitando o nome de um componente não existente, o Unity sugere a criação de uma *script* (Figura 1.20).

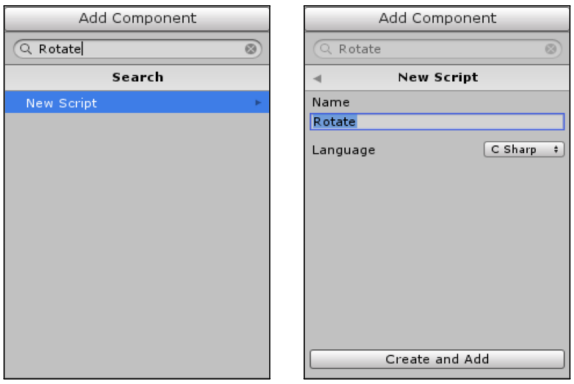


FIGURA 1.20 – Interface de criação de uma *script*

Depois de criada, a *script* aparece na pasta do projeto, aparecendo, também, como um componente no *Inspetor* (Figura 1.21).

FIGURA 1.21 – Visualização de uma *script* como um componente

A *script* pode ser editada fazendo duplo clique sobre o seu nome, seja no *Inspetor*, ou no *Projeto*. Dependendo do seu sistema operativo e das opções de instalação, poderá surgir o Visual Studio ou o MonoDevelop.


O MonoDevelop está disponível para qualquer arquitetura. No entanto, no Windows é usado o Visual Studio, já que contém funcionalidades mais avançadas. Em todo o caso, o MonoDevelop é mais leve, sendo por vezes escolhido por alguns programadores. Também pode configurar um qualquer editor externo, usando a opção `Preferences` no menu `Edit`.

A criação de uma *script* gera automaticamente um esqueleto de código, apresentado de seguida. Todas as *scripts* que correspondam a componentes têm, obrigatoriamente, de ser subclasses da classe `MonoBehaviour`, que define um conjunto de métodos relevantes:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Rotate : MonoBehaviour {
        // Use this for initialization
        void Start () {
        }

        // Update is called once per frame
        void Update () {
        }
}
```

A programação destes componentes pode ser vista como baseada em eventos. Por exemplo, no código apresentado, o método `Start` é invocado quando o componente inicia a sua execução e o método `Update` é invocado sempre que o Unity faz a renderização do ecrã (a cada *frame*).

Estas classes são especiais e não devem definir um construtor. Qualquer inicialização pode ser feita no método `Start`. Por sua vez, o método `Update` será invocado várias vezes por segundo, uma vez que é invocado sempre que o ecrã é atualizado. Supondo um rácio de desenho de 60 *frames* por segundo (habitualmente designado por *frame rate* ou *fps*),

este método será invocado pelo Unity esse mesmo número de vezes por segundo. Sendo que cada componente tem um método `Update`, a quantidade de código executada em cada *frame* é relativamente alta, pelo que estes métodos devem ser o mais eficientes possível.

Para exemplificar o tipo de programação do Unity, considere o seguinte método `Update`, que faz uma rotação de 15 graus à volta do eixo *y*, ao objeto a que o componente está associado:

```
void Update () {
   transform.Rotate (Vector3.up, 15);
}
```

Este componente pode ser ainda mais interessante se permitir que o programador defina a velocidade de rotação para cada cubo. Isso pode ser feito adicionando uma variável pública, com a quantidade de graus a rodar em cada *frame*, o que corresponde à velocidade de rotação que será observada nos cubos:

```
using UnityEngine;

public class Rotate : MonoBehaviour {
   public float RotationSpeed;

   void Update () {
      transform.Rotate (Vector3.up, RotationSpeed);
   }
}
```

Depois de gravada a alteração, voltando ao Unity, o *Inspetor* referente a esta *script* mudou, apresentando um campo para a configuração da velocidade de rotação, como demonstrado na Figura 1.22.

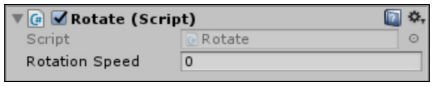


FIGURA 1.22 – Componente de uma *script* com um campo público

Ou seja, as variáveis de classe que são declaradas como públicas são acessíveis ao programador através do *Inspetor* do Unity, o que permite que se possam configurar comportamentos de forma simples, diretamente no editor.

1.3.7 CONTROLO DE VERSÕES

Antes de terminar a visita guiada, há um aspeto importante a referir para todos aqueles que usam sistemas de controlo de versões, como o Git ou o SubVersion. O Unity não é, de todo, "simpático" na sua interação com este tipo de ferramenta, mas existe um conjunto de opções que podem ser configuradas de forma a que esta convivência seja mais pacífica.

A pasta de um projeto Unity é composta pela pasta `Assets`, que corresponde à pasta de recursos apresentada pelo separador *Projeto*, pela pasta `Project Settings`, com ficheiros de configuração do projeto, e pela pasta `Library`, que inclui recursos pré-compilados pelo Unity. Destas pastas, a última (`Library`) não deve ser colocada no sistema de controlo de versões, uma vez que funciona como uma *cache*.

Embora as restantes pastas devam ser colocadas no repositório, é de realçar que o Unity usa, por omissão, ficheiros com metadados escondidos (que habitualmente o explorador do sistema operativo esconde) e, além disso, faz a serialização de recursos em ficheiros de cariz binário. Note que as ferramentas de controlo de versões não lidam bem com ficheiros escondidos, nem com documentos em formatos binários (que não permitem facilmente o cálculo de diferenças).

A Figura 1.23 mostra o painel de configuração do editor, que pode ser obtido através do menu `Edit` → `Project Settings` → `Editor`.

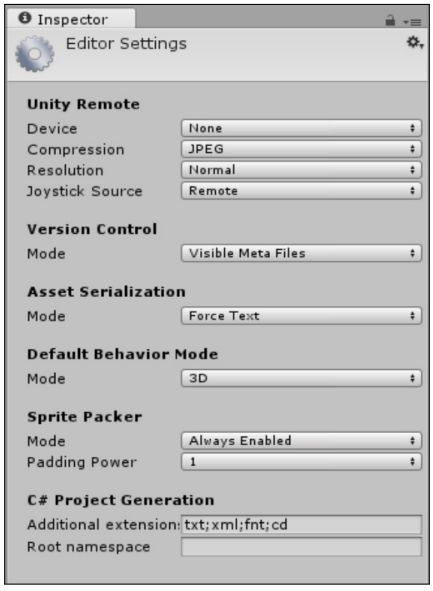


FIGURA 1.23 – Painel de configuração do editor

Neste painel, e para uma melhor integração com os sistemas de controlo de versões referidos, é importante que a opção `Version Control` apresente como valor `Visible Meta Files`. Além desta, a opção `Asset Serialization` deverá ter a opção `Force text` selecionada.

Grande parte dos sistemas de controlo de versões permite a definição de um conjunto de ficheiros e pastas a ignorar (por exemplo, `.gitignore` para o Git). Sugere-se, assim, que se ignorem (e não sejam adicionadas ao repositório) as pastas:

- `temp/`
- `obj/`
- `Library/`
- `.vs/`

2 Terrenos, Água e Céu

Embora o Unity não seja uma aplicação de modelação, como o Blender ou o Autocad 3ds Max[4], inclui uma ferramenta para a modelação de terrenos. A definição de terrenos diretamente dentro do Unity permite que este possa otimizar a forma como são renderizados. Neste capítulo será apresentado então o editor de terrenos do Unity e explicado o processo de adição de texturas e de outros modelos. Posteriormente, também serão incluídas zonas com água e um céu.

2.1 MODELAÇÃO DE UMA ILHA

Neste e nos próximos capítulos é desenvolvido um pequeno jogo, no qual a personagem terá de se deslocar numa ilha, com montanhas e árvores. A Figura 2.1 mostra uma cena possível da ilha que será modelada.

Figura 2.1 – Vista a partir da ilha

[4] O uso deste tipo de ferramentas é imprescindível para a preparação de modelos tridimensionais originais. Se o leitor tiver interesse na modelação, sugere-se a consulta dos títulos *Produção 3D com Blender de Personagens Bípedes* e *3ds MAX – Curso Completo* da FCA.

Ao longo deste capítulo, são descritas as ferramentas que o Unity disponibiliza para a modelação de terreno e indicados os passos necessários para preparar a ilha. O 1.º passo é a criação de um novo projeto.

2.2 CRIAÇÃO DE TERRENOS

Os terrenos modelados no Unity são baseados no conceito de *height map*, ou mapa de alturas, que consiste numa grelha de quadrados em que, a cada vértice, corresponde uma altura. Este tipo de técnica é bastante simples, mas tem algumas limitações, por exemplo, não permite escarpas (já que cada ponto só poderá ter uma altura) nem cavernas. O número de nodos associado ao terreno permite um maior detalhe, mas aumenta o custo de processamento. A Figura 2.2 representa este conceito.

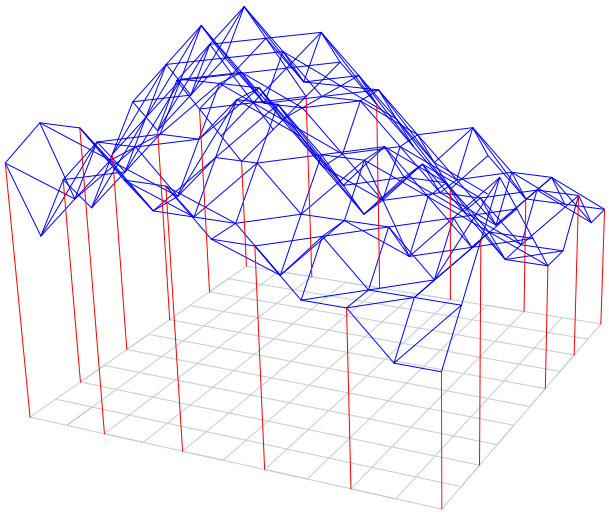


FIGURA 2.2 – Representação de um terreno baseado em *height map*

Os terrenos são criados usando o menu `Game Object` → `3D Object` → `Terrain`. À primeira vista, um terreno é pouco ou nada diferente de um plano. Analisando o *Inspetor* apresentado na Figura 2.3, verifica-se que os terrenos são compostos por dois componentes:

- *Terrain* — Corresponde a toda a informação do terreno, quer no que toca à sua estrutura tridimensional, quer no que respeita às texturas aplicadas, bem como a árvores e a outros objetos colocados no terreno.
- *Terrain Collider* — Com base na informação do componente *Terrain*, é capaz de calcular colisões com o terreno e com as árvores existentes.

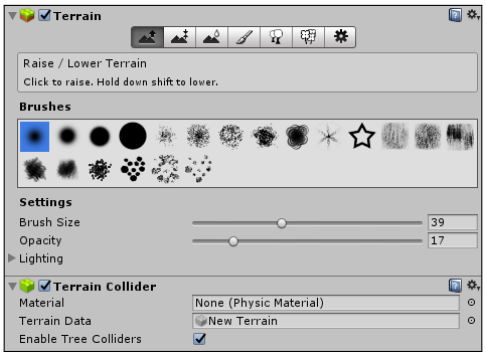


FIGURA 2.3 – *Inspetor* de um terreno recém-criado

O componente *Terrain* apresenta sete botões que permitem aceder a diversas funcionalidades. Os três primeiros são usados para esculpir o terreno, o quarto é usado para aplicar texturas, o quinto para plantar árvores, o sexto para plantar relva e outros objetos, e o último para aceder às suas variáveis de configuração. Ao fundo, existe uma zona referente a como o terreno se irá comportar em relação às fontes de luz (secção `Lighting` do *Inspetor*).

Embora seja possível iniciar logo o processo de esculpir o terreno desejado, se mais tarde se optar por alterar alguma configuração do terreno, é muito provável que se perca todo o trabalho feito. A análise das variáveis de configuração permitirá compreender um pouco melhor o funcionamento dos terrenos, pelo que inicialmente se descreverão os vários parâmetros de configuração (apresentados na Figura 2.4).

- Opções básicas referentes ao terreno:
  - → `Draw` — Indica se o terreno deve ser desenhado.
  - → `Pixel Error` — Margem de erro, em píxeis, entre o terreno gerado e a posição das texturas e objetos. Um valor maior resulta em menos qualidade, mas também em menor peso computacional.
  - → `Base Map Distance` — Distância a partir da qual as texturas poderão ser substituídas por outras com menos qualidade. Incrementar este valor resulta em maior peso computacional.
  - → `Cast Shadows` — Indica se o terreno deve produzir sombras.
  - → `Material` — Tipo de material usado na renderização do terreno.
  - → `Reflection Probes` — Tipo de suporte para o cálculo de reflexos nas texturas.

→ `Thickness` — Grossura do terreno (calculada no eixo das abcissas, na direção oposta do terreno). É especialmente útil para facilitar o cálculo de colisões com objetos que se movimentam a grande velocidade.

- Opções referentes a árvores:

  → `Draw` — Indica se as árvores devem ser desenhadas.

  → `Bake Light Probes For Trees` — Permite que sejam calculados reflexos para as árvores. No entanto, diminui a eficiência do terreno.

  → `Detail Distance` — Distância da câmara a partir da qual os objetos do terreno (que não são árvores) deixam de ser desenhados.

  → `Collect Detail Patches` — Controla se a informação referente aos objetos é mantida sempre em memória (opção desselecionada) ou se deve ser libertada para aqueles que não estejam visíveis (opção selecionada).

  → `Detail Sensity` — Informação sobre a densidade de objetos a serem renderizados a cada momento. Este valor corresponde à percentagem de objetos a serem apresentados.

  → `Tree Distance` — Distância da câmara a partir da qual as árvores deixam de ser desenhadas.

  → `Billboard Start` — Distância da câmara a partir da qual as árvores deixam de ser modelos tridimensionais e passam a ser representadas por texturas 2D pré-calculadas.

  → `Fade Length` — Intervalo de distâncias em que os modelos das árvores devem passar a texturas 2D.

  → `Max Mesh Trees` — Número máximo de árvores a serem desenhadas como modelos tridimensionais a cada instante.

- Opções relativas ao vento para a relva:

  → `Speed` — Velocidade do vento, que irá fazer a relva balancear.

  → `Size` — Tamanho da onda de vento representada na relva.

  → `Bending` — Curvatura, em graus, da relva, ao ser agitada pelo vento.

  → `Grass Tint` — Cor-base para a relva.

- Opções relativas à resolução do terreno:

  → `Resolution Width`, `Length` e `Height` — Largura e comprimento do terreno, bem como a sua altura máxima. Esta altura é a diferença entre o ponto mais alto e o ponto mais baixo do terreno. Quanto maior a altura, maiores diferenças no relevo são possíveis.

→ `Heightmap Resolution` — Número de interseções da grelha do *height map*. Este valor é obrigatoriamente uma potência de 2 acrescida de uma unidade: $resolution = 2^n + 1$. Note que a resolução é a mesma para a largura e para o comprimento do terreno, pelo que se uma destas dimensões for extremamente maior do que a outra, o relevo irá ficar com um aspeto estranho.

→ `Detail Resolution` — Resolução da matriz que armazena informação sobre os objetos colocados no terreno. Quanto maior for mais preciso será o posicionamento dos objetos, mas maior será o custo computacional.

→ `Detail Resolution Per Patch` — Número de objetos que podem ser colocados em cada uma das células da matriz definida pela configuração anterior.

→ `Control Texture Resolution` — Resolução do mapa de junção/sobreposição de texturas.

→ `Base Texture Resolution` — Resolução da textura apresentada quando o terreno se encontra a uma distância da câmara superior ao valor indicado em `Base Map Distance`.

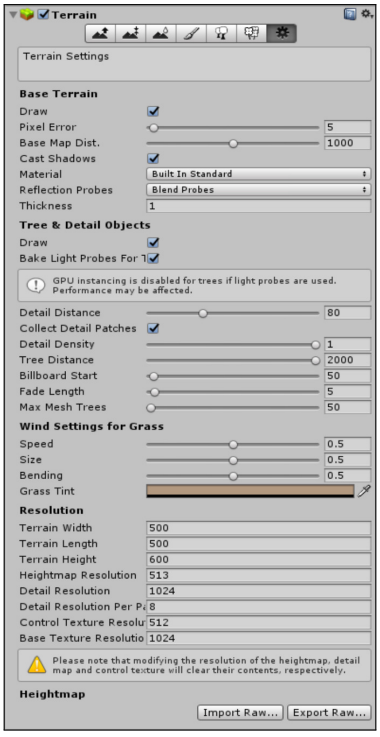


FIGURA 2.4 – Parâmetros de configuração de um terreno

Além das opções indicadas, existem dois botões, no fundo, que permitem exportar ou importar o relevo do mapa num ficheiro de imagem (*raw*). Nestas imagens, em tons de cinzento, quanto mais clara for a cor, mais elevado será o ponto.

A ilha do jogo não precisa de ser muito grande, pelo que as dimensões do *Height Map* devem ser alteradas para um quadrado de 200×200 unidades e uma altura máxima de 300 unidades. Em relação às outras opções de configuração, sugere-se que se mantenham no seu valor por omissão. A Figura 2.5 mostra o terreno criado.

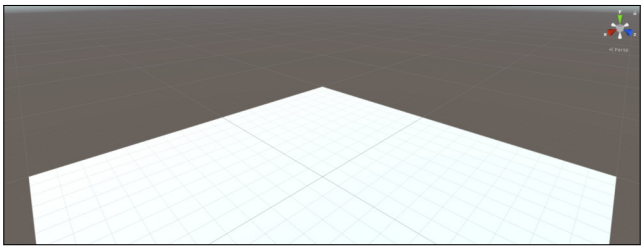


FIGURA 2.5 – Terreno com 200×200 unidades de área

2.2.1 MODELAÇÃO

A primeira tarefa, depois de configurar o tamanho desejado para o terreno, é iniciar o processo de o esculpir. Para o efeito são usadas as três primeiras ferramentas do componente *Terrain*.

A primeira ferramenta permite criar montes e vales (Figura 2.6). É possível escolher o padrão que será usado, o tamanho do padrão e a opacidade. Note que com um tamanho nulo ou uma opacidade nula, o terreno não irá sofrer qualquer alteração. A opacidade corresponde, basicamente, à força da modelação. Clicando no terreno, serão criados montes; clicando no terreno e na tecla **shift** em simultâneo serão criados vales. É importante realçar que a criação de montes e vales está limitada ao valor de altura definido nas propriedades do terreno.

A ferramenta seguinte permite criar planaltos (Figura 2.7). Para tal, é necessário definir a altura no campo `Height` ou clicando num declive juntamente com a tecla **shift**. Posteriormente, ao usar a ferramenta, o terreno irá subir (ou descer) até à altura definida.

Nesta ferramenta, existe ainda o botão `Flatten` que é usado para transformar todo o terreno num plano a determinada altura. Esta opção é especialmente útil quando se pretendem criar terrenos com montes e vales ou lagos. Assim, é possível alisar o plano e, posteriormente, fazer buracos. Caso contrário, e como o plano é criado à altura mínima do terreno, seria necessário içar todo o terreno para, depois, se poderem fazer os referidos buracos.

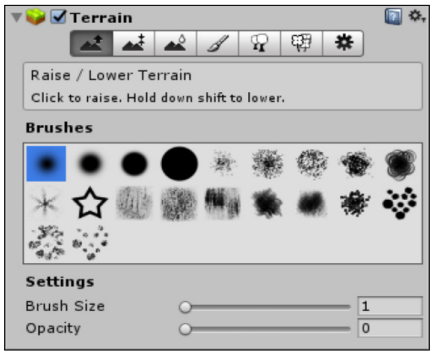


FIGURA 2.6 – Modelador de terrenos: levantar e baixar terreno

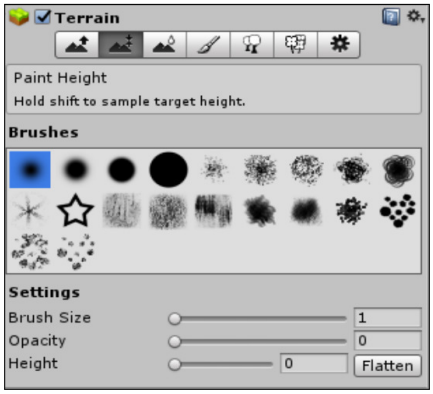


FIGURA 2.7 – Modelador de terrenos: criação de planaltos

Finalmente, a terceira ferramenta permite suavizar arestas do terreno (Figura 2.8). O seu uso é extremamente simples, estando limitado ao tamanho da área da ferramenta e da força com que esta irá suavizar o terreno.

Para a preparação da ilha, será necessário criar um monte que corresponda não só à parte acima do nível do mar, mas também um pouco à encosta submersa:

1) Começar por altear todo o terreno, para as 100 unidades, usando o `Flatten`. É natural que ao realizar esta operação o terreno deixe de ser visível na *Cena*. Isto deve-se ao facto de a sua posição ter sido alterada. Para que este volte a estar visível, poderá selecionar o terreno na *Hierarquia* e, passando o rato por cima da *Cena*, usar a tecla **f**.

2) Através da ferramenta de modelação, toda a borda do terreno deve ser transformada numa encosta decrescente (ver na Figura 2.22 o terreno parcialmente submerso com água).

3) Posteriormente, dar algum relevo à ilha e suavizar as arestas.

4) Antes de avançar, sugere-se que grave a cena atual, usando a opção `File` → `Save Scenes`, com o nome `ilha`. Repare que a cena aparecerá no separador *Projeto*.

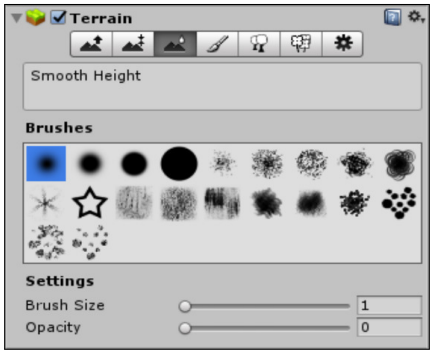


FIGURA 2.8 – Modelador de terrenos: suavizador

A Figura 2.9 mostra uma possível ilha obtida seguindo estes passos.

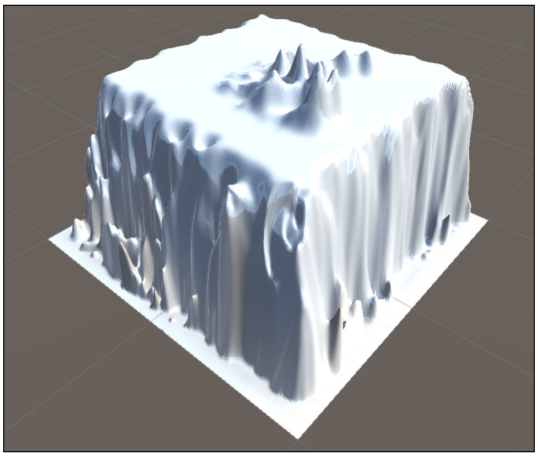


FIGURA 2.9 – Terreno modelado para simular uma ilha

2.2.2 TEXTURIZAÇÃO

A quarta ferramenta do componente *Terrain* permite aplicar texturas ao terreno. Para que isso seja possível, é necessário importar, previamente, texturas para o projeto

Unity. Uma forma rápida de o fazer, sem necessidade de se procurarem imagens na Internet, é importar um pacote com recursos vários para a criação de terrenos, que é disponibilizado em cada versão do Unity. O processo de importação de pacotes do Unity é bastante simples, bastando aceder ao menu `Assets` → `Import Package` → `Environment`, para importar o pacote *Environment*.

Ao importar um pacote, será apresentada uma janela com a lista de ficheiros pertencentes a esse pacote, tal como demonstrado na Figura 2.10. Cada ficheiro pode ser selecionado (ou desselecionado), indicando se deve ser, ou não, importado. Do lado direito, é apresentado um ícone que indica se o ficheiro é novo ou se já existe no projeto e, em caso de ser um ficheiro que já exista no projeto, se o seu conteúdo é semelhante, ou não. Aceitando a importação, será criada uma pasta designada por `Standard Assets` na pasta do projeto. Esta pasta é usada pelos recursos do Unity e não deve ser usada para qualquer outra finalidade.

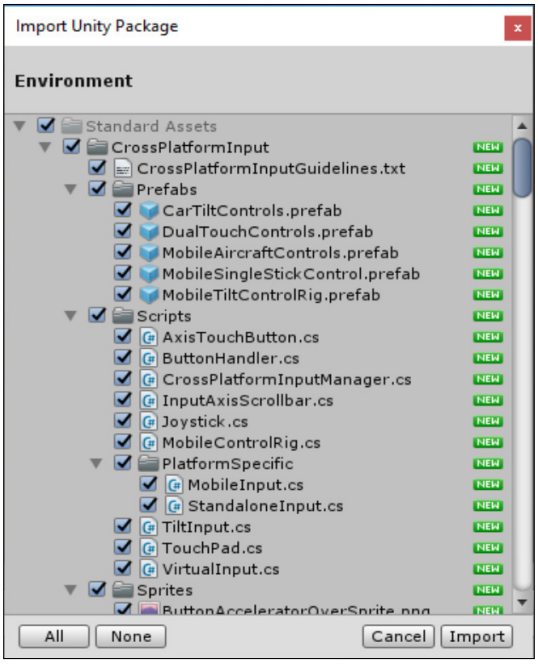


FIGURA 2.10 – Importador de pacotes Unity

Importado o pacote, é então possível associar texturas ao terreno. A Figura 2.11 mostra as opções disponíveis para a ferramenta. O 1.º passo consiste em associar texturas. A primeira textura associada é especial, porque vai ser aplicada automaticamente a todo o terreno. Assim, deve ser escolhida, em primeiro lugar, a textura que irá sobrepor-se à maior parte do terreno. Para adicionar texturas, usa-se o botão `Edit Textures` → `Add Texture`. Aparecerá uma janela semelhante à apresentada na Figura 2.12.

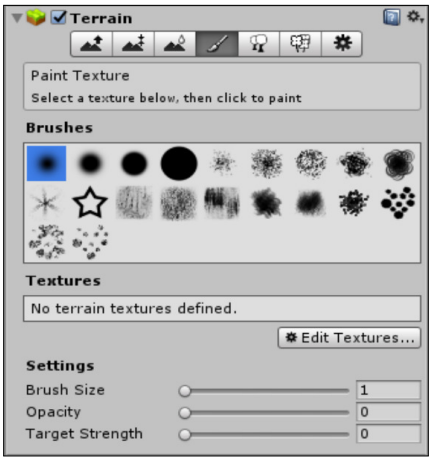


FIGURA 2.11 – Aplicador de texturas

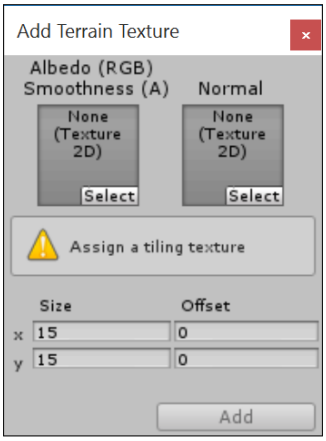


FIGURA 2.12 – Selecionador de texturas

Esta janela permite selecionar uma textura (`Albedo`), obrigatória, que corresponde à textura a ser aplicada no terreno. Por sua vez, a textura `Normal` corresponde a uma imagem, com o mesmo tamanho do que a `Albedo`, mas em que a informação de cores é usada para associar relevo à textura. Abaixo, é possível indicar o número de vezes que a textura será repetida em cada eixo (coluna `Size`) e indicar se a imagem deve ser deslocada antes de ser aplicada (coluna `Offset`).


No Unity as texturas devem ser quadradas e cujos lados tenham como dimensão uma potência de 2 (ou seja, $largura = 2^n$).

A Figura 2.13 mostra a ferramenta de aplicação de texturas depois de adicionadas três texturas. A primeira textura, representativa de areia, será aplicada a todo o terreno automaticamente. As outras duas têm de ser aplicadas manualmente, selecionando o tipo de cursor, o seu tamanho (`Brush Size`) e opacidade (`Opacity`). Tal como em ferramentas de manipulação de imagem, ao passar o cursor várias vezes pelo mesmo local, a textura é aplicada várias vezes, aumentando, assim, a sua opacidade. Para facilitar a aplicação de texturas com pouca opacidade, é possível definir a opacidade máxima que será aplicada (`Target Strength`).

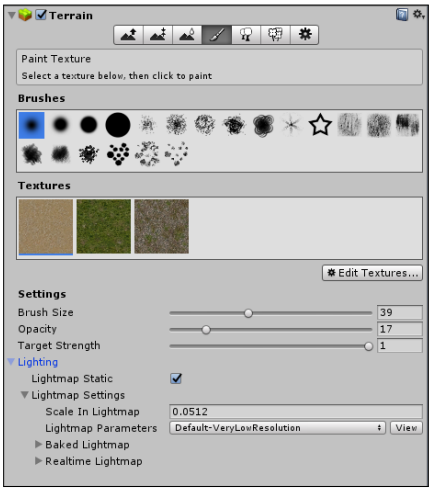


FIGURA 2.13 – Aplicador de texturas depois de adicionadas algumas texturas

Para a ilha, poderá ser adicionada uma textura de areia (`SandAlbedo`), que cubra todo o terreno, a textura de relva (`GrassHillAlbedo`) para as zonas planas da ilha e uma textura rochosa (`GrassRockyAlbedo`) para as zonas mais montanhosas. A Figura 2.14 apresenta a ilha depois de aplicadas essas texturas.

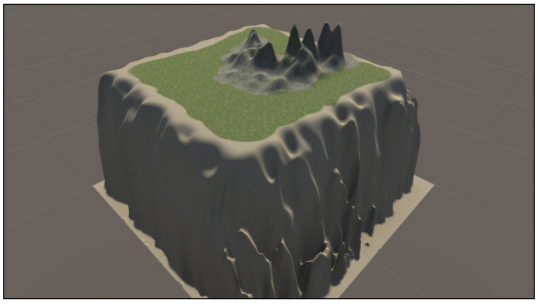


FIGURA 2.14 – Terreno modelado e texturizado

2.2.3 PLANTAÇÃO DE ÁRVORES

O processo de plantação de árvores é semelhante ao processo de aplicação de texturas. É necessário começar por indicar os modelos de árvores que vão ser utilizados. A Figura 2.15 apresenta a janela de aplicação de árvores antes de se terem definido os modelos a usar.

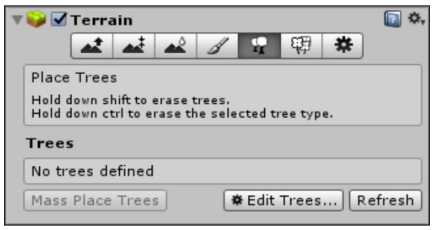


FIGURA 2.15 – Plantador de árvores

O processo de adição de um modelo de árvore é muito semelhante ao processo de adição de uma textura. Depois de escolhida a ferramenta correspondente, usa-se o botão `Edit Trees` → `Add Tree`. A janela da Figura 2.16 é apresentada e, clicando no pequeno círculo do lado direito da opção `Tree Prefab`, surgirá um seletor de objetos, onde deverá ser escolhido o modelo da árvore a usar. Nesta mesma janela, a opção `Bend Factor` permite indicar um nível de curvatura máximo para a árvore.

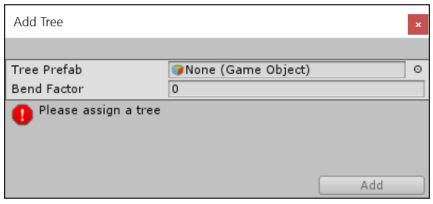


FIGURA 2.16 – Janela de adição de árvore

Adicionada a árvore, será possível aplicar árvores no terreno. Usando a tecla **shift**, a ferramenta removerá as árvores; usando a tecla **ctrl**, removerá as árvores do tipo selecionado. A janela da ferramenta de aplicação de árvores apresenta algumas opções extra (Figura 2.17).

O `Brush Size` corresponde ao tamanho da zona de seleção e a `Tree Density` à quantidade de árvores que será plantada. Por sua vez, as opções `Tree Height` e `Tree Width` permitem definir um intervalo de valores que limitem a altura e a grossura das árvores. No entanto, é possível remover a opção `Random` e, nesse caso, as árvores serão plantadas todas com o mesmo tamanho. Como habitualmente as árvores mais altas também são mais grossas, é possível ativar a opção `Lock Width to Height`, de modo a que o Unity calcule

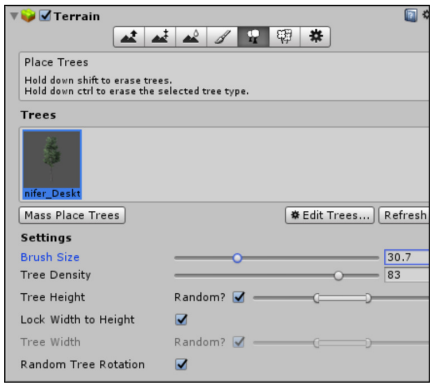


FIGURA 2.17 – Plantador de árvores com uma árvore disponível

automaticamente uma proporção da grossura de cada árvore. Mais abaixo, a opção `Random Tree Rotation` indica se cada árvore deverá ter uma rotação aleatória. Existe ainda o botão `Mass Place Trees`, que permite plantar um número predefinido de árvores em todo o terreno, de forma completamente automática.

No que toca à plantação de árvores na ilha do jogo, sugere-se o uso de poucas árvores, dando preferência às zonas planas, tal como demonstrado na Figura 2.18.

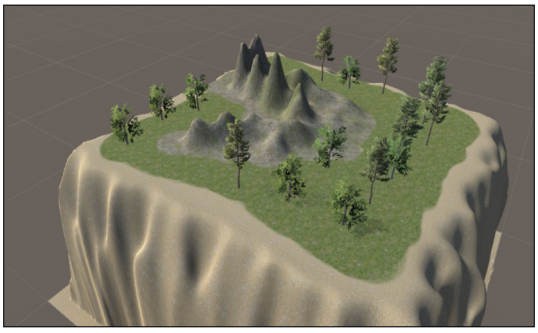


FIGURA 2.18 – Terreno modelado, texturizado e com árvores

2.2.4 RELVA E OUTROS DETALHES

Do mesmo modo que se podem plantar árvores, também é possível colocar zonas com relva ou com outros detalhes, como pedras. A relva é baseada em texturas bidimensionais que são animadas, simulando o vento. Já os restantes detalhes consistem em quaisquer objetos tridimensionais que tenham sido importados para o projeto.

A Figura 2.19 apresenta a ferramenta para aplicar relva. Tal como no caso das texturas e das árvores, é necessário, em primeiro lugar, importar os objetos em causa. Repare que existem dois botões diferentes, um para adicionar relva e outro para adicionar objetos tridimensionais. A Figura 2.20 apresenta as interfaces para adição de relva e de outros objetos, respetivamente.

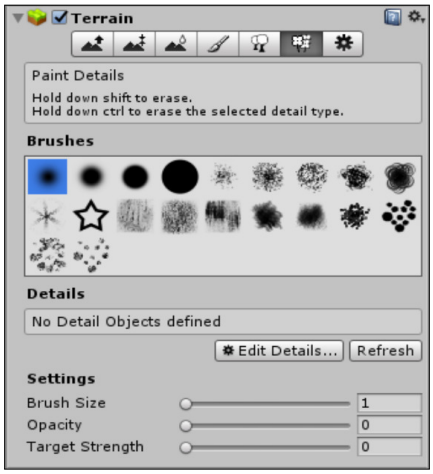


FIGURA 2.19 – Aplicador de relva e outros detalhes

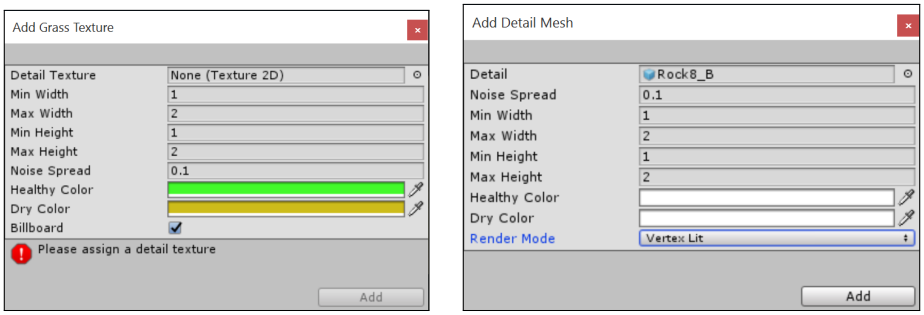


FIGURA 2.20 – Interface para adição de relva e de outros objetos

As opções disponíveis em cada uma das janelas são bastante similares. A primeira permite escolher a textura correspondente à relva ou o modelo tridimensional do objeto a adicionar. Seguem-se as dimensões (largura e altura) mínimas e máximas de cada pedaço de relva (ou de qualquer outro objeto). A opção `Noise Spread` corresponde a um valor de aleatoriedade para a geração de ruído. Podem-se configurar duas cores que correspondem à cor da relva saudável e da relva seca. Por incrível que pareça, para outros objetos também se definem duas cores que, embora possam não ser as cores de pedras saudáveis ou secas, correspondem às que serão usadas aleatoriamente para colorir os objetos adicionados.

Finalmente, do lado da relva existe uma opção para usar, ou não, a técnica de `Billboard`, situação em que as texturas são rodadas de maneira a que estejam diretamente viradas para a câmara. No entanto, por vezes, esta técnica produz efeitos estranhos, pelo que pode ser desligada. Já do lado dos objetos tridimensionais, existe o `Render Mode` que indica o modo como os objetos devem ser renderizados: diretamente como objetos tridimensionais (opção `Vertex Lit`), ou como relva (o objeto tridimensional é convertido para uma textura 2D).


Ao contrário das árvores, a relva e quaisquer outros detalhes adicionados ao terreno não geram colisões (pelo que a personagem é capaz de os atravessar). Quando as colisões são importantes deverá ser usado um objeto tradicional, independente do terreno.

Para a ilha, poderá usar-se alguma relva, escolhendo a textura `GrassFrond01-AlbedoAlpha` e tendo o cuidado de não exagerar na quantidade, já que o seu uso excessivo pode, facilmente, comprometer a eficiência do jogo desenvolvido (Figura 2.21). Uma vez que não existe outro tipo de detalhe no pacote *Environment*, sugere-se que não se apliquem outros objetos além da relva.

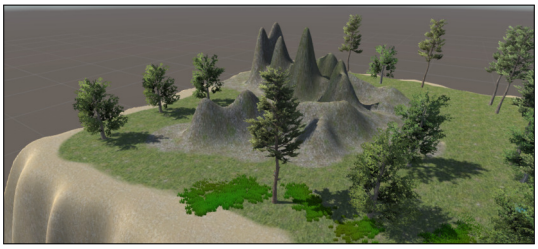


FIGURA 2.21 – Terreno modelado, texturizado e com árvores e relva

2.3 ÁGUA

A modelação de água realista é uma tarefa complexa e os detalhes da sua implementação vão além dos objetivos deste livro, pelo que esta secção descreve, apenas, o uso de um *prefab* que é parte integrante de um dos pacotes disponibilizados pelo Unity. No entanto, o resultado é bastante aceitável e a sua utilização bastante simples.

O Unity disponibiliza vários *prefabs* para a modelação de água, todos eles incluídos no pacote *Environment*, já referido. Todos os modelos funcionam como um plano (alguns quadrangulares, outros circulares) que deve ser colocado ao nível do mar desejado. Dentro da pasta `Standard Assets/Environment` encontra-se:

- `/Water (Basic)/Prefabs` — Inclui dois *prefabs* básicos que simulam água: um para cenas diurnas; outro para noturnas. Trata-se de planos circulares que simulam a água de forma simples, são relativamente eficientes e pouco realistas.
- `/Water/Water/Prefabs` — Inclui versões avançadas dos dois *prefabs* referidos no item anterior.
- `/Water/Water4/Prefabs` — Inclui dois *prefabs* para planos retangulares: um mais simples, com menos propriedades de configuração mas mais eficiente; outro mais avançado e capaz de resultados mais realistas. A Figura 2.22 apresenta um exemplo de uma ilha atravessada por um plano de água, usando um destes *prefabs*.

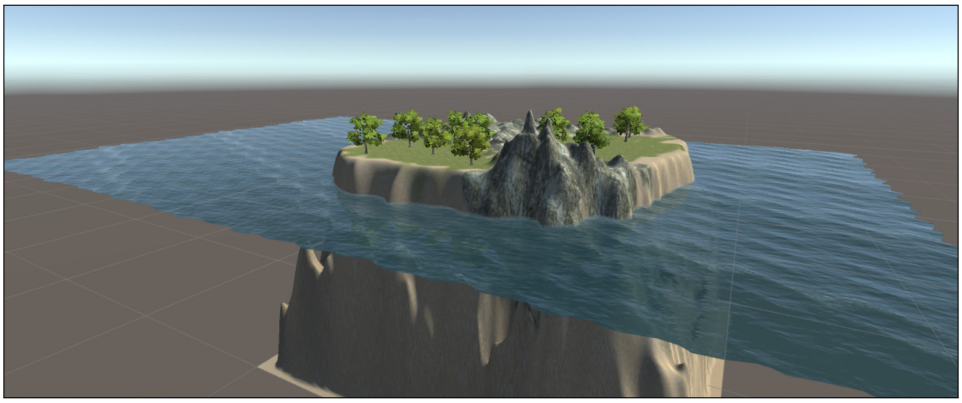


FIGURA 2.22 – Exemplo de uma ilha atravessada por um plano de água


Note que as pastas que se encontram ao mesmo nível da `Prefabs` incluem os materiais, modelos, texturas e *scripts* necessários para que os *prefabs* funcionem, pelo que não devem ser removidas.

Foge ao âmbito deste livro a explicação da forma como cada uma das propriedades destes *prefabs* alteram o seu comportamento. No entanto, com algumas experiências, poder-se-á perceber de que modo algumas delas funcionam.

Para a criação da ilha, poderá ser adicionado o *prefab* de água retangular (`Water4`). Ao adicionar este *prefab* irão surgir dois objetos na *Hierarquia*: o objeto responsável por desenhar a água e um objeto responsável pelo cálculo dos reflexos na água. Deverá ter-se o cuidado de, no primeiro objeto, alargar a área do mar, de modo a que o jogador não consiga notar o limite da água no horizonte, como demonstrado na Figura 2.23.

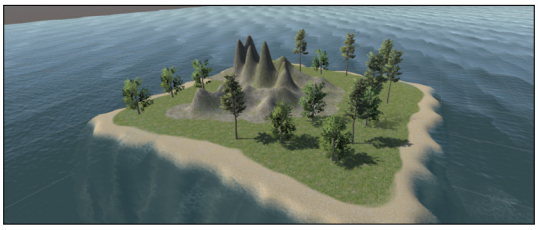


FIGURA 2.23 – Ilha no meio do mar

2.4 CÉU

Além da modelação do terreno e da adição de lagos ou mares, é importante algum cuidado com o céu e o horizonte. O céu e o horizonte são definidos por uma *skybox* que corresponde a um conjunto de texturas responsáveis por simular o que a personagem vê quando olha para o céu, para o chão, ou para cada uma das quatro direções à sua volta. Ou seja, supondo que a personagem se encontra dentro de um cubo, será necessária uma imagem para preencher cada uma das suas seis faces internas. Estas seis imagens podem ser independentes, ou em formato *cubemap*, como se pode ver na Figura 2.24.

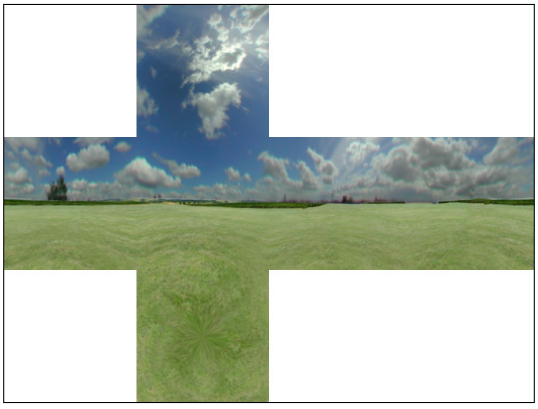


FIGURA 2.24 – *Skybox* em formato *cubemap*

 É importante ter em atenção que é necessário que cada aresta de cada imagem seja consistente com a aresta correspondente da face adjacente.

Sempre que se cria uma câmara com o Unity é usada uma *skybox* muito simples, apenas com um céu azulado e um horizonte acastanhado. Embora a criação de uma *skybox* não seja propriamente simples, o uso de uma que seja boa pode mudar completamente

o aspeto de um jogo. Existem também muitas *skyboxes* disponíveis na Internet, embora grande parte com direitos de autor associados.


Nos recursos disponibilizados, o leitor poderá encontrar a *skybox* apresentada na Figura 2.24, e que será aplicada no jogo.

Para criar e usar uma *skybox* terão de ser executados os passos seguintes:

1) Importar a imagem para o Unity é trivial, bastando arrastá-la para uma das pastas no separador *Projeto*.

2) Como o Unity usa imagens para diferentes finalidades, como texturas, botões, *sprites*, *skyboxes*, entre outras, é necessário indicar qual a finalidade da imagem importada. Para tal seleciona-se a imagem, no separador *Projeto* e, no *Inspetor*, altera-se a `Texture Type` para `Default` e a `Texture Shape` para `Cube`. Para finalizar as alterações, é necessário aplicá-las, usando o botão `Apply` (lado esquerdo da Figura 2.25).

Depois desta alteração, o ícone da imagem no separador *Projeto* mudará e ficará com um aspeto esférico.

3) Posteriormente, é necessário criar um *Material*, que indique de que forma a imagem deverá ser apresentada. Para o efeito, deve clicar-se na zona do separador *Projeto* com o botão direito do rato e escolher a opção `Create` → `Material` e atribuir-lhe um nome.

No *Inspetor* do material será necessário mudar a opção `Shader` para `Skybox` → `Cubemap`. A opção `6 Sided` permite associar seis imagens separadas ao cubo. A opção `Procedural` permite a criação de *skyboxes* de forma procedimental.

Alteradas estas opções, o próximo passo consiste em arrastar a imagem preparada no ponto anterior para o seletor `Cubemap (HDR)`.

4) Finalmente, é necessário associar o material criado à câmara. Para isso, seleciona-se a câmara na *Hierarquia*, e adiciona-se o componente *Skybox*,[5] que deverá ser preenchido com o material criado no ponto anterior.

Ao terminar este processo poderá, tal como foi apresentado no Capítulo 1, alterar a posição e ângulo usados na *Cena*, para uma posição em que se veja devidamente a ilha, e sincronizar essa perspetiva com a posição da câmara (selecionar a câmara na *Hierarquia* e usar a opção `GameObject` → `Align With View`). Posteriormente, use o botão `Play` para visualizar o resultado do trabalho realizado.

[5] Durante o avançar do livro, alguns processos básicos irão deixar de ser explicados passo a passo, com o intuito de que o leitor se vá familiarizando com a terminologia e com a interface.

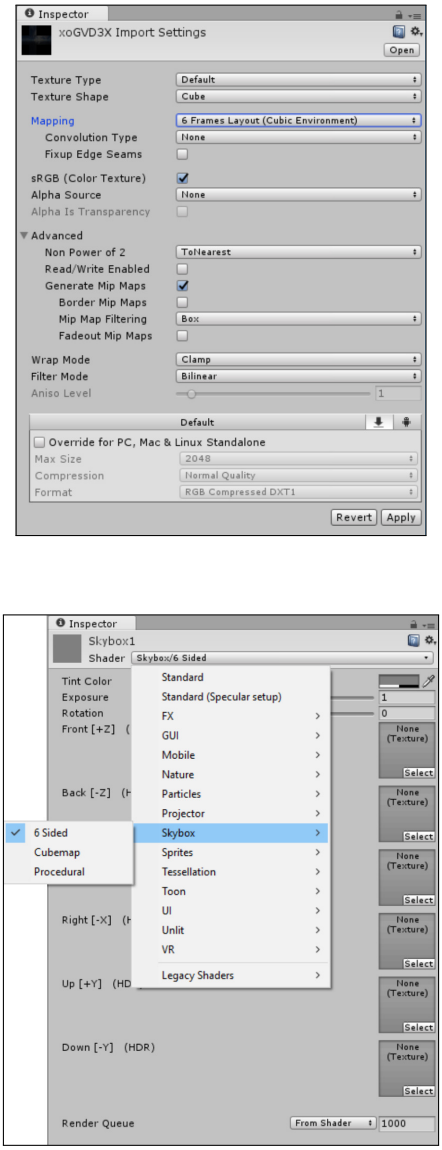


FIGURA 2.25 – *Inspetor* de importação da imagem, e de material, para a criação de uma *skybox*

3 MOVIMENTO E ANIMAÇÕES

Neste capítulo é introduzida a personagem principal do jogo. Em primeiro lugar é apresentada uma forma simples de animar personagens tridimensionais usando um serviço gratuito na Internet. Posteriormente, essas animações são importadas para o Unity e mapeadas às diferentes ações que o jogador poderá realizar. Finalmente, adiciona-se movimento à personagem, de acordo com o controlo do jogador.

3.1 ANIMAÇÃO DE PERSONAGENS

O Unity permite que algumas animações sejam realizadas diretamente no seu editor. No entanto, este editor é ideal para pequenas animações e não para animações complexas, como o movimento de um humanoide. Este tipo de animação é habitualmente realizado em aplicações de modelação e animação, que fogem ao âmbito deste livro.

Para garantir alguma completude, será usado um serviço da Mixamo[6] que permite a criação de um esqueleto para uma personagem e o mapeamento de animações predefinidas para esse esqueleto.


A personagem principal do jogo é uma formiga, modelada em Blender por Frederico Gonçalves, e está disponível no sítio do livro em `www.fca.pt`, com o nome `ant.fbx`.

O 1.º passo corresponde à criação de um utilizador Adobe, que será necessário para se autenticar no sítio da Mixamo (ligação `Sign-up`). Depois de o ter criado e de se ter autenticado (ligação `Sign-in`), aparecerá a interface de seleção de animações para uma personagem, tal como é apresentado na Figura 3.1.

É possível, usando as ligações no topo, alternar entre uma lista de personagens (`characters`) e de animações (`animations`). Escolhidas a personagem e a animação dese-

[6] Disponível gratuitamente em `http://mixamo.com`. Note-se que o serviço gratuito da Mixamo é apenas para humanoides. Durante a escrita deste livro o tipo de serviço prestado mudou, embora a funcionalidade-base tenha sido mantida. Assim sendo, é natural que no futuro surjam novas alterações quer às funcionalidades, quer às suas políticas de uso.

jadas, é possível configurar o movimento alterando parâmetros como a velocidade, amplitude de braços, entre outros, e descarregar o modelo para ser importado para o Unity.

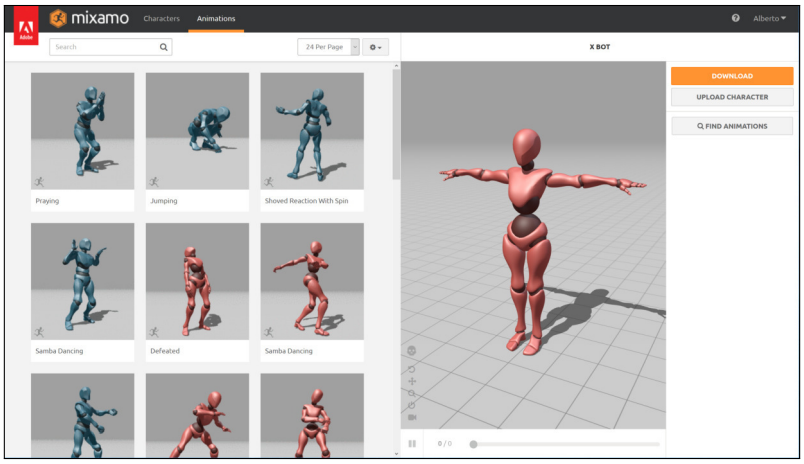


FIGURA 3.1 – Interface inicial do serviço da Mixamo

No caso concreto do jogo em desenvolvimento, pretende-se usar um modelo concreto que não está disponível, e portanto, é necessário carregá-lo para o sistema da Mixamo. É possível importar modelos que já incluam um esqueleto ou tirar partido de um sistema automático de definição do esqueleto. Note que, para poder fazer uso de um esqueleto já existente, este tem de ser semelhante ao usado pela Mixamo.

Se o esqueleto existente for compatível, então o utilizador irá ser direcionado de novo para a interface de animações, já com o modelo desejado. Se o esqueleto existe e for possível mapear os seus ossos aos ossos do esqueleto da Mixamo, então o utilizador terá de passar por um processo delicado e algo moroso de associar ossos do esqueleto utilizado no modelo ao reconhecido pelo serviço.

Para o caso do modelo da formiga, que não inclui um esqueleto, então será processado por um algoritmo que irá, se possível, associar um esqueleto ao modelo.

A Figura 3.2 apresenta o 1.º passo, que corresponde à definição de rotação-base para a personagem, que deve ser colocada de frente.

De seguida, será necessário indicar a posição do queixo (`chin`), pulsos (`wrists`), cotovelos (`elbows`), joelhos (`knees`) e virilha (`groin`), tal como demonstrado na Figura 3.3. Nesta etapa é possível definir qual o tipo de esqueleto desejado, com mais ou menos segmentos. Habitualmente a opção `Standard Skeleton` permite bons resultados.

Estes dois passos são suficientes para que seja construído um esqueleto para a personagem. No passo seguinte, será necessário confirmar que o esqueleto está correto e, no último passo, será avisado de que o uso deste novo modelo irá substituir o último modelo

em edição. Daqui será redirecionado para a interface inicial, para configuração das animações, mas desta vez já com o modelo da formiga (Figura 3.4).

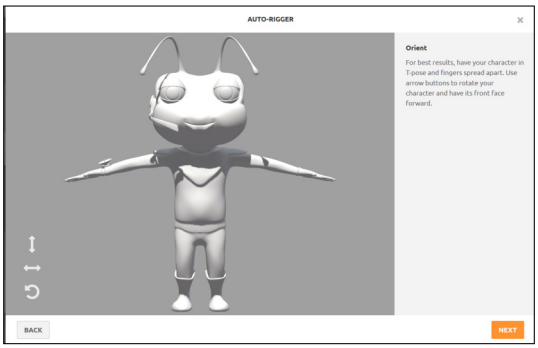


FIGURA 3.2 – Definição da rotação-base da personagem

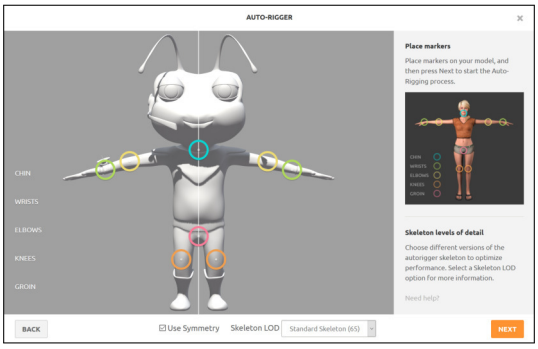


FIGURA 3.3 – Definição das articulações-base da personagem

De seguida, é possível aplicar animações, escolhidas a partir de um grande conjunto de animações disponíveis. Grande parte destas tem duas variantes, uma para o género masculino (avatar azul) e outra para o género feminino (avatar rosa). Escolhido o movimento desejado, o modelo irá simular essa animação e será possível definir um conjunto de parâmetros, como o espaçamento entre braços, a velocidade, o número de *frames*, entre outros. Além disso, existe uma opção para inverter uma animação (`Mirror`) e outra para que a animação seja realizada sem mudança de posição (`In Place`).

Embora seja possível o uso de animações `In Place`, e a princípio estas possam parecer mais simples de usar, será apresentado o uso de animações em que há movimento da personagem, já que resolve um maior número de situações. Assim, as animações usadas no projeto terão todas esta opção desligada. A Figura 3.5 apresenta a interface Mixamo com a personagem no movimento de caminhada, bem como os parâmetros existentes para a respetiva animação.

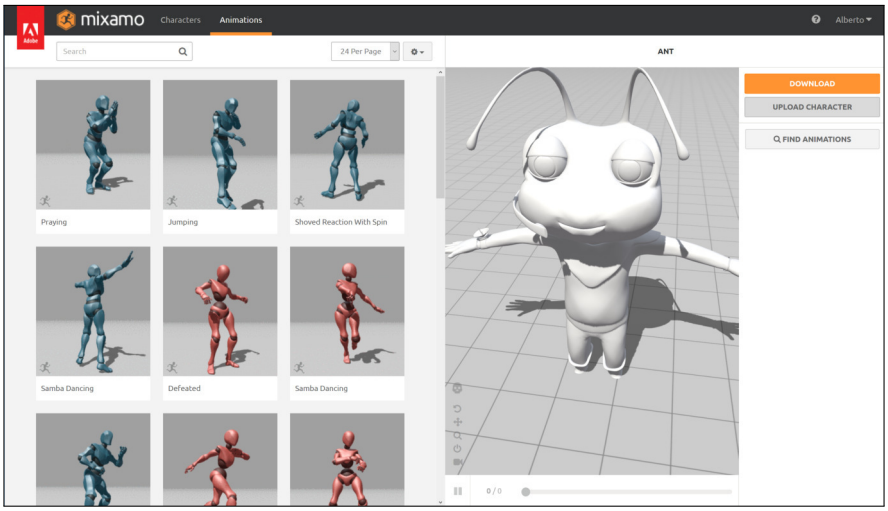


FIGURA 3.4 – Interface para escolha e configuração de animações

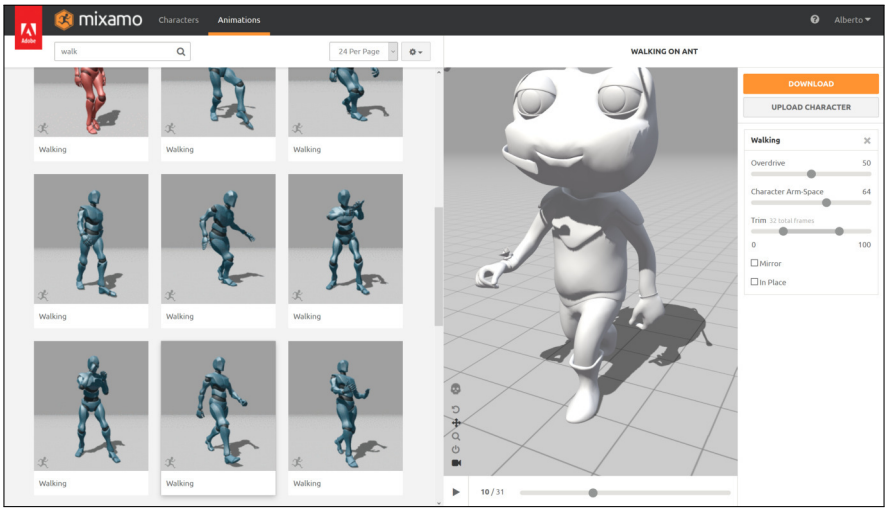


FIGURA 3.5 – Configuração de parâmetros da animação de caminhada

Para o jogo em desenvolvimento, serão necessárias as seguintes animações: andar (`Walking`), correr (`Running`), parado (`Neutral Idle`), recolha de um objeto (`Taking Item`) e rodar para a direita[7] (`Right Turn`). Deverá escolher cada animação, configurar os parâ-

[7] Dada a grande variedade de animações de rotação, optamos por escolher a animação para um dos lados e implementar a rotação no outro sentido com simetria no próprio Unity. Além disso, é importante escolher uma animação em que o torso da personagem não se movimente, caso contrário, o processo de animação do Unity poderá tornar a animação estranha.

metros respetivos para um movimento natural e usar o botão `Download`. Surgirá a janela apresentada na Figura 3.6, que permite escolher o tipo de modelo a descarregar.

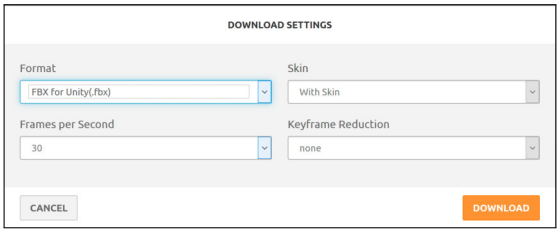


FIGURA 3.6 – Configuração das opções do modelo a descarregar

Para o projeto deste livro, sugere-se que se escolha o formato `FBX for Unity` e se garanta a inclusão do modelo (`With Skin`). Isto significa que o ficheiro exportado terá em consideração alguma compatibilidade com o Unity, sendo que para além das animações será exportado o modelo original depois de adicionado o esqueleto. As duas outras opções permitem aumentar o número de *frames* por segundo (para aumentar a qualidade da animação) e reduzir as *frames* principais da animação, o que pode gerar animações mais compactas, mas menos fluidas.


No sítio do livro (`www.fca.pt`) estão disponíveis, na pasta `Ant/Rigged`, os ficheiros correspondentes às animações.

3.2 CONTROLO DE ANIMAÇÕES

A personagem usada tem bastantes texturas, pelo que, para simplificar o seu uso, foi criado um pacote Unity que as inclui, juntamente com os respetivos materiais. Assim, o 1.º passo na inclusão da personagem no jogo é a importação do pacote *Ant Textures* (`AntTextures.unitypackage`).

Os pacotes do Unity consistem em ficheiros comprimidos que incluem os ficheiros relevantes (neste caso concreto, as imagens e os materiais), bem como um conjunto de ficheiros de metainformação. A criação destes pacotes pode ser feita a partir do próprio Unity, usando o menu `Assets` → `Export Package`. Por sua vez, a importação de pacotes incluídos no próprio Unity, como já foi visto, é feita a partir da lista de pacotes disponível no menu `Assets` → `Import Package`. Este mesmo menu inclui a opção `Custom Package`, que permite escolher um pacote disponível no sistema de ficheiros e realizar a sua importação. Depois de importado, ficará disponível no projeto uma pasta `AntModel`, que inclui as pastas `Materials` e `Textures`.

De seguida, serão importadas as animações. Todos os ficheiros obtidos da Mixamo (ou os ficheiros da pasta `Rigged` disponível no sítio da FCA) devem ser arrastados para a pasta `AntModel` dentro do separador *Projeto* (ou usada a opção `Import New Asset` do menu `Assets` para cada um dos ficheiros). A Figura 3.7 apresenta a estrutura da pasta `AntModel` depois de importados os materiais, texturas e os modelos animados.

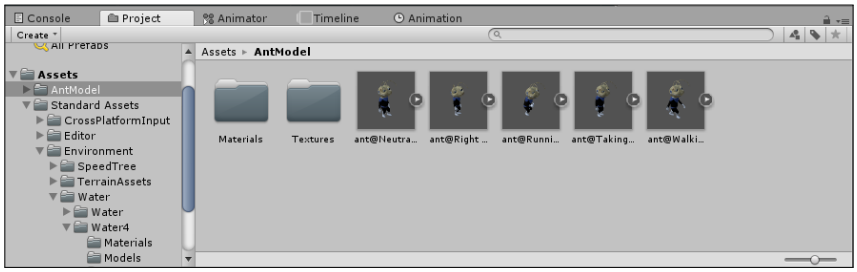


FIGURA 3.7 – Estrutura de pastas para os recursos da personagem


Embora habitualmente não seja importante a pasta em que são colocados os recursos importados, neste caso, as texturas foram importadas previamente e é fundamental que o Unity as reuse e não tente criar novos materiais.

Antes de usar as animações, é necessário, para cada uma, configurá-la como um modelo humanoide. Para isso, deverá selecionar todas as animações importadas, no separador *Projeto*, e aceder no *Inspetor* à opção `Rig` e indicar que o tipo de animação é humanoide, tal como apresentado na Figura 3.8. A cada alteração, deverá usar o botão `Apply`.

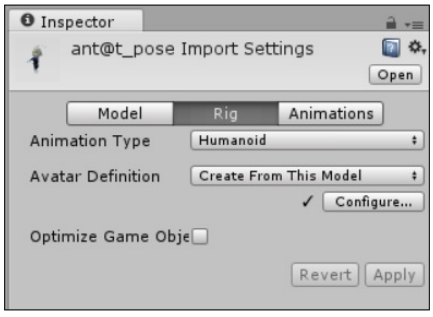


FIGURA 3.8 – Opção para animação humanoide

De seguida, ao arrastar a personagem (`ant@Neutral Idle`) para a *Cena*, deverá surgir a simpática formiga da Figura 3.9. A este objeto será associado o identificador `formiga` para ser mais fácil de o referir. O identificador poderá ser alterado no topo do *Inspetor*, ou diretamente na *Hierarquia*.

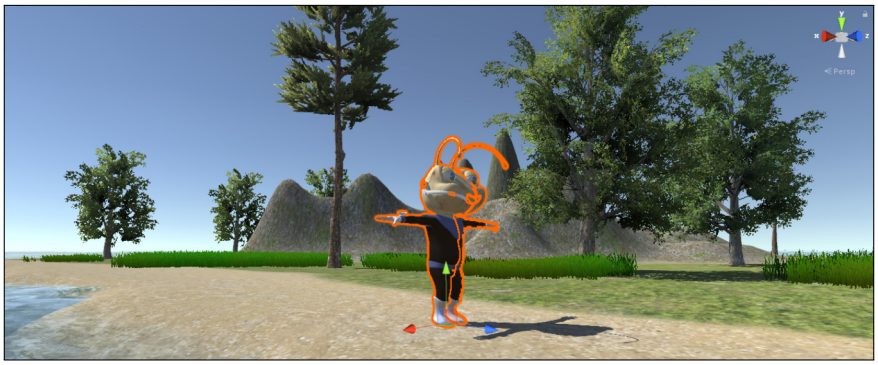


FIGURA 3.9 – Modelo da formiga na cena

A `formiga` é um objeto composto, ou seja, não é apenas um objeto mas uma árvore de objetos, que pode ser expandida na *Hierarquia*. O objeto de topo inclui os componentes *Transform* e *Animator*. O *Animator* é responsável por associar um controlador de animações (*Animator Controller*) a um objeto.

Antes da criação de um controlador, poderá aproveitar-se a ocasião para garantir que a opção `Apply Root Motion` no componente *Animator* se encontra ativa, o que irá permitir que o movimento das animações seja analisado e usado como se se tratasse do movimento do objeto na cena de jogo, o que irá simplificar a implementação do movimento-base sem necessidade de implementar qualquer linha de código.

Inicialmente não existe um controlador associado ao *Animator*. A criação de um novo *Animator Controller* é feita usando o menu `Create` do separador *Projeto*, ou usando a opção com o mesmo nome no menu `Assets`. Sugere-se que se coloque o ficheiro criado na pasta referente à personagem, para ser fácil de encontrar posteriormente. Por sua vez, a associação deste controlador à personagem é feita arrastando-o para o campo `Controller` do componente *Animator* (Figura 3.10).

A definição do controlador é feita num editor específico, disponível como um separador do Unity. Pode aceder a esse editor com um duplo clique sobre o controlador. A Figura 3.11 apresenta o editor de um controlador vazio.

Um controlador consiste numa máquina de estados. Quando o componente é iniciado, o controlador encontra-se no estado `Entry` e transita para o primeiro estado que lhe esteja ligado (neste momento, ainda nenhum). O 1.º passo corresponde à criação de um novo estado, clicando com o botão direito do rato sobre a área do editor e escolhendo a opção para adicionar um estado vazio (`Create State` → `Empty`). Ao criar este novo estado, a transição a partir do estado inicial é automaticamente criada e é apresentado o *Inspetor* do estado acabado de criar.

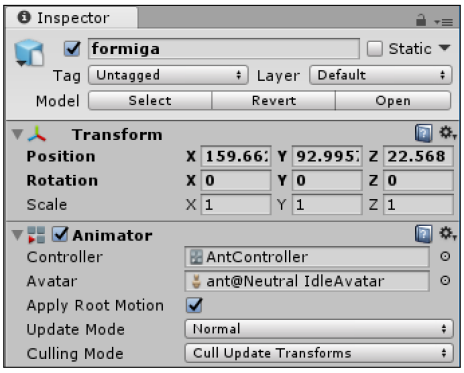


FIGURA 3.10 – Componente *Animator*

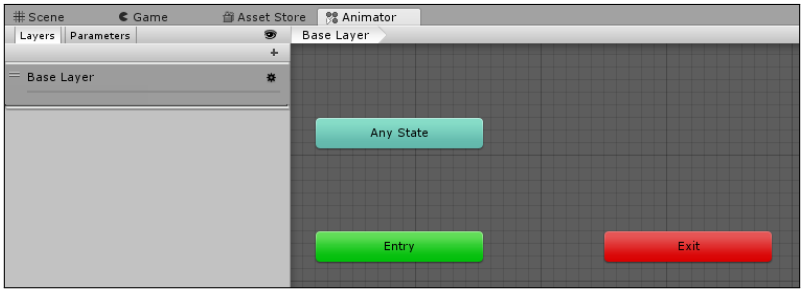


FIGURA 3.11 – Editor de controladores de animações (*Animator Controller*)

Este primeiro estado irá representar a personagem em descanso. Para tornar o jogo mais realista, será usada a animação `Neutral Idle` para a fazer respirar enquanto se encontra parada. O nome do estado pode ser alterado no *Inspetor* respetivo. No campo `Motion` deverá ser associada a animação correspondente. A Figura 3.12 apresenta o resultado destas operações. Se tudo estiver correto, na próxima execução[8] a formiga deixará de estar imóvel e começará a respirar.

Infelizmente, a formiga rapidamente se irá cansar, ficando estática. Isto acontece porque as animações, quando são criadas, não são configuradas para funcionar em ciclo (*loop*). Para o alterar, será necessário selecionar a animação, no separador *Projeto* e, no *Inspetor* respetivo, ativar as opções `Loop Time` e `Loop Pose`, e usar o botão `Apply` para aplicar as alterações efetuadas (Figura 3.13). Aproveitando a ocasião, poderá também ativar o funcionamento em ciclo para as animações de caminhada (`ant@Walking`) e de corrida (`ant@Running`). A opção `Loop Time` indica que a animação deverá ser repetida indefinidamente. A opção `Loop Pose` permite que o movimento seja mais fluido, não repetindo as posições iniciais e finais da animação.

[8] Para garantir que a formiga aparece em cena, será necessário colocar a câmara a focá-la.

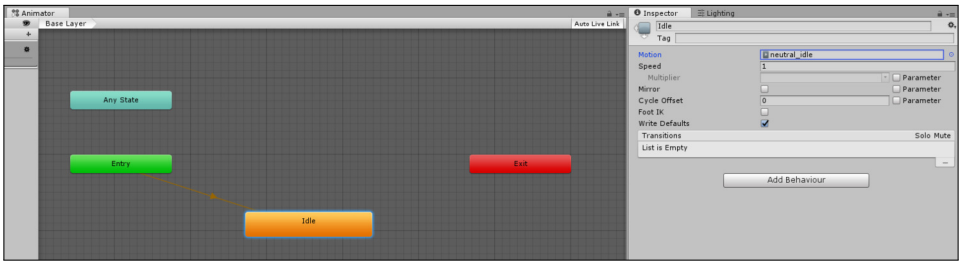


FIGURA 3.12 – Estado `Idle` da personagem

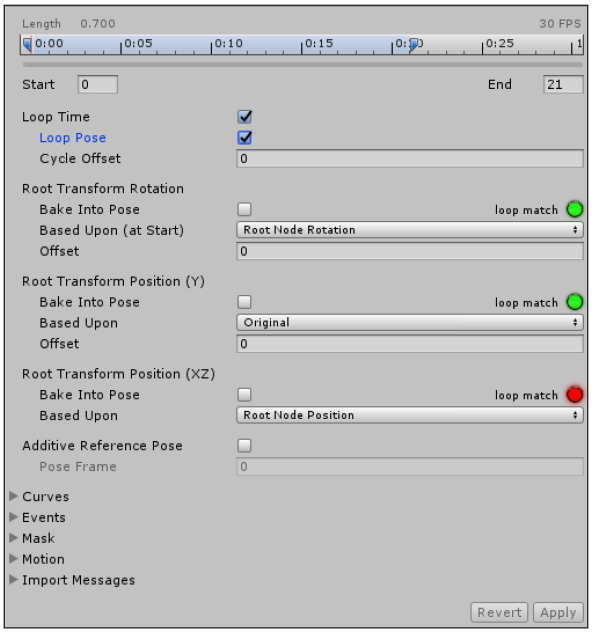


FIGURA 3.13 – Ativação das opções `Loop Time` e `Loop Pose`

O próximo passo é a criação de um novo estado para a caminhada (`Walk`). O processo é o mesmo da criação do estado anterior e deverá, neste caso, associar a animação `ant@Walking`. No entanto, será necessário criar as transições entre o estado `Idle` e o estado `Walk`, clicando com o botão direito do rato sobre o estado de origem, escolhendo a opção `Make Transition` e selecionando o estado de destino. A Figura 3.14 mostra a máquina de estados resultante.

Da forma como foram criadas, estas transições vão ser sempre executadas, já que não incluem quaisquer condições. A animação `Idle` irá executar uma vez, passando à animação `Walk` e voltando de novo à animação `Idle`, continuando neste ciclo indefinidamente.

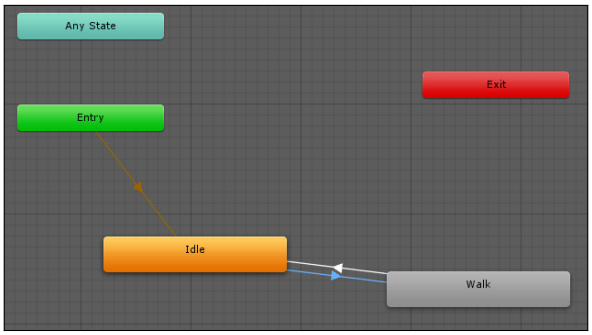


FIGURA 3.14 – Máquina de estados com os estados `Idle` e `Walk`

O resultado é uma formiga que ora anda, ora se cansa, e volta de novo a andar. Pretende-se que a transição para o estado de caminhada só seja realizada quando o jogador pretender movimentar-se. Do mesmo modo, a transição de regresso deverá ser ativada quando o jogador pretender parar. Estas transições vão depender do estado da personagem, que será representado por um parâmetro.

Os parâmetros podem ser de diferentes tipos. O menu de criação de parâmetros é apresentado na Figura 3.15. Para manter a máquina de estados simples, serão usados apenas os tipos booleano e real. Para o movimento, será usado um parâmetro de tipo booleano designado por `walk`. Os valores destes parâmetros podem ser alterados através de código e podem servir para controlar a transição entre diferentes estados.

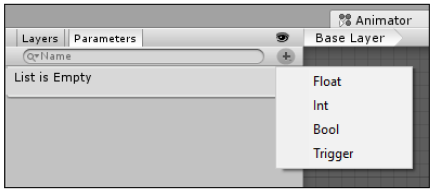


FIGURA 3.15 – Adicionar parâmetro à máquina de estados

As transições são configuradas selecionando a respetiva transição e alterando as suas propriedades. Na Figura 3.16 são apresentadas as propriedades das duas transições criadas.

Cada uma destas transições necessita de uma condição para que a transição ocorra. As condições são adicionadas ao fundo do *Inspetor*. Várias condições correspondem à conjunção lógica de todas as condições indicadas. No caso das transições em causa, as condições são definidas com base no parâmetro que foi criado. No primeiro caso, a transição tem lugar quando o parâmetro for verdadeiro e no segundo a transição é realizada quando o parâmetro for falso.

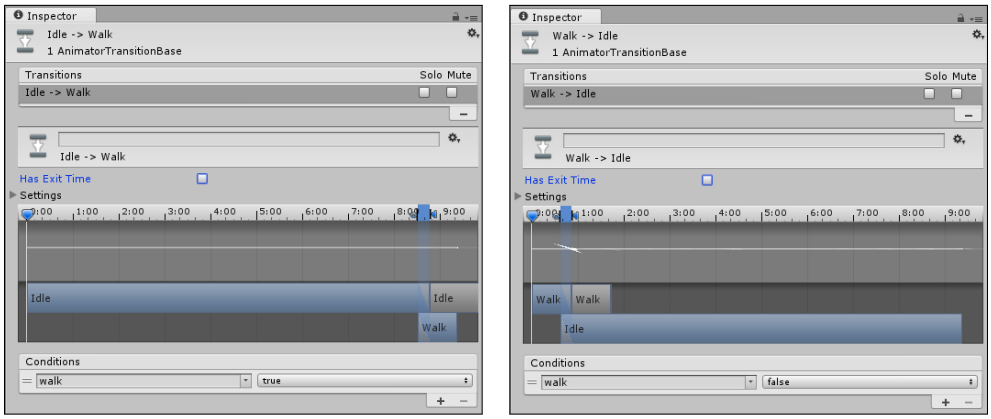


FIGURA 3.16 – Propriedades das transições entre `Idle` e `Walk` e entre `Walk` e `Idle`, respetivamente

Além das condições, é importante desativar a opção `Has Exit Time` em ambas as transições. Quando esta opção está ativa, a transição só será efetuada quando a animação do estado de origem tiver terminado. Com a opção desligada, a transição irá interromper a animação em curso, iniciando a animação do estado de destino.

Existem certas animações que devem ser terminadas e não interrompidas: saltos, o carregamento de uma arma, o pressionar de um botão, etc.

Para que estas alterações tenham algum efeito, é necessário implementar uma pequena *script*[9], denominada `AntInput`, e associá-la à formiga:

```
using UnityEngine;

public class AntInput : MonoBehaviour {
  Animator animator;

  void Start() {
    animator = GetComponent<Animator>();
  }
  void Update() {
    if (Input.GetAxis("Vertical") > 0.1f)
      animator.SetBool("walk", true);
    else
      animator.SetBool("walk", false);
  }
}
```

[9] O processo de criação de uma *script* é explicado na secção 1.3.6.

Nesta *script* é criada uma variável privada, do tipo `Animator`,[10] para armazenar a referência ao componente *Animator*, responsável por gerir a máquina de estados. Para a associação do componente à variável, seria possível a criação de uma variável pública, em que se arrastaria o componente para a variável apresentada no *Inspetor* da *script*. A abordagem apresentada usa o método `GetComponent` para obter uma referência a um componente do tipo indicado que esteja presente nesse mesmo objeto. Esta associação é feita no método `Start` para que seja executada apenas uma vez. A partir dessa altura, a variável `animator` é uma referência ao componente.

A gestão de *input* é realizada no método `Update`. Embora se possa validar o estado de teclas específicas, o Unity implementa um sistema abstrato de gestão de *input* que traz várias vantagens e deve ser preferido em detrimento da sua validação manual.

O Unity designa por *Eixo* (*Axis*) cada tecla, botão ou *input* de dispositivos de controlo. Existem vários eixos predefinidos e é também possível definir novos.

Para a gestão de movimento (usando um comando de consola, *joystick*, teclas cursoras, ou as teclas **AWSD**) existem dois eixos, *vertical* e *horizontal*, que armazenam valores no intervalo $[-1, 1]$ indicando a direção. No caso das teclas, e para o eixo *vertical*, o valor será 0 se nenhuma estiver ativa, 1 se estiver ativa a tecla **W** ou a tecla cursora para cima, e -1 se estiver ativa a tecla **S** ou a tecla cursora para baixo. No caso dos controlos analógicos, os valores irão variar de acordo com a inclinação do respetivo controlo. O eixo *horizontal* funciona de modo análogo, mas para as teclas **A** e **D** e cursoras esquerda e direita.[11] O valor destes eixos é obtido através do método `Input.GetAxis`, fornecendo-lhe o nome do eixo em questão. Por sua vez, o método `SetBool` permite alterar um parâmetro de tipo booleano da máquina de estados. Agora a formiga deverá ser capaz de caminhar para a frente.

O próximo passo será permitir que a formiga corra. Para tal será criado um novo estado, com a animação `ant@Running`, e estabelecidas ligações entre este e os outros dois estados já criados (`Walk` e `Idle`), e vice-versa, tal como demonstrado na Figura 3.17. São necessárias todas as ligações para permitir que a formiga inicie a corrida, quer esteja parada, quer esteja a andar e que passe da corrida para a caminhada ou para o estado parado.

Do mesmo modo que foi criado um parâmetro booleano para controlar o estado de caminhada, será criado um outro booleano, designado como `run`, para controlar a corrida. As transições criadas deverão ter as condições apresentadas na Tabela 3.1.[12] Note que a condição da transição entre os estados `Idle` e `Walk` também deve ser alterada, adicionando uma restrição extra, de modo a permitir que não interfira com a transição entre `Idle` e `Run`. Será igualmente necessário desligar a opção `Has Exit Time` em todas estas transições.

[10] Cada componente tem um tipo de dados equivalente, cujo nome é semelhante, mas sem espaços. Para distinguir, sempre que se referir um componente em C# será usado um tipo de letra diferente.

[11] A vantagem do uso de eixos predefinidos do Unity é a possibilidade de o utilizador os redefinir ao seu gosto, sem necessidade de reimplementação.

[12] Na Tabela 3.1, o símbolo $\wedge$ corresponde à conjunção lógica, pelo que deverão ser adicionadas várias condições para essa transição.

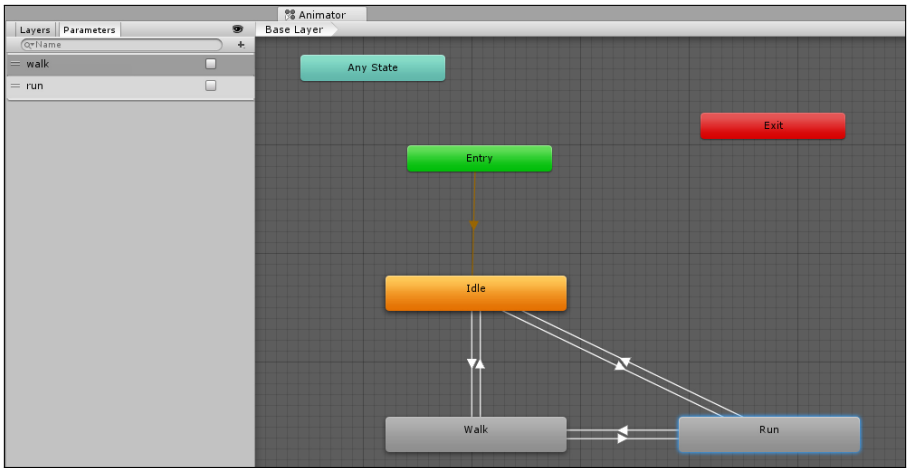


FIGURA 3.17 – Máquina de estados com os estados `Idle`, `Walk` e `Run`

ORIGEM	⟶	DESTINO	CONDIÇÃO
`Idle`	⟶	`Run`	$walk = true \wedge run = true$
`Run`	⟶	`Idle`	$walk = false \wedge run = false$
`Walk`	⟶	`Run`	$run = true$
`Run`	⟶	`Walk`	$run = false$
`Idle`	⟶	`Walk`	$walk = true \wedge run = false$

TABELA 3.1 – Condições para as transições entre os estados `Idle`, `Walk` e `Run`

Para que estas transições sejam selecionadas e executadas, a *script* terá de alterar o valor do parâmetro `run` se o jogador estiver a pressionar as teclas correspondentes. O Unity inclui teclas virtuais para várias ações, mas não inclui nenhuma para indicar a corrida, pelo que será aproveitada a ocasião para explicar a criação de novas teclas virtuais.

O menu `Edit` → `Project Settings` → `Input` permite aceder à configuração das teclas virtuais. Esta configuração consiste num *array* de nomes associados a um conjunto de eixos (teclas). A Figura 3.18 apresenta a configuração de teclas por omissão, na qual foi alterado o número de eixos (valor no topo) e adicionada uma entrada ao fundo, dando-lhe um nome e indicando a tecla correspondente (no campo `Positive Button`), neste caso, a tecla **shift** esquerda. Uma boa forma de perceber a configuração de teclas é espreitando os eixos já existentes, como o *horizontal*, o *vertical* ou mesmo o *jump*.

Os eixos que são configurados apenas com uma tecla de ação positiva são designados por *botão*. Para validar o estado de um botão, existem três métodos: `GetButton`, `GetButtonDown` e `GetButtonUp`. O primeiro retorna um valor verdadeiro quando o botão se encontra pressionado. Os outros dois retornam um valor verdadeiro quando o estado do botão muda de pressionado para não pressionado e vice-versa.

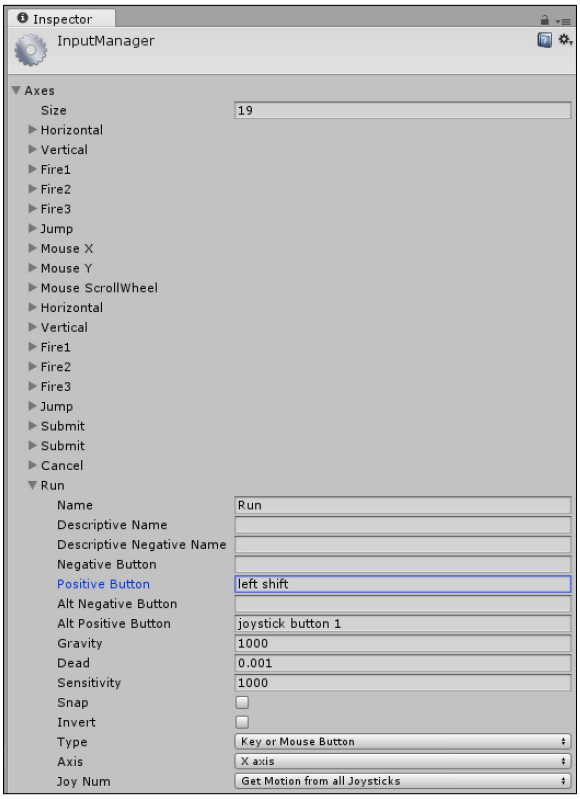


FIGURA 3.18 – Configuração de teclas e adição de uma nova entrada

Para que a formiga corra, é necessário que o jogador se encontre com o botão de corrida pressionado, pelo que o código deverá usar o método `GetButton`, tal como se segue. Note que é necessário usar o nome do eixo como foi definido nas propriedades e o nome do parâmetro como foi definido na máquina de estados.[13]

```
void Update() {
  animator.SetBool("walk", Input.GetAxis("Vertical") > 0.1f);
  animator.SetBool("run", Input.GetButton("Run"));
}
```


A caminhada e a corrida poderiam ser implementadas recorrendo ao conceito de *Blend Tree*, que permite uma mescla entre várias animações. No entanto, optou-se pelo uso de diferentes estados a fim de simplificar a implementação.

[13] O código do método `Update` definido anteriormente foi reescrito de modo a tornar-se mais compacto.

A animação de rotação só será usada quando o jogador estiver parado, já que durante a caminhada ou a corrida será apenas alterada a rotação da personagem no Unity, como veremos na secção 3.3.

Adicionados os dois estados de rotação (para a esquerda e para a direita), será necessário associar as animações. Na rotação para a direita, bastará escolher a animação respetiva. Para a rotação inversa, em vez de usar uma animação diferente, será escolhida, de novo, a rotação para a direita e indicada a opção `Mirror` (espelhar) no *Inspetor* do estado, tal como se verifica na Figura 3.19.

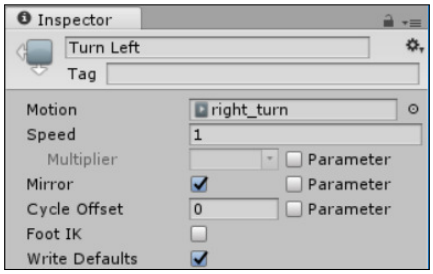


FIGURA 3.19 – Opção para espelhar a animação de um humanoide

Posteriormente, deverão ser criadas as transições de e a partir do estado `Idle`, para cada um dos estados de rotação, como se pode ver na Figura 3.20. Também foi criado um novo parâmetro `turn`, de tipo `Float`, que indicará se o movimento corresponde a virar para a esquerda (valor negativo) ou para a direita (valor positivo).

As novas transições terão associadas as condições discriminadas na Tabela 3.2.

ORIGEM	$\longrightarrow$	DESTINO	CONDIÇÃO
`Idle`	$\longrightarrow$	`Turn Right`	$turn > 0$
`Idle`	$\longrightarrow$	`Turn Left`	$turn < 0$
`Turn Right`	$\longrightarrow$	`Idle`	—
`Turn Left`	$\longrightarrow$	`Idle`	—

TABELA 3.2 – Condições das transições entre os estados `Idle`, `Turn Right` e `Turn Left`

Destas quatro transições, as duas primeiras terão a propriedade `Has Exit Time` inativa, já que se pretende que a formiga inicie a rotação (quando parada) de imediato. Por outro lado, as duas últimas transições não têm quaisquer condições. Para que estas transições funcionem, a propriedade `Has Exit Time` deve estar ativa, de modo a que a máquina de estados só regresse ao estado `Idle` depois de completar a rotação.

Com vista à utilização destas duas transições, será necessário alterar de novo a *script* de controlo da formiga, como se mostra de seguida:

```
void Update() {
  animator.SetBool("walk", Input.GetAxis("Vertical") > 0.1f);
  animator.SetBool("run", Input.GetButton("Run"));
  animator.SetFloat("turn", Input.GetAxis("Horizontal"));
}
```

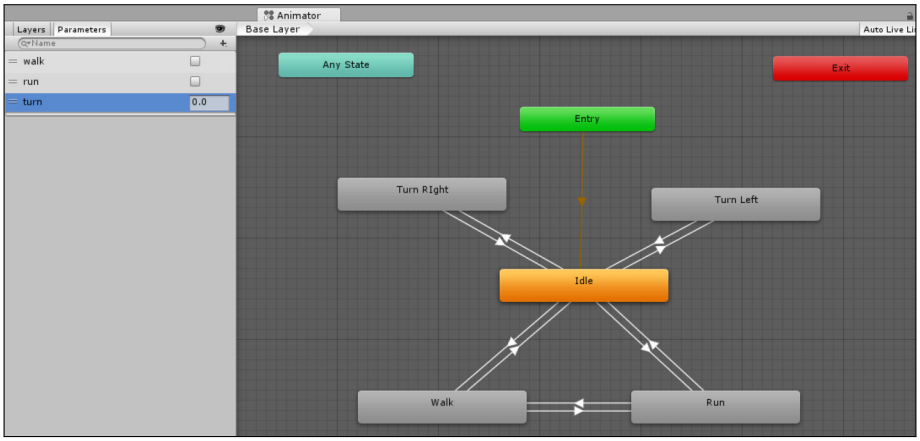


FIGURA 3.20 – Máquina de estados com os estados de rotação adicionados

Finalmente, será adicionado o estado de ação, em que a formiga se abaixa para apanhar alguma coisa do chão. Esse estado, designado por `Take`, terá associada a animação `ant@Taking Item`, e só estará ligado ao estado `Idle`, já que a formiga só poderá iniciar essa ação quando estiver devidamente parada. Para controlar este estado, será criado mais um parâmetro booleano, `take`, e as transições com as seguintes condições da Tabela 3.3.

ORIGEM	$\longrightarrow$	DESTINO	CONDIÇÃO
`Idle`	$\longrightarrow$	`Take`	$take = true$
`Take`	$\longrightarrow$	`Idle`	—

TABELA 3.3 – Condições das transições entre os estados `Idle` e `Take`

Mais uma vez, repare-se que a primeira deverá ter a propriedade `Has Exit Time` inativa e a segunda ativa. Já na *script*, iremos usar o botão `Fire1`, predefinido no gestor de teclas do Unity, para desencadear esta ação:

```
void Update() {
  animator.SetBool("walk", Input.GetAxis("Vertical") > 0.1f);
  animator.SetBool("run", Input.GetButton("Run"));
  animator.SetBool("take", Input.GetButton("Fire1"));
  animator.SetFloat("turn", Input.GetAxis("Horizontal"));
}
```

3.3 MOVIMENTO

Neste ponto, e porque foi usada a técnica de aplicação do movimento das animações ao objeto (`Apply Root Motion`), a formiga já se movimenta livremente, andando e correndo, e já é possível mudar de direção, desde que esteja parada. Além disso, quando está parada, também já é capaz de se abaixar. Nesta secção são apresentadas as funcionalidades em falta: a mudança de direção em movimento e a adaptação ao declive, subindo-o ou descendo-o. Também será adicionada uma câmara para seguir a formiga.

Para a rotação durante o movimento, será alterada a *script* de modo a que a formiga seja rodada através da animação quando estiver parada (se os parâmetros `run` e `walk` forem falsos) e através da rotação do objeto quando se encontrar em movimento (bastando para isso validar se o parâmetro `walk` é verdadeiro, já que será sempre verdadeiro durante a caminhada ou a corrida).

Para a rotação, serão usados dois valores: um será a velocidade de rotação-base, quando a formiga se encontra em caminhada (em radianos, denominada `RotationSpeed`), e um fator multiplicativo que indicará quão mais rápida será a rotação durante a corrida (denominado `RunRotationFactor`). Estas duas variáveis são declaradas na classe como públicas para que estejam disponíveis no *Inspetor* da *script* e permitam ao programador adaptar os valores facilmente sem necessidade de editar código. Os valores predefinidos serão 1.5 para a velocidade de rotação e 2 para o fator de rotação em corrida (o que corresponde a uma rotação de 3 durante a corrida):

```
public class AntInput : MonoBehaviour {
  Animator animator;
  public float RotationSpeed = 1.5f;
  public float RunRotationFactor = 2.0f;

  void Start() {
    // ...
  }

  void Update () {
    animator.SetBool("walk", Input.GetAxis("Vertical") > 0);
    animator.SetBool("run", Input.GetButton("Run"));

    if (animator.GetBool("walk") || animator.GetBool("run")) {
      float rotation = Input.GetAxis("Horizontal");
      if (animator.GetBool("run"))
        rotation *= RunRotationFactor;
      transform.Rotate(Vector3.up, rotation * RotationSpeed);
    }
```

```
    else {
      animator.SetFloat("turn", Input.GetAxis("Horizontal"));
      animator.SetBool("take", Input.GetButtonDown("Fire1"));
    }
  }
}
```

Antes de adicionar o fator gravidade, será adicionada uma câmara que siga a formiga, de modo a que seja mais fácil validar o seu movimento (e porque facilitará o processo de jogo). Em vez de se definir o comportamento da câmara, será usada uma câmara predefinida do Unity, disponível no pacote *Cameras*. Para adicionar este pacote, deve ser usada a opção `Assets` → `Import Package` → `Cameras`. O Unity apresentará os conteúdos do pacote, tal como demonstrado na Figura 3.21.

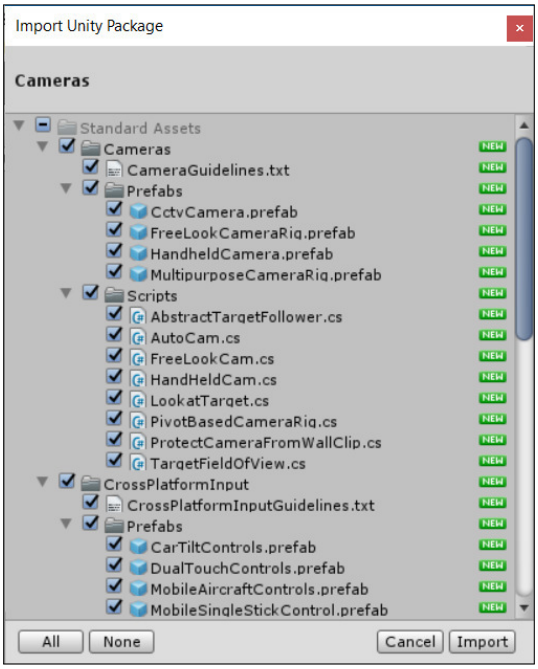


FIGURA 3.21 – Conteúdo do pacote *Cameras*


Dado o tamanho reduzido deste pacote e a grande interligação entre os diferentes ficheiros que contém, sugere-se a importação de todo o seu conteúdo.

Depois de importado, estará disponível uma câmara denominada `Multipurpose Camera Rig` na pasta `Standard Assets/Cameras/Prefabs` (Figura 3.22). Para a usar, esta

deverá ser arrastada para a *Cena*, ou para a *Hierarquia*. No entanto, não é possível ter mais do que uma câmara na cena de jogo[14], pelo que a câmara já presente deverá ser apagada (selecionando-a na *Hierarquia* e usando a tecla **Delete**).

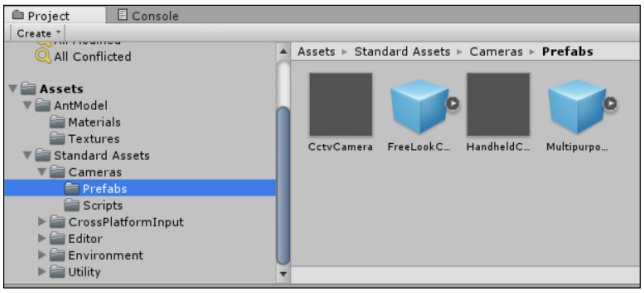


FIGURA 3.22 – Localização do *prefab* `Multipurpose Camera Rig`

Esta câmara está preparada para seguir o jogador mas, para isso, precisa de saber qual é o objeto que corresponde à personagem do jogador. Assim, é necessário selecionar o objeto `formiga` na *Hierarquia*, e alterar a sua etiqueta (*tag*) no *Inspetor* para `Player`, tal como demonstrado na Figura 3.23. Estas etiquetas permitem distinguir objetos em diferentes tipos, de modo a que se possam selecionar objetos não só pelo seu nome, mas também pela etiqueta que têm associada.

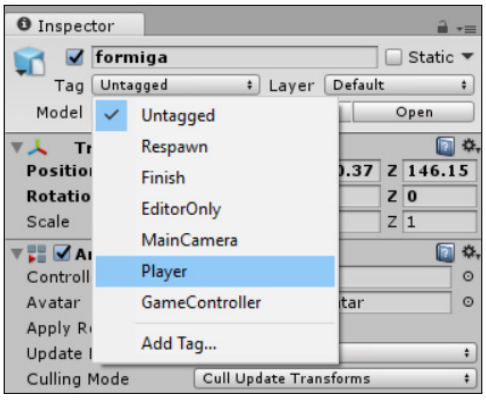


FIGURA 3.23 – Definição da *tag* de um objeto


Existem várias opções de configuração do comportamento desta câmara no seu *Inspetor*. No entanto, a explicação está fora do âmbito introdutório deste livro.

[14] Exceto em certas situações, que não serão abordadas neste livro.

Finalmente, será adicionada gravidade à formiga. Para tal será necessário voltar a selecionar as animações da formiga, que foram importadas para a pasta `AntModel` e, para cada uma, indicar que a posição vertical do objeto durante a animação deve ser embebida no modelo. Isto irá permitir que o motor de física seja capaz de alterar a posição da formiga. Assim, e para todas as animações, deverá aceder ao *Inspetor* e, sob a secção `Root Transform Position (Y)`, ativar a opção `Bake Into Pose` (Figura 3.24), seguindo-se a aplicação das alterações através do botão `Apply`.

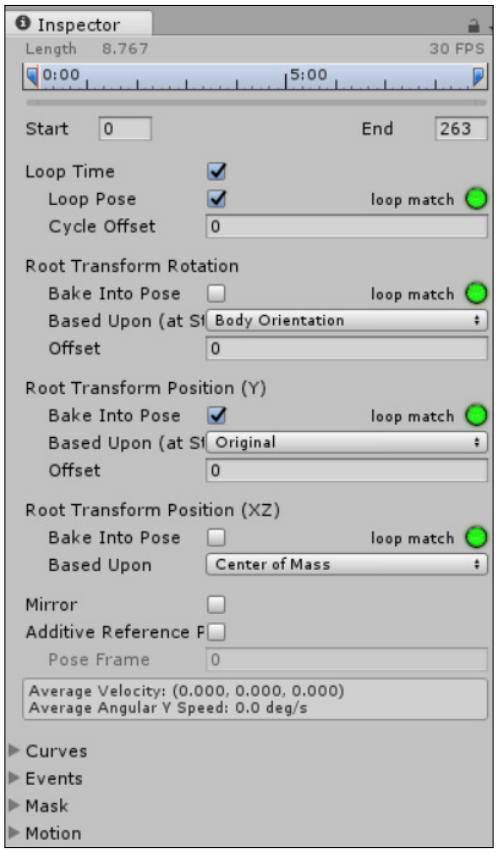


FIGURA 3.24 – Configuração da animação para que a posição da personagem seja externamente controlável

Terminada esta operação, será selecionado o objeto `formiga` no *Inspetor* e adicionados dois componentes: um *Rigidbody*, para indicar que o objeto deverá ser sensível à gravidade, e um *Capsule Collider*, que indicará a forma volumétrica a ser usada para calcular colisões.


Será usada uma cápsula por ser cilíndrica e ter os cantos arredondados. A forma arredondada inferior irá permitir que a formiga seja capaz de subir pequenos declives sem que para isso seja necessário escrever qualquer código.

Adicionados os componentes, o próximo passo é a sua configuração. No componente *Rigidbody* será necessário ativar as opções `Freeze Rotation` para todos os eixos (x, y e z), sob a opção `Constraints` (Figura 3.25), o que irá limitar a rotação da formiga pelo motor de física, permitindo apenas rotação programada. Assim, garante-se que a formiga não cai nem muda de rotação ao subir ou descer declives.

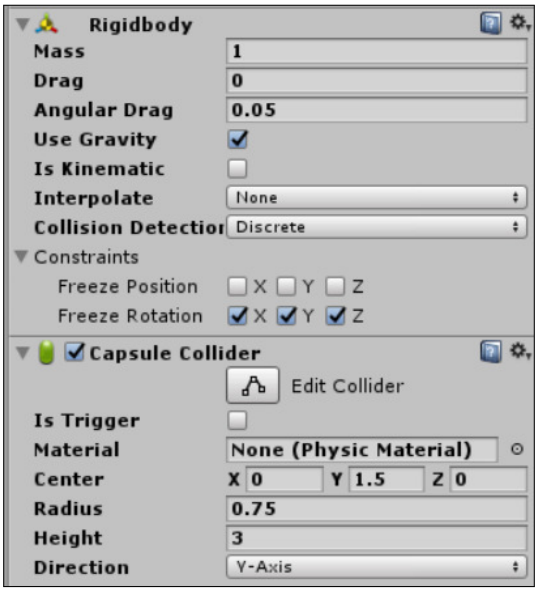


FIGURA 3.25 – Opções de configuração dos componentes *Rigidbody* e *Capsule Collider*

No *Capsule Collider* será necessário configurar a posição da cápsula, seja copiando os valores aqui apresentados, ou experimentando e analisando a posição relativa da cápsula em relação à formiga, como demonstrado na Figura 3.26. Esta cápsula irá representar os limites sólidos da personagem.

Nesta altura, a formiga deverá ser capaz de andar, correr, virar-se quando estiver parada ou em movimento, apanhar objetos do chão, bem como subir e descer colinas, sendo que a câmara deverá segui-la.

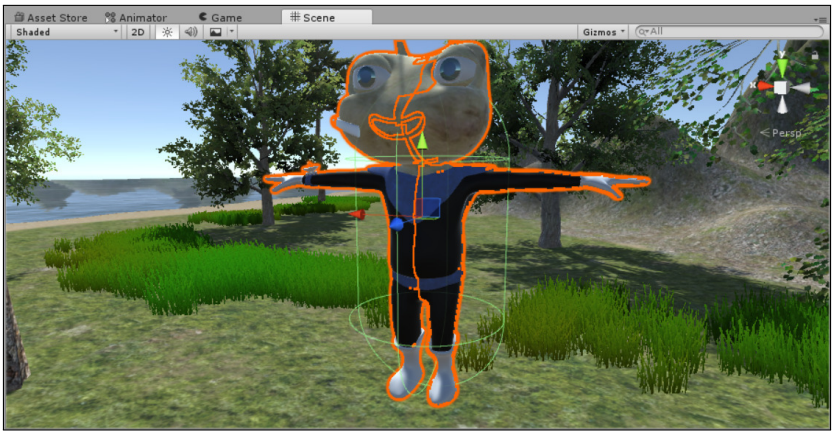


FIGURA 3.26 – Configuração do *Capsule Collider* de modo a incluir todo o modelo da formiga (exceto os braços)

3.4 COLISÕES

Nesta secção é adicionada uma funcionalidade simples, mas importante. As formigas não sabem nadar, e seria muito aborrecido que, por distração, o jogador afogasse a personagem. Este é um problema geral em muitos jogos: a definição da área de jogo sem que o jogador perceba que se encontra preso, num mundo fechado. No caso da ilha, isso fica mais ou menos claro, mas noutras situações, será necessário adicionar objetos que limitem o movimento da personagem (como montanhas íngremes, grandes valas, casas, etc.).

Na ilha, o movimento da formiga será limitado por paredes transparentes, que a impedirão de se deslocar para o mar. Estas paredes serão vários componentes *Collider*. A fim de facilitar a gestão dos objetos na *Hierarquia*, será criado um objeto vazio (*Empty Game Object*) denominado `limites` e, dentro deste, tantos objetos vazios quantos os necessários para criar paredes virtuais à volta da ilha (clicando com o botão direito do rato sobre o objeto `limites` e escolhendo a opção `Create Empty`). Cada um destes limites terá um *Box Collider* que deverá ser colocado a delimitar a ilha, como demonstrado na Figura 3.27.

Embora seja possível alterar o tamanho do *Collider* manualmente, alterando o tamanho no *Inspetor*, é mais simples usar o botão `Edit Collider`, que permite a edição interativa do *Collider*. Quando um *Collider* se encontra selecionado, é representado por um cubo verde na *Cena* e, em cada face, é apresentado um ponto que pode ser usado para arrastar e aumentar o seu tamanho. Aproveitando que a ilha é quadrada, só foram necessários quatro componentes *Collider* como é ilustrado na Figura 3.28.

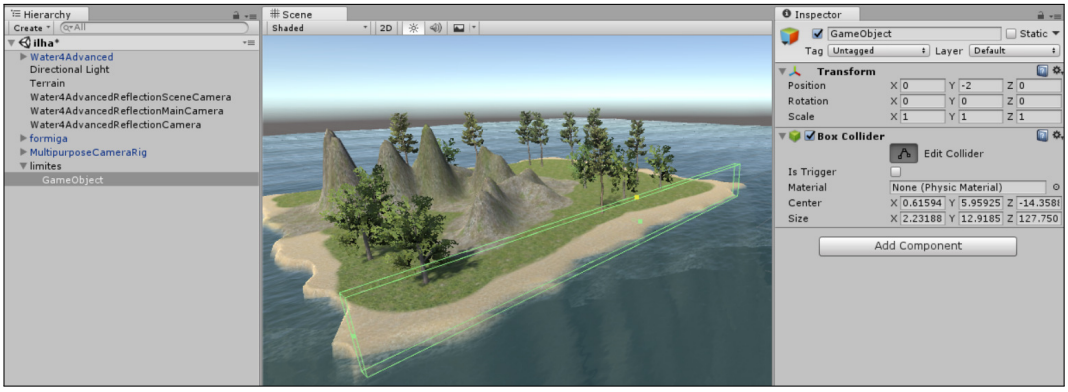


FIGURA 3.27 – Criação de um dos limites da ilha

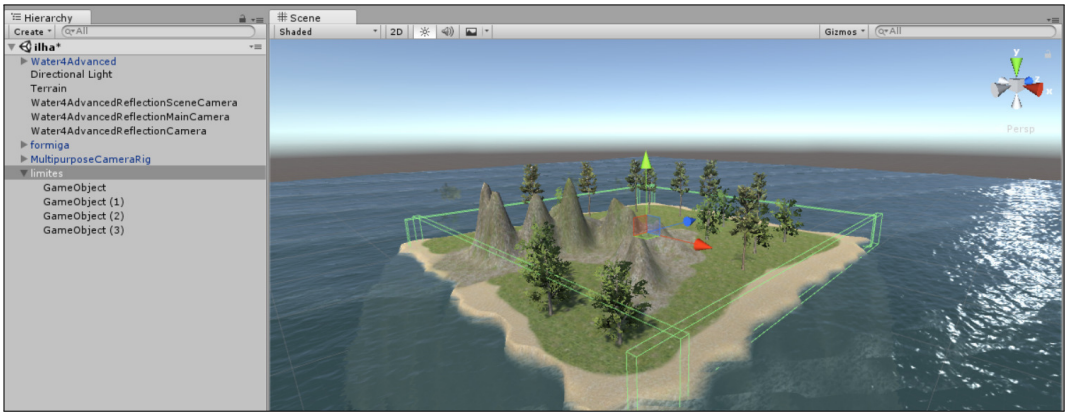


FIGURA 3.28 – Definição dos limites da ilha


Para facilitar o processo de colocação dos *Collider*, poderá definir um para uma das margens da ilha e depois duplicá-lo, movê-lo e rodá-lo para criar as restantes margens.

Definidos estes limites, a formiga não conseguirá sair da ilha, garantindo-se, assim, que o jogador não a afogará por distração.

4 Lógica de Jogo

Neste capítulo é implementada a lógica de jogo. A formiga terá como objetivo apanhar todos os cogumelos existentes na ilha. No entanto, a sua energia irá diminuir ao longo do tempo e a única forma de a aumentar é comendo cogumelos. Além disso, existirão algumas armadilhas que farão a energia da formiga esvair-se mais rapidamente.

4.1 *COLLIDERS* E *TRIGGERS*

Até ao momento, a formiga abaixa-se sempre que é usada a tecla **Ctrl** esquerda, como foi definido no controlador. Mas não faz sentido que a formiga se abaixe quando não há nada para ser apanhado. Assim, num 1.º passo serão adicionados cogumelos à cena, e posteriormente, alterado o código da formiga, de modo a que esta só se abaixe quando existir um cogumelo para ser apanhado.

Será usado um pacote designado por *Sinuous Shrooms*, com modelos de cogumelos, que é disponibilizado gratuitamente na *Asset Store*[15]. Depois de importado, ficarão disponíveis vários *prefabs* na pasta `SinuousShrooms/Prefabs`.

Após ter escolhido qual o cogumelo a usar e o ter arrastado para a *Cena*, será preparado todo o código necessário para que a formiga o consiga apanhar. Assim que estiver devidamente configurado, será criado um *prefab* para facilmente se povoar todo o terreno com cogumelos.

O tamanho original dos cogumelos é ligeiramente grande, quando comparado com o tamanho da formiga, pelo que será boa ideia alterar a sua escala. Posteriormente, e para que seja possível a interação da formiga com eles, será adicionado um componente *Box Collider* que deverá ficar ajustado ao tamanho do cogumelo, como é mostrado na Figura 4.1.


Se for difícil apanhar o cogumelo, pelo facto de a área de colisão ser demasiado pequena, poderá aumentar o tamanho do *collider* de modo a que a zona em que a formiga é capaz de apanhar o cogumelo seja maior.

[15] Também está disponível no sítio `www.fca.pt` (ficheiro `Sinuous Shrooms.unitypackage`).

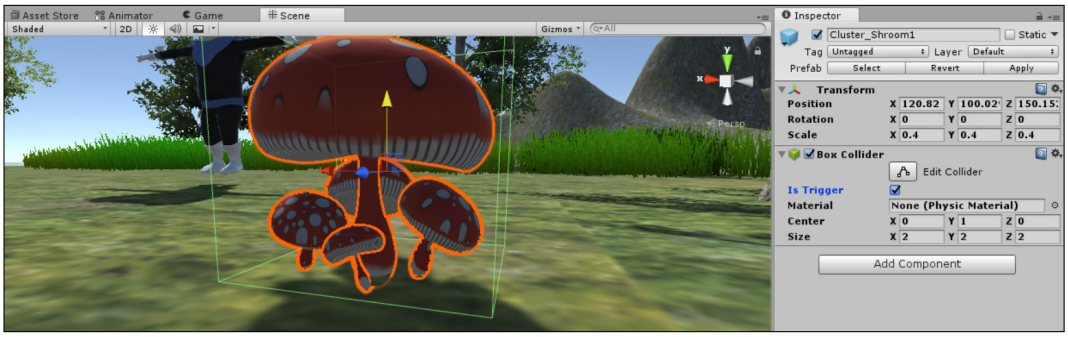


FIGURA 4.1 – Cogumelo com *collider* configurado como `trigger`

O problema em colocar um *collider* nesse objeto é que, a partir desse momento, todos os outros objetos passam a colidir com ele, e não é esse o objetivo, pois a formiga não deverá colidir e ficar presa ao tentar passar por um cogumelo. Para isso, deverá ser ativada a opção `Is Trigger`, que transforma um *collider* num detetor de objetos: é ativado se outro objeto está dentro da área definida pelo *collider*, mas não gera colisões físicas.


Embora não exista uma colisão, quando um dos *colliders* tem a opção `Is Trigger` ativa, é habitual continuar a referir que são detetadas colisões, pelo que no texto que se segue esse termo será referido com frequência.

Quando existe uma colisão entre dois objetos que contenham *colliders*, o Unity ativa um evento em cada um dos objetos. Se ambos tiverem a opção `Is Trigger` desligada, são usados eventos na forma `OnCollision`. Se pelo menos um dos *colliders* tiver essa opção ligada, serão invocados os eventos da forma `OnTrigger`. A Tabela 4.1 resume os métodos existentes.

Para apanhar cogumelos será usado o método `OnTriggerStay`. Ou seja, é preciso que a formiga se encontre em colisão com o cogumelo para que ele possa ser apanhado. No entanto, este evento é invocado quando qualquer objeto que tenha um *trigger* colida com a formiga, pelo que é preciso detetar se o objeto em questão é um cogumelo. Para isso, será usada uma *tag* (etiqueta), como a que foi usada anteriormente na formiga, quando se pretendia que a câmara a seguisse. É possível criar novas *tags*, bastando para isso abrir a opção de escolha da *tag* do cogumelo e selecionar a opção `Add Tag`, como demonstrado na Figura 4.2.

Surgirá o *Inspetor* com a lista de *tags* definidas pelo utilizador, que estará vazia. Usando o botão +, é possível adicionar uma nova entrada na lista. Posteriormente, deverá aplicar a etiqueta aos cogumelos. Feita esta alteração, será alterada a *script* `AntInput`, associada à formiga, de modo a validar a existência de colisão com o cogumelo.

MÉTODO	DESCRIÇÃO
`OnCollisionEnter`	É invocado quando o *collider* começa a tocar num outro *collider* ou *gameobject*.
`OnCollisionExit`	É invocado quando o *collider* deixa de tocar num outro *collider* ou *gameobject*.
`OnCollisionStay`	É invocado em cada *frame* para cada *collider* que se encontra em colisão com um outro *collider* ou *gameobject*.
`OnTriggerEnter`	É invocado para o *collider* que entra na zona de um *trigger*, assim como no objeto que define esse mesmo *trigger*.
`OnTriggerExit`	É invocado para o *collider* que sai da zona de um *trigger*, assim como no objeto que define esse mesmo *trigger*.
`OnTriggerStay`	É invocado em cada *frame* para o *collider* que se encontra na zona de um *trigger*, bem como no objeto que define esse mesmo *trigger*.

TABELA 4.1 – Métodos invocados nos eventos de colisão

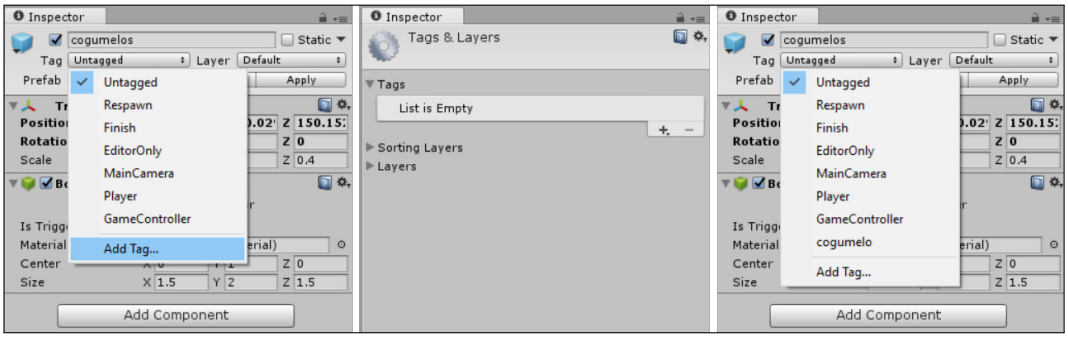


FIGURA 4.2 – Criação de uma nova *tag*

Em primeiro lugar, será alterado o método `Update`, removendo a linha que invoca a animação com a formiga para apanhar o cogumelo, resultando no seguinte código:

```
void Update () {
  animator.SetBool("walk", Input.GetAxis("Vertical") > 0);
  animator.SetBool("run", Input.GetButton("Run"));

  if (animator.GetBool("walk") || animator.GetBool("run")) {
    float rotation = Input.GetAxis("Horizontal");
    if (animator.GetBool("run")) {
      rotation *= RunRotationFactor;
    }

    transform.Rotate(Vector3.up, rotation * RotationSpeed);
  }
```

```
    else {
      animator.SetFloat("turn", Input.GetAxis("Horizontal"));
    }
  }
```

Dado que o *collider* dos cogumelos está configurado como um *trigger*, será necessário usar um evento deste tipo. Além disso, a formiga deverá conseguir apanhar o cogumelo a qualquer momento, e não apenas quando entra em contacto com ele, pelo que será usado o método `OnTriggerStay`:

```
  void OnTriggerStay(Collider c) {
    if (c.gameObject.tag == "cogumelo" &&
        Input.GetButtonDown("Fire1")   &&
        !animator.GetBool("walk"))
    {
      Destroy(c.gameObject);
      animator.SetBool("take", true);
    }
  }
```

Os métodos `OnTrigger` recebem a referência ao *collider* que está configurado como *trigger*. Então, em primeiro lugar, verifica-se se esse *collider* corresponde a um cogumelo. Em caso afirmativo, verifica-se se o jogador carregou, nesse momento, na tecla configurada como **Fire1** e se não se encontra a correr. Se todas estas condições forem validadas, então, o cogumelo é destruído e é iniciada a animação da formiga.

Infelizmente, esta abordagem não funciona. Ao usar o método `SetBool`, o valor do parâmetro no controlador muda para verdadeiro, mas não existe nenhum código que o volte a colocar com um valor falso, pelo que depois de a formiga se levantar, voltará a baixar-se para tentar apanhar de novo o cogumelo, sem nunca mais parar. Seria necessário voltar a colocar a variável com o valor falso. Para resolver este problema, será alterado o parâmetro `take`, mudando-o do tipo *booleano* para o tipo *trigger*.

 Por coincidência, a funcionalidade *trigger* dos *colliders* e a funcionalidade com o mesmo nome do *Animator Controller* são aqui apresentados ao mesmo tempo. No entanto, não há qualquer relação entre eles.

A diferença entre parâmetros booleanos e do tipo *trigger* reside no facto de, este último, depois de ser ativado, só ser desativado quando alguma transição o consumir, altura em que volta ao estado inativo. Esta é uma funcionalidade bastante útil e poderosa, mas que também pode produzir algumas dores de cabeça, já que um *trigger* ativado pode vir a ser consumido depois de várias transições terem sido feitas.

No caso em questão, como o código implementado garante que o parâmetro `take` será ativado apenas quando a formiga se encontra num estado capaz de apanhar um cogumelo, não deverão surgir quaisquer problemas.

Depois de alterar[16] o tipo do parâmetro `take`, será necessário adaptar a transição do estado `Idle` para o estado `Take`, tal como demonstrado na Figura 4.3.

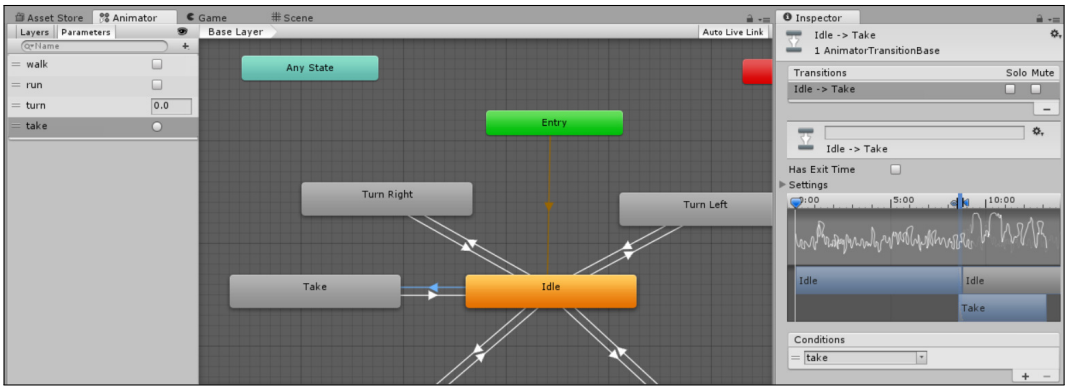


FIGURA 4.3 – Configuração da transição `Idle` para `Take` usando um *trigger*

Note que ao configurar uma transição com base num *trigger* não é preciso especificar o valor necessário para que a transição seja desencadeada. Nestes casos, a transição é iniciada se o *trigger* tiver sido ativado e ainda não tiver sido consumido por nenhuma transição prévia.

Para ativar o *trigger* é necessário alterar o código anterior, fazendo uso do método `SetTrigger`, como se mostra de seguida:

```
void OnTriggerStay(Collider c) {
  if (c.gameObject.tag == "cogumelo" &&
      Input.GetButtonDown("Fire1")   &&
      !animator.GetBool("walk"))
  {
    Destroy(c.gameObject);
    animator.SetTrigger("take");
  }
}
```

Neste momento, a formiga já se deverá abaixar devidamente e apenas uma vez. No entanto, demora algum tempo a abaixar-se até conseguir chegar ao chão. Para que o cogumelo não desapareça antes de a formiga se abaixar, poderá ser usado um segundo

[16] Infelizmente, não é possível alterar o tipo de um parâmetro diretamente. Será necessário apagar o anterior e criar um novo, com o tipo desejado.

argumento no método `Destroy`, que corresponde a um temporizador do tempo que falta para o objeto ser realmente destruído:

```
Destroy(c.gameObject, 1.5f);
```


Embora esta abordagem funcione, não é a ideal. O Unity permite associar a invocação de métodos a determinadas *frames* de animações, pelo que se poderia verificar exatamente em que *frame* a formiga se encontra abaixada e aplicar aí a invocação à destruição do objeto.

Ainda que mais tarde se façam outras alterações ao cogumelo, neste momento está pronto para ser criado um *prefab*. Para isso, basta arrastá-lo da *Hierarquia* para uma pasta desejada no separador *Projeto*, apagando-se, de seguida, o objeto original da *Hierarquia* (ou da *Cena*).

Depois de criado o *prefab*, deverão ser colocadas cópias deste ao longo do terreno, para que a formiga tenha vários cogumelos para apanhar.


Seria interessante que os cogumelos fossem colocados automaticamente no terreno, no entanto, preferiu-se simplificar, pelo que deverão ser colocados manualmente pelo programador.

4.2 ESTADO DE JOGO: *SINGLETON*

Todos os jogos têm informação importante para manter ao longo de cada jogada. Num jogo de tabuleiro é capaz de ser suficiente guardar as peças que estão em jogo e as suas posições, mas à medida que forem adicionadas mecânicas ao jogo e informação de estado sobre o jogador ou os seus inimigos, a quantidade e diversidade de informação a armazenar aumenta.

No jogo em desenvolvimento, será necessário ter alguma informação, como o número de cogumelos em jogo e qual o nível de energia atual da formiga. Com o desenrolar do jogo será diminuído o valor de energia ao longo do tempo, contarar-se-á o número de cogumelos apanhados e aumentar-se-á a energia da formiga a cada cogumelo apanhado.

Existem muitas formas de armazenar esta informação, que poderão ser mais centralizadas ou distribuídas. Uma abordagem distribuída irá guardar cada detalhe de informação num objeto distinto, de acordo com a proveniência dessa informação. Numa abordagem centralizada, será armazenada toda a informação num único objeto.

Esta segunda abordagem é aquela que será mais fácil de manter, já que qualquer alteração à lógica do jogo poderá ser facilmente realizada, sem necessidade de percorrer todos os objetos envolvidos para atualizar o código implementado.

A questão seguinte prende-se com como e onde guardar essa informação. Uma abordagem sensata poderia ser o uso do objeto da formiga, visto que se pretende guardar dados que estão diretamente relacionados com ela. Mas, a abordagem mais comum é a criação de um objeto especial, responsável apenas pela manutenção do estado de jogo. Uma das vantagens será a forma fácil como se poderá aceder a estes dados, usando diretamente uma variável estática de uma classe. Outra das vantagens é que, se o jogo for multijogador, não haverá dois objetos com informação do estado do jogo, mantendo-se um único objeto com a informação sobre todos os jogadores.

Assim, será criado um objeto vazio (*Empty Game Object*), designado por `GameManagerObject`. A este objeto será associada uma nova *script* denominada `GameManager` e que centralizará toda a informação relativa ao estado do jogo.

Sendo este objeto tão central ao funcionamento do jogo, será natural que vários objetos interajam com ele. A forma habitual de realizar este tipo de interação (uma *script* que invoca métodos de outra) é criar uma variável de instância pública, tendo como tipo de dados a classe da *script* que se pretende invocar, para que, usando o editor do Unity, se possa arrastar o objeto em causa para essa variável. Para demonstrar este processo, adicione-se no `GameManager` código para contar o número inicial de cogumelos em jogo e para disponibilizar um método para a contagem do número de cogumelos apanhados pela formiga. Para isso, serão criadas duas variáveis públicas, já que será necessário aceder-lhes a partir de outras *scripts*, que irão armazenar o número total de cogumelos em jogo, e saber quantos destes já foram recolhidos:

```
using UnityEngine;

public class GameManager : MonoBehaviour {

  public int TotalMushrooms;
  public int PickedMushrooms = 0;

  void Start() {
    TotalMushrooms = GameObject.FindGameObjectsWithTag("cogumelo").Length;
  }

  public void PickMushroom() {
    PickedMushrooms++;
  }
}
```

Para que facilmente se possam plantar novos cogumelos sem necessidade de alterar na *script* o número total de cogumelos disponíveis, optou-se por, no método `Start`, contar o número de objetos em cena que tenham a etiqueta `cogumelo` atribuída. O método `FindGameObejctsWithTag` retorna um *array* com todos os objetos que estejam atualmente na cena de jogo e que tenham determinada etiqueta (*tag*) atribuída. Por sua vez, é definido um método para contar o número de cogumelos apanhados.

Para que este código conte alguma coisa, a formiga deverá notificar o `GameManager` sempre que come um cogumelo. Aplicando a abordagem utilizada anteriormente, será definida uma variável pública no início da *script* de gestão da formiga,

```
public class AntInput : MonoBehaviour {
  public GameManager gameManager;
  // ...
```

e, por sua vez, o código referente ao apanhar do cogumelo será substituído por:

```
  void OnTriggerStay(Collider c) {
    if (c.gameObject.tag == "cogumelo" &&
        Input.GetButton("Fire1")       &&
        !animator.GetBool("walk"))
    {
       Destroy(c.gameObject, 1.5f);
       animator.SetTrigger("take");
       gameManager.PickMushroom();
    }
  }
```

Finalmente, para que tudo funcione, deverá, dentro do editor do Unity, arrastar o objeto `GameManagerObject` da *Hierarquia* para o campo `Game Manager` do componente *AntInput* no *Inspetor* da formiga.

Embora esta abordagem funcione perfeitamente e sem problemas de eficiência, não é uma solução prática para o programador que, em todas as *scripts* em que pretenda aceder ao `GameManager`, terá de definir a variável pública e arrastar o objeto para o campo respetivo. Isto é especialmente aborrecido, já que, muitas vezes, quando ao escrever o código da funcionalidade em questão, é esquecido o preenchimento do campo respetivo.

Uma solução é o uso de um *Singleton*. Este é um padrão conhecido das linguagens de programação orientadas a objetos que permite tornar o acesso a uma instância de determinado objeto simples e direta. A desvantagem da abordagem é que só funciona se o objeto em causa só puder ter, a cada momento, uma instância. Como durante o jogo só haverá um `GameManager`, esta questão não é muito relevante, pelo que o seu uso não deverá levantar quaisquer problemas.

A implementação de um *Singleton* passa pela definição de uma variável estática e pública, que terá um tipo idêntico ao da classe em questão. O construtor da classe, que deverá ser invocado apenas uma vez para a única instância que irá existir, é responsável por armazenar a instância criada nessa variável:

```
public class GameManager : MonoBehaviour {
    public static GameManager instance = null;
    // ...
```

Como as classes que derivam de `MonoBehaviour` não podem implementar um construtor, o código que lá seria colocado irá ser definido no método `Awake`:

```
  void Awake() {
    if (instance == null)
      instance = this;
    else
      Destroy(gameObject);
  }
```

O método `Awake` é semelhante ao método `Start`, mas é invocado antes deste. Além disso, é invocado uma única vez ao longo da vida da *script*, enquanto o método `Start` pode ser invocado várias vezes.

Um dos problemas típicos da implementação deste padrão é a possibilidade de o programador, por engano, criar mais do que uma instância deste objeto, o que poderá rapidamente causar problemas de memória. Para tentar minimizar esse problema, optou-se por destruir qualquer objeto que seja criado como uma instância da *script* `GameManager` depois de já existir uma criada.

Este código é suficiente para que o padrão *Singleton* seja implementado. Agora, é só uma questão de o usar devidamente. Na *script* `AntInput`, deverá **eliminar** a linha

```
  public GameManager gameManager;
```

e o método para o apanhar do cogumelo deverá ser reescrito do seguinte modo:

```
  void OnTriggerStay(Collider c) {
    if (c.gameObject.tag == "cogumelo" && Input.GetButton("Fire1") &&
        !animator.GetBool("walk"))     {
      Destroy(c.gameObject, 1.5f);
      animator.SetTrigger("take");
      GameManager.instance.PickMushroom();
    }
  }
```

Como se pode ver neste código, o uso de um *Singleton* torna-se simples, bastando aceder à variável que armazena a (única) instância do `GameManager`.

4.3 GESTOR DO JOGO

Na secção anterior foi criada a base para o gestor de jogo onde irá ser adicionada alguma lógica para o seu funcionamento. Como explicado no início deste capítulo, a lógica será bastante simples:

- No início, a formiga tem a energia a 100%. A cada segundo, a energia decresce um valor percentual.
- Existem vários cogumelos em jogo. A formiga terá de os apanhar a todos para terminar o jogo.
- A cada cogumelo apanhado, a energia da formiga sobe 25 valores percentuais, sendo que não é possível ultrapassar a energia máxima de 100%.

Além das variáveis criadas anteriormente, será necessário adicionar uma nova, correspondente à energia da formiga:

```
public class GameManager : MonoBehaviour {
    public static GameManager instance = null;

    public int TotalMushrooms;      // calculado inicialmente
    public int PickedMushrooms = 0; // cogumelos apanhados
    public float Energy = 100;      // energia inicial
```

Por sua vez, no método para apanhar cogumelos, além de se incrementar o número de cogumelos apanhados, incrementa-se a energia. A fim de facilitar a implementação, será usado o método `Clamp`, que permite alterar um valor, garantindo que este não ultrapassa os limites definidos:

```
  public void PickMushroom() {
     PickedMushrooms++;
     Energy = Mathf.Clamp(Energy + 25, 0, 100);
  }
```

Este método recebe um valor (primeiro parâmetro) e um intervalo (segundo e terceiro parâmetros, valores mínimo e máximo). Se o valor indicado estiver dentro do intervalo definido, então, não é alterado. No entanto, se estiver fora do intervalo, será retornado o limite inferior (se o valor for inferior ao intervalo definido) ou o limite superior (se o valor for superior ao intervalo definido).

Para decrementar o valor da energia, será usada a classe estática `Time`, que armazena informação temporal sobre o jogo. Esta classe inclui o campo `Time.deltaTime`, que corresponde ao número de segundos que decorreram desde que o método `Update` foi invocado pela última vez. Por exemplo, se o jogo estiver a executar a uma velocidade de 60 *fps*, o método `Update` é invocado 60 vezes durante um segundo, pelo que, se o rácio com que é invocado o método `Update` for constante, o valor de `Time.deltaTime` será $1/60 = 0.01667$ segundos. Usando este valor, o método `Update` pode ser implementado do seguinte modo:

```
void Update() {
    Energy -= Time.deltaTime;
    if (Energy < 0)
        Debug.Log("Formiga morta!");
}
```

O método decrementa o valor de energia e verifica se a formiga morreu. Neste momento, apenas será impressa, na consola, uma mensagem a indicar que a formiga se encontra morta, usando o método `Debug.Log`. Os métodos da classe `Debug` são bastante úteis durante o desenvolvimento para ajudar na depuração.

4.4 GUI DE JOGO

Implementada a lógica de jogo, torna-se importante dar informação ao jogador, de modo a que este esteja a par do seu progresso durante o jogo. Para isso, será colocado no canto superior esquerdo do ecrã o número de cogumelos já apanhado e na restante margem superior uma barra de energia, que irá sendo preenchida de forma percentual, relativa à energia da formiga. Serão também usadas três cores — verde, laranja e vermelho — de acordo com a energia restante. A Figura 4.4 mostra a interface que será criada nesta secção e que é composta por quatro objetos: a imagem do cogumelo, o número de cogumelos apanhados, a barra de energia (energia atual) e os limites da barra de energia (zona total).

FIGURA 4.4 – Interface durante o jogo

A construção desta interface usará um conjunto de objetos que estão disponíveis no menu `Game Object` → `UI`. Todos os objetos deste menu implicam a existência de um objeto-pai que inclua o componente *Canvas*. De seguida, será explicado o funcionamento do *Canvas* e de que modo este será aplicado na construção da interface com o utilizador. Posteriormente, será introduzido o funcionamento do componente *Rect Transform*, terminando com a construção, passo a passo, da interface do jogo.

4.4.1 *CANVAS*

O componente *Canvas* é responsável pela renderização, na câmara, dos componentes da interface que sejam seus filhos. A ordem pela qual estão ordenados na *Hierarquia* é a ordem usada para a sua renderização. Assim, os objetos que apareçam primeiro serão desenhados primeiro, pelo que os restantes serão renderizados sobre os primeiros.

A Figura 4.5 mostra o componente *Canvas*, bem como os componentes *Rect Transform*, *Canvas Scaler* e *Graphic Raycaster*.

O *Rect Transform* é uma especialização do *Transform*, com funcionalidades específicas para a construção de interfaces com o utilizador. No entanto, como se pode ver na Figura 4.5, neste objeto este componente está inativo, pelo que apenas será explicado com detalhe na secção 4.4.2.

Em relação ao componente *Canvas*, a sua primeira opção permite definir em que moldes a interface será renderizada. Existem três modos possíveis:

- `Screen Space - Overlay` — Neste modo, a interface é desenhada diretamente sobre o ecrã, como se se tratasse de uma folha de acetato colada à câmara onde os diferentes controlos são desenhados. Este é o modo mais simples de controlar e aquele que será usado.
- `Screen Space - Camera` — Este modo é semelhante ao anterior, mas a referida folha de acetato não é colada à câmara, mas colocada a certa distância. Se a câmara for configurada com a opção `Perspective`, então, a interface será desenhada em perspetiva. Isto permite que a interface possa parecer tridimensional, sem na verdade o ser. No entanto, tal como no modo anterior, sempre que a câmara se move, a interface move-se de forma semelhante.
- `World Space` — Se o *Canvas* for configurado para funcionar em `World Space`, aparecerá como um objeto tridimensional típico do Unity, com posição real e a rotação desejada, o que permite que a interface apareça à frente ou por trás de outros objetos de acordo com a sua posição.

Mantendo o modo `Overlay`, existem três campos que podem ser configurados. O primeiro, denominado `Pixel Perfect`, indica se a interface deve ser desenhada com

anti-alising (processo que tenta disfarçar mudanças bruscas nas cores dos píxeis) ou se deve ser perfeita, mantendo com precisão os píxeis a serem desenhados. Por sua vez, o `Sort Order` permite definir a ordem de renderização de diferentes objetos com componentes *Canvas*. É possível escolher em que dispositivo será renderizado este *Canvas*, o que permite que jogos que suportem mais do que um ecrã possam definir em que ecrã a interface deve aparecer. Para o jogo em desenvolvimento, estas opções serão mantidas inalteradas.

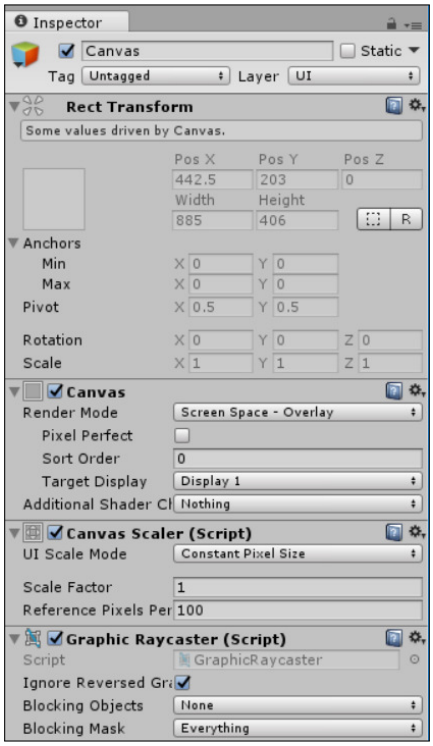


FIGURA 4.5 – Componente *Canvas* e seus componentes auxiliares

A opção `Additional Shader Channels` permite a aplicação de *shaders* específicos apenas à interface. No entanto, esta é uma opção avançada que não será aqui discutida.


Dado o cariz introdutório deste livro, não serão explicados os detalhes dos outros dois tipos de *Canvas*. Porém, o leitor é desafiado a fazer experiências, alterando o valor dessa propriedade e verificando de que modo o Unity se comporta.

O componente *Canvas Scaler* é responsável por controlar de que modo a renderização da interface é afetada pela resolução do ecrã (número de píxeis) e pelo seu tamanho físico. Existem três modos possíveis:

- `Constant Pixel Size` — Os componentes mantêm o seu número, seja qual for o tamanho do ecrã ou a sua resolução.
- `Scale With Screen Size` — Os componentes aumentam e diminuem de tamanho, de acordo com o tamanho físico do ecrã.
- `Constant Physical Size` — Os componentes mantêm o seu tamanho físico, seja qual for o tamanho do ecrã ou a sua resolução.

Dependendo do modo escolhido, as propriedades disponíveis irão variar. No jogo em desenvolvimento, serão mantidas as opções predefinidas.


Mais uma vez, para o contexto deste livro, não se adequa a explicação detalhada do comportamento de cada um destes modos. Sugere-se a consulta da documentação do Unity para compreender melhor de que forma se poderá controlar a aparência da interface em diferentes dispositivos.

O componente *Graphics Raycaster* é responsável por detetar quais os eventos sobre o ecrã que deverão ser intercetados pela interface. Em condições típicas, este componente não precisa de ser configurado e poderá ser mantido na sua configuração inicial.

4.4.2 *RECT TRANSFORM*

O componente *Rect Transform* substitui o componente *Transform* nos elementos de interface gráfica. A forma como se usa não é tão simples e direta, pelo que esta secção se dedica apenas à explicação dos conceitos associados a esse componente.

Dependendo da forma de uso do componente, os campos disponíveis podem mudar. A Figura 4.6 mostra duas imagens deste componente, com diferentes campos, que serão explicados de seguida.

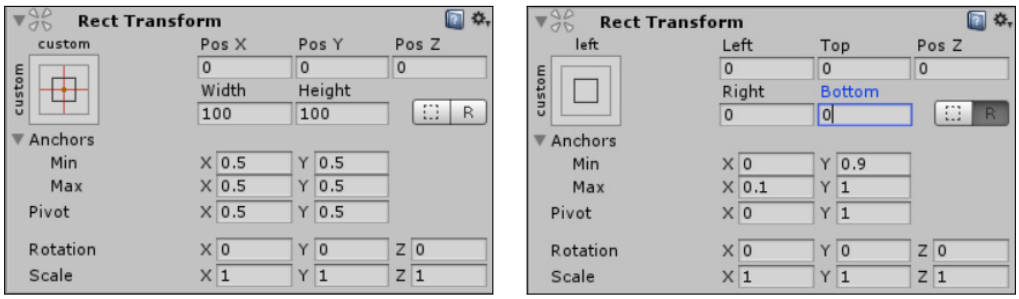


FIGURA 4.6 – Componente *Rect Transform* e campos disponíveis

Neste componente, o principal conceito corresponde à definição de âncora (*anchor*). Uma âncora indica de que forma o objeto que está a ser configurado será colocado dentro do seu elemento-pai. A âncora pode corresponder a um único ponto (se os valores máximo e mínimos de cada eixo forem iguais) ou a uma área. Estes valores são percentuais (entre 0 e 1), sendo que o canto inferior esquerdo corresponde ao ponto $(0, 0)$ e o canto superior direito corresponde ao ponto $(1, 1)$. Com os valores por omissão, apresentados à esquerda na Figura 4.6, é definida uma única âncora, que se situa no centro do seu objeto-pai. É neste ponto que o pivô (*pivot*) do objeto será colocado, de forma percentual, sobre as dimensões do objeto.


Embora a posição dos elementos possa ser definida interativamente, essa abordagem é problemática para jogos que tenham de funcionar em diferentes dispositivos com tamanhos de ecrã e resoluções diferentes. Daí que neste livro se dê especial atenção aos conceitos de âncora e pivô.

De seguida, serão analisados alguns exemplos, de modo a ilustrar melhor como estes dois elementos funcionam em conjunto. Nas figuras seguintes as âncoras são representadas por quatro pequenos triângulos e o pivô do objeto por um círculo negro. O retângulo a tracejado corresponde ao objeto-pai, sobre o qual se está a incluir o objeto desenhado com linha contínua.

A situação apresentada na Figura 4.7 corresponde à posição por omissão: o pivô está no centro do objeto, e as âncoras correspondem a uma única posição, no centro do objeto-pai. Assim, ao colocar o pivô sobre o ponto definido pelas âncoras, o objeto fica centrado.

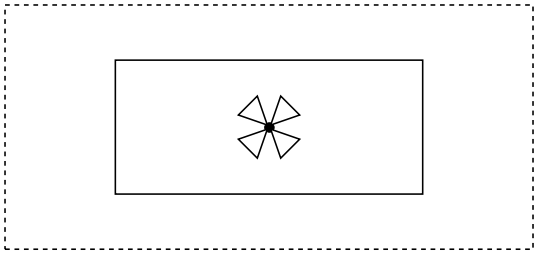


FIGURA 4.7 – Pivô e âncoras centrados

Supondo a alteração do pivô do objeto para a posição $X = 0.25$ e $Y = 0.75$, ou seja, a 25% da margem esquerda e a 75% da margem inferior e se mantivessem as âncoras definidas para o centro do objeto-pai ($Max\,X = Max\,Y = 0.5$), o resultado seria o que se apresenta na Figura 4.8.

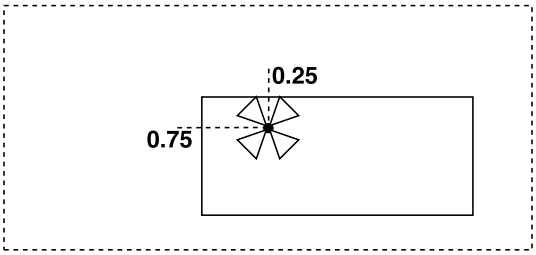


FIGURA 4.8 – Alteração da posição do pivô

Mantendo ainda as âncoras como um único ponto, mas colocando-as no canto superior esquerdo do objeto-pai, com os valores $Max\,X = Min\,X = 0$ e $Max\,Y = Min\,Y = 1$, e definindo o pivô também no canto superior esquerdo, mas do objeto (coordenadas $(0, 1)$), o resultado será o apresentado na Figura 4.9.

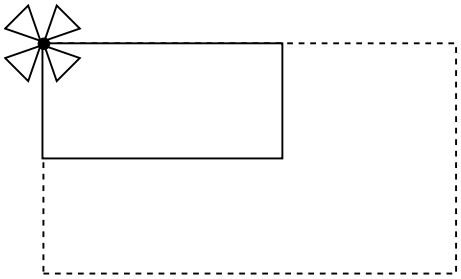


FIGURA 4.9 – Pivô e âncoras ao canto

Repare que, em qualquer um dos três casos até agora apresentados, a largura e a altura do objeto têm de ser definidas de forma independente das âncoras, já que apenas se está a indicar em que posição o pivô do objeto é colocado em relação ao tamanho do seu objeto-pai. Estes casos correspondem ao inspetor apresentado à esquerda na Figura 4.6. Os campos `Width` e `Height` definem a largura e altura do objeto em píxeis. Por sua vez, os campos `Pos X` e `Pos Y` definem uma margem (*offset*) do objeto em relação ao pivô, também em píxeis. Por exemplo, na última situação apresentada, se estes dois valores fossem definidos como $Pos\,X = 5$ e $Pos\,Y = -5$, então, seria criada uma margem de 5 píxeis nas margens esquerda e superior (o pivô seria deslocado 5 píxeis para a direita e 5 píxeis para baixo).

Considere, agora, que as âncoras não foram definidas como um único ponto, mas como quatro cantos de um retângulo. Neste caso, para o eixo x, $Min\,X = 0.25$ (25%), $Max\,X = 0.75$ (75%) e para o eixo y, $Min\,Y = 0$ e $Max\,Y = 1$. Mantendo-se o pivô no centro do objeto $(0.5, 0.5)$, este irá expandir-se, ocupando a área vertical total do objeto-pai (já que o mínimo para o eixo y corresponde a 0 e o máximo a 1) e irá ocupar metade da área horizontal, deixando margens de 25% de cada lado (Figura 4.10).

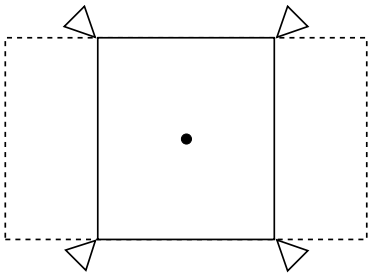


FIGURA 4.10 – Pivô centrado e âncoras percentuais

Nesta situação, o *Inspetor* irá mudar, ficando semelhante ao da direita na Figura 4.6. Uma vez que a largura e a altura estão a ser definidas pelas âncoras, o *Inspetor* irá apresentar os campos `Left`, `Top`, `Right` e `Bottom`, que permitem definir margens, em píxeis.

É importante realçar que o comportamento habitual, ao mudar os valores das âncoras ou do pivô, o Unity tenta adaptar os valores, em píxeis, para que o objeto não mude de sítio. Este comportamento pode ser bastante aborrecido, quando se pretende definir a interface usando apenas âncoras. Para facilitar este processo, poderá ser ativado o botão `R` (como demonstrado no *Inspetor* à direita na Figura 4.6).

Este último exemplo mostra uma combinação. Os valores máximos e mínimos das âncoras do eixo y foram definidas com o valor 0.5: $Max\,Y = Min\,Y = 0.5$. Já no eixo x, as âncoras foram definidas com $Min\,X = 0.25$ e $Max\,X = 0.75$. Neste caso, a largura do objeto é completamente definida pelas âncoras, portanto, o *Inspetor* irá mostrar os campos `Left` e `Right`. Já a altura do objeto não está definida, pelo que o *Inspetor* irá apresentar os campos `Pos Y` e `Height`, para permitir definir um *offset* em relação ao pivô e, ao mesmo tempo, definir a altura desejada para o objeto (Figura 4.11).

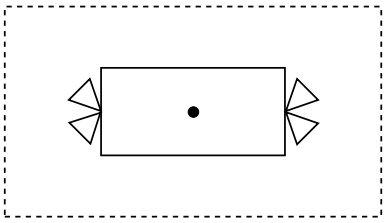


FIGURA 4.11 – Âncoras mistas: percentuais e centradas verticalmente

Finalmente, o *Inspetor* também inclui a posição no eixo z, especialmente útil quando o *Canvas* é definido em `World Space`. Em baixo, é ainda possível definir a rotação e a escala. Estas duas propriedades são herdadas do componente *Transform*.

De seguida, as técnicas aqui apresentadas serão postas em prática para a construção da interface de jogo desejada.

4.4.3 CRIAÇÃO DA GUI

A Figura 4.12 mostra a interface desejada, com algumas zonas demarcadas, que vão facilitar o posicionamento dos diversos componentes da interface.

FIGURA 4.12 – Interface de jogo e respetivas zonas de construção

O 1.º passo será a inclusão da imagem do cogumelo, usando um objeto com o componente *Image*, que pode ser criado acedendo ao menu `Game Object` → `UI` → `Image`. Ao adicionar este objeto, será automaticamente adicionado também um objeto com o componente *Canvas* denominado `Canvas`.

O objeto criado contém os componentes *Rect Transform*, *Canvas Renderer* e *Image*. O primeiro já foi suficientemente explicado na secção 4.4.2. O *Canvas Renderer* é um componente não configurável, responsável por renderizar os componentes de interface no componente *Canvas*. Finalmente, o componente *Image* será o responsável por associar uma imagem a este objeto.


A imagem do cogumelo está disponível no sítio do livro (`www.fca.pt`), mas qualquer imagem em formato `.png` e com fundo transparente poderá ser usada.

Será necessário importar a imagem do cogumelo, o que pode ser feito arrastando-a para a pasta do projeto. No entanto, é necessário configurar de que forma esta imagem será importada e interpretada pelo Unity. O processo é semelhante ao da configuração da

importação de uma animação, como foi discutido no Capítulo 3. Basicamente, ao selecionar a imagem, no separador *Projeto*, surgirá no *Inspetor* as propriedades de importação, tal como se mostra na Figura 4.13.

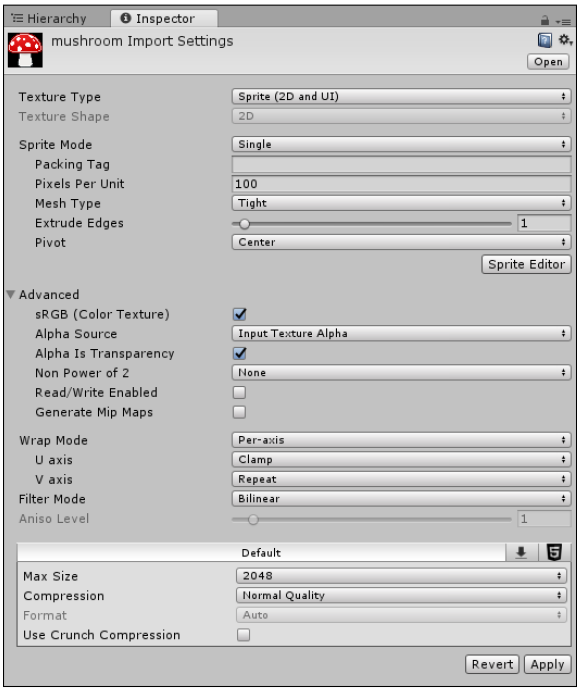


FIGURA 4.13 – Opções de importação de uma imagem

Nas configurações da imagem, é importante alterar o tipo de textura, de modo a que possa ser usado na interface (alterando o tipo da textura — `Texture Type` — para `Sprite (2D and UI)`. Além disso, e dado que a imagem tem transparências, é essencial que a opção `Alpha Source` indique a importação da transparência (`Import Texture Alpha`, e que a opção `Alpha Is Transparent` esteja ativa. Posteriormente, é necessário usar o botão `Apply`. Realizadas estas alterações é, então, possível arrastar a imagem do separador *Projeto* para o campo `Source Image` do objeto `Image` criado.

Depois de configurada a imagem, irá reparar que ela não aparece no ecrã. Na verdade, o componente *Canvas* está configurado, por omissão, para funcionar como `Screen Space - Overlay`, pelo que, para se poder ver, corretamente, qual o aspeto da interface, deverá selecionar-se o separador *Jogo*. Em princípio, o cogumelo já aparecerá no centro do ecrã.[17]

[17] Pode acontecer que ainda não apareça, por ter o *Rect Transform* configurado com coordenadas que não estejam entre os limites do ecrã. Se for o caso, altere os campos `Pos X` e `Pos Y` para 0.

Colocar o cogumelo na posição desejada é uma questão de alterar valores no *Rect Transform*. Como se vê na Figura 4.12, pretende-se que o cogumelo fique no canto superior esquerdo, numa zona que corresponda a 10% da largura e a 10% da altura do ecrã. Para isso, bastará definir as âncoras como $Min\,X = 0$, $Max\,X = 0.1$, $Max\,Y = 1$ e $Min\,Y = 0.9$. Tendo em atenção que foi usado o botão `R` como descrito anteriormente, ou os valores de `Left`, `Bottom`, `Top` e `Right` foram colocados com o valor zero, então, o cogumelo já deverá aparecer no canto do ecrã (embora deformado).

Para que a imagem seja apresentada mantendo sempre o mesmo rácio entre largura e altura e não apareça deformada, deverá ser ativada a opção `Preserve Aspect` no componente *Image*.

O alinhamento do cogumelo ao lado direito da zona definida pode ser obtido alterando o pivô para o ponto com coordenadas $(1.0, 1.0)$. Para conferir uma melhor aparência à interface, será simpático separar o cogumelo alguns píxeis da margem superior do ecrã. A Figura 4.14 mostra o *Inspetor* com as configurações finais.

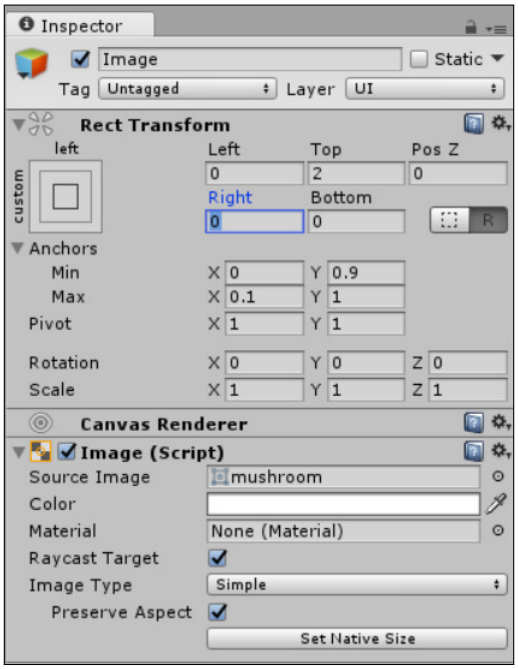


FIGURA 4.14 – *Inspetor* com as configurações finais dos componentes do cogumelo

De seguida será adicionado um novo objeto, do tipo *Text*, para indicar o número de cogumelos já apanhados. Será também criado como filho do `Canvas` já existente, pelo que para o criar poderá clicar com o botão direito do rato sobre o objeto `Canvas` e escolher a opção `UI` → `Text`. Tal como aconteceu com o objeto criado para apresentar o cogumelo,

também este objeto tem um *Rect Transform* e um *Canvas Renderer*. Além disso, tem o componente *Text*, que permite a adição de texto na interface. Grande parte das propriedades deste componente são fáceis de perceber, já que correspondem a operações típicas realizadas diariamente em processadores de texto.

No sentido de se perceber melhor a posição do texto, ao alterar o seu *Rect Transform*, será inicialmente configurado o componente com algum texto de teste (por exemplo, o número 2), e a configuração inicial do tamanho, cor e tipo de letra, tal como demonstrado na Figura 4.15.

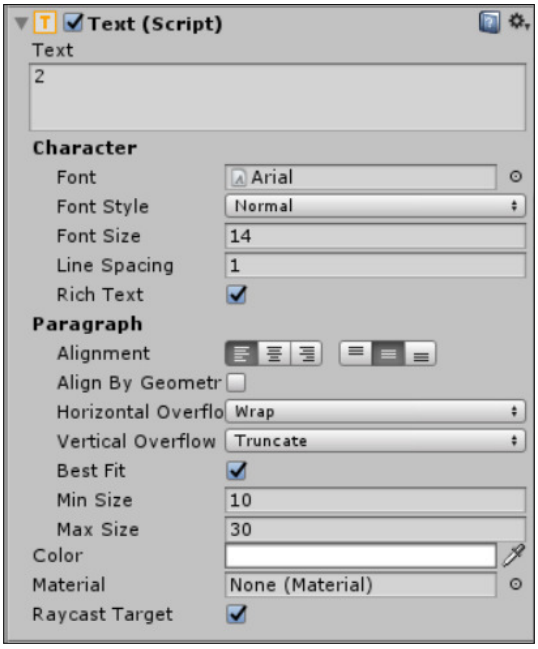


FIGURA 4.15 – *Inspetor* com configurações do componente *Text*

O campo para a escrita do texto permite que se coloque algum texto inicial predefinido, embora seja possível alterá-lo depois via programação. Será essa a abordagem, pelo que o valor 2 lá colocado servirá apenas de exemplo de teste.

Por omissão o Unity só suporta o tipo de letra *Arial*, mas podem ser adicionados outros bastando arrastar o ficheiro com o tipo de letra[18] para o separador *Projeto*.

Segue-se o estilo do tipo de letra, bem como o seu tamanho, o espaçamento entre linhas, e se o texto deverá ser processado como *rich text*, ou seja, em que as etiquetas HTML para itálico e negrito (`<i>` e `<b>`, respetivamente) são interpretadas e o texto devidamente formatado.

[18] Ficheiro do tipo *True Type Font* e extensão `.ttf`.

Mais abaixo, a configuração do parágrafo permite definir o seu alinhamento horizontal e vertical. Uma vez que se pretende que o número fique colado do lado direito do cogumelo, deve ser mantido o alinhamento à esquerda. Já em relação ao alinhamento vertical, sugere-se que escolha a opção correspondente a centrar o texto verticalmente. É ainda possível configurar o que deve acontecer quando o texto é demasiado extenso para a área definida.

Será também usada a opção `BestFit`, que permite que o tamanho do texto mude automaticamente de acordo com a área disponível. No entanto, e para que o texto não fique demasiado grande ou demasiado pequeno, é possível definir os tamanhos mínimos e máximos permitidos (10 e 30 respetivamente).

É igualmente possível configurar a cor e o material, se assim desejar, e se este objeto deverá intercetar tentativas de *raycasting*[19].

Passando ao posicionamento deste texto, a área a usar, de acordo com o diagrama da Figura 4.12, corresponde a 10% da largura, afastada 10% da margem esquerda (área definida para o cogumelo), e deve ocupar 10% da altura, devidamente encostada ao topo. Assim, as âncoras devem ser definidas como $Min\,X = 0.1$, $Max\,X = 0.2$, $Min\,Y = 0.9$ e $Max\,Y = 1$. Também se deverá adicionar alguma margem esquerda, de 3 ou 4 píxeis, de modo a que o texto não fique demasiado próximo do cogumelo.

Para facilitar a leitura, será ainda adicionado o componente *Outline* a este mesmo objeto. Este componente permite adicionar uma linha, noutra cor à volta do texto. A sua configuração é extremamente simples, sendo apenas possível definir a cor, a posição relativa ao texto e se a cor da borda deve ser multiplicada pela cor de fundo.

A construção da barra de energia será um pouco mais difícil, já que dependerá de dois objetos (duas imagens) em que uma será filha da outra. Para começar, será criado um novo objeto do tipo *Image* diretamente dentro do objeto `Canvas`. Para facilitar a referência a esta imagem, será usado o nome `Energy`.

Antes de colocar a barra na posição desejada, será escolhida a imagem, sendo que será usada uma imagem, existente num dos pacotes já importados, para os botões. A forma mais simples de a incluir é clicar no pequeno círculo à direita do campo `Source Image` e escolher a imagem `InputFieldBackground`, como demonstrado na Figura 4.16.

De acordo com o diagrama da Figura 4.12, o *Rect Transform* deve ser configurado para ter as âncoras com os valores $Min\,X = 0.2$, $Max\,X = 0.95$, $Min\,Y = 0.925$ e $Max\,Y = 0.975$.

Esta imagem irá corresponder à zona para a barra de energia e não à própria barra. Para lhe dar um efeito interessante, será alterada a cor da imagem, adicionando-lhe alguma transparência (componente `A` no seletor de cor, como se verifica na Figura 4.17).

[19] O conceito de *raycasting* será explicado com maior detalhe na secção 5.1.3.

FIGURA 4.16 – Seleção da imagem `InputFieldBackground`

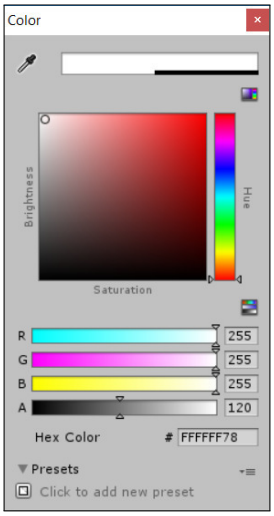


FIGURA 4.17 – Seletor de cor e alteração da transparência (*alpha*)

A barra de energia corresponderá a outro objeto, com o componente *Image*, mas criado como filho da imagem `Energy` e não diretamente dentro do `Canvas`. A imagem propriamente dita será a mesma aplicada na imagem anterior (`InputFieldBackground`). A cor será, para já, verde, apenas para testes. Finalmente, o *Rect Transform* deverá estender-se por toda a área, mas tendo em conta que a âncora máxima do eixo horizontal será usada para representar a percentagem de energia atual. Assim, e supondo que a formiga está com a energia a 75%, este objeto deverá ter as âncoras definidas como $Min\,X = 0$, $Max\,X = 0.75$, $Min\,Y = 0$ e $Max\,Y = 1$.

Falta apenas ligar o estado do jogo com a interface, fazendo-a atualizar-se à medida que o jogo se vai desenrolando.

4.4.4 ATUALIZAÇÃO DA GUI

Preparada a interface, é necessário que esta se atualize de acordo com o estado do jogo. Para o apanhar dos cogumelos, será adicionada uma *script* de nome `SyncMushrooms` ao objeto responsável pela apresentação do número de cogumelos apanhados (portanto, objeto `Text` dentro do `Canvas`, criado na secção 4.4.3). A *script* será bastante simples. Terá uma variável privada, do tipo *Text*, que irá armazenar uma referência ao componente *Text* associado ao objeto. No método `Start`, esta variável será inicializada; no método `Update`, o número de cogumelos será atualizado no campo `Text`[20]:

```
using UnityEngine;
using UnityEngine.UI;

public class SyncMushrooms : MonoBehaviour {
  Text mushroomCount;

  void Start () {
    mushroomCount = GetComponent<Text>();
  }

  void Update () {
    mushroomCount.text =
          GameManager.instance.PickedMushrooms.ToString() + "/" +
          GameManager.instance.TotalMushrooms.ToString();
  }
}
```

Implementado este código, já será possível testar o jogo e tentar apanhar um cogumelo. Se o jogador for bem comportado e só apanhar o cogumelo uma vez, o código irá funcionar perfeitamente, mas se o jogador carregar duas vezes seguidas na tecla para apanhar o cogumelo, o contador irá contabilizar dois cogumelos, o que não é aceitável. Isto acontece porque o cogumelo não é destruído imediatamente (para sincronizar o seu desaparecimento com a animação da formiga). A forma mais simples de resolver este problema é manter a destruição do objeto em diferido, mas destruir o componente *Collider*, responsável por detetar as colisões, logo que o jogador carregue na tecla para apanhar o cogumelo. Para solucionar esta situação, na *script* `AntInput` deve ser alterado o método `OnTriggerStay`.

[20] Note a adição da importação da biblioteca `UnityEngine.UI`, que inclui as definições dos componentes usados na construção da interface.

```
void OnTriggerStay(Collider c) {
  if (c.gameObject.tag == "cogumelo" &&
      Input.GetButtonDown("Fire1")   &&
      !animator.GetBool("walk"))
  {
     Destroy(c.gameObject.GetComponent<BoxCollider>());
     Destroy(c.gameObject, 1.5f);
     animator.SetTrigger("take");
     GameManager.instance.PickMushroom();
  }
}
```

Tenha em atenção que o método `Destroy` pode ser utilizado para destruir um objeto de jogo (`GameObject`) mas também para remover um componente: primeiro, é removido o componente *Box Collider*, e só depois se pede a destruição diferida do objeto.

Embora esta solução já permita que o cogumelo não seja contabilizado duas vezes, o incremento na interface não está sincronizado com o desaparecimento do cogumelo, o que pode tornar-se estranho. O Unity tem dois métodos que permitem a invocação diferida de métodos definidos pelo utilizador. Desse modo, a última linha do código anterior poderá ser substituída por

```
Invoke("DeferredPick", 1.5f);
```

e deverá ser criado um novo método, nesta mesma *script*, que atualize o estado do jogo:

```
void DeferredPick() {
  GameManager.instance.PickMushroom();
}
```

O método `Invoke` executa o método com o nome indicado assim que se passarem tantos segundos quantos os solicitados no segundo argumento. Note-se que o método invocado deverá ser `void`, não poderá receber quaisquer parâmetros e tem de estar definido no mesmo componente a partir do qual é invocado.

 Existe também o método `InvokeRepeating` que invoca o método indicado de forma repetida. Além do nome do método a invocar, este método recebe dois valores: o tempo necessário para que o método seja invocado pela primeira vez e o tempo de espera entre invocações sucessivas.

De seguida, será implementado o funcionamento da barra de energia. Além da percentagem de preenchimento, pretende-se que a sua cor mude, entre verde, laranja e vermelho, de acordo com o valor de energia restante.

Tal como para o número de cogumelos apanhados, para a barra de energia será criada uma nova *script*, que será associada à imagem que está dentro do objeto `Energy` criado anteriormente (a imagem da barra a verde). Esta *script* denominar-se-á `SyncEnergy`.

Serão definidas variáveis privadas para guardar referência ao `RectTransform` do objeto, a referência à imagem renderizada e às três cores que serão usadas:

```
public class SyncEnergy : MonoBehaviour {
  RectTransform energyRectTransform;
  Image energyImage;
  Color green, red, orange;
```

O método `Start` será responsável por obter a referência ao `RectTransform` e garantir que este está a ocupar toda a área da barra (energia a 100%), para definir as três cores, e para obter a referência à imagem e atribuir-lhe a cor verde:

```
  void Start () {
    energyRectTransform = GetComponent<RectTransform>();
    energyRectTransform.anchorMax = new Vector2(1, 1);

    green  = new Color(  18f / 255f, 153f / 255f, 20f / 255f);
    orange = new Color( 249f / 255f, 180f / 255f, 17f / 255f);
    red    = new Color( 211f / 255f,  19f / 255f,  2f / 255f);

    energyImage = GetComponent<Image>();
    energyImage.color = green;
  }
```

Para definir as âncoras de uma variável do tipo `RectTransform` existem dois campos disponíveis: `anchorMin` e `anchorMax`. São ambos do tipo `Vector2`, pelo que o primeiro campo guarda os valores $(Min\,X, Min\,Y)$ e o segundo $(Max\,X, Max\,Y)$. Assim, é o primeiro valor da variável `anchorMax` que irá corresponder à percentagem de energia da formiga. Como não é possível atribuir apenas esse valor, são atribuídos os valores máximos para ambos os eixos.

Para a criação das cores, é usado o construtor da classe `Color`, que recebe três valores reais, entre 0 e 1, e que correspondem às quantidades das cores vermelha, verde e azul[21] que a cor final deve ter. Como a maior parte das ferramentas retorna um valor entre 0 e 255, optou-se por indicar diretamente a fórmula que calcula a percentagem desejada.[22]

Nas últimas linhas, obtém-se a referência para o componente `Image` e altera-se a cor-base com que a imagem é desenhada.

[21] Que correspondem ao modelo RGB: *red*, *green* e *blue* amplamente usado na *Web*.

[22] Note que este tipo de operações é realizado em tempo de compilação, pelo que não torna a execução do jogo mais lenta.

O método `Update` limita-se a realizar operações semelhantes para alterar a percentagem de energia e a cor, de acordo com o valor atual de energia da formiga:

```
void Update () {
  float currentEnergy = GameManager.instance.Energy;

  energyRectTransform.anchorMax = new Vector2(currentEnergy / 100f, 1);

  if (currentEnergy > 40f)
    energyImage.color = green;
  else if (currentEnergy > 10f)
    energyImage.color = orange;
  else
    energyImage.color = red;
}
```

Terminado este capítulo, a formiga deverá ser capaz de apanhar cogumelos; a interface deverá indicar o estado de jogo — a energia atual da formiga e o número de cogumelos apanhados; e também já deverá surgir uma mensagem a indicar a morte da formiga, quando se esgota o nível de energia.

5 INIMIGOS

Neste capítulo são implementados dois tipos de inimigos: uma planta estranha, que não sai do sítio, mas atira ervilhas venenosas, e uma aranha, que passeia pela ilha e, quando deteta a formiga, persegue-a e ataca-a (Figura 5.1).

FIGURA 5.1 – Vista da cena de jogo com a formiga e a aranha

5.1 PLANTA VENENOSA

O primeiro inimigo que será construído corresponde a uma planta que cospe ervilhas. Esta é uma personagem conhecida de outros jogos, e que será designada por `PeaShooter`.

No sítio do livro (`www.fca.pt`) estão disponíveis, na pasta `PeaShooter`, o modelo e a textura que serão usados de seguida. Salienta-se que a autoria deste modelo pertence a Kirom Nur Sholikin.

O 1.º passo será criar um *prefab* que corresponda ao `PeaShooter`. Para isso, e uma vez que serão necessários vários ajustes ao modelo, sugere-se que seja criada uma nova

cena, apenas temporária. Para isso é usada a opção `File` → `New Scene`. Note que, ao criar uma nova cena, são criados automaticamente os objetos para a câmara e para a fonte de luz. Também é importante realçar que, embora se possam criar diversas cenas, a pasta de projeto é partilhada entre elas, pelo que é boa prática mantê-la arrumada, de modo a que seja sempre fácil encontrar os recursos pretendidos.

5.1.1 HIERARQUIA DE OBJETOS

Criada a nova cena, segue-se a importação do modelo e da textura da planta. Para tal, e para manter o projeto organizado, sugere-se que se arraste a pasta `PeaShooter`, dos recursos disponibilizados, para a pasta `Assets`. Depois de importada, será automaticamente criada uma pasta `Materials`, dentro da pasta `PeaShooter`. No entanto, o Unity não associa automaticamente a textura que foi importada ao material criado, pelo que deve ser selecionada a textura de nome `ea`, dentro da pasta de materiais, e, no respetivo *Inspetor*, alterar a imagem associada ao campo `Albedo` (Figura 5.2). A forma mais simples de o fazer é usar o pequeno círculo à esquerda da etiqueta `Albedo`, e selecionar a textura `pea` (note que pode procurar usando a barra superior da janela de seleção de texturas).

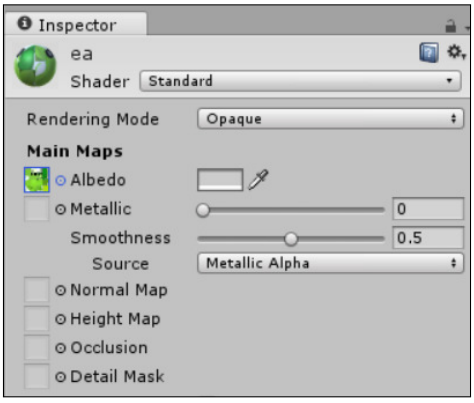


FIGURA 5.2 – Definição do campo `Albedo` no material `ea`

Antes de adicionar a planta à cena, será ainda necessário ajustar a sua escala. É que os modelos importados de outras ferramentas nem sempre partilham uma escala adequada à do Unity. Para alterar a escala usada na importação, deverá ser selecionado o modelo `peashooter` no *Projeto* e, no *Inspetor* respetivo, alterar o campo `Scale Factor` para o valor 10, clicando depois no botão `Apply`, como demonstrado na Figura 5.3.

Alterada a escala do modelo, este pode ser colocado na cena, arrastando-o. Infelizmente, ao arrastar o modelo, o Unity colocará uma escala de 100 unidades, que deverá ser alterada para 1 em todos os eixos. A rotação, por sua vez, deve ser mantida, já que o modelo original não se encontra na vertical.

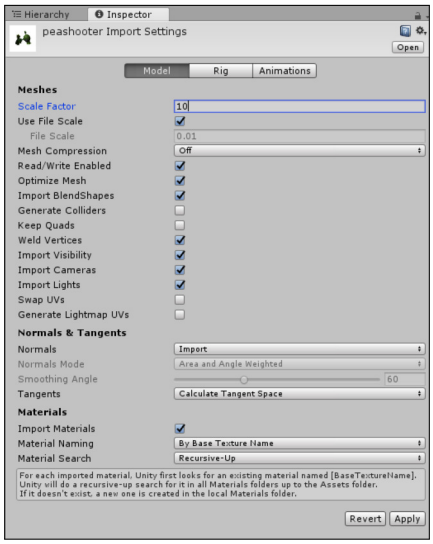


FIGURA 5.3 – Definição do fator de escala na importação de um modelo

Para que mais tarde seja possível criar facilmente ervilhas no sítio adequado (boca da planta), será criado um objeto vazio com coordenadas relativas ao modelo. Para tal, devem ser seguidos os seguintes passos:

1) Colocar o modelo na origem da cena — posição $(0, 0, 0)$.

2) Criar um novo objeto, vazio, denominado `Plant`, e que também deverá ser colocado na posição $(0, 0, 0)$. Neste momento os dois objetos estão alinhados, pelo que poderá ser arrastado o `PeaShooter`, na *Hierarquia*, para dentro do objeto `Plant`, criando assim uma hierarquia de objetos.

3) Para descobrir a posição em que as ervilhas devem surgir, será criado e posicionado um novo objeto, com uma esfera, dentro do objeto `Plant`. A forma mais simples de o fazer é clicar com o botão direito do rato sobre o objeto `Plant`, na *Hierarquia*, e escolher a opção `3D Object` → `Sphere`. A esfera criada é demasiado grande para o fim a que se destina, pelo que a sua escala deve ser alterada para, por exemplo, um valor de 0.2 em todos os eixos. Arrastando a esfera, deverá colocá-la junto à boca da planta (Figura 5.4), de modo alinhado. Será essa a posição em que as ervilhas serão criadas antes de serem disparadas.

4) Decidida a posição da ervilha, e porque ela não deverá aparecer logo na cena, serão removidos os componentes *Sphere*, *Sphere Collider* e *Mesh Renderer* da esfera (ficando apenas o componente *Transform*), e alterado o nome do objeto para *SpawnPoint*.

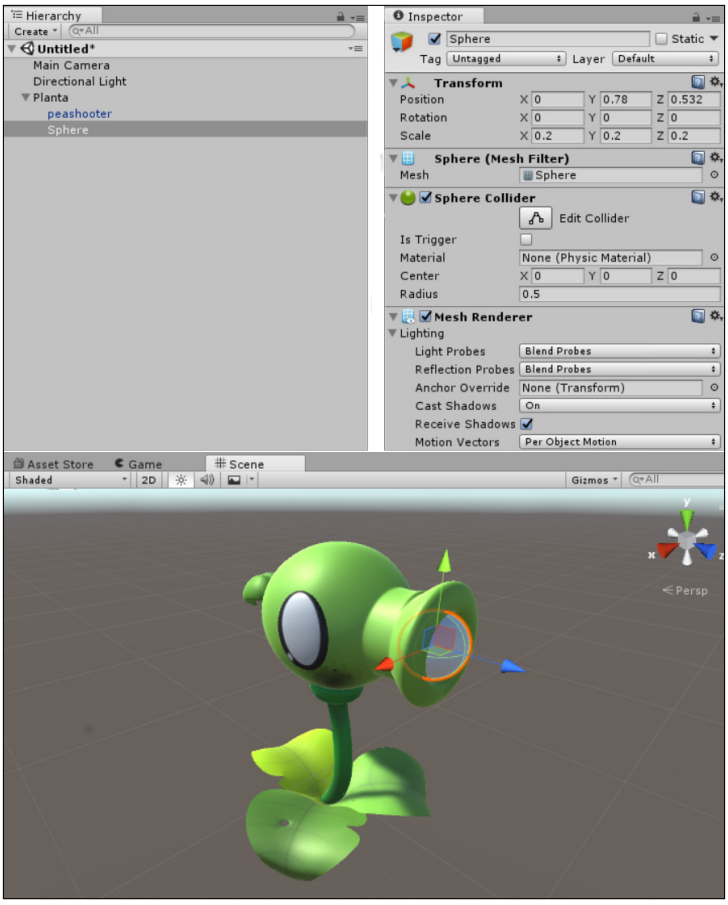


FIGURA 5.4 – *Hierarquia* de objetos e propriedades da esfera

Para testar o disparo das ervilhas, será criada uma *script* que irá fazer crescer uma ervilha, disparando-a de seguida. A *script* será criada no objeto `Plant`, de nome `ShootPea`. Uma vez que é importante manter o projeto organizado, a *script* pode ser arrastada para a pasta `PeaShooter`.

5.1.2 CORROTINAS

No sentido de tornar o jogo mais apelativo do ponto de vista visual, em vez de ser criada automaticamente uma ervilha com o tamanho desejado e disparada logo, será criada uma pequena animação. A ervilha aparecerá muito pequena e crescerá progressivamente. Quando chegar ao tamannho desejado, sairá disparada. Para este tipo de animação é útil o uso de corrotinas.

A *script* define duas variáveis: uma privada, que será uma referência ao ponto em que as ervilhas devem ser criadas, e uma pública, com a força com que a ervilha deverá ser disparada:

```
public class ShootPea : MonoBehaviour
{
  private Transform SpawnPoint;
  public float ShootForce = 500f;
```

Uma corrotina é um método especial, que retorna um enumerador (de tipo `IEnumerator`) e que realiza pequenas operações, retornando um estado intermédio. O Unity terá a responsabilidade de o voltar a invocar, para que ele realize uma nova operação e volte a retornar. Assim, o método irá aumentar, a cada iteração, o tamanho da ervilha e retornar o controlo para o Unity. Quando o método tiver sido invocado o número de vezes suficiente para a ervilha ter atingido o tamanho desejado, o método irá prosseguir com o seu funcionamento normal. A corrotina será implementada do seguinte modo:

```
  IEnumerator Shoot() {
    GameObject pea = GameObject.CreatePrimitive(PrimitiveType.Sphere);
    pea.transform.position = SpawnPoint.position;
    pea.transform.localScale = Vector3.one * 0.01f;

    Material peaMaterial = pea.GetComponent<MeshRenderer>().material;
    peaMaterial.color = new Color(.4f,.6f,.4f);

    while (pea.transform.localScale.x < 0.2f) {
       pea.transform.localScale += Vector3.one * 0.1f;
       yield return null;
    }

    Rigidbody rb = pea.AddComponent<Rigidbody>();
    rb.AddForce(transform.forward * ShootForce);
  }
```

À parte da definição da corrotina com o tipo necessário, como já foi referido, o método começa por criar um novo objeto. Neste caso, é usado o método `CreatePrimitive`, que permite a criação de primitivas tridimensionais do Unity (as mesmas disponíveis no menu `GameObject` → `3D Object`. O único argumento deste método é o tipo de objeto a ser criado, sendo que o método retorna uma referência ao objeto recém-criado.

As duas linhas seguintes definem a cor da esfera. O método `GetComponent` obtém o componente `MeshRenderer` através do qual se acede ao campo `material`, que contém a referência ao material do objeto. Posteriormente, é alterada a cor desse material modificando o valor do campo `color`.

Definida a cor, indica-se a posição onde a ervilha aparecerá (que corresponde à posição determinada anteriormente e que será em breve armazenada na variável `SpawnPoint`) e o seu tamanho inicial: $(0.01, 0.01, 0.01)$. De realçar que a escala de um objeto é associada ao campo `localScale`.


Em várias situações, a criação de vetores pode ser simplificada usando o acessor `one` da classe `Vector3`, que retorna um vetor com uma unidade em cada um dos três eixos $(1, 1, 1)$ e que pode ser facilmente multiplicado por outro valor, para definir um vetor com qualquer outro valor em cada um dos eixos.

O ciclo que se segue é o coração da corrotina, que irá iterar até que o tamanho do objeto atinja o valor desejado. A cada iteração, o tamanho da ervilha é aumentado em 0.01 em cada eixo, e é feito um retorno especial, usando a palavra reservada `yield`. Este retorno indica ao gestor de corrotinas que uma iteração terminou. Posteriormente, este mesmo gestor irá de novo invocar a corrotina e a execução iniciará neste mesmo ponto, pelo que o ciclo continuará. O tamanho da ervilha será de novo incrementado e o controlo retornado ao gestor de corrotinas. Este processo terminará quando o tamanho da ervilha for o desejado. Findo este ciclo, a execução segue com o código restante.

Depois do ciclo de incremento do tamanho da ervilha, é adicionado o componente `Rigidbody` à ervilha, sendo que também lhe é aplicada uma força através do método `AddForce`, que aplica a força definida ao objeto. Não esquecer de, no *Inspetor*, definir a força com um valor elevado como 500.

Para invocar a corrotina (e definir o ponto de origem das ervilhas), será usado o seguinte código:

```
void Start () {
  SpawnPoint = transform.Find("SpawnPoint");
  StartCoroutine(Shoot());
}
```

O método `Find`, quando invocado sobre um objeto do tipo `Transform`, procura um objeto com o nome indicado que seja seu filho, o que permite que o processo seja mais eficiente do que quando se usa o método equivalente da classe `GameObject`, que iria procurar na cena completa objetos com esse nome.

O método `StartCoroutine` recebe como parâmetro a invocação de um método que tenha sido definido como uma corrotina (retornando o tipo `IEnumerable`).

Embora se vá fazer mais alterações à *script*, está na altura de criar o *prefab*. A forma mais simples é arrastar o objeto `Plant` da *Hierarquia*, e colocá-lo na pasta `PeaShooter`. Neste momento, a cena criada no início deste capítulo pode ser descartada.

5.1.3 *RAYCASTING*

Ao contrário do que foi implementado inicialmente, a planta não irá disparar ervilhas a todo o momento, mas apenas quando a formiga estiver suficientemente perto. Espera-se que a planta tenha uma área de deteção, quase como um sensor de movimento, à espera de que a formiga apareça. Nessa altura, irá rodar sobre si mesma na sua direção e disparar uma ervilha.

Para a deteção de movimento, será usada uma técnica denominada *raycasting* que pode ser vista como o uso de um *laser*, de determinado comprimento, que deteta se algum movimento o interseta. A Figura 5.5 mostra uma configuração possível destes raios, em que a planta verifica a existência de movimento em cinco direções predefinidas.

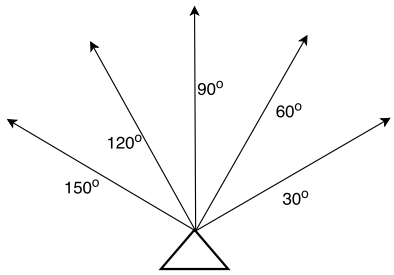


FIGURA 5.5 – Deteção de movimento usando *raycasting*

Embora estes raios não sejam realmente visíveis durante o jogo, é possível solicitar ao Unity que os desenhe, facilitando assim a sua validação. Assim, antes de se implementar a deteção de movimento, será adicionado código à *script* `ShootPea` para desenhar os raios:

```
void Update() {
  for (int i = 0; i < 5; i++) {
    float radians = Mathf.Deg2Rad * (30f * i + 30f);
    Vector3 direction = new Vector3(Mathf.Cos(radians), 0,
                            Mathf.Sin(radians));
    direction = transform.TransformDirection(direction);
    Debug.DrawRay(transform.position + Vector3.up,
                  direction * 5, Color.red);
  }
}
```

É realizado um ciclo para os cinco raios. Para cada um, é calculada a sua rotação em graus: 30°, 60°, …, 150°. Uma vez que as funções trigonométricas funcionam em radianos e não em graus, a rotação é multiplicada pela constante `Deg2Rad`, que corresponde a 0.0174, e permite facilmente a conversão entre graus e radianos.

Dada a rotação desejada, é possível calcular um vetor unitário com essa rotação usando as funções cosseno e seno. Infelizmente, a rotação obtida é correspondente à rotação-base do mundo (de acordo com o sistema de eixos apresentado no canto superior direito da cena), e não à rotação da planta, pelo que é necessário usar o método `TransformDirection` para converter um vetor que representa uma direção no mundo global, num vetor representando a mesma direção, em relação ao sistema de coordenadas local ao objeto. Obtido o vetor unitário com a direção desejada e comprimento 1, é possível usar o método `DrawRay` que, dada uma origem o, e uma deslocação Δ, desenha uma linha entre os pontos o e $o+\Delta$. Para a origem, foi somada 1 unidade virtual (`Vector3.up`) para que o raio calculado não fique junto ao solo, mas a uma unidade de altura. O vetor unitário foi multiplicado pelo valor 5, de modo a que o raio tenha um comprimento de 5 unidades. A Figura 5.6 mostra o resultado, depois de colocado o *prefab* da planta na cena.

Figura 5.6 – *Gizmos* a representar os *raycasts*


Estes raios são designados por *Gizmos* e desenhados, por omissão, apenas na *Cena*, embora seja possível ativar o seu desenho também no separador *Jogo* usando o botão `Gizmos`, disponível na barra de botões desse mesmo separador.

Agora que os raios foram corretamente definidos e validados, o seu desenho deixa de ser importante, passando a ser relevante a deteção de algum objeto que intersete com algum destes raios. Esta deteção é realizada usando o método `Raycast`. Existem várias assinaturas para este método, mas a que será usada receberá três parâmetros: um objeto do tipo `Ray`, que corresponde ao raio a ser emitido; um parâmetro de saída do tipo `RaycastHit`, que armazena informação sobre a possível colisão entre o raio e determinado objeto; e, finalmente, a distância ou comprimento do raio. O método retorna um valor verdadeiro caso haja alguma colisão:

```
  void Update() {
    for (int i = 0; i < 5; i++) {
      float radians = Mathf.Deg2Rad * (30f * i + 30f);
      Vector3 direction = new Vector3(Mathf.Cos(radians), 0,
                              Mathf.Sin(radians));
      direction = transform.TransformDirection(direction);
      Ray ray = new Ray(transform.position + Vector3.up, direction);
      RaycastHit hit;
      if (Physics.Raycast(ray, out hit, 5)) {
        if (hit.transform.tag == "Player") {
          StartCoroutine(RotateAndShoot());
          break;
        }
      }
    }
  }
```

Caso exista colisão, o método `Raycast` irá popular a variável `hit` com informação sobre o objeto que intersetou o raio. Essa variável tem um campo `transform` que permite aceder ao componente `Transform` do objeto que intersetou o raio e, a partir daí, à sua *tag*. Caso a *tag* tenha como valor `Player`, então, será iniciada uma corrotina responsável por rodar a planta na direção da formiga e, posteriormente, disparar a ervilha.

Para a implementação do método de rotação, será necessário saber onde se encontra a formiga. Não é eficiente, a cada disparo, procurar na cena o objeto correspondente à formiga, pelo que será criada uma variável que armazenará a instância da formiga e que será populada no método `Start`[23]:

```
public class ShootPea : MonoBehaviour {
  public float ShootForce = 100f;
  Transform SpawnPoint;
  Transform Player;

  void Start () {
    Player = GameObject.FindGameObjectWithTag("Player").transform;
    SpawnPoint = transform.Find("SpawnPoint");
  }
```

Foi usado o método `FindGameObjectWithTag`, que permite procurar um objeto com determinada *tag*, sobre o qual se está a aceder ao componente `Transform`. Sempre que for necessário saber a posição da formiga, poderá ser consultada a variável `Player`.

[23] Note que já existe um método `Start`, que foi implementado apenas para teste e que deve ser substituído pelo aqui apresentado.

A rotação e o disparo podem ser obtidos com o seguinte código:

```
IEnumerator RotateAndShoot() {
  Vector3 targetDir = Player.position - transform.position;
  targetDir.y = 0;

  while (Vector3.Angle(transform.forward, targetDir) > 2) {
    Vector3 newDir = Vector3.RotateTowards(transform.forward,
                                           targetDir, 0.2f, 0f);
    transform.rotation = Quaternion.LookRotation(newDir);
    yield return null;

    targetDir = Player.position - transform.position;
    targetDir.y = 0;
  }
  StartCoroutine(Shoot());
}
```

Mais uma vez, foi usada uma corrotina que, a cada iteração, irá rodar a planta ligeiramente na direção da formiga. Para isso, é criado o vetor `targetDir` com a direção da formiga. Neste vetor, é alterada a coordenada y colocando-a a zero, já que as posições verticais da planta e da formiga são diferentes. Se não se tivesse este cuidado, a planta tentaria rodar não só sobre o eixo vertical (y), para apontar para a formiga, mas também sobre um dos eixos horizontais (x), para conseguir apontar para o centro da formiga.

Obtido este vetor, pretende-se que a planta rode sobre si mesma, para apontar nessa direção. Ora, esta rotação só é necessária enquanto o ângulo entre este vetor e o vetor que aponta para a frente da planta for superior a 2°. Para que esta rotação seja progressiva e não brusca, é usado o método `RotateTowards`, que calcula um vetor interpolado entre os dois vetores indicados, sendo que a cada invocação o vetor é aproximado em 0.2 radianos. O último argumento indica a margem de erro desejada. Obtido o vetor, usa-se o método `LookRotation`, a fim de obter a rotação, para que um objeto esteja virado nessa direção. Após cada incremento, é feito o retorno da corrotina. Quando o controlo é de novo obtido, é calculado um novo objetivo, tendo em conta a nova posição da personagem. Assim que a rotação tiver terminado, é iniciado o processo do disparo.

Embora este algoritmo funcione em teoria, existe um pequeno problema. Como as corrotinas são executadas em paralelo, e podem ser executadas tantas corrotinas quantas as que se desejar, à medida que a formiga for passeando junto à planta, esta vai iniciando várias corrotinas, em que cada uma irá forçar uma rotação, sendo que o resultado final não será o desejado. Para resolver este problema, será usada uma variável booleana, definida no início da classe, que será verdadeira quando a planta se está a mexer ou a disparar, e falsa quando se encontra parada. Além disso, e para que a planta não dispare ervilhas

demasiado depressa, como se fosse uma metralhadora, será adicionado um compasso de espera após cada disparo.

Estas alterações começam pela definição da seguinte variável booleana:

```
bool stopped = true;
```

Esta variável indica se a planta se encontra parada, ou se já se encontra em rotação ou a disparar. No método `Update` o processo de rotação e de disparo de ervilhas só será iniciado se a planta estiver parada (não estiver a disparar qualquer ervilha), pelo que será colocada uma estrutura condicional a validar o conteúdo da variável `stopped`. Do mesmo modo, assim que o processo de disparo iniciar, esta variável terá de ser alterada para um valor falso:

```
void Update() {
  if (stopped) {
    for (int i = 0; i < 5; i++)  {
      float radians = Mathf.Deg2Rad * (30f * i + 30f);
      Vector3 direction = new Vector3(Mathf.Cos(radians), 0,
                                      Mathf.Sin(radians));
      direction = transform.TransformDirection(direction);
      Ray ray = new Ray(transform.position + Vector3.up, direction);
      RaycastHit hit;
      if (Physics.Raycast(ray, out hit, 5)) {
        if (hit.transform.tag == "Player") {
          stopped = false;
          StartCoroutine(RotateAndShoot());
          break;
        }
      }
    }
  }
}
```

Por sua vez, a corrotina `Shoot` irá terminar do seguinte modo:

```
  // ...
  rb.AddForce(transform.forward * ShootForce);
  yield return new WaitForSeconds(0.5f);
  stopped = true;
}
```

A classe `WaitForSeconds` pode ser usada em corrotinas para a fazer aguardar alguns segundos, indicados como parâmetro. Assim, o fluxo de execução irá esperar meio

segundo, até que o controlo volte à corrotina e, então, se indique que o processo de disparo terminou.

Neste momento, está a faltar a validação de colisões da ervilha com a formiga ou com qualquer outro objeto.

5.1.4 COLISÕES E DANO

Preparada a planta, para que dispare ervilhas, é necessário que estas sejam nefastas, para que possam, de algum modo, provocar algum dano à formiga. Este processo será definido à base de colisões. Será adicionada uma pequena *script* à ervilha, que será responsável por se autodestruir em caso de colisão e, se a colisão for com a formiga, será alterado o nível de energia da formiga.

Esta *script*, denominada `PeaController`, será adicionada ao objeto em tempo de execução, pelo que, para a criar, será necessário usar o menu `Assets` ou o botão `create` no separador *Projeto*. O seu conteúdo será tão simples quanto se apresenta de seguida:

```
public class PeaController : MonoBehaviour {
  void OnCollisionEnter(Collision c) {
    if (c.gameObject.tag == "Player")
      GameManager.instance.TakeDamage(5f);
    Destroy(gameObject);
  }
}
```

É definido o evento `OnCollisionEnter` e verificado o objeto contra o qual a ervilha colidiu. Se essa colisão tiver sido com o objeto `formiga`, então, será invocado um novo método, `TakeDamage`, no gestor de jogo. Em qualquer caso, a ervilha será destruída, quer tenha colidido contra a formiga ou contra qualquer outro objeto da cena. Assim, garantir-se-á que não existirão muitas ervilhas na cena, já que logo que esta caia sobre o terreno será destruída.


Não é boa prática, para a eficiência de um jogo, fazer a instanciação e destruição de muitos objetos. O que habitualmente se faz criar uma *cache* de objetos que vão sendo reutilizados. Por exemplo, poderiam existir 100 ervilhas em cena, inativas, que seriam colocadas nos devidos sítios sempre que fosse necessário serem disparadas.

De modo a que a ervilha tenha o comportamento descrito nesta *script*, é necessário associá-la ao objeto. Para isso, na *script* `ShootPea`, o método `Shoot` terá de ser alterado, adicionando uma linha antes que retorne o `WaitForSeconds`:

```
// ...
Rigidbody rb = pea.AddComponent<Rigidbody>();
rb.AddForce(transform.forward * ShootForce);
pea.AddComponent<PeaController>();
yield return new WaitForSeconds(0.5f);
// ...
```

Também será necessária a adição do método `TakeDamage` ao gestor de jogo (`GameManager`):

```
public void TakeDamage(float amount) {
  Energy -= amount;
}
```

Ao ser definido deste modo, o método pode ser reutilizado para outros inimigos, alterando simplesmente a quantidade de dano produzido.

Terminado todo este processo, é possível duplicar o *prefab* e criar tantos inimigos quantos os desejados.

5.2 ARANHA

Além das plantas venenosas, a ilha é habitada por uma aranha, que se movimenta num pequeno raio, deambulando pela ilha. Sempre que a formiga se aproximar demasiado da aranha, esta começa a segui-la. Felizmente, a aranha anda mais devagar, pelo que deverá ser fácil para a formiga fugir do seu encalço.

No sítio do livro (`www.fca.pt`), existe o pacote *Spider* (`spider.unitypackage`) que deverá ser importado, usando o menu `Assets` → `Import Package` → `Custom Package`. A Figura 5.7 mostra o conteúdo deste pacote.

Este pacote irá criar a pasta `CustomColliders`, que inclui um componente especial para a definição de *colliders* em forma de *donut*, denominado *Torus Collider*. Como a aranha é bastante larga, mas baixa, não é possível usar um dos *colliders* tradicionais do Unity, pelo que foi necessário usar uma outra técnica.


A explicação de como construir este componente não é introdutória, por isso, optou-se por não a incluir neste livro. No entanto, um leitor mais interessado poderá analisar o código disponibilizado e, com alguma atenção, conseguirá perceber o seu funcionamento.

Por sua vez, a pasta `Editor` inclui uma *script* usada para indicar ao Unity como o componente *Torus Collider* deve ser apresentado no *Inspetor*.

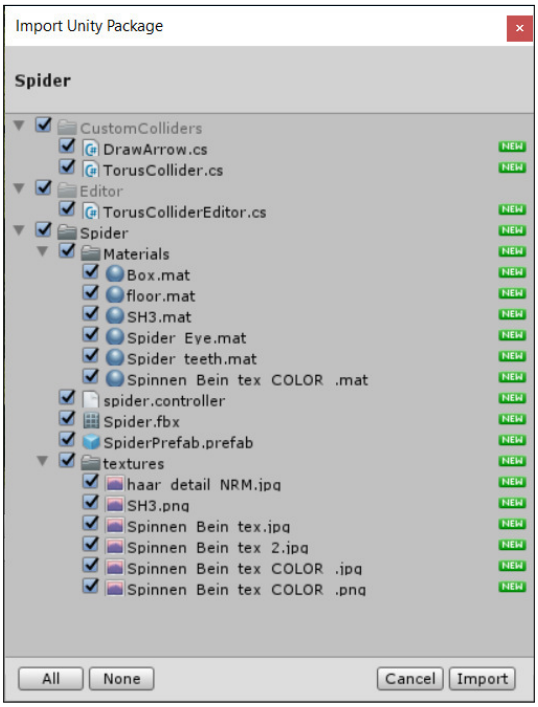


FIGURA 5.7 – Conteúdo do pacote *Spider*

Finalmente, a pasta `Spider` inclui não só o modelo, os materiais e as texturas necessários, mas também um *prefab* pronto a usar, já com o componente *Animator* devidamente configurado para o comportamento da aranha quando parada.

5.3 *NAVMESH*

Para movimentar a aranha no terreno, é necessário saber de que forma a aranha se pode deslocar (por exemplo, não deverá poder andar sobre a água), bem como que encostas poderá subir ou descer, de acordo com a inclinação do terreno. Para além desta informação local, a cada posição possível, é necessário saber qual o percurso que a aranha deve percorrer para se deslocar de um a outro ponto da ilha. O cálculo deste percurso pode ser feito diretamente usando vetores, mas não será fácil ter em conta as zonas que a aranha não poderá percorrer.

Este é um problema genérico na maioria dos jogos, pelo que o Unity inclui um módulo de inteligência artificial que implementa o cálculo de caminhos sobre um conjunto de objetos. Para que este cálculo de caminhos funcione, é necessário criar uma *NavMesh*,

que corresponde a uma malha, semelhante à usada para renderizar os objetos, mas que neste contexto é utilizada para o cálculo de caminhos. Nesta grelha, constituída apenas por triângulos, o Unity sabe que se pode deslocar entre dois triângulos, desde que ambos façam parte das zonas por onde a personagem pode caminhar e exista um conjunto de triângulos com esta mesma propriedade que liguem os dois triângulos iniciais (sendo que há caminho entre dois triângulos sempre que estes partilharem uma aresta). Felizmente, a compreensão desta teoria não é imprescindível para usar esta técnica.

O controlo da navegação sobre *NavMesh* é configurado no separador *Navegação* (`Window` → `Navigation`), que é composto, na versão atual do Unity, por quatro abas, tal como apresentado na Figura 5.8.

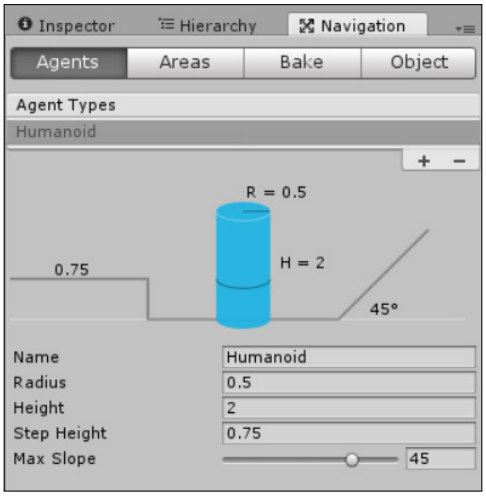


FIGURA 5.8 – Separador *Navegação*

As abas deste separador são as seguintes:

- `Agents` — Aqui são definidos os diferentes tipos de agentes que serão controlados pela *NavMesh*. Diferentes personagens podem, ou não, usar diferentes tipos de agentes. Se todas as personagens têm um tamanho idêntico e se movimentam de forma semelhante, é possível usar um único agente para todas. Por sua vez, se algumas são mais altas, mais gordas, ou menos ágeis, pode ser necessário definir agentes com propriedades diferentes.

  Cada agente define um conjunto de características físicas da personagem. Para além de um identificador de agente, inclui o raio da personagem (para calcular a sua largura), a sua altura (de modo a saber por onde a personagem poderá passar), a altura máxima dos degraus que é capaz de subir, e a inclinação máxima do terreno que a personagem consegue escalar.

Note que, para usar este tipo de agentes, é necessário algum trabalho extra, pelo que a abordagem aqui apresentada não usará agentes diferentes.

- `Areas` — Esta aba permite definir diferentes tipos de áreas: aquelas por onde as personagens podem andar (*walkable*), por onde não se podem movimentar (*non walkable*) e onde podem saltar (*jumpable*). Também é possível a definição de novas áreas, para permitir um maior controlo sobre o processo de locomoção.

- `Bake` — O uso de *NavMesh* pode ser realizado em tempo real, em que a malha de navegação é calculada durante a execução do jogo, ou previamente, usando um processo habitualmente denominado cozinhar ou cozer (*bake*). Esta aba é usada para realizar o processo de cozedura. É possível definir um tamanho de agente genérico, que será usado para todas as personagens (caso não exista nenhum definido no separador respetivo).

- `Object` — Finalmente, nesta aba são definidos os objetos que fazem parte da malha de navegação. Sempre que um terreno é criado, o Unity considera-o para o cálculo de *NavMesh*.

A Figura 5.9 apresenta as dimensões da aranha, pelo que o 1.º passo corresponderá à definição das dimensões do agente genérico (na aba `Bake`), com 2 unidades de raio, 1.5 de altura, 0.25 como altura máxima do degrau e 10° de inclinação.

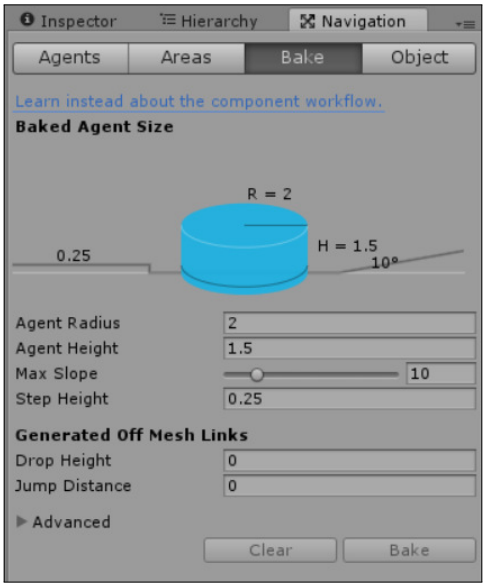


FIGURA 5.9 – Dimensões do agente genérico

Depois, poderá ser cozinhada a *NavMesh*, usando o botão `Bake`. Nessa altura, o Unity irá mostrar, na *Cena*, as zonas por onde a aranha poderá andar, tal como demonstrado na Figura 5.10.

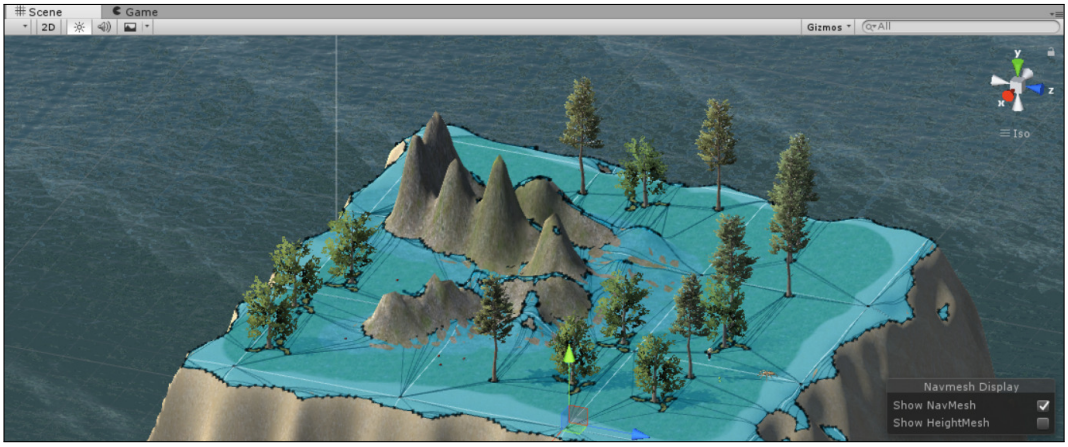


FIGURA 5.10 – Representação visual da *NavMesh* depois de cozinhada

Repare que o Unity calculou perfeitamente as zonas íngremes em que a aranha não poderá andar.

5.3.1 DEAMBULAR

O próximo passo corresponde à colocação do *prefab* da aranha no terreno, bem como a adicionar-lhe comportamento. Assim, o algoritmo corresponderá a calcular um ponto, ao acaso, no terreno e indicar à aranha para caminhar para esse destino. Se, porventura, o destino não for alcançável (não existir um caminho por onde a aranha possa chegar a esse destino), então, será calculado um novo ponto. Nessa altura, a aranha irá deslocar-se até esse local e, quando lá chegar, decidirá ir passear para um novo local.

Para além deste comportamento, a aranha estará atenta à sua vizinhança e, assim que detetar a formiga a uma determinada distância, passará ao modo de ataque, até que a formiga morra ou até que seja capaz de sair do raio de ação da aranha.

Para facilitar o controlo do estado da aranha, será criado um tipo enumerado, com quatro valores possíveis: em ataque, em perseguição, em passeio, ou parada (quando chega a um destino).

Depois de colocado o *prefab* (`Spider/SpiderPrefab.prefab`) no terreno, deve ser adicionado o componente *Nav Mesh Agent*, tal como apresentado na Figura 5.11.

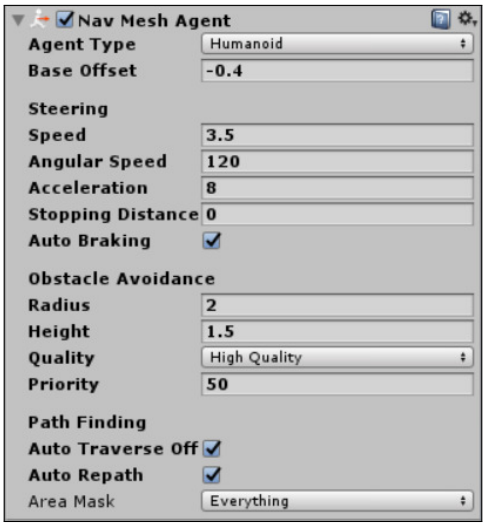


FIGURA 5.11 – Configuração do componente *Nav Mesh Agent*

Neste componente, é importante configurar alguns aspetos, nomeadamente o parâmetro `Base Offset`, que corresponde à distância do agente em relação ao terreno. No caso da aranha, deverá ser indicado um valor de -0.4 (para baixar a aranha). Por sua vez, e embora se tenha definido o tamanho do agente, também é necessário definir o seu tamanho em relação à deteção de obstáculos (colocar o valor do raio — `radius` — a 2 unidades e a altura — `height` — a 1.5 unidades).


O processo de deteção de obstáculos e o cálculo das zonas de passeio são independentes e, em várias situações, pode ser necessário definir diferentes tamanhos.

Depois de adicionado este componente, já é possível controlar o seu comportamento de acordo com a sua posição na *NavMesh*. Será adicionada uma nova *script* à aranha, de nome `SpiderController`, que começará por definir o tipo enumerado `SpiderStatus`:

```
enum SpiderStatus {
    Idle,
    Wander,
    Chase,
    Attack
}
```

Já na classe do componente, será criada uma variável para armazenar o estado atual da aranha, variáveis para aceder aos componentes *Nav Mesh Agent* e *Animator*, que serão

posteriormente usados para animar o movimento da aranha, e variáveis para controlar o movimento de deambulação: o destino atual e o raio de deambulação (distância máxima de cada passeio em determinada direção). A variável `wanderRadius` é pública permitindo que, para diferentes aranhas, se possa definir facilmente, através do *Inspetor*, o raio da área de deambulação. Por sua vez, a variável `wanderTarget` não é pública, já que será calculada programaticamente e não deverá ser alterada pelo utilizador. Assim, o início da *script* `SpiderController` terá as seguintes variáveis definidas[24]:

```
public class SpiderController : MonoBehaviour {

    public float wanderRadius = 20f;
    Vector3 wanderTarget;

    SpiderStatus status = SpiderStatus.Idle;

    NavMeshAgent agent;
    Animator animator;
```

O método `Start` começará por obter os componentes desejados e invocará o método `StartWander`, responsável por calcular o primeiro destino de passeio da aranha:

```
  void Start () {
    animator = GetComponentInChildren<Animator>();
    agent = GetComponent<NavMeshAgent>();

    StartWander();
  }
```

Para obter o `Animator` usou-se o método `GetComponentInChildren`, que procura um componente de determinado tipo nos objetos que sejam filhos do `Transform` atual, visto que este componente não se encontra diretamente no objeto de topo da aranha.

O método `StartWander` é um pouco mais complexo, uma vez que possui um conjunto de considerações práticas para que seja calculado um destino de passeio apropriado:

```
 void StartWander() {

    float angle = Random.Range(0, 2 * Mathf.PI);
    wanderTarget = transform.position +
                   wanderRadius * new Vector3(Mathf.Sin(angle), 0,
                                              Mathf.Cos(angle));
```

[24] Note que, para além da diretiva `using UnityEngine;` já presente, será preciso adicionar a `using UnityEngine.AI;`, que é responsável pela definição das classes de inteligência artificial.

```
    NavMeshHit myHit;
    if (NavMesh.SamplePosition(wanderTarget, out myHit, 5, -1)) {
        wanderTarget = myHit.position;
        agent.SetDestination(wanderTarget);
        agent.isStopped = false;
        status = SpiderStatus.Wander;
    }
}
```

Em primeiro lugar é calculada uma posição, na vizinhança da aranha, à distância definida na variável `wanderRadius`. Para isso, é calculado um ângulo aleatório, em radianos, que irá variar entre 0 e 2π, permitindo assim obter uma qualquer direção ao redor da aranha. Este ângulo é posteriormente usado para calcular uma posição no círculo com raio `wanderRadius`, que será somada à posição atual da aranha.

Embora este processo seja simples e funcional, poderá acontecer que o ponto calculado esteja fora das zonas por onde a aranha pode caminhar. Por exemplo, se a aranha se encontrar à beira do limite da ilha, nada previne que o ponto calculado se encontre no meio do mar. Do mesmo modo, se a aranha estiver perto de uma montanha, o ponto calculado pode ficar numa zona montanhosa, para onde ela não se consegue deslocar.

Este problema é resolvido usando o método `SamplePosition` que permite, a partir de um ponto na cena, calcular um outro ponto, o mais próximo possível do primeiro, que se encontre na *NavMesh*. Este método recebe o ponto que se pretende usar, uma referência a uma variável do tipo *NavMeshHit* que será preenchida com os dados referentes ao ponto mais próximo calculado. Para além disso, também é indicada a distância máxima ao ponto original (pelo que se não for encontrado um ponto na *NavMesh* a esta distância, o método falha) e, finalmente, uma variável que indica qual o tipo de zona da *NavMesh* na que se pretende encontrar o ponto (o valor -1 indica que qualquer ponto é aceitável).

Se um ponto for encontrado na *NavMesh*, então é definida essa posição como o destino do movimento de passeio, primeiramente armazenando-se o valor na variável `wanderTarget` e, seguidamente, invocando o método `SetDestination`. Segue-se a atribuição de um valor falso à variável `isStopped` que indica ao componente *NavMeshAgent* que deve colocar o agente em movimento. Finalmente, é alterado o valor do estado da aranha, indicando que esta se encontra no modo passeio.

Já no método `Update` será necessário verificar o estado da aranha e, caso esta tenha chegado ao seu destino, reiniciar o movimento de passeio. Embora este código tenha de ser alterado muito em breve para permitir que a aranha possa perseguir a formiga, o processo de passeio pode ser testado com:

```
void Update () {
   switch (status) {
```

```
        case SpiderStatus.Wander:
            if ((transform.position - wanderTarget).magnitude < 1f)
               SetIdle();
            break;
        case SpiderStatus.Idle:
           StartWander();
           break;
        case SpiderStatus.Chase:
           break;
        case SpiderStatus.Attack:
           break;
    }
}
```

O método começa por verificar o estado atual da aranha. Se esta se encontra em modo de passeio, é validada a sua distância relativamente ao destino desejado. Como a aranha tem um raio de 2 unidades, considera-se que ela chegou ao destino se a distância do seu centro à posição calculada na *Nav Mesh* for inferior a 1 unidade. Nesse caso, é invocado o método `SetIdle`, apresentado já de seguida. Por sua vez, se a aranha se encontrar no estado parado (`Idle`), então, é iniciado o passeio. Embora, neste momento, ainda não seja possível à aranha entrar em modo de perseguição, para que o compilador não apresente um erro pelo facto de o código atual não considerar essa possibilidade, adicionou-se essa situação, sem qualquer código associado.

O método `SetIdle` é definido simplesmente por:

```
    void SetIdle() {
        agent.isStopped = true;
        status = SpiderStatus.Idle;
    }
```

O modelo da aranha é animado e o pacote que foi importado inclui já um controlador de animação com o estado da aranha parada, em que se move, lentamente, para baixo e para cima. Com o código implementado, a aranha desliza, já que não foi criado um estado referente ao caminhar. Para terminar o modo passeio, será alterado o projeto de forma a que, quando se movimenta, seja usada a animação correta.

Na pasta `Spider`, encontra-se o *Animator Controller* de nome `spider`, ao qual deve ser adicionado um novo estado, denominado `Walk`, as transições de, e para, o estado `Idle`, e o parâmetro `velocity` do tipo `float`, tal como apresentado na Figura 5.12. Ao estado deverá associar-se a animação `walk_ani_vor`.

As transições devem ser criadas desativando a opção `Has Exit Time` e usando as condições apresentadas na Tabela 5.1.

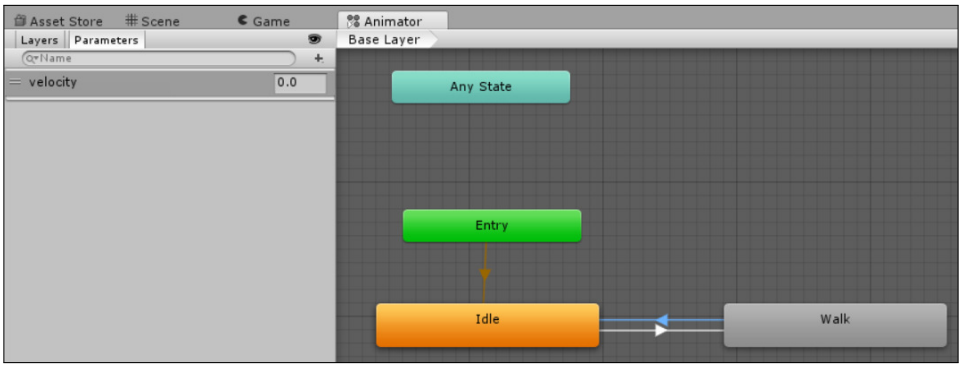


FIGURA 5.12 – Controlador de animações da aranha

ORIGEM	⟶	DESTINO	CONDIÇÃO
`Idle`	⟶	`Walk`	$velocity > 0.01$ (*Greater*)
`Walk`	⟶	`Idle`	$velocity < 0.01$ (*Less*)

TABELA 5.1 – Transições do controlador de animações da aranha


Os parâmetros do tipo `float` são, tal como as variáveis desse mesmo tipo, representados internamente como valores de precisão limitada, o que significa que não devem ser comparados usando igualdades (já que será muito pouco provável que o número seja representado internamente com a precisão exata desejada). É essa a razão pela qual, ao definir as transições, só estão disponíveis as comparações *maior que* e *menor que* mas não *maior ou igual que* ou *menor ou igual que*.

Para completar o processo, fica apenas a faltar a alteração, no método `Update`, ao parâmetro `velocity` definido no controlador. Tal pode ser realizado adicionando a seguinte linha exatamente antes da última chaveta do método:

```
animator.SetFloat("velocity", agent.velocity.magnitude);
```

Nesta linha, usa-se o campo `velocity` do *Nav Mesh Agent* para obter o vetor composto, que inclui a direção e velocidade da aranha, e é usado o acessor `magnitude` para retirar deste vetor o valor real da velocidade.

5.3.2 DETEÇÃO DA FORMIGA E PERSEGUIÇÃO

O próximo passo corresponde à deteção de proximidade da formiga (dentro de um intervalo previamente definido) para que, quando esta se encontre suficientemente perto da

aranha se inicie perseguição. Durante a perseguição, a aranha poderá desistir, se a formiga fugir do seu raio de ação, ou poderá atacar a formiga, matando-a de imediato.

Em primeiro lugar, serão criadas duas variáveis, uma para guardar a referência ao componente `Transform` da formiga, de modo a ser possível saber a sua posição, e outra para armazenar a distância a partir da qual a aranha consegue detetar a formiga. Por sua vez, no método `Start`, será calculada a referência à formiga:

```
public class SpiderController : MonoBehaviour {
  // ...
  public float followDistance = 10;
  private Transform player;

  void Start () {
    // ...
    player = GameObject.FindGameObjectWithTag("Player")
                       .GetComponent<Transform>();
  }
```

O método `Update` deverá também ser alterado. No caso `Wander` do bloco do `switch` ficará com:

```
case SpiderStatus.Wander:
  if ((transform.position - player.position).magnitude < followDistance) {
    SetChase();
  }
  else if ((transform.position - wanderTarget).magnitude < 1f) {
    SetIdle();
  }
  break;
```

Ou seja, se a formiga estiver ao alcance da aranha, a perseguição inicia-se. Caso contrário, continuará com o comportamento de deambulação previamente definido.

O método `SetChase` é bastante simples: limita-se a ativar o `NavMeshAgent` e a definir o seu destino — a posição atual da formiga:

```
  void SetChase() {
     agent.isStopped = false;
     status = SpiderStatus.Chase;
     agent.SetDestination(player.position);
  }
```

Ao contrário do algoritmo de deambulação, em que a aranha não muda o seu percurso enquanto não chegar ao seu destino previamente calculado, durante a perseguição,

o seu destino muda constantemente. Por isso, no método `Update` é necessário atualizar, constantemente, a posição para a qual o `NavMeshAgent` se desloca. Além disso, é necessário validar se a formiga conseguiu escapar ao raio de ação da aranha e, nesse caso, parar a perseguição. Este par de comportamentos pode ser implementado codificando a condição `Chase` do comando `switch`:

```
case SpiderStatus.Chase:
  if ((transform.position - player.position).magnitude > followDistance) {
    SetIdle();
  }
  else {
    agent.SetDestination(player.position);
  }
  break;
```

5.3.3 ATAQUE

A aranha só poderá atacar a indefesa formiga quando, no estado de perseguição, se encontre a 2 unidades da formiga. Nessa altura, será ativada a animação de ataque. Ao mesmo tempo, os *colliders* da aranha deverão colidir com a formiga e, nessa altura, será invocado o método para infligir o respetivo dano.

Mais uma vez, será necessário alterar o código correspondente à condição `Chase`, do seguinte modo:

```
case SpiderStatus.Chase:
  if ((transform.position - player.position).magnitude < 2.0f) {
      SetAttack();
  }
  else
   if ((transform.position - player.position).magnitude > followDistance) {
      SetIdle();
  }
  else {
      agent.SetDestination(player.position);
  }
  break;
```

No estado de perseguição, a aranha começa sempre por validar a sua distância em relação à formiga e, se esta for menor do que 2 unidades, inicia o ataque. Caso contrário, se a formiga ultrapassar a distância de deteção, então, a perseguição é cancelada . Se nenhuma destas condições ocorrer, a aranha continua a perseguição.

O método `SetAttack` também será bastante simples:

```
void SetAttack() {
    animator.SetTrigger("attack");
    agent.isStopped = true;
    status = SpiderStatus.Attack;
}
```

O método começa por ativar o *trigger* para a animação de ataque, que será adicionada, já de seguida, ao controlador da aranha. Depois, a aranha passa a estar parada e altera o seu estado para o ataque.

Ainda antes de alterar o controlador de animações da aranha, será adicionado um método para detetar a colisão com a formiga e, nessa situação, infligir o dano respetivo:

```
void OnTriggerEnter(Collider c) {
    if (c.gameObject.tag == "Player") {
        GameManager.instance.TakeDamage(100f);
    }
}
```

Sempre que a aranha colidir com a formiga, esta irá receber um dano de 100 unidades, o que garante que nunca sobreviverá a um ataque bem sucedido da aranha.

Agora que o comportamento está todo codificado, falta apenas abrir o controlador da aranha, criar um novo parâmetro, `attack`, com o tipo `Trigger`, e adicionar um novo estado, denominado `Attack`, como demonstrado na Figura 5.13. Este estado deverá ter associada a animação `Attack` da aranha.

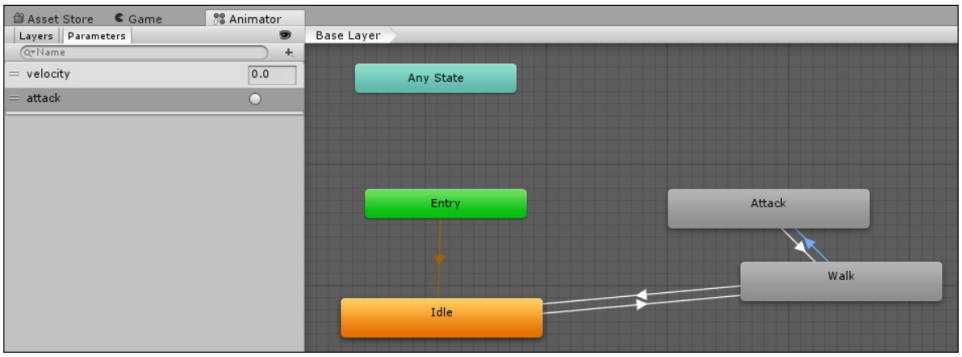


FIGURA 5.13 – Controlador da aranha com estado `Attack`

A transição será do estado `Walk` para o estado `Attack` e terá como condição o *trigger* `attack` estar ativo.

Embora a formiga não consiga resistir a um ataque da aranha, é capaz de conseguir fugir, pelo que se deverá adicionar também uma transição no sentido inverso (Figura 5.13). Neste caso, a transição não terá qualquer condição e terá a opção `Has Exit Time` ativa (Tabela 5.2).

ORIGEM	⟶	DESTINO	CONDIÇÃO
`Walk`	⟶	`Attack`	*attack*
`Attack`	⟶	`Walk`	—

TABELA 5.2 – Transições do controlador de animações da aranha para o estado `Attack`

Se, no final do ataque, a formiga não tiver morrido, será necessário que a aranha volte ao seu estado normal, deambulando, o que pode ser obtido alterando o código referente ao estado `Attack` no `switch` definido anteriormente para o seguinte código:

```
case SpiderStatus.Attack:
   if (animator.GetCurrentAnimatorStateInfo(0).IsName("Walk")) {
      SetIdle();
   }
   break;
```

Depois de um ataque falhado, a aranha terá de voltar ao ataque ou ao seu passeio. Para facilitar a implementação desta dupla possibilidade de resposta, a solução aqui usada corresponde a verificar se, ainda no estado de ataque, o *Animator Controller* já regressou à animação de passeio (`Walk`). É possível obter todos os detalhes relativos a um `Animator`. No caso apresentado, usa-se o método `GetCurrentAnimatorStateInfo` para obter dados sobre o estado atual. O seu único argumento corresponde ao nível do *Controller* a que se pretende aceder, mas como todos os *Controllers* definidos neste livro têm apenas um nível, será usado o valor zero. O método `IsName` valida o nome de um determinado estado.

5.3.4 ATUALIZAÇÃO DO *PREFAB*

Embora no início da secção 5.2 se tenha usado um *prefab* da aranha, foram realizadas alterações (adicionados componentes e alteradas algumas propriedades) que não são automaticamente atualizadas no *prefab* original.

Sempre que se edita um objeto na *Cena*, que foi obtido a partir de um *prefab*, o topo do *Inspetor* mostra três botões para a gestão do *prefab* como demonstrado na Figura 5.14.

Estes três botões são usados para:

- `Select` — Permite selecionar automaticamente o *prefab*, sem necessidade de o procurar no separador *Projeto*. Funciona apenas como um atalho.

- `Revert` — Permite repor as configurações originais do *prefab*. Ou seja, todas as alterações que tiverem sido efetuadas no objeto serão perdidas.
- `Apply` — Aplica todas as alterações que foram realizadas na instância do objeto ao *prefab*, o que permite que o programador possa usar uma instância para configurar o comportamento do *prefab* e aplicar essas alterações diretamente ao *prefab* sem necessidade de o voltar a criar.

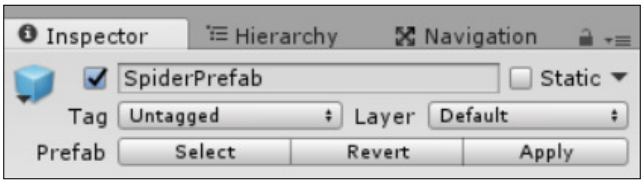
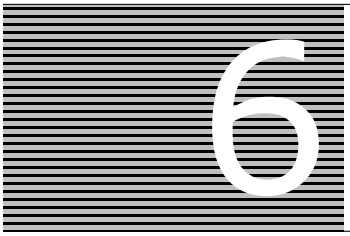


FIGURA 5.14 – *Inspetor* de uma instância de um *prefab*

Como foram realizadas muitas alterações ao *prefab* original, deverá ser usado o botão `Apply` para aplicar as alterações ao *prefab*. A partir deste momento, poderão ser adicionadas várias aranhas, bastando arrastar o respetivo *prefab* para a *Cena*.



Sons e Efeitos Visuais

Dois aspetos especialmente importantes num jogo são o som e os efeitos visuais. O som é útil para indicar ao jogador o que se passa na sua vizinhança, seja a mudança de comportamento de um inimigo, o disparo de uma bala, ou a recuperação energia. Os efeitos visuais, embora não sejam imprescindíveis, são cativantes, ajudando o jogador a divertir-se e a associar funcionalidades a determinadas ações.

6.1 SOM

No Unity, existem dois componentes para a gestão de som: o *Audio Listener* e o *Audio Source*. Esta secção descreve as funcionalidades de cada um destes componentes, e explica a adição de vários sons ao jogo.

6.1.1 *AUDIO LISTENER*

O componente *Audio Listener* é extremamente simples. Sempre que uma câmara é criada, o Unity associa-lhe um destes componentes, que funciona como um microfone, colocado na cena, que recolhe os sons que o rodeiam. Por esta razão, só é possível ter um *Audio Listener* ativo em cada momento.

Embora se possa associar este componente a outros objetos, é habitual usar-se a câmara. A razão é simples: se a câmara funciona como olhos para o jogador, é natural que os sons tenham a mesma posição relativa, de modo a que, ao ouvir um som vindo da esquerda, o jogador automaticamente o associe a algo que se passa à sua esquerda (à esquerda dos seus olhos, ou seja, à esquerda da câmara).

Desafia-se o leitor a experimentar, no final desta secção, a alternar o *Audio Listener* entre a câmara e, por exemplo, a formiga, de modo a perceber as diferenças e o seu impacto na jogabilidade.

Como este componente já existe na câmara, não é necessário adicioná-lo. O componente também não tem propriedades de configuração, a não ser a possibilidade de alterar o volume de jogo programaticamente.

6.1.2 *AUDIO SOURCES*

Num mundo tridimensional, os sons surgem dos vários objetos que os produzem. Por exemplo, quando a planta venenosa atira uma ervilha, é de esperar que o som surja da posição da planta. Do mesmo modo, quando a aranha inicia a perseguição, o som deverá ser proveniente da direção de onde ela se encontra.

Durante esta secção, serão adicionados vários componentes de fonte de som, associados efeitos sonoros a cada um e codificada a reprodução desses sons de acordo com o desenrolar dos vários eventos ao longo do jogo.


No sítio do livro (`www.fca.pt`) existe a pasta `Sounds` com os ficheiros correspondentes aos diferentes sons que serão usados nesta secção. No entanto, o leitor poderá procurar outros sons e aplicá-los ao seu gosto.

Esta secção explicará, passo a passo, como adicionar cada som. À medida que o leitor for ganhando prática neste processo, os detalhes apresentados serão progressivamente menos.

Serão adicionados sons aos seguintes comportamentos:

- Ao disparo da ervilha (quando esta sai da planta).
- À colisão da ervilha com a formiga.
- À morte da formiga.
- Ao ataque da aranha à formiga.
- À ingestão de um cogumelo.
- À perseguição da formiga, feita pela aranha.

Além destes sons, será adicionada alguma música de fundo (secção 6.1.2.6).

A importação de sons é semelhante à importação de qualquer outro tipo de recurso, bastando arrastá-los para o separador *Projeto*. Sugere-se que se arraste toda a pasta, de modo a que os sons fiquem devidamente arrumados.

6.1.2.1 DISPARO DA ERVILHA

O primeiro som a configurar corresponde ao disparo da ervilha. Quem dispara a ervilha é a planta, pelo que deverá ser a este objeto que o *Audio Source* terá de ser adicionado. A Figura 6.1 mostra o componente *Audio Source* já configurado.

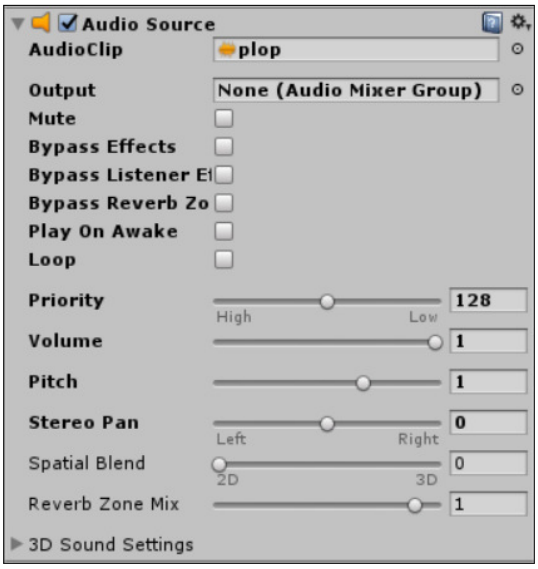


FIGURA 6.1 – *Audio Source* para o som de lançamento da ervilha

Neste componente, o primeiro campo corresponde ao trecho de áudio que deve ser reproduzido. De seguida, o campo `Output` permite que o som seja reproduzido através de outro objeto, responsável pela mistura de sons, o que não corresponde ao nosso exemplo. A opção `Mute` indica se o som deve ter volume, e portanto, ser audível, ou se por outro lado, deve ter volume nulo. As três opções que se seguem indicam se os sons provenientes deste objeto devem, ou não, ser processados por componentes de efeitos ou de *reverb*. As últimas duas opções deste grupo são mais importantes: o `Play On Awake` deve estar desativado, de forma a que o som não seja reproduzido quando a cena inicia, e o `Loop` indica se o som deverá ser reproduzido em ciclo.

As opções que se seguem permitem configurar prioridade, volume, tom, entre outras propriedades que não serão aqui analisadas.

Depois de adicionado e devidamente configurado o componente, poderá ser adicionado o código que irá instruir o Unity a reproduzir o som. Será adicionada a invocação do método `Play` na *script* `ShootPea`, no método `Shoot`, depois do ciclo que calcula o tamanho da ervilha:

```
IEnumerator Shoot() {
    // ...

    GetComponent<AudioSource>().Play();
    Rigidbody rb = pea.AddComponent<Rigidbody>();
    // ...
```

6.1.2.2 COLISÃO DA ERVILHA COM A FORMIGA

Para reproduzir o som da colisão da ervilha com a formiga, existem duas hipóteses: adicionar um *Audio Source* à ervilha e fazer com que esta reproduza o som na colisão; ou adicionar o *Audio Source* à formiga e fazer com que a ervilha, ao colidir com a formiga, invoque o método de reprodução de som definido na *script* da formiga.

Embora pareça que a primeira hipótese é mais simples, a forma como o código da *script* associada à ervilha foi desenvolvido não o permite. Logo após a colisão a ervilha é destruída e um objeto inexistente não pode reproduzir som. A solução seria manter a ervilha na cena durante alguns segundos até que o som terminasse de ser reproduzido, mas a outra abordagem descrita é mais simples de implementar.

A solução será adicionar o *Audio Source* à formiga, atribuir-lhe o trecho de som `argh` e desativar a opção `Play On Awake`. Uma vez que poderão existir outras plantas na *Cena*, deverá usar-se o botão `Apply` no topo do *Inspetor* da planta para propagar a adição deste componente ao *prefab* original.

Para que a formiga reaja às colisões da ervilha, bastará adicionar algum código à função `TakeDamage`, que está implementada na *script* `GameManager`, do seguinte modo:

```
public void TakeDamage(float amount) {
    Energy -= amount;
    player.GetComponent<AudioSource>().Play();
}
```

Ou seja, ao mesmo tempo que o dano é infligido, acede-se ao componente *Audio Source* da formiga e solicita-se a sua reprodução. O problema é que, neste momento, ainda não existe uma referência ao jogador. Para solucionar, poderá ser adicionada uma variável no topo desta *script*:

```
GameObject player;
```

No método `Start` será obtida a referência ao objeto usando a sua etiqueta (*tag*):

```
void Start() {
  TotalMushrooms = GameObject.FindGameObjectsWithTag("cogumelo").Length;
  player = GameObject.FindGameObjectWithTag("Player");
}
```

6.1.2.3 MORTE DA FORMIGA

Um som com propriedades semelhantes às do anterior é o que se pretende reproduzir quando a formiga morre. Este som também será emitido pela formiga, pelo que se poderá aproveitar algum do código alterado na secção 6.1.2.2.

No método `TakeDamage`, ao invés de reproduzir diretamente o som da formiga, poderá ser invocado um método, na formiga, para reproduzir um de dois sons: a colisão da ervilha (`PlayArgh`), ou o som relativo à morte da formiga (`PlayDying`). No entanto, não é esta a única situação em que a formiga morre. Por exemplo, quando o tempo expirar, será também necessário ativar a reprodução desse som.

Uma boa prática no desenvolvimento de um jogo é não misturar diferentes tipos de comportamento numa mesma *script*, pelo que os métodos referidos no parágrafo anterior não serão adicionados à *script* `AntInput`. Será criada uma nova associada à formiga, denominada `AntSounds`:

```
public class AntSounds : MonoBehaviour {

    public AudioClip ArghSound;
    public AudioClip DyingSound;

    AudioSource audioSource;

    void Start() {
       audioSource = GetComponent<AudioSource>();
    }
```

Nesta classe estarão disponíveis dois campos públicos do tipo `AudioClip`, para armazenar cada um dos dois sons possíveis: o som para a colisão da ervilha (`argh`) e o som para a morte da formiga (`dying`) — sons que deverão ser arrastados no editor para os devidos campos. Também se criou uma variável privada, do tipo `AudioSource`, para armazenar uma referência ao componente com o mesmo nome, que é inicializada no método `Start`.

Os métodos para a reprodução dos dois sons serão bastante semelhantes. Convém chamar a atenção para o facto de que estes métodos terão de ser públicos, já que serão usados a partir de uma classe externa:

```
    public void PlayArgh() {
        audioSource.clip = ArghSound;
        audioSource.Play();
    }

    public void PlayDying() {
        audioSource.clip = DyingSound;
        audioSource.Play();
    }
```

Nestes métodos atribui-se o som a ser reproduzido e, posteriormente, é invocado o método `Play` para o reproduzir.

Depois destas alterações, pode alterar-se o *script* `GameManager`, em particular o método `TakeDamage` do seguinte modo:

```
public void TakeDamage(float amount) {
    if (!dead) {
        Energy -= amount;
        if (Energy >= 0) {
            player.GetComponent<AntSounds>().PlayArgh();
        }
    }
}
```

Esta implementação tira partido do facto de se saber que, quando a energia passar a negativa, a formiga morreu, portanto, o som a reproduzir será diferente daquele que será reproduzido quando é atingida por uma ervilha. Se a formiga tiver morrido, outra parte do código (no método `Update`) tratará de reproduzir o som correto.

Na secção 4.3, ao implementar-se o método `Update` na classe `GameManager`, foi deixada uma mensagem a indicar a morte da formiga. Este método deverá ser alterado para que também reproduza o som da morte, mas como é invocado a cada *frame*, temos de prevenir a possibilidade de o *Unity* tentar reproduzir o som a cada *frame* renderizada após a morte da formiga. Para resolver esse problema, será criada uma variável booleana, denominada `dead`, que indicará se a morte da formiga já foi processada. No início da classe `GameManager` define-se:

```
public class GameManager : MonoBehaviour {
    // ...
    bool dead = false;
```

Por sua vez, no método `Update` altera-se a condição da morte, validando se esta já ocorreu. Em caso negativo, é processada, e o valor da variável alterado:

```
void Update() {
    Energy -= Time.deltaTime;
    if (!dead && Energy < 0) {
        dead = true;
        player.GetComponent<AntSounds>().PlayDying();
        Debug.Log("Formiga morta!");
    }
}
```

Esta implementação garante que o som só será reproduzido uma vez depois da morte da formiga.

6.1.2.4 INGESTÃO DE COGUMELOS

Ainda no que toca a sons produzidos pela formiga, também se pretende que se ouça algo quando ela come um cogumelo. Mais uma vez, poderá tirar-se partido da *script* `AntSounds` criando uma variável extra, pública, para o som em questão, para onde deve ser arrastado o som `eating`:

```
public class AntSounds : MonoBehaviour {

    public AudioClip ArghSound;
    public AudioClip DyingSound;
    public AudioClip EatSound;
```

Relembra-se que estes três sons devem ser preenchidos, usando o editor do Unity, como se vê na Figura 6.2.

FIGURA 6.2 – Configuração do *Audio Source* e da *script* `AntSounds`

Tal como foram implementados os métodos para a colisão da ervilha e da morte da formiga, será também criada um método semelhante para reproduzir o som da formiga a ingerir o cogumelo:

```
    public void PlayEat() {
        audioSource.clip = EatSound;
        audioSource.Play();
    }
```

Fica apenas a faltar a invocação do método `PlayEat` na altura certa. Na *script* `AntInput` foi criado um método `DeferredPick` (secção 4.4.4) que é invocado de modo diferido, a fim de sincronizar a ingestão do cogumelo com o incremento do número de cogumelos. Este mesmo método poderá ser modificado para reproduzir, atempadamente, o som de ingestão do cogumelo:

```
  void DeferredPick () {
     GameManager.instance.PickMushroom();
     GetComponent<AntSounds>().PlayEat();
 }
```


A solução aqui implementada não é propriamente a mais versátil, já que não permitirá a reprodução de diferentes sons ao mesmo tempo. Para a colisão da ervilha ou morte da formiga não é um problema, já que apenas um poderá acontecer a cada momento, mas a ingestão do cogumelo pode ser interrompida por um dos outros dois eventos, pelo que a reprodução poderá não ser a melhor. Uma solução alternativa seria a criação de um *Audio Source* noutro objeto, apenas para reproduzir o som referente à ingestão do cogumelo.

6.1.2.5 PERSEGUIÇÃO E ATAQUE

A aranha, por sua vez, também emite alguns sons. Por um lado, haverá uma música sempre que a aranha estiver em perseguição e um som quando esta realiza o seu ataque mortífero. Seguindo a mesma abordagem realizada com a formiga, será adicionado um *Audio Source* à aranha (com a opção `Play On Awake` desligada) e uma nova *script*, denominada `SpiderSounds` com o seguinte conteúdo[25]:

```
public class SpiderSounds : MonoBehaviour {
    public AudioClip chase;
    public AudioClip bark;

    private AudioSource soundSource;

    void Start() {
       soundSource = GetComponent<AudioSource>();
    }

    public void BeginChase() {
        soundSource.clip = chase;
        soundSource.loop = true;
        soundSource.Play();
    }

    public void EndChase() {
        if (soundSource.isPlaying)
            soundSource.Stop();
    }
```

[25] Tal como no caso da planta, está a ser alterada uma instância de um *prefab* pelo que, depois de ter adicionado o componente *Audio Source* e a *script* `SpiderSounds`, deverá utilizar o botão `Apply` no topo do *Inspetor* para propagar as alterações ao *prefab* original.

```
    public void Bark() {
        EndChase();
        soundSource.clip = bark;
        soundSource.loop = false;
        soundSource.Play();
    }
}
```

Esta *script* segue a mesma estrutura da sua irmã. A principal diferença reside na utilização da propriedade `loop` para indicar que a música de perseguição deve repetir-se indefinidamente e no facto de se ter adicionado um método para terminar a reprodução dessa mesma música com o método `Stop`. De realçar que, antes de terminar a reprodução, é usada a propriedade `isPlaying` para garantir que música se encontra em curso.

Note que será necessário arrastar e música e o efeito sonoro para as respetivas variáveis: `suspense` para a variável `chase` e `bark` para a variável `bark` (Figura 6.3).

No método `SetIdle` da *script* `SpiderController` será adicionada uma linha, exatamente no final deste método, para terminar a perseguição, caso esta se encontre ativa:

```
GetComponent<SpiderSounds>().EndChase();
```

Do mesmo modo, no início da perseguição, será necessário ativar a música, adicionando a seguinte linha ao final do método `SetChase`:

```
GetComponent<SpiderSounds>().BeginChase();
```

No método que inicia o ataque da aranha (`SetAttack`) será adicionado:

```
GetComponent<SpiderSounds>().Bark();
```

6.1.2.6 MÚSICA DE FUNDO

Para a música de fundo, não será necessário escrever qualquer linha de código, sendo que bastará escolher um objeto onde adicionar o *Audio Source* e configurá-lo diretamente no *Inspetor*. Para que não haja variações de volume, o ideal será optar por um objeto que esteja sempre à mesma distância da câmara, por exemplo, o próprio objeto da câmara (`Camera`), que se encontra sob o objeto `Pivot`, dentro do objeto `MultipurposeCameraRig`.

 A vantagem de colocar o componente na própria câmara é que, se num determinado momento pretender aceder-lhe, é fácil obter a câmara atual usando o campo `Camera.main`.

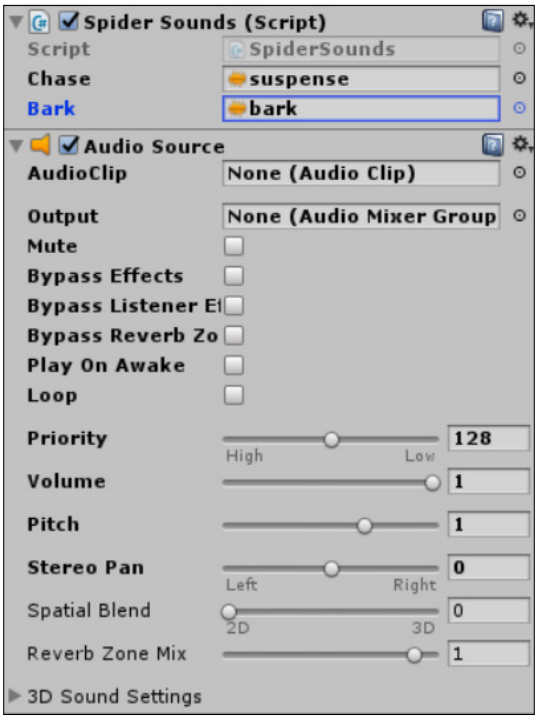


FIGURA 6.3 – Configuração do *Audio Source* e da *script* `SpiderSounds`

Adicionado o componente *Audio Source*, será uma questão de indicar qual o `clip` que deve ser reproduzido, que a reprodução deverá iniciar assim que o objeto for criado (`Play On Awake`) e que deverá repetir a música (`Loop`). Poderá ainda alterar-se o volume, de modo a que não se sobreponha ao resto dos efeitos sonoros. A Figura 6.4 mostra uma possível configuração deste componente.

6.2 EFEITOS VISUAIS

Embora a mecânica de um jogo seja o ponto central no seu desenvolvimento, a verdade é que o uso de recursos atrativos permite mais facilmente captar a atenção dos jogadores. De seguida serão apresentados três tipos de efeitos visuais: reflexos, rastos e sistemas de partículas.

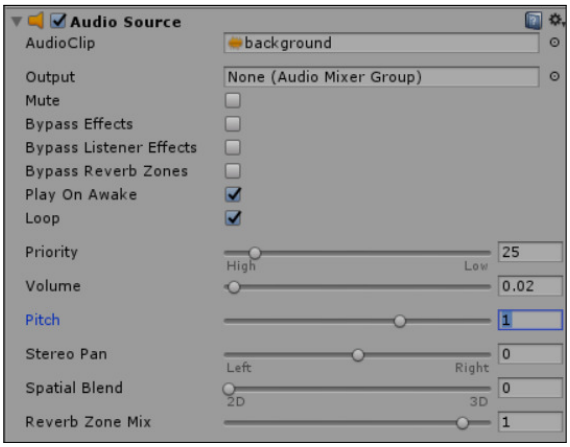


FIGURA 6.4 – Configuração do *Audio Source* para a música de fundo

6.2.1 REFLEXOS

Um efeito simples de obter e que permite algum espetáculo visual é o uso de reflexos (*flares*) da luz na câmara de jogo. Embora seja possível a criação de reflexos de raiz, nesta secção serão usados reflexos disponíveis nos pacotes *standard* do Unity, pelo que deverá importar o pacote *Effects*, usando o menu `Assets` → `Import Package` → `Effects`. Surgirá a janela que lista todo o conteúdo do pacote. Poderá importar-se tudo ou, sabendo que apenas se pretendem os reflexos, desselecionar todos os itens, com exceção da árvore referente aos `Light Flares`, como se vê na Figura 6.5.

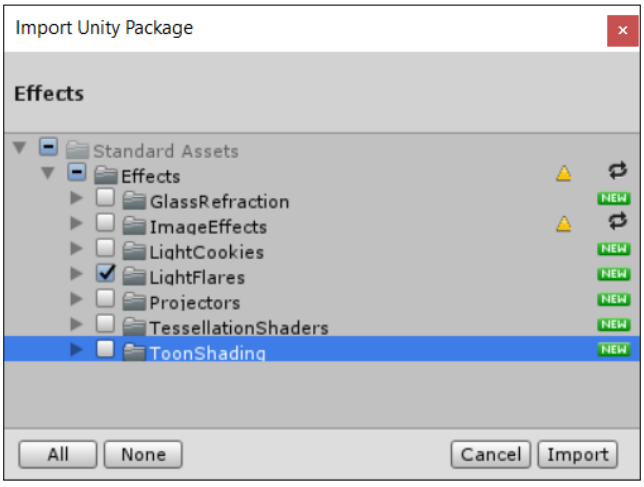


FIGURA 6.5 – Importação de reflexos do pacote *Effects*

Para que os reflexos sejam visíveis, é necessário que a câmara que esteja a fazer a renderização do ambiente de jogo tenha o componente *Flare Layer* associado. Este, não tem quaisquer configurações, mas, se não existir, o uso de reflexos não vai produzir qualquer resultado.

A configuração do tipo de reflexo é, habitualmente, adicionada a um objeto vazio que conterá apenas o componente *Lens Flare*. Assim, deve ser criado um objeto vazio a que se poderá dar o nome de `Reflexos` e ao qual deverá ser adicionado o componente *Lens Flare*, como é demonstrado na Figura 6.6.

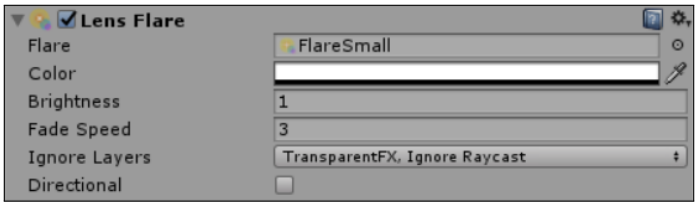


FIGURA 6.6 – Componente *Lens Flare*

No pacote predefinido do Unity existem três tipos de reflexos, que se encontram na pasta `Standard Assets/Effects/Light Flares/Flares`. Um reflexo consiste numa textura e num conjunto de elementos, que indicam o número de vezes e a forma como a textura será usada nos reflexos.

Na configuração do componente *Lens Flare* é possível, além de definir o tipo de reflexo, definir a sua cor, a quantidade de brilho e a velocidade de desvanecimento. É também possível definir um conjunto de camadas de objetos que devam ser ignorados, bem como se o cálculo dos reflexos deve ter em conta a iluminação global ou direcional.

Aplicando o reflexo `50mmZoom` e alterando ligeiramente a cor definida no *Inspetor* (`Color`), é possível obter resultados como o apresentado na Figura 6.7. Para obter um bom resultado deverá experimentar diferentes posições para o objeto `Reflexos` até conseguir o efeito desejado.

6.2.2 RASTOS

Seja apenas como um efeito visual, seja como forma de representar visualmente a velocidade ou movimento de um objeto, os rastos (*trail*) são um efeito fácil de obter e com alguma força visual.

Para demonstrar o uso de rastos, será criado um para cada uma das ervilhas, o que dará um efeito interessante, quase como se a ervilha deixasse um rasto de fumo verde por onde passa.

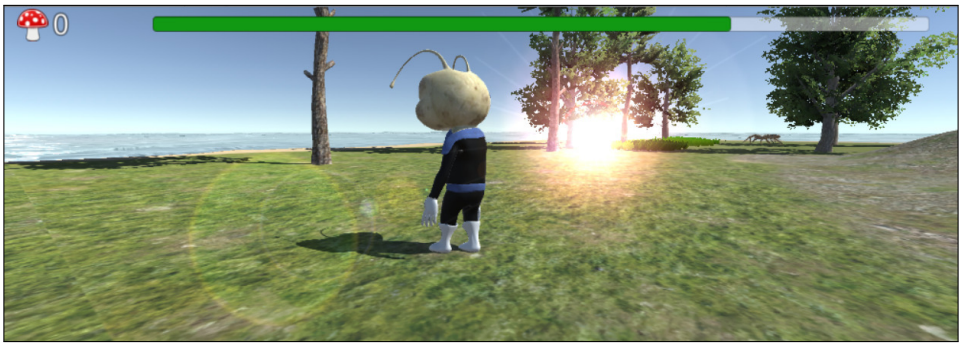


Figura 6.7 – Exemplo de reflexo

Embora a configuração de rastos seja simples, a forma como as ervilhas estão a ser criadas e os respetivos componentes associados não permite que se teste facilmente a adição de rastos. Ao implementar essa funcionalidade no Capítulo 5, o código para gerar a ervilha usa o método `CreatePrimitive` para criar uma esfera, à qual eram posteriormente adicionados, manualmente, um *Rigidbody* e a *script* `PeaController`. Para seguir esta mesma lógica, seria necessário adicionar e configurar, via código, o componente *Trail Renderer*.

Para facilitar a adição e configuração deste componente, em primeiro lugar será criado um *prefab* para representar a ervilha e alterado o código já existente para que continue a funcionar usando o *prefab* ao invés da esfera inicial. Posteriormente, será adicionado o componente *Trail Renderer* e discutida a sua configuração.

6.2.2.1 *PREFAB* PARA A ERVILHA

Nesta secção será alterada a forma como as ervilhas estão a ser criadas. Não será adicionada qualquer funcionalidade nova, apenas se irá reimplementar a funcionalidade atual, usando uma outra abordagem.

Em primeiro lugar, será criada uma esfera na *Cena*, a que se dará o nome de `Pea`. A esta esfera, serão adicionados os componentes *Rigidbody* e a *script* `PeaController`. No componente *Rigidbody* será desligado o uso da gravidade, que será ativado programaticamente. Neste momento este objeto é suficiente para servir de modelo às ervilhas, pelo que será criado um *prefab* a partir dele. Isto pode ser obtido arrastando o objeto `Pea` da *Hierarquia* para o separador *Projeto*. Criado o *prefab*, a esfera original pode ser apagada da *Cena*.

O passo seguinte será alterar o código da *script* `ShootPea`, de modo a usar um *prefab* e não uma esfera criada programaticamente. Para isso, será criada, inicialmente, uma variável pública, do tipo `GameObject`, que irá armazenar uma referência ao *prefab*:

```
public class ShootPea : MonoBehaviour {
  public GameObject peaModel;
  // ...
```

Posto isto, deverá ser arrastado o *prefab* da ervilha para este campo, que deverá aparecer em todas as plantas existentes no nível.


Note que só terá de executar este processo numa das plantas. Depois, deverá usar o botão `Apply` para aplicar as alterações ao *prefab* da planta e, assim, propagar as alterações a todas as plantas na *Cena*.

Posteriormente, no método `Shoot`, terão de ser feitas algumas alterações. O código que criava a esfera

```
GameObject pea = GameObject.CreatePrimitive(PrimitiveType.Sphere);
pea.transform.position = SpawnPoint.position;
pea.transform.localScale = Vector3.one * 0.01f;
```

deverá ser substituído por:

```
GameObject pea = Instantiate(peaModel,
                             SpawnPoint.position,
                             Quaternion.identity);
pea.transform.localScale = Vector3.one * 0.01f;
```

Ou seja, em vez de se obter uma ervilha através da criação de uma esfera, obtém-se através da instanciação de um *prefab* existente. O método `Instantiate` recebe como primeiro parâmetro uma referência ao objeto-modelo que deve ser criado, a posição onde o objeto deve ser colocado e a sua rotação. Já que a rotação de uma esfera é irrelevante, usamos a propriedade `identity` da classe `Quaternion` que representa a rotação nula.

Como o *prefab* acabado de criar já inclui os componentes *Rigidbody* e a *script* `PeaController`, também será necessário alterar o seguinte código (ainda no método `Shoot`)

```
Rigidbody rb = pea.AddComponent<Rigidbody>();
rb.AddForce(transform.forward * ShootForce);
pea.AddComponent<PeaController>();
```

pelas seguintes linhas

```
Rigidbody rb = pea.GetComponent<Rigidbody>();
rb.useGravity = true;
rb.AddForce(transform.forward * ShootForce);
```

que correspondem a obter uma referência ao componente *Rigidbody* já existente (e não à criação de um novo), a ativar a gravidade e a adicionar-lhe a força necessária.

Neste ponto, o comportamento original do disparo da ervilha deve ter sido totalmente replicado.

6.2.2.2 *TRAIL RENDERER*

Para se obter um rasto em qualquer objeto bastará adicionar um *Trail Renderer*. Este componente permite a configuração de como deverá ser desenhado o rasto, as suas cores, o material e o seu tamanho.

A Figura 6.8 mostra a configuração deste componente depois de adicionado ao *prefab* da ervilha. As principais configurações são descritas de seguida.


Note que pode selecionar o objeto `pea` diretamente no separador *Projeto* e aceder ao *Inspetor* do *prefab*, onde poderá adicionar e configurar componentes.

As duas primeiras opções permitem definir se o rasto deve produzir sombras (como fumo) e se outros objetos devem produzir sombras sobre o próprio rasto. Quando este componente é criado, estas duas opções aparecem ativas. Segue-se uma configuração de como o rasto é calculado: de acordo com o movimento do objeto ou da sua velocidade.

Abaixo, existe um *array*, com um único elemento (mas que pode ser alargado) que corresponde ao material que será usado para renderizar o rasto. Poderá escolher qualquer material que se deseje, mas sugere-se que se use o material predefinido `Default-Particle`, que é um material com bordas difusas e adequado para este tipo de efeito. Por sua vez, a opção `Lightmap Parameters` permite definir um conjunto de propriedades sobre como o rasto poderá interagir com as propriedades globais de iluminação.

O parâmetro `Time` corresponde ao tempo de vida do rasto — quanto maior for, mais longo será. Por sua vez, a medida `Min Vertex Distance` permite definir a distância mínima necessária de movimento para que o rasto seja criado. A opção `Autodestruct` permite que o próprio objeto que produz o rasto se autodestrua, se estiver parado durante o tempo indicado no parâmetro `Time`.

Mais interessantes em termos de aparência são as propriedades seguintes. A propriedade `Width` permite definir um valor, para obter uma largura constante do rasto, ou uma curva que corresponde à variação do tamanho do rasto de acordo com o tempo de vida do rasto. Na Figura 6.8, o rasto será mais largo quando é produzido (junto à ervilha) e irá diminuir de tamanho à medida que se afasta. Esta curva permite definir alguns efeitos interessantes, como rastos que aumentam e diminuem ao longo do tempo.

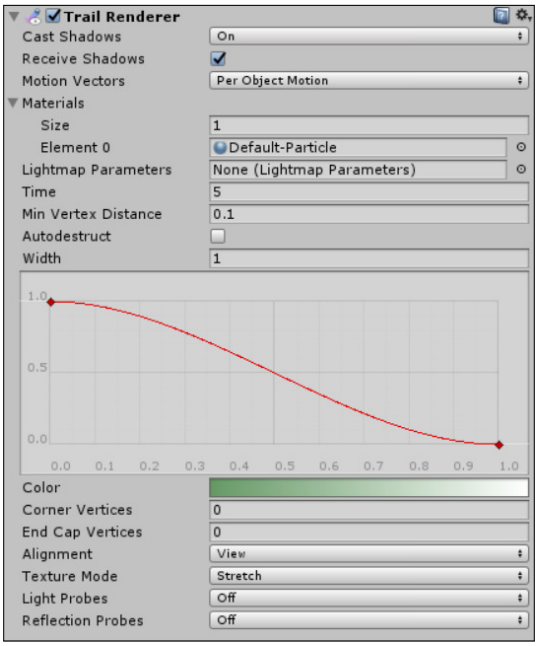


FIGURA 6.8 – Configuração do componente *Trail Renderer*

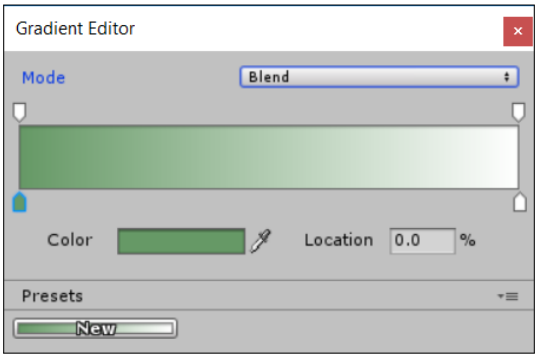


FIGURA 6.9 – Janela de edição da cor ao longo do tempo

A configuração da cor permite a definição de cores sólidas ou gradativas, como a que se mostra na Figura 6.8, campo `Color`. A alteração desta gradação é definida numa janela à parte, como a apresentada na Figura 6.9. Para cada ponto na curva, é colocada uma marca onde se pode definir uma cor diferente. No caso apresentado, apenas existem duas marcas, uma no início e outra no fim, em que foram escolhidas, respetivamente, as cores verde e branca. O modo `blend` permite definir o gradativo (caso contrário, existirá uma alteração brusca da cor).

Existem ainda outras opções que não serão aqui discutidas,visto que correspondem a configurações mais avançadas.

Feita esta configuração, as ervilhas deverão apresentar um rasto, tal como demonstrado na Figura 6.10.

Figura 6.10 – Efeito do *Trail Renderer*

6.2.3 SISTEMAS DE PARTÍCULAS

Os sistemas de partículas são recursos interessantes para simular alguns efeitos reais como o fogo, fumo ou água — situações em que o que se pretende obter é um conjunto de pequenos objetos com comportamentos similares.

Assim, um sistema de partículas corresponde a um emissor que gera um conjunto de n objetos (partículas) por segundo. Cada uma destas partículas tem uma velocidade e um tempo de vida predefinidos. É possível especificar como cada partícula vai ser renderizada e como se comportará ao longo da sua vida. Exemplos de comportamentos são a mudança da cor, de tamanho ou de velocidade, ao longo do tempo de vida. A conjugação de todos estes fatores num sistema de partículas realista é demorada e, até certo ponto, é uma arte.

Não é fácil descrever todas as funcionalidades dos sistemas de partículas. Existem muitas propriedades que se podem associar a um sistema de partículas, e é mesmo possível a criação de subsistemas de partículas. Daí que neste capítulo não se irá descrever, item a item, as configurações possíveis. A abordagem consistirá em descrever dois sistemas de partículas distintos e o modo como estes se podem implementar, indicando (e explicando) os parâmetros a configurar. Em relação aos parâmetros não usados nos dois exemplos apresentados, sugere-se que o leitor os experimente, um a um, no sentido de perceber de que forma afetam o sistema de partículas.

6.2.3.1 DEGUSTAÇÃO DO COGUMELO

O primeiro sistema de partículas que será implementado irá ser usado para dar um efeito energético sempre que a formiga come um cogumelo. Tratar-se-á de uma espiral colorida que subirá à sua volta, demonstrando o caráter refrescante e energizante do cogumelo (Figura 6.11).

FIGURA 6.11 – Sistema de partículas durante a degustação do cogumelo

Os sistemas de partículas podem ser adicionados a qualquer objeto, usando para isso o componente *Particle System*, ou pode ser criado um objeto apenas com um sistema de partículas através da opção `Particle System` no menu `GameObject`. O principal problema em adicionar um sistema de partículas a um objeto já existente é que algumas das estruturas de emissão (que serão explicadas de seguida) emitem as partículas em direção ao eixo z do objeto (*forward*) e não permitem ser rodadas, obrigando à rotação do objeto, o que não é uma solução.

Por exemplo, se tentasse criar o sistema de partículas diretamente no objeto da formiga, as partículas não iriam subir como se pretende, mas iriam sair na horizontal, na direção para a qual a formiga está virada. Para conseguir que esse sistema emita as partículas na direção desejada, poderia tentar rodar todo o objeto, mas, nesse caso, a formiga ficaria deitada, o que não seria útil.

Assim, a solução passará por criar um objeto vazio, filho do objeto referente à formiga. Para tal, deverá ser usado o menu de contexto na *Hierarquia* (botão direito do rato sobre o objeto da formiga). A vantagem deste método é que o objeto será já criado como filho da formiga e centrado no seu pivô. Este objeto será denominado `particulas_powerup` (Figura 6.12).

FIGURA 6.12 – Objeto vazio sob o objeto da formiga

O passo seguinte corresponde à adição do componente *Particle System* ao objeto acabado de criar. Como se observa parcialmente na Figura 6.13, o *Inspetor* do sistema de partículas é algo complexo.

Na parte inicial do componente estão presentes várias propriedades genéricas, imprescindíveis ao sistema de partículas. Abaixo destas, existe um conjunto de grupos de propriedades, que podem ou não ser utilizados. Na Figura 6.13 vê-se que os grupos de propriedades referentes à emissão (`Emission`) e à forma (`Shape`) estão selecionados, o que indica que as propriedades desse grupo afetam o comportamento do sistema de partículas.

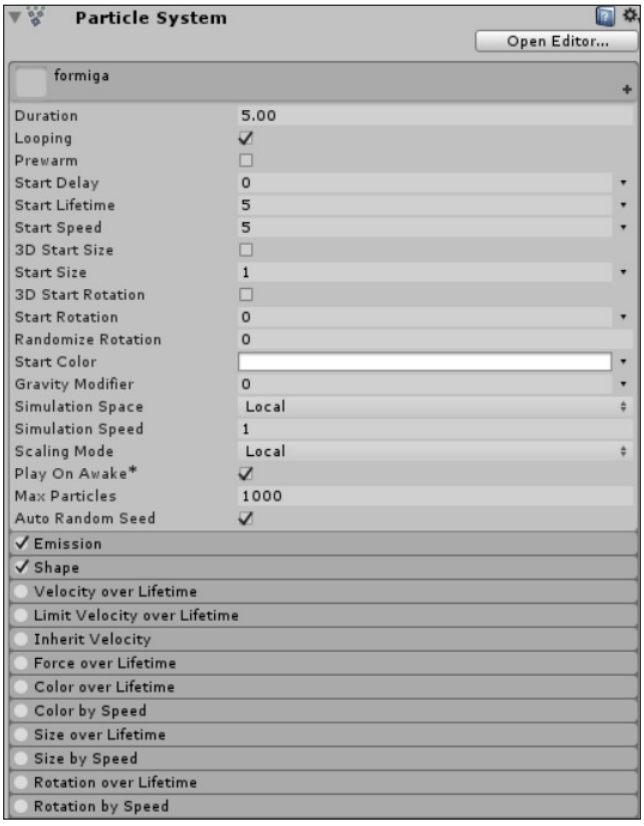


FIGURA 6.13 – Vista parcial do componente *Particle System*

Ao adicionar o componente, em princípio já são visíveis algumas partículas na *Cena*. No entanto, é bastante provável que tenham uma cor e forma estranhas, como demonstrado na Figura 6.14. Além das partículas, aparece um pequeno controlador, no canto inferior direito, que permite controlar a forma como o sistema está a ser simulado na *Cena*, de modo a poder analisar o seu comportamento.

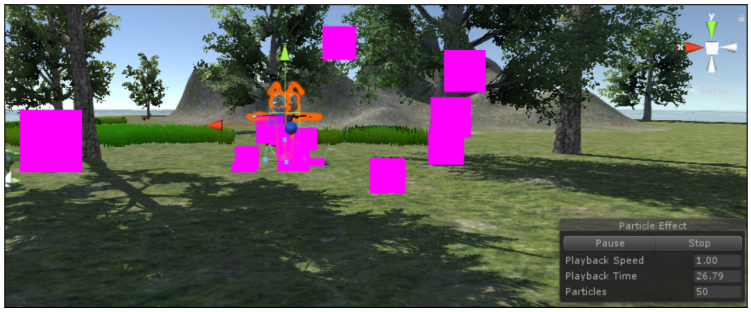


FIGURA 6.14 – Sistema de partículas inicial

A cor e forma estranhas das partículas são devidas à falta de configuração do grupo de propriedades relativas à renderização das partículas (grupo *Renderer*). Nestas propriedades é necessário alterar o material a utilizar para as partículas, devendo ser usado o `Default-Particle`, como se mostra na Figura 6.15.

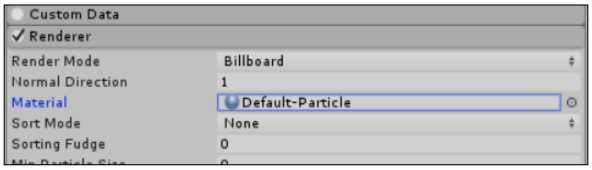


FIGURA 6.15 – Configuração do modo de renderização (grupo *Renderer*)

Posteriormente, será alterada a forma da emissão. Para isso, deverá abrir-se o grupo correspondente: *Shape*. Este grupo configura a forma (e direção) segundo a qual as partículas são geradas. Existem diversas alternativas. Uma esfera gera partículas em todas as direções; um cone gera partículas dentro de si mesmo (que se vão afastando); um círculo gera partículas num plano, em todas as direções; etc. No caso do efeito que se pretende, será usado um cone, mas adaptado, já que se irá definir o ângulo do cone (`Angle`) a 0°, o que o transformará em algo mais próximo de um cilindro.

Repare que, depois de ter escolhido a forma indicada, as partículas ainda não se movem na direção desejada, porque o cone está deitado. Para solucionar este problema, o objeto que contém o sistema de partículas deve ser rodado -90° sobre o eixo x, alterando esta informação no respetivo componente *Transform*.

Ainda no grupo de propriedades referentes à forma do emissor, será necessário configurar um conjunto extra de propriedades. O raio será mantido a 1, já que é suficiente para incluir o modelo da formiga, mas a grossura da zona de emissão (`Radius Thickness`) deverá ser alterada para 0, o que indicará que as partículas serão emitidas apenas na zona exterior do cilindro (o seu perímetro) e não em qualquer posição dentro do seu volume. O arco será mantido a 360°, uma vez que se pretende a geração de partículas a toda a volta da formiga. No entanto, para que o efeito se pareça com uma espiral, o modo de geração deve ser alterado para `loop`, que indica que as partículas devem ser geradas a toda a volta, rodando sobre o cilindro. As alternativas seriam o modo aleatório (`random`) que gera partículas em qualquer posição do cilindro, e o modo `ping-pong`, que se assemelha ao `loop`, mas que, após completar uma volta, inverte a sua direção. Ao escolher o modo `loop`, é possível definir o espaçamento entre partículas geradas (`spread`) que será mantido a 0, e a velocidade de rotação (`speed`), que será alterada para 2.5.

Para terminar a configuração da forma do emissor, será necessário alterar a propriedade `Emit from` escolhendo o valor `Base`. Esta propriedade define que as partículas devem ser emitidas a partir da base do cone.

A Figura 6.16 resume as propriedades que devem ser alteradas. Após estas configurações, as partículas já devem movimentar-se em espiral.

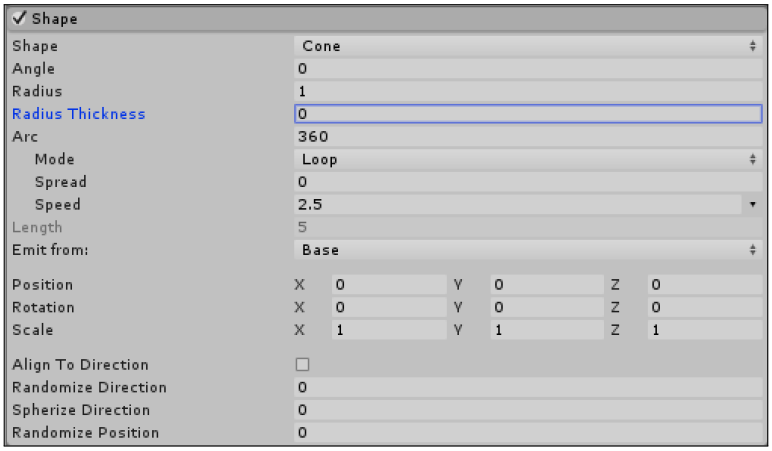


FIGURA 6.16 – Configuração da forma de geração (grupo *Shape*)

O número de partículas geradas por segundo é relativamente pequeno. A velocidade com que são criadas as novas partículas pode ser controlada no grupo de propriedades *Emission*, no qual a primeira propriedade permite controlar o número de partículas geradas por segundo, como se mostra na Figura 6.17. Este mesmo grupo permite também definir rajadas de partículas, de modo que regularmente sejam emitidas maiores quantidades.

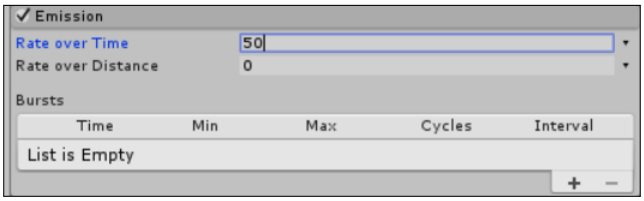


FIGURA 6.17 – Configuração da emissão de partículas (grupo *Emission*)

Nas opções gerais do sistema de partículas também será necessário configurar algumas propriedades:

- A duração (`duration`) será de 5 segundos.
- A opção de execução repetitiva (`Looping`) ficará desligada, já que a animação de partículas só será executada uma única vez por cada cogumelo comido.
- Assim que o sistema de partículas for ativado, pretende-se que estas sejam logo emitidas, pelo que o valor de `Start Delay` deve ser 0.
- Ao gerar uma partícula, o sistema irá associar-lhe um tempo de vida, findo o qual a partícula é destruída. Neste caso, o `Start Lifetime` será definido com 2 segundos.
- A opção *Start Speed* corresponde à velocidade com que as partículas se movem, quando são criadas, e terá o valor 2.5.
- Finalmente, o sistema de partículas só deverá ser executado ao comer um cogumelo, pelo que também será necessário desligar a opção `Play On Awake`. Caso contrário, o sistema começaria a gerar partículas assim que o jogo iniciasse.

A Figura 6.18 mostra estas configurações.

A próxima configuração está relacionada com a cor das partículas. É possível definir uma cor estática ou um conjunto de cores pelas quais as partículas devem passar. É possível controlar as cores dependendo da sua velocidade (`Color by Velocity`) ou do seu tempo de vida (`Color over Lifetime`) das partículas.

A Figura 6.19 mostra a configuração, fazendo a cor variar, ao longo do tempo de vida, entre o vermelho e o amarelo[26]. Do lado direito, é apresentada a configuração da gradação de cores. Usando as marcas inferiores, é possível definir as cores e usando as marcas superiores, o nível de transparência.

[26] É possível consultar algumas das imagens do livro a cores no sítio da FCA (`www.fca.pt`).

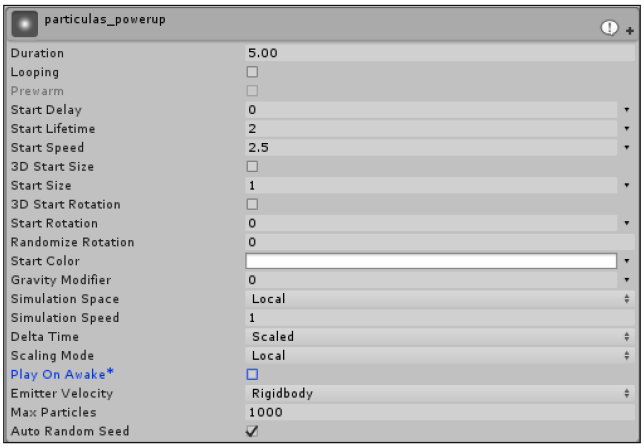


Figura 6.18 – Parâmetros gerais do sistema de partículas

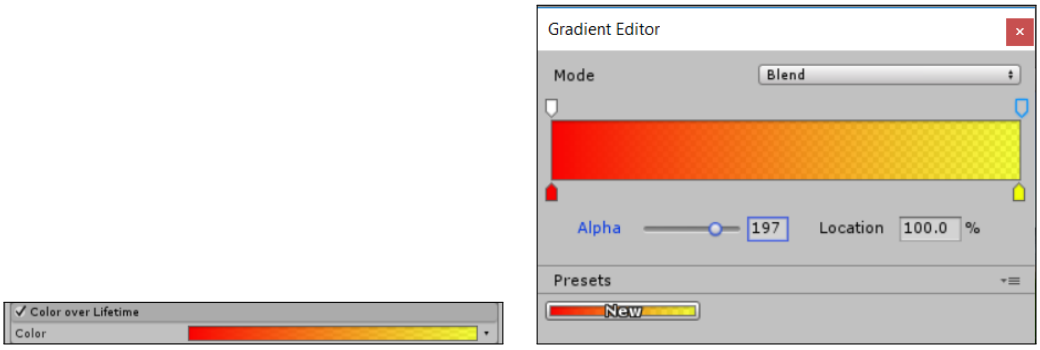


Figura 6.19 – Configuração da cor das partículas ao longo do tempo de vida

O tamanho da partículas também pode variar ao longo do tempo, seja dependendo da sua velocidade (`Size by Velocity`) ou dependendo do seu tempo de vida (`Size over LifeTime`). Mais uma vez, será usada uma variação baseada no tempo de vida da partícula, como se mostra na Figura 6.20. Repare que a configuração de variação de valores numéricos é representada pelo Unity como uma curva, o que permite grande controlo na forma como os valores irão variar ao longo do tempo. Também permite que se escolha uma curva predefinida de um conjunto de curvas disponíveis. No fundo do *Inspetor* do sistema de partículas serão apresentados todos os gráficos de todos os valores numéricos configurados. Esta sobreposição é útil para que se possam sincronizar eventos.

O que se pretende é que, ao longo do tempo, as partículas diminuam de tamanho, até desaparecerem. A forma como essa diminuição é realizada pode ser linear, ou mais sinusoidal, como a que se apresenta na Figura 6.20. Assim, fica ao cuidado do leitor a definição do modo como o tamanho da partícula irá variar.

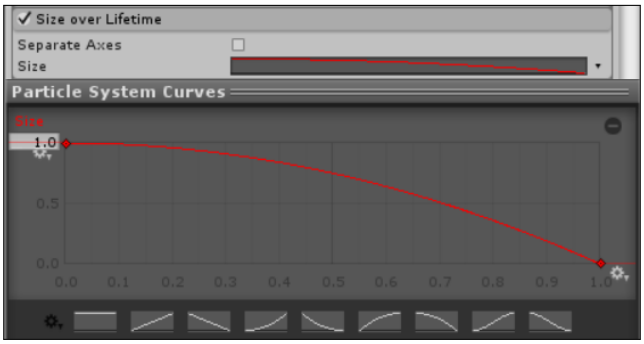


FIGURA 6.20 – Configuração do tamanho das partículas ao longo do tempo de vida

Finda a configuração do sistema de partículas, é necessária a sincronização da sua animação com a deglutição do cogumelo. Para isso, o método `DeferredPick` da *script* `AntInput` deve ser reescrito:

```
void DeferredPick() {
   GameManager.instance.PickMushroom();
   GetComponent<AntSounds>().PlayEat();
   transform.Find("particulas_powerup")
            .GetComponent<ParticleSystem>().Play();
}
```

Usou-se o método `Find` sobre um objeto do tipo `Transform`. A diferença em relação ao método com o mesmo nome da classe `GameObject` é que esta última procura um objeto com a designação indicada em toda a cena, ao passo que quando invocado sobre o `Transform`, será procurado um objeto com esse nome que seja filho do objeto ao qual esse `Transform` pertence. Por sua vez, acede-se ao componente *Particle System* e invoca-se o método `Play`, que irá iniciar a simulação do sistema de partículas.

6.2.3.2 RECEÇÃO DE DANO

O segundo sistema de partículas que será criado é bastante simples. Será invocado sempre que a formiga receber algum dano externo, seja de uma ervilha ou da aranha. Para isso, será usado o método `TakeDamage`, onde será ativado o sistema de partículas.

Este sistema de partículas vai, de alguma forma, simular algumas gotas de sangue. O 1.º passo será a construção de um novo material, o que é feito diretamente no separador *Projeto*, usando o menu `Create` → `Material`. Será criado um material novo, a que se poderá dar o nome de `blood`. Selecionando o material, surgirá o respetivo *Inspetor*, similar ao apresentado na Figura 6.21.

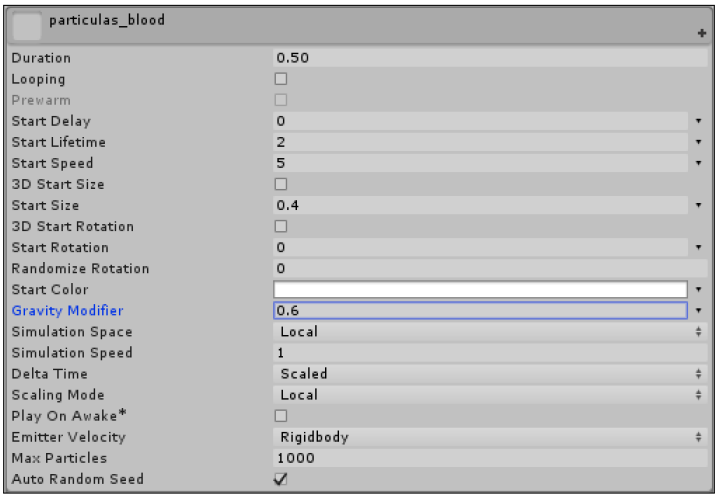


Figura 6.21 – Material `blood`

Neste caso, foi escolhida uma cor vermelho-escuro para o `Albedo` e foi usada a propriedade `Metallic` juntamente com a propriedade `Smoothness` para dar um tom mais metálico ao material.

O próximo passo consistirá na criação de um objeto vazio, sob a formiga, tal como foi feito para o sistema de partículas anterior, ao qual será adicionado o componente *Particle System*. A este objeto foi atribuído o nome `particulas_blood`, tal como se apresenta na *Hierarquia* da Figura 6.22.

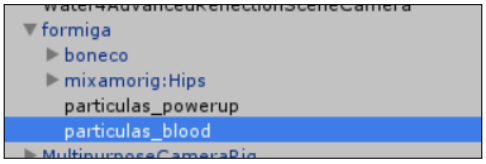


Figura 6.22 – *Hierarquia* da formiga, com o novo objeto `particulas_blood`

O efeito vai ser semelhante a uma chuva de gotas vermelhas, a sair da formiga, que cai e desaparece à sua volta. Tal como no sistema anterior, deverá ser desligada a opção `Play On Awake`, uma vez que o sistema será iniciado programaticamente. Também a opção `Looping` deverá ser desligada, já que o sistema deve emitir apenas uma vez algumas partículas.

A Figura 6.23 apresenta as propriedades gerais do sistema. Além das duas opções referidas no parágrafo anterior, é importante realçar a duração, que será de meio segundo (`Duration` $= 0.50$), o tempo de vida máximo de cada partícula (`Start Lifetime` $= 2$), o seu

tamanho original (`Start Size` $= 0.4$), e um modificador de gravidade (`Gravity Modifier` $= 0.6$). Destas opções, é relevante explicar a última. O modificador de gravidade, que inicialmente está definido com o valor 0, indica que não deverá ser aplicada qualquer força de gravidade às partículas. Se esse valor for alterado para 1, então, as partículas serão sujeitas a uma força de gravidade convencional. Deste modo, é possível criar partículas mais ou menos sensíveis à gravidade, ou mesmo com comportamentos inversos à gravidade, indicando apenas um valor real. O valor usado, 0.6, corresponde a um efeito de gravidade pouco forte, para que as partículas não caiam demasiado depressa.

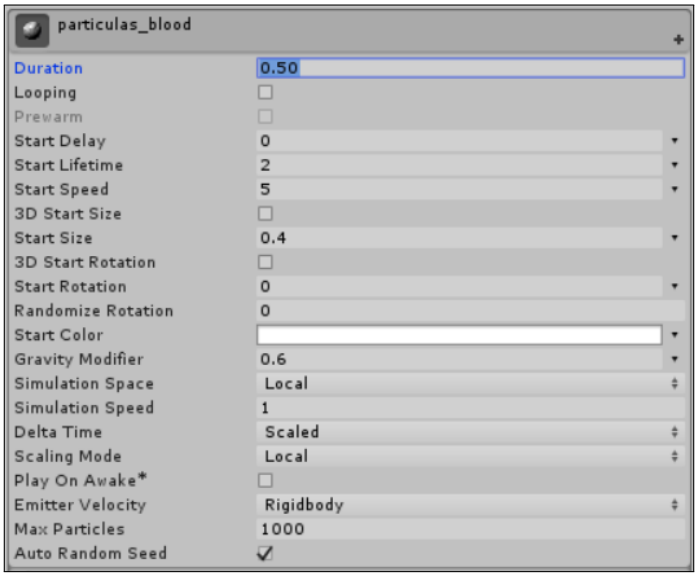


FIGURA 6.23 – Propriedades gerais do sistema de partículas

As partículas vão ter a forma de gotas ou de pequenas esferas. A forma dos objetos é definida na secção `Renderer`, tal como apresentado na Figura 6.24.

A primeira opção relevante é `Render Mode`, que deve ser alterada de `Billboard`, a opção por omissão, que simplesmente desenha uma imagem 2D no ecrã, para a opção `Mesh`, que permite que cada partícula seja um objeto tridimensional. Ao escolher esta opção, surgirá um campo `Mesh` onde deverá selecionar o modelo tridimensional a usar que, neste caso, será uma simples esfera. Também deverá ser alterada a propriedade `Material`, escolhendo o material criado anteriormente.

De seguida, será alterada a forma do emissor de partículas, configurando as propriedades do grupo *Shape*, tal como apresentado na Figura 6.25. Será usada uma forma bidimensional, um círculo, de raio 1 e grossura 0. O modo de emissão será aleatório, pelo que não se alterará qualquer outra propriedade.

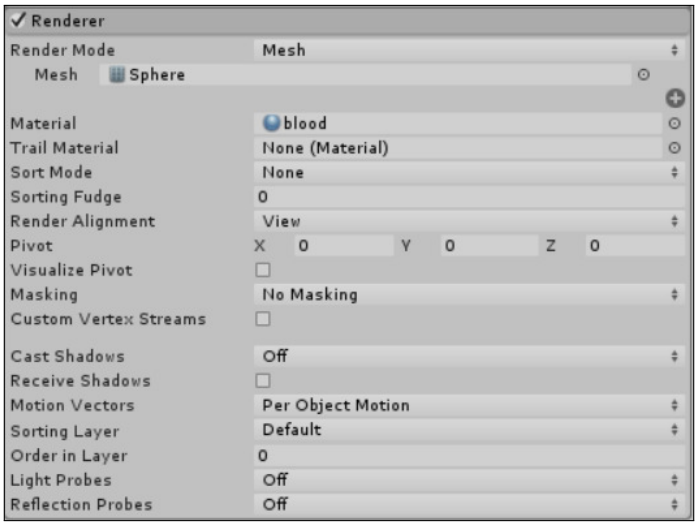


FIGURA 6.24 – Propriedades de renderização do sistema de partículas (grupo *Renderer*)

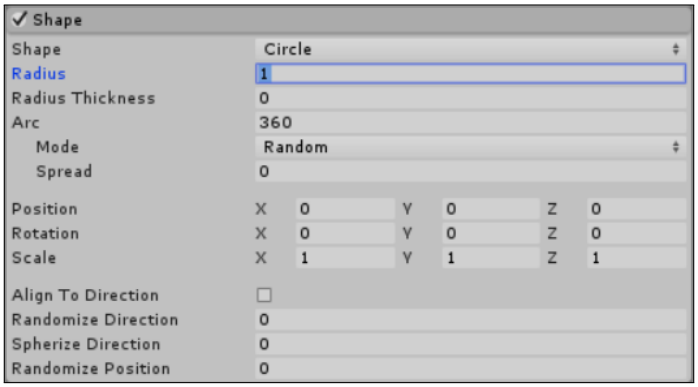


FIGURA 6.25 – Propriedades da forma do emissor de partículas (grupo *Shape*)

Infelizmente, o círculo apresenta uma rotação estranha. Para a corrigir, deverá ser alterada a rotação do eixo x do componente em 90°. Além disso, o centro do círculo deve ser mudado especificando a sua posição como $(0, 1.3, 0)$, o que o colocará um pouco mais acima, ao nível do tórax da formiga, como se vê na Figura 6.26.

Pretende-se igualmente que as partículas diminuam de tamanho à medida que vão caindo. O processo é semelhante ao usado no sistema de partículas anterior, alterando o gráfico do grupo `Size over Lifetime`. Finalmente, e porque por omissão os sistemas de partículas só emitem 10 partículas, o número de partículas a emitir deve ser alterado para 35 no grupo `Emission`.

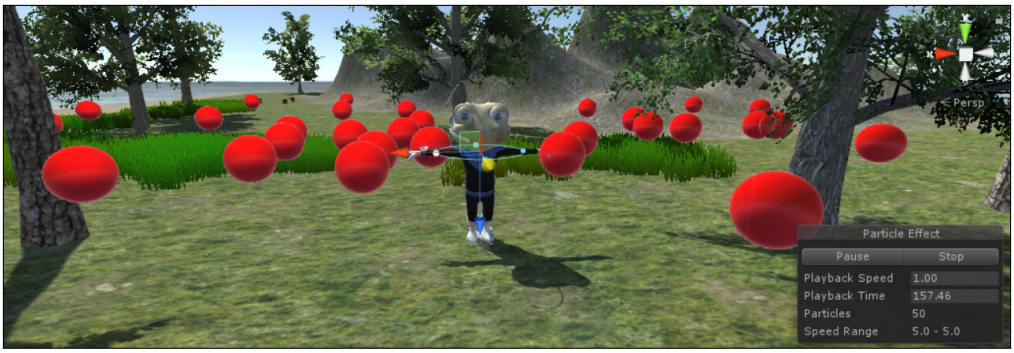


FIGURA 6.26 – Posição e rotação do emissor de partículas

Terminada a configuração do sistema de partículas, este poderá ser ativado adicionando uma linha ao código do método `TakeDamage` da *script* `GameManager`:

```
// ...
if (Energy >= 0) {
   player.GetComponent<AntSounds>().PlayArgh();
   player.transform.Find("particulas_blood")
         .GetComponent<ParticleSystem>().Play();
}
// ...
```

7 Gestão de Cenas

Um jogo de computador é mais do que a parte jogável, incluindo habitualmente um menu inicial, uma zona com as melhores classificações, ou mesmo a possibilidade de sair para o sistema operativo. Este capítulo dedica-se a estes complementos de funcionalidade.

7.1 MENU INICIAL

Quando um jogo é iniciado, além de um ou mais ecrãs com informação sobre os seus produtores e nalguns casos, uma imagem com o título, é apresentado aos jogadores um menu com algumas opções, como a possibilidade de jogar num ou diferentes modos de jogo, a configuração da qualidade gráfica, volume de som, etc., a consulta dos créditos ou das melhores classificações e a possibilidade de cancelar o jogo e voltar ao sistema operativo.

A forma mais simples de gerir diferentes zonas num jogo, ou diferentes ecrãs, é a criação de várias cenas. Nesta secção será criada uma cena para apresentar ao jogador um pequeno menu, com apenas quatro opções: iniciar o jogo, consultar as melhores classificações, alterar algumas configurações (volume do som e da música) ou sair do jogo. A Figura 7.1 mostra o menu que será construído.

7.1.1 CENÁRIO

Para começar, será criada uma cena nova, usando o menu `File` → `New Scene`. Ao criar uma nova cena, o Unity mostrará no separador *Hierarquia* dois objetos — uma fonte de luz e uma câmara — tal como aconteceu com a cena inicial, quando o projeto foi criado.

Criada a cena, e antes de se iniciar a definição do menu, será construído o cenário, adicionando um plano (`GameObject` → `3D Object` → `Plane`) e atribuindo uma textura de relva. Uma vez que não há necessidade de relevo, será usado um plano e não um terreno, dado que é bastante mais eficiente. De seguida, serão alteradas as dimensões do plano de forma manual, alterando a escala nos eixos x e z. A escala necessária variará de acordo com a posição em que o plano foi colocado e a sua relação com a posição da câmara. Pretende-se que não se veja o fim do plano, pelo que as suas dimensões devem dar a ideia de que o terreno é infinito (tal como demonstrado na Figura 7.1).

Figura 7.1 – Menu inicial do jogo

Enquanto num terreno se pode aplicar, de forma relativamente simples, uma textura, no caso do plano isso não é tão linear. O plano, tal como todos os outros objetos (com exceção do terreno), necessita da criação de um material. Esse material pode ser criado automaticamente, arrastando a textura diretamente para o plano mas, por trás, o que o Unity faz é criar um novo material de forma automática, com um nome predefinido. Para manter o projeto organizado, sugere-se a criação manual dos materiais. Assim, é possível, por exemplo, criar uma pasta denominada `Menu` onde sejam colocados os recursos que são apenas usados neste contexto. Depois de criar um novo material, ao qual se deu o nome `Grass`, atribuiu-se no campo `Albedo` a textura `GrassHillAlbedo`, que foi importada na altura da criação do terreno da cena de jogo.

Por omissão o Unity irá aplicar a textura uma única vez ao plano, esticando-a, de modo a que cubra todo o objeto. Como a textura permite a aplicação repetida, como azulejos (*tiling*), deve ser alterado o número de vezes que a textura deve ser usada no *Inspetor* do material, no campo `Tiling`, tal como é demonstrado na Figura 7.2.

Os cogumelos deverão ser colocados manualmente, tal como foram postos na cena de jogo, bastando arrastar os respetivos *prefabs*.

Resta-nos colocar a formiga. Em primeiro lugar, deverá ser arrastado o respetivo *prefab* (`ant@NeutralIdle`) para a cena, devidamente colocado sobre o plano. Embora não se pretenda que a formiga se desloque, será possível usar o mesmo *animator controller* que se usou na cena de jogo, já que não haverá uma *script* a alterar os parâmetros, portanto,

a formiga nunca deixará a sua posição de partida, ou seja, a animação em *Idle*. Deverá, então, ser arrastado o controlador `antController` para o campo `controller` do componente *Animator*.

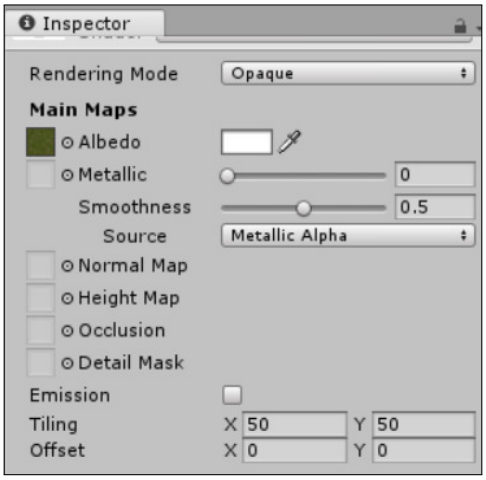


FIGURA 7.2 – Material para o plano com *tiling*

Para terminar o cenário, falta apenas alterar a *skybox*, seguindo os mesmos passos realizados no Capítulo 2: aceder à câmara, adicionar o componente *Skybox* e alterar o campo `skybox` a usar. De seguida, posicionar a câmara devidamente, de modo a ter a formiga de lado, tal como apresentado na Figura 7.1. Para o fazer, é possível movimentar diretamente a posição do objeto da câmara ou, de forma mais simples, colocar no separador *Cena* o terreno e a formiga nas posições desejadas, rodando a vista. Assim que o aspeto da cena estiver na posição desejada, deve ser selecionada a câmara, na *Hierarquia*, e usar a opção `Game Object → Align With View`.

7.1.2 INTERFACE

O título e os quatro botões do menu são construídos com base no componente *Canvas*, já usado no Capítulo 4. Para se utilizarem componentes de interface, é necessário criar um objeto com o componente *Canvas*, que irá incluir os restantes objetos de interface.


Relembre que, para poder ver o aspeto da interface que está a desenhar, deverá usar o separador *Jogo*, já que todos os objetos de interface necessitam de uma câmara com *GUI Layer* que os renderize.

O título do jogo consiste numa imagem (`antroid.png`[27]) que deverá ser importada de uma das seguintes formas: arrastá-la para a pasta do projeto, ou usando o menu `Assets` → `Import New Asset`. Depois de importada, no respetivo *Inspetor*, deverá ser alterado o seu tipo para `Sprite (2D and UI)`.

Para apresentar a imagem no ecrã, será usado um objeto do tipo *Image*, que deverá ser criado como filho do objeto `Canvas`. Este objeto tem três componentes: um *Rect Transform*, um *Canvas Renderer* e um *Image*. O 1.º passo será associar a imagem a este último componente, no campo `Source Image`.

Também será necessário configurar a sua posição no *Rect Transform*, alterando as âncoras horizontais para $Min\,X = 0.05$, e $Max\,X = 0.95$ para que o título use o ecrã inteiro, apenas com uma margem de 5% de cada lado; e as âncoras verticais para $Min\,Y = 0.7$ e $Max\,Y = 0.95$, usando assim 30% do ecrã, com uma margem superior de 5%.

No entanto, se o Unity seguir estes pedidos linearmente, pode acontecer que a imagem seja distorcida, já que o rácio entre a sua largura e altura poderá alterar. Para garantir que isso não acontece é necessário ativar a opção `Preserve Aspect`, do componente *Image*.

A Figura 7.3 apresenta o *Inspetor* da imagem com todas estas alterações.

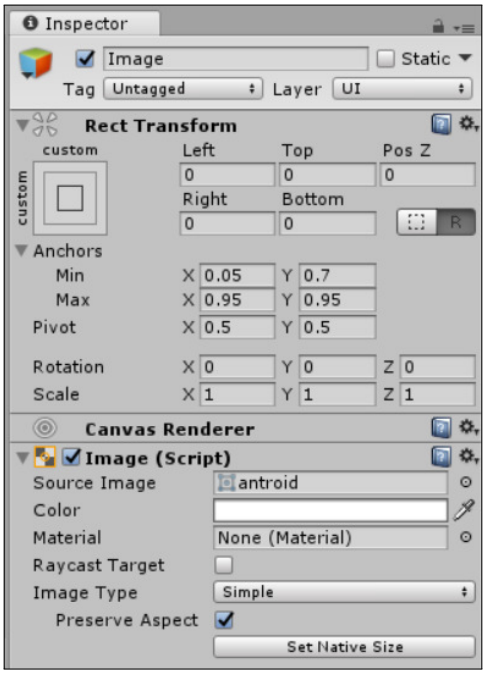


FIGURA 7.3 – Objeto `Image` e seus componentes configurados

[27] Disponível no sítio do livro (`www.fca.pt`).

Para a colocação dos botões, será criado, primeiramente, um objeto do tipo *Panel*, o que permitirá que depois de criar os botões, se consiga movimentá-los facilmente, bem como escondê-los desativando apenas o painel. Como se usarão vários painéis para vários menus, o painel acabado de criar será designado por `MainMenu-Panel`. Serão criados quatro botões dentro deste painel. A estrutura final deverá ser semelhante à apresentada na Figura 7.4 (os botões foram renomeados de acordo com o seu objetivo, para facilitar a leitura).

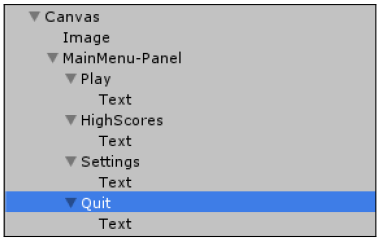


FIGURA 7.4 – *Hierarquia* de objetos dentro do `Canvas`

Depois de criado o painel, deverá ser removido o componente *Image*, já que não se pretende colocar uma imagem de fundo. A remoção de um componente é possível usando a pequena roda dentada no canto superior direito do *Inspetor* do componente em causa. O *Rect Transform* deverá ser alterado, colocando as âncoras em $Min\,X = 0.2$, $Max\,X = 0.55$, $Min\,Y = 0.3$ e $Max\,Y = 0.6$, de acordo com a Figura 7.5.

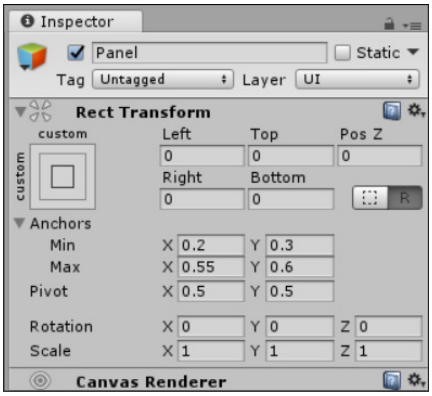


FIGURA 7.5 – Objeto `MainMenu-Panel` e respetivo *Rect Transform*

Cada um dos botões vai ocupar, verticalmente, aproximadamente $\frac{1}{4}$ do painel (o primeiro será ligeiramente mais alto), com uma separação de 5%, e, horizontalmente, a largura total do painel. A Tabela 7.1 resume as âncoras destes quatro botões, também apresentadas na Figura 7.6.

BOTÃO	MIN X	MAX X	MIN Y	MAX Y
`Play`	0.00	1.00	0.75	1.00
`HighScores`	0.00	1.00	0.50	0.70
`Settings`	0.00	1.00	0.25	0.45
`Quit`	0.00	1.00	0.00	0.20

TABELA 7.1 – Âncoras dos quatro botões

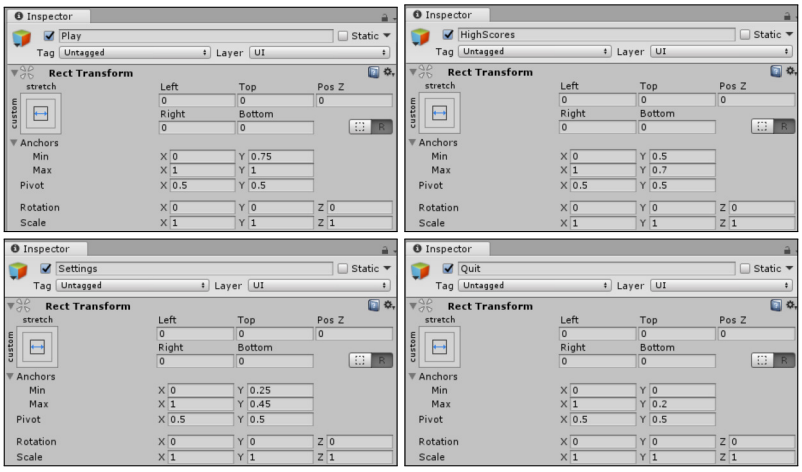


FIGURA 7.6 – Configuração dos *Rect Transform* para os quatro botões

Cada botão é um objeto composto. Por um lado, o objeto de topo inclui, além do *Rect Transform* e *Canvas Renderer*, existentes em todos os objetos relacionados com o componente *Canvas*, os componentes *Image* e *Button*. Por sua vez, o seu objeto-filho inclui os componentes *Rect Transform*, *Canvas Renderer* e *Text*, sendo que este último é o responsável pelo conteúdo textual do botão. Em cada botão será editado o seu objeto-filho, alterando o texto a ser apresentado. A Figura 7.7 (à esquerda) mostra o componente *Text* para o primeiro botão. Para o texto dos restantes botões deverá consultar a Figura 7.1.

Para tornar os botões um pouco mais simpáticos, será alterada a sua transparência e serão configurados de modo a que, quando o rato passe por cima deles, a cor de fundo mude. Como todos os botões terão este comportamento, é possível selecioná-los e configurá-los todos de uma vez.

A Figura 7.7 (à direita) apresenta a configuração de cores escolhida. A `Normal Color` é aplicada aos botões no seu estado-base. Repare-se na pequena barra sob a cor, que indica a transparência. Quanto maior for a zona branca dessa barra, menos transparente será o botão. A `Highlighted Color` é aplicada quando o rato se move sobre o botão. A `Pressed Color` é usada quando o utilizador pressiona o botão. A `Disabled Color` é usada quando o botão se encontra inativo.

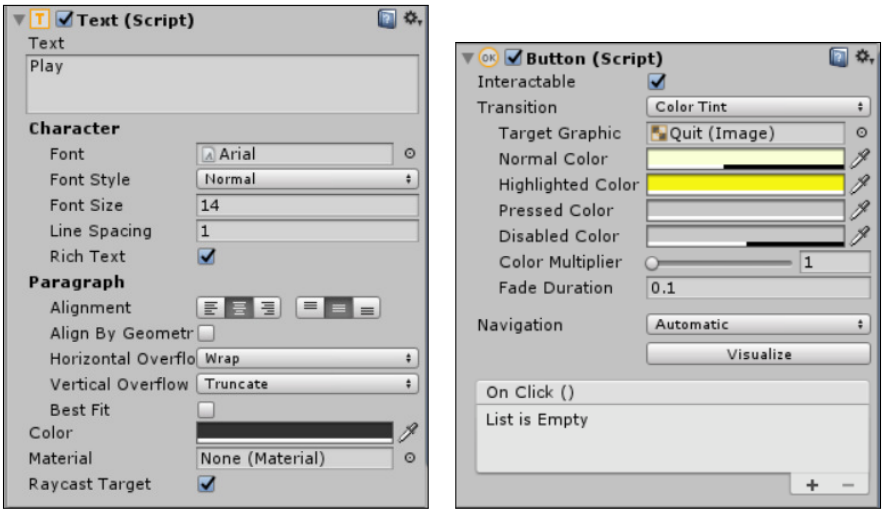


FIGURA 7.7 – Configuração do componente *Text* do botão `Play` (lado esquerdo) e das suas cores (lado direito)

No seletor de cor, apresentado na Figura 7.8, existe uma zona, no no canto inferior esquerdo, denominada `Presets`, onde é possível guardar cores escolhidas, de modo a que mais tarde possam ser reutilizadas. Como as cores aplicadas nestes botões serão úteis para os restantes menus, sugere-se que se adicionem as cores definidas à zona `Presets` para não ser necessário voltar a defini-las.

7.1.3 AÇÕES DOS BOTÕES

Criada a interface, é necessário associar comportamentos aos vários botões. O processo é simples e baseia-se na invocação de métodos definidos pelo utilizador. A *script* onde esses métodos se definem não é relevante, desde que esta esteja associada a um objeto em cena. Uma boa prática é associar a *script* ao `Canvas` que irá conter todos os objetos da interface.

Criando, então, uma *script* de nome `MainMenuCode` e associando-a ao `Canvas`, passa a ser possível definir comportamentos, como o que se apresenta de seguida, para abandonar o jogo e que irá ser associado ao botão `Quit`:

```
public class MainMenuCode : MonoBehaviour {
    public void ExitGame() {
        Application.Quit();
    }
}
```

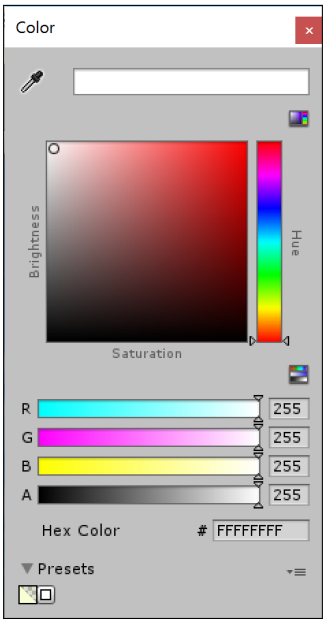


FIGURA 7.8 – Seletor de cor

Os métodos a serem invocados a partir da interface devem ser públicos. Neste caso, resume-se à invocação do método `Quit` para abandonar o jogo.

Para associar este método ao botão em questão, será usada a zona de configuração `On Click` do componente *Button*. Usando o sinal "mais" é possível adicionar uma ação para ser executada quando o botão é pressionado. Cada ação é composta por três ou quatro campos, dependendo do tipo de método a invocar, tal como se vê na Figura 7.9.

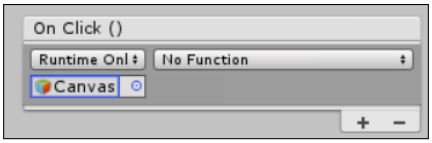


FIGURA 7.9 – Configuração de uma ação da interface

Os vários campos devem ser preenchidos por colunas. Primeiro, no canto superior esquerdo, deve ser escolhido se o evento deverá funcionar apenas em tempo de execução, ou se se pretende um botão que funcione também no editor do Unity. A opção típica é `Runtime Only`, que corresponde a um botão que só irá funcionar em tempo de execução. De seguida, e por baixo desta opção, deverá ser selecionado o objeto que contém a *script* onde se definiu a ação que se pretende executar. Uma vez que a *script* criada foi adicionada ao objeto `Canvas`, é este que deverá ser escolhido.

Após a escolha do objeto que define o método a invocar, em cima, do lado direito, surgirá a possibilidade de escolher qual o método que deverá ser executado. Veja-se na Figura 7.10 um exemplo das opções disponíveis: para cada componente do objeto existirá uma entrada no menu, que, quando selecionada, apresentará os vários métodos presentes nesse componente. Também existem alguns métodos que não foram definidos, mas que estão disponíveis. Estes são métodos definidos na classe-pai de todas as *scripts*, a `MonoBehavior`. Poderá, então, selecionar-se o método `ExitGame()` disponível no menu sob o nome da classe `MainMenuCode`.

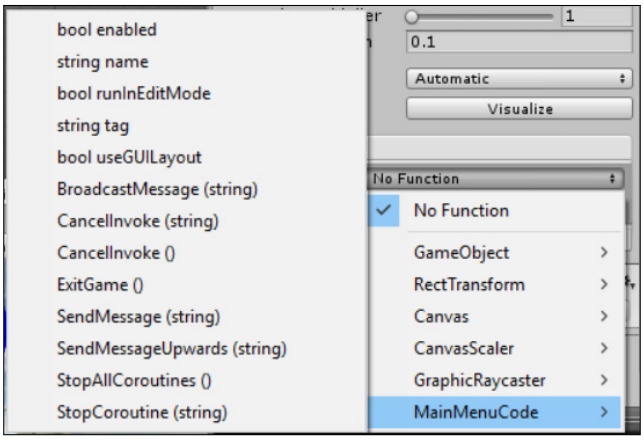


FIGURA 7.10 – Métodos disponíveis na configuração de uma ação da interface

Quando o método escolhido recebe um parâmetro, no canto inferior direito da configuração da ação, irá surgir um campo para a definição desse mesmo parâmetro.


O método `Application.Quit` não produz qualquer efeito enquanto se testa o jogo dentro do Unity. Para se poder testar devidamente o botão, será necessário criar um executável do jogo, como será apresentado no Capítulo 8.

O botão para iniciar o jogo não terá código muito mais complicado, bastando usar o método `LoadScene`, como se apresenta de seguida:

```
public void PlayGame() {
    SceneManager.LoadScene("ilha");
}
```

Note que o nome, entre aspas, corresponde ao nome da cena a carregar, devendo ser escrito tal como foi definido quando se gravou a cena de jogo pela primeira vez. Também será necessário incluir a referência à biblioteca `UnityEngine.SceneManagement`.

O Unity tem um pequeno *bug*, há vários anos, que coloca a cena um pouco mais escura quando esta é carregada usando o método `LoadScene`. Este *bug* só afeta o jogo enquanto em desenvolvimento (no editor), já que, quando é construído o seu executável, o problema não persiste. Se essa diferença incomodar, deverá aceder ao *Inspetor* referente à iluminação (`Window` → `Lighting`), desativar a opção `Auto Generate` no fundo da janela e clicar no botão `Generate Lighting`. É de salientar que este processo terá de ser executado para cada uma das cenas.

Criado este método na *script* `MainMenuCode`, passa-se à associação do método ao botão desejado, usando o mesmo processo definido para o botão `Quit`.

Infelizmente, mesmo depois de se definir o evento corretamente, o botão não irá funcionar, uma vez que será necessário indicar ao Unity quais as cenas que devem ser usadas em tempo de execução, sendo que só as cenas que foram devidamente identificadas na interface do Unity poderão ser carregadas. Usando a opção `File` → `Build Settings`, surgirá uma janela semelhante à apresentada na Figura 7.11.

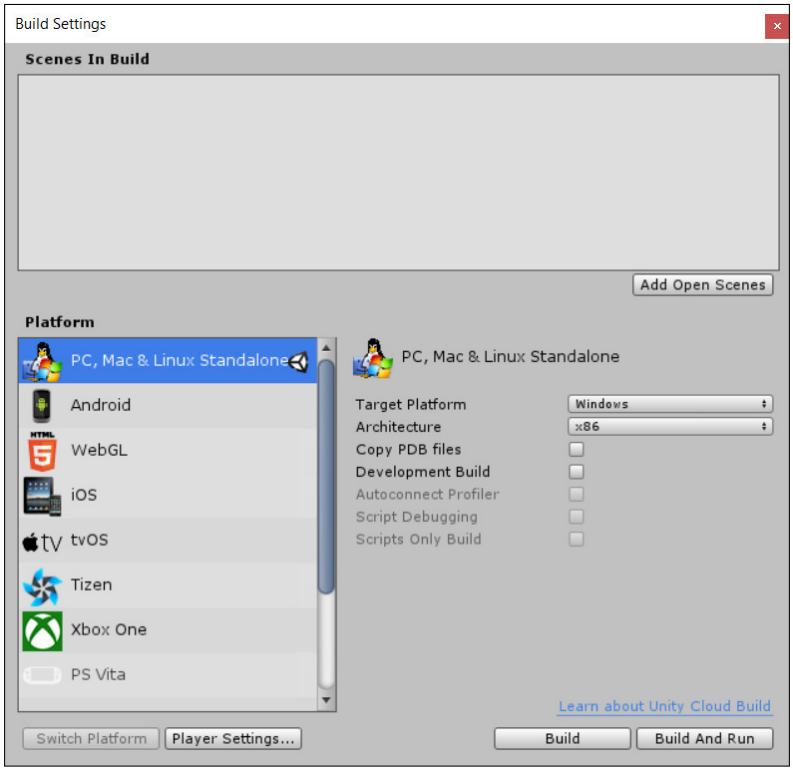


FIGURA 7.11 – Opções de compilação (`Build Settings`)

Esta janela é composta por duas zonas. Na superior, são definidas as cenas que devem ser disponibilizadas na compilação. Em baixo, são definidas propriedades sobre a arquitetura para a qual se pretende compilar o código e que serão discutidas no Capítulo 8.

Assim, na zona superior, será necessário colocar as cenas que se pretendem que estejam disponíveis durante o jogo. Atualmente, são apenas duas: a cena referente ao menu e a cena referente ao jogo. Para adicionar a cena atual, poderá usar-se o botão `Add Open Scenes`. Para adicionar cenas extra, estas poderão ser arrastadas a partir do separador *Projeto*. A Figura 7.12 mostra como deverá ficar esta lista.

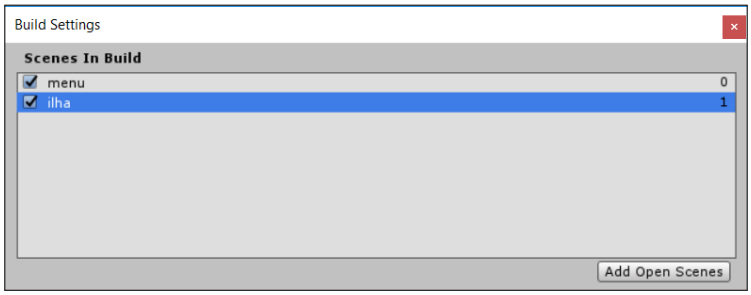


FIGURA 7.12 – Opções de compilação: cenas a disponibilizar


A ordem de apresentação das cenas não é relevante. Apenas a escolha da primeira a ser colocada na lista é importante: corresponde à cena inicial que o Unity deverá carregar ao executar o jogo.

Adicionadas as cenas, a janela pode ser fechada (usando o botão do canto superior direito da janela). A partir deste momento, o botão de início do jogo já deverá ter o comportamento esperado.

Nas secções 7.2 e 7.3 serão implementados os comportamentos dos botões `Setting` e `HighScores`, respetivamente.

7.1.4 VOLTAR AO MENU

Durante o jogo, também deverá ser possível sair, sem obrigar o jogador a morrer. Para isso, será usada uma técnica relativamente simples: assim que o jogador carregar na tecla **Esc**, iniciará uma contagem decrescente de 30 segundos. Se, durante esse tempo, o jogador voltar a carregar nessa tecla, então, confirmará que deseja abortar o jogo e voltará ao menu principal. Por outro lado, se o tempo expirar, será necessário reiniciar o processo para conseguir abandonar o jogo.

A fim de o jogador perceber o que se está a passar, sempre que o temporizador for ativado, aparecerá uma mensagem no fundo do ecrã a sugerir que volte a carregar na tecla para abandonar o jogo. Assim que o temporizador expirar, a mensagem desaparecerá.

Para a implementação deste processo será necessário voltar à cena de jogo. Depois, para que exista um objeto responsável por mostrar a mensagem indicativa ao jogador, será criado um novo objeto com o componente *Text*, dentro do objeto `Canvas` existente, de nome `ExitText` tal como demonstrado na Figura 7.13, à esquerda.

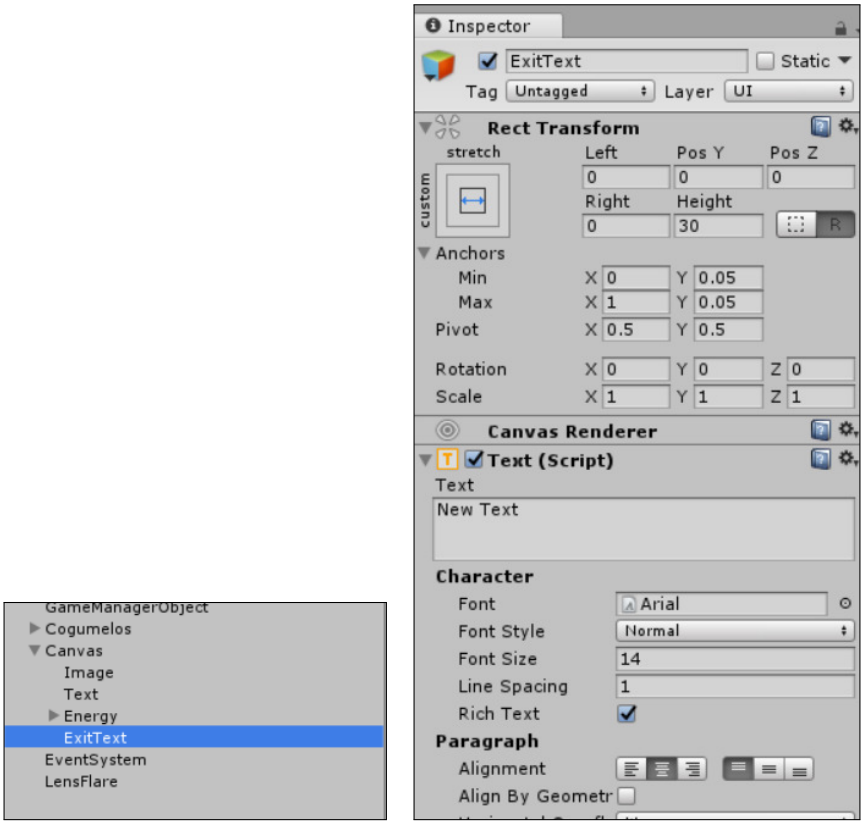


FIGURA 7.13 – *Hierarquia* do `Canvas` alterado e *Inspetor* do objeto `ExitText`

Na configuração do *Rect Transform* deste objeto, definem-se as âncoras horizontais a $MinX = 0$ e $MaxX = 1$ para que a mensagem ocupe toda a largura do ecrã. Depois, a posição vertical é definida a 5% do ecrã, colocando $MinY = MaxY = 0.05$. Para garantir que o texto aparece, é necessário que esteja definida a altura ($Height = 30$) e, para que o texto apareça centrado, será necessário que, no componente *Text*, sob a secção `Paragraph` se coloque o alinhamento (`alignment`) como centrado.

Terminada a configuração gráfica do objeto, será associada uma nova *script*, designada por `ExitToMenu`, que terá o seguinte código inicial[28]:

```
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.SceneManagement;

public class ExitToMenu : MonoBehaviour {
  Text ui;
  float timer;
```

Nesta *script* são definidas duas variáveis. A primeira, do tipo `Text`, será uma referência ao componente do mesmo tipo, associado ao objeto `ExitText`. A segunda será o contador, que irá garantir a contagem dos 30 segundos.

No método `Start` será apenas obtida a referência ao componente `Text`, que, logo de seguida, será colocado inativo, para que o texto não apareça. A propriedade `enabled` indica se o componente deve, ou não, ser executado. O estado interno deste componente (ativo ou inativo) funcionará para saber se o jogador já carregou na tecla **Esc**.

```
  void Start() {
     ui = GetComponent<Text>();
     ui.enabled = false;
  }
```

O método `Update` é um pouco mais longo, mas não é mais complicado:

```
  void Update() {

     if (ui.enabled) {
        if (timer < 0) {
           ui.enabled = false;
        }
        else {
           if (Input.GetKeyDown(KeyCode.Escape)) {
              SceneManager.LoadScene("menu");
           } else {
               timer -= Time.deltaTime;
               ui.text = "Press Esc again to exit (" +
                           ((int)timer) + " secs remaining)";
           }
        }
     }
```

[28] Para poder usar os componentes de interface numa *script* terá de importar a biblioteca `UnityEngine.UI`.

```
    else {
        if (Input.GetKeyDown(KeyCode.Escape)) {
            timer = 30;
            ui.enabled = true;
        }
    }
}
```

A primeira condição irá verificar se o contador já está ativo. Em caso afirmativo, será verificado se o tempo já expirou, situação na qual o contador é de novo escondido, sendo desativado.

Por sua vez, se o contador está ativo, mas o tempo não expirou, será validado o estado da tecla **Esc**. Se o utilizador a pressionou, então, já é a segunda vez que o faz, pelo que o jogo deverá terminar e carregar o menu. Se, no entanto, o utilizador não pressionou o **Esc**, o temporizador é decrementado e o texto a apresentar ao utilizador é atualizado (fazendo uma conversão do tempo decorrido para inteiros, para melhor legibilidade do jogador).

Finalmente, se o contador não está ativo, é apenas validado se o jogador pressionou a tecla **Esc**, situação na qual o contador deve ser reiniciado aos 30 segundos e o texto devidamente ativado.

7.2 CONFIGURAÇÕES DE VOLUME

Das opções disponíveis em jogos, as mais comuns são o controlo do volume da música de fundo e dos efeitos sonoros, pelo que se irá construir uma interface semelhante à apresentada na Figura 7.14.

FIGURA 7.14 – Menu de configuração do volume da música e dos efeitos sonoros

A interface será ativada através do botão `Settings` definido anteriormente. Esta secção descreve a implementação desta funcionalidade dividindo essa tarefa em três partes: a definição da estrutura gráfica da interface (secção 7.2.1); o código necessário para transitar do menu inicial para as opções e respetivo código de configuração dos volumes (secção 7.2.2); e finalmente, o código, durante o jogo, que garante que as opções de volume definidas serão tidas em conta durante o jogo (secção 7.2.3).

7.2.1 INTERFACE

A interface será um painel semelhante ao criado para o menu principal. Na verdade, para poupar algum tempo, poderá copiar-se toda a estrutura do painel criado anteriormente para um novo, a que se chamará `Prefs-Panel`. O botão `Quit` poderá ser aproveitado, mudando-se o nome para `Ok` e alterando o texto a apresentar. Os restantes botões devem ser apagados. Por sua vez, serão criados dois painéis dentro do painel `Prefs-Panel` que se chamarão `Effects` e `Music`. Cada um destes terá como filhos um objeto com o componente *Text* e outro com o componente *Slider*. A Figura 7.15 mostra a hierarquia de objetos criada para este menu.

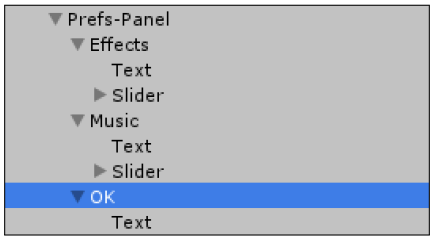


FIGURA 7.15 – Hierarquia de objetos para o menu de preferências


Para que possa ver devidamente o painel que está a configurar, poderá desativar o painel do menu principal, usando o seletor junto ao nome do painel, no seu *Inspetor*. Dada a forma como estes painéis irão ser manipulados, é irrelevante se os mantém ativos ou inativos.

O *Rect Transform* do painel `Prefs-Panel` será semelhante ao painel do menu principal. Por sua vez, as âncoras dos *Rect Transform* dos restantes objetos estão apresentadas na Tabela 7.2. De realçar que os elementos *slider*, por terem os pivôs verticais iguais, deverão ter uma altura predefinida (`Height` = 20).

Deverá, também, alterar o texto apresentado nos objetos `Effects/Text` e `Music/Text`, que poderá ser definido, respetivamente, como `Effects` e `Music`. Além disso, o alinhamento do texto deve ser colocado ao centro, quer horizontalmente, quer verticalmente.

OBJETO	MIN X	MAX X	MIN Y	MAX Y
`Effects`	0.00	1.00	0.70	1.00
`Effects/Text`	0.05	0.35	0.00	1.00
`Effect/Slider`	0.30	0.95	0.50	0.50
`Music`	0.00	1.00	0.35	0.65
`Music/Text`	0.05	0.35	0.00	1.00
`Music/Slider`	0.30	0.95	0.50	0.50
`Ok`	0.00	1.00	0.00	0.30

TABELA 7.2 – Âncoras para os vários objetos do menu de preferências

Cada *slider* é composto por vários objetos e vários componentes, de acordo com a estrutura apresentada na Figura 7.16.

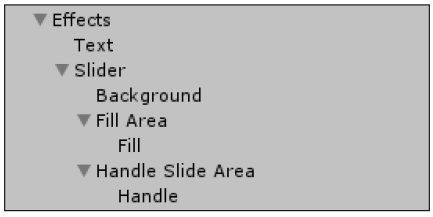


FIGURA 7.16 – Estrutura de objetos de um *slider*

O objeto de topo (`Slider`) tem o componente com o mesmo nome, apresentado na Figura 7.17. Das várias opções disponíveis, salientam-se as seguintes:

- A opção `Interactable` indica se é possível o jogador alterar o valor apresentado pelo *slider*, ou se este irá funcionar apenas para apresentar informação ao utilizador (e, portanto, permanecerá bloqueado).
- De seguida, é possível definir de que modo será visualizada a transição de valores. Por omissão é usada a mudança de cor. Outras opções são a mudança de uma textura ou o uso de uma animação.
- Mais abaixo encontra-se um grupo de opções para a configuração de cores. A opção `Target Graphic` indica qual o objeto que será sujeito à transição de cores. As quatro cores seguintes são: a cor do manípulo do *slider* (`Normal Color`); a cor do manípulo quando se encontra selecionado (`Highlighted Color`); a cor usada quando o jogador altera a posição do manípulo (`Pressed Color`); e a cor usada quando o *slider* se encontra inativo (`Disabled Color`).
- Existem duas referências a dois objetos, o `Fill Rect` e o `Handle Rect`, que são dois dos filhos do `Slider` e correspondem à zona que demonstra o preenchimento (tipicamente à esquerda do manípulo) e ao próprio manípulo.

- De seguida, é possível indicar se o *slider* irá funcionar da esquerda para a direita, ou vice-versa, quais os valores-limite, se deverão ser usados valores inteiros ou reais e, finalmente, qual o valor por omissão.

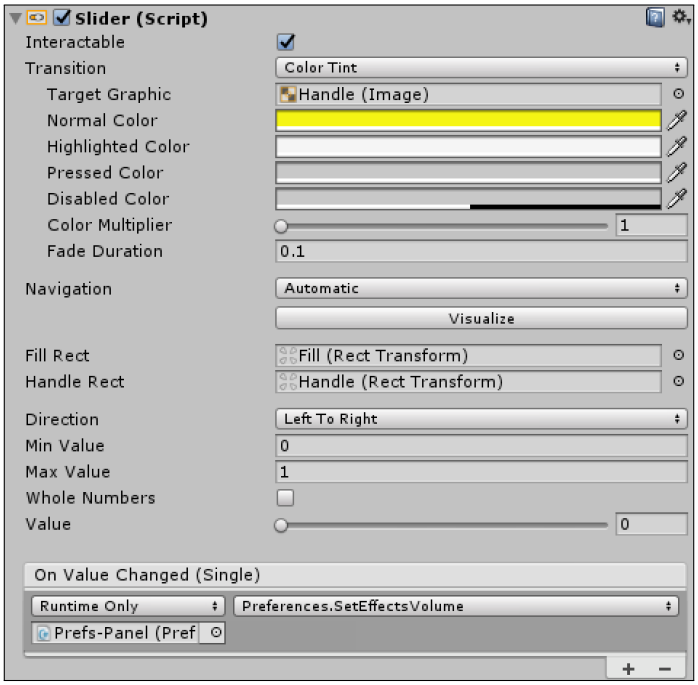


FIGURA 7.17 – Componente *Slider*

O objeto `Background` consiste numa imagem que será usada como fundo do *slider* (por onde o manípulo irá deslizar). Deverá usar este objeto para configurar a cor de fundo, se assim o entender.

Já o objeto `Fill Area` contém uma imagem (no seu filho `Fill`) que é a zona preenchida do *slider* (zona à esquerda do manípulo).

Finalmente, o objeto `Handle Slide Area` contém uma imagem (no seu filho `Handle`) que corresponde ao manípulo. Note que a cor desta imagem será alterada pelo componente *Slider*, pelo que não a deve alterar diretamente.

7.2.2 CONTROLO DA INTERFACE

Esta interface tem um conjunto de particularidades. Em primeiro lugar, quando se seleciona o botão do menu principal correspondente às configurações, deverá desaparecer

o painel do menu inicial e surgir o painel do menu de configurações, e ao usar o botão `Ok` do menu de preferências, regressar ao menu inicial. Além disso, no menu de preferências, deverá ser possível alterar o volume e permitir ao jogador ouvir alguns exemplos de sons, de modo a validar a escolha.

7.2.2.1 LIGAÇÃO AO MENU PRINCIPAL

Como ao longo do desenho da interface surgirão outros painéis, a *script* `MainMenuCode` será alterada de forma a que, mais tarde, seja simples de alterar e adicionar suporte aos novos painéis.

Serão criadas variáveis para guardar referências aos vários painéis. Embora neste momento só existam dois, pode aproveitar-se e adicionar já todas as variáveis que serão necessárias:

```
public class MainMenuCode : MonoBehaviour {

   public GameObject MainManuPanel;
   public GameObject PrefsPanel;
   public GameObject NoScorePanel;
   public GameObject AddScorePanel;
   public GameObject HighScoresPanel;

   void Awake() {

      MainManuPanel.SetActive(false);
      PrefsPanel.SetActive(false);
      // AddScorePanel.SetActive(false);
      // HighScoresPanel.SetActive(false);
      // NoScorePanel.SetActive(false);

      MainManuPanel.SetActive(true);
   }
```

No método `Awake` serão desativados todos os painéis disponíveis, usando o método `SetActive`. Este método permite manipular o seletor à esquerda de cada objeto, no seu respetivo *Inspetor*. Note-se que se comentaram três linhas, referentes a três painéis que ainda não foram preparados. À medida que isso for feito, os comentários destas linhas deverão ser removidos. A última linha ativa o menu principal. A razão para se desativar e voltar a ativar é simples: mais tarde, será adicionado algum código e nem sempre será o menu principal a ser ativado. No editor, será ainda necessário associar os objetos `MainMenu-Panel` e `Prefs-Panel` às respetivas variáveis da *script* `MainMenuCode`.

O método que deverá ser associado, no painel do menu principal, ao evento de clique do botão "clicar no botão para ativar o menu de configurações" será implementado do seguinte modo:

```
public void ShowPrefs()    {
   MainManuPanel.SetActive(false);
   PrefsPanel.SetActive(true);
}
```

Por sua vez, ao botão `Ok` no menu de preferências, será associado um método inverso, para regressar ao menu principal:

```
public void CancelButton() {
   PrefsPanel.SetActive(false);
   MainManuPanel.SetActive(true);
}
```

7.2.2.2 DEFINIÇÃO DO VOLUME

O Unity inclui uma classe, denominada `PlayerPrefs`, que permite armazenar, de forma não volátil, propriedades do jogo, entre várias execuções. Deste modo, o programador não tem de se preocupar em arranjar um ficheiro onde guardar essa informação, o que seria especialmente trabalhoso ao programar para diferentes arquiteturas, já que diferentes sistemas operativos usam pastas diferentes para o armazenamento temporário de ficheiros.

Esta classe é implementada como um dicionário, que associa valores a propriedades, valores esses que que podem ser inteiros, reais, ou *strings*. O método `HasKey` valida a existência de determinada chave, os métodos `SetFloat`, `SetInt` e `SetString` permitem armazenar propriedades do tipo real, inteiro ou *string*, e os métodos `GetFloat`, `GetInt` e `GetString` permitem obter, de volta, esses valores. Finalmente, o método `Save` permite forçar a que os valores sejam armazenados em determinado momento (caso contrário, o próprio Unity irá tratar disso quando lhe parecer necessário).

No início do método `Awake` da *script* `MainMenuCode`, será validada a existência da definição do volume dos efeitos sonoros e da música de fundo. Se não estiverem definidos, serão armazenados valores por omissão:

```
void Awake() {
   if (!PlayerPrefs.HasKey("MusicVolume"))
      PlayerPrefs.SetFloat("MusicVolume", 0.75f);
   if (!PlayerPrefs.HasKey("EffectsVolume"))
      PlayerPrefs.SetFloat("EffectsVolume", 0.95f);
   // ...
```

Quando o utilizador alterar o volume dos efeitos sonoros ou da música, convém que exista algum *feedback* sonoro que lhe permita validar se o volume está adequado. Para a música, será simples: será adicionado um *Audio Source* ao objeto `Canvas`, que reproduzirá uma música de fundo (a mesma que é reproduzida durante o jogo, tal como configurado na Figura 6.4) e, ao alterar o volume da música, a própria música irá ser reproduzida com um volume diferente.

Ao painel `Prefs-Panel` será adicionada uma nova *script* para manipular os *sliders*, e alterar o volume da música:

```
using UnityEngine;
using UnityEngine.UI;

public class Preferences : MonoBehaviour {
    void Awake() {
        float MusicVolume = PlayerPrefs.GetFloat("MusicVolume");
        float EffectsVolume = PlayerPrefs.GetFloat("EffectsVolume");

        transform.Find("Effects/Slider")
                 .GetComponent<Slider>().value = EffectsVolume;
        transform.Find("Music/Slider")
                 .GetComponent<Slider>().value = MusicVolume;
    }
```

Quando o menu for ativado, o método será executado e os volumes atuais serão obtidos das preferências do utilizador. Posteriormente, será usada a propriedade `value` do componente *Slider* para colocar o manípulo do seletor na posição correspondente.

Quando o utilizador usa o seletor, será invocado o seguinte método para atualizar o volume da música:

```
public void SetMusicVolume (float volume) {
    PlayerPrefs.SetFloat("MusicVolume", volume);
    transform.GetComponentInParent<AudioSource>().volume = volume;
}
```

O método recebe por parâmetro o volume a ser definido e guarda-o nas preferências. Além disso, procura um componente do tipo `AudioSource` em todos os seus pais. O primeiro que irá encontrar é o que se encontra associado ao `Canvas` e, sobre este componente, é alterado o volume atual da música.

Tal como foi alterado o volume ao usar o seletor de configuração do volume, também deveremos, ao ser apresentado o menu, quando se inicia o jogo, adaptar o volume da música de fundo ao volume solicitado pelo jogador. Assim, na *script* `MainMenuCode`, no

método `Awake`, será adicionada uma linha, logo após a configuração dos valores de volume por omissão:

```
void Awake() {
    if (!PlayerPrefs.HasKey("MusicVolume"))
       PlayerPrefs.SetFloat("MusicVolume", 0.75f);
    if (!PlayerPrefs.HasKey("EffectsVolume"))
       PlayerPrefs.SetFloat("EffectsVolume", 0.95f);

    GetComponent<AudioSource>().volume =
                           PlayerPrefs.GetFloat("MusicVolume");
    // ...
```

Para que a configuração do volume da música funcione, falta apenas adicionar a referência ao método `SetMusicVolume` no *slider* respetivo, como se mostra na Figura 7.18.

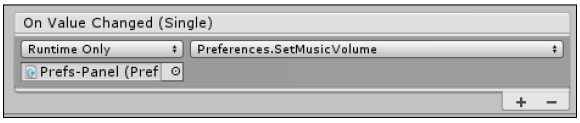


FIGURA 7.18 – Método associado ao evento `On Value Changed`

Para a configuração do volume dos efeitos sonoros, será feito algo semelhante. No entanto, não haverá um efeito sonoro a ser reproduzido a todo o momento, mas apenas quando o jogador alterar o volume atual, pelo que será adicionado um outro *Audio Source*, mas neste caso ao painel `Prefs-Panel`, que deverá ser configurado com um dos sons usados durante o jogo (por exemplo, o `Argh`) e deverá ter as opções `Play On Awake` e `Loop` desligadas (Figura 7.19).

Na *script* `Preferences`, será implementado o método `SetEffectsVolume`, que será associado ao respetivo *slider*, tal como o método `SetMusicVolume`, definido anteriormente:

```
public void SetEffectsVolume(float volume) {
    PlayerPrefs.SetFloat("EffectsVolume", volume);
    GetComponent<AudioSource>().volume = volume;
    GetComponent<AudioSource>().Play();
}
```

Neste método guarda-se a informação sobre o volume desejado nas preferências. De seguida, atualiza-se o volume do *Audio Source* acabado de criar e reproduz-se o som de teste, para que o jogador o oiça e valide o volume.

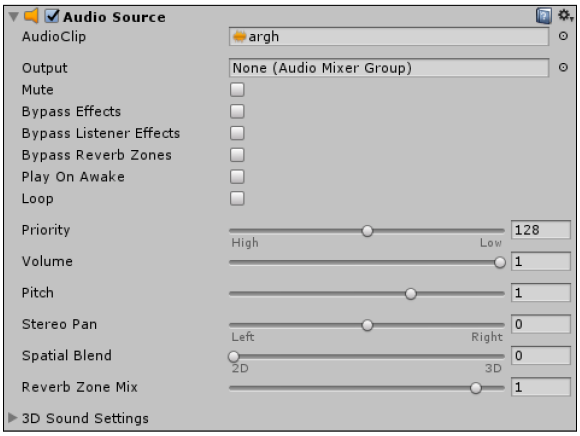


FIGURA 7.19 – *Audio Source* associado ao `Prefs-Panel`

7.2.3 CONTROLO DO VOLUME NO JOGO

Para que todo este trabalho para um menu de configuração do volume de som faça sentido, é necessário que o próprio jogo tenha essas propriedades em conta. Nesta secção serão alteradas algumas *scripts* usadas na cena de jogo, de modo a que, ao reproduzirem som, tenham em atenção o volume configurado.

A implementação tirará partido da *script* `GameManager`, visto que, por ser um *Singleton*, é facilmente acessível a partir de qualquer outra *script*.

Em primeiro lugar, serão criadas duas variáveis, estáticas e públicas, para armazenar informação sobre o volume da música e dos efeitos sonoros:

```
public class GameManager : MonoBehaviour {
    public static float effectsVolume;
    public static float musicVolume;
    // ...
```

Por sua vez, no método `Awake`, a informação sobre os volumes é obtida das preferências do utilizador e armazenada nas variáveis:

```
void Awake() {
    effectsVolume = PlayerPrefs.GetFloat("EffectsVolume");
    musicVolume = PlayerPrefs.GetFloat("MusicVolume");
    // ...
```

Estando os volumes disponíveis, pode ser adicionado o código que os irá utilizar. Ainda na *script* `GameManger`, no método `Start`, pode aceder-se à câmara principal e, daí, ao

componente *Audio Source* criado anteriormente com a música de fundo, para alterar o seu volume:

```
void Start () {
  Camera.main.GetComponent<AudioSource>().volume = musicVolume;
  // ...
```

Esta linha de código tira partido do facto de se ter associado o componente *Audio Source* à câmara e do facto de o componente *Camera* ter um campo estático (`main`) que permite aceder à câmara principal que está a ser usada em determinado momento.

Fica a faltar a alteração do volume de cada efeito sonoro. Felizmente, foram criadas *scripts* independentes para tratar dos sons da formiga e da aranha, portanto, é relativamente fácil encontrar o código (quase todo) que necessita de alteração.

Para os sons da formiga, será alterada a *script* `AntSounds`, adicionando uma linha ao método `Start`, que configura o volume de acordo com o desejado:

```
void Start() {
   audioSource = GetComponent<AudioSource>();
   audioSource.volume = GameManager.effectsVolume;
}
```

Por sua vez, para a aranha, será alterada a *script* `SpiderSounds`, usando uma abordagem semelhante à anterior:

```
void Start() {
   soundSource = GetComponent<AudioSource>();
   soundSource.volume = GameManager.effectsVolume;
}
```

Finalmente, será também necessário alterar o volume do som produzido pela planta, ao disparar as ervilhas. Este comportamento está implementado na *script* `ShootPea`, que levará também uma linha extra no método `Start`:

```
void Start () {
   Player = GameObject.FindGameObjectWithTag("Player").transform;
   SpawnPoint = transform.Find("SpawnPoint");
   GetComponent<AudioSource>().volume = GameManager.effectsVolume;
}
```

7.3 TOP DE PONTUAÇÕES

Por fim, será criado um sistema de pontuações. Para tal, vão ser criados vários novos painéis, na cena do menu principal: um para o jogador adicionar o seu nome, a fim de aparecer na lista das melhores pontuações; outro para indicar ao jogador que, infelizmente, não conseguiu atingir os valores mínimos necessários para aparecer nas melhores pontuações; e outro para mostrar a lista dos jogadores com as melhores classificações.

Para que isto seja possível, é necessário definir o que é uma pontuação e o que é uma boa pontuação. Uma possibilidade é indicar o número de cogumelos apanhados: quanto mais cogumelos, maior a pontuação. O único dilema é que, se mais que um jogador conseguir apanhar todos os cogumelos, não há nada que os distinga. Para que haja alguma distinção, será usado o tempo total despendido para obter os cogumelos.

Então, a pontuação de um jogador será um par $p_i = \langle n, t \rangle$, em que n é o número de cogumelos apanhados e t o tempo demorado. Assim, a pontuação de um jogador p_i é *melhor* do que a pontuação de um jogador p_j se:

$$p_i < p_j \Leftrightarrow \begin{cases} t_i < t_j & \text{se } n_i = n_j \\ n_i > n_j & \text{caso contrário} \end{cases}$$

De seguida, serão descritos todos os passos para implementar este sistema de pontuações, que inclui um grande conjunto de pormenores. De início, serão alteradas partes do código da *script* `GameManager` de modo a que o resto da implementação seja possível (secção 7.3.1). Seguidamente, serão definidas as partes gráficas da interface dos três painéis necessários para a gestão das melhores classificações (secção 7.3.2). Finalmente, será implementado todo o código necessário para colar os vários painéis e armazenar as melhores classificações como preferências do utilizador (secção 7.3.3).

7.3.1 PRELIMINARES

Neste momento, o código implementado não contabiliza o tempo de jogo. Felizmente, essa é uma tarefa simples, bastando definir uma nova variável na *script* `GameManager`:

```
public class GameManager : MonoBehaviour {
    public static float effectsVolume;
    public static float musicVolume;

    public float GameTime;
    // ...
```

Esta variável terá de ser inicializada a 0 no método `Start`:

```
void Start () {
   // ...
   player = GameObject.FindGameObjectWithTag("Player");
   GameTime = 0;
}
```

No método `Update` esta variável terá de ser incrementada com o tempo decorrido desde a última vez que este mesmo método foi executado:

```
void Update () {
   GameTime += Time.deltaTime;
   Energy -= Time.deltaTime;
   // ...
```

Depois desta alteração, a *script* `GameManager` tem toda a informação necessária para calcular a classificação: o tempo de jogo (variável `GameTime`) e o número de cogumelos comidos (`PickedMushrooms`). No entanto, sempre que é carregada uma nova cena, o Unity destrói todos os objetos na cena atual, pelo que, se não se acautelar essa situação, os dados são perdidos ao carregar a cena do menu principal.

A solução passa por usar o método `DontDestroyOnLoad`, que indica que determinado objeto não deverá ser destruído durante o carregamento de uma nova cena.

 Note que, depois de indicar que um objeto não deve ser destruído, este jamais será destruído, a não ser que o indique explicitamente. Tal pode constituir um problema porque, se voltar a carregar a cena original, será criada uma nova instância.

Então, o método `Awake` da *script* `GameManager` passará a:

```
void Awake() {
   effectsVolume = PlayerPrefs.GetFloat("EffectsVolume");
   musicVolume = PlayerPrefs.GetFloat("MusicVolume");

   if (instance == null) {
      instance = this;
      DontDestroyOnLoad(gameObject);
   }
   else Destroy(gameObject);
}
```

Antes de se passar ao desenho da interface, será ainda adicionado o código necessário para que, do jogo, se volte à cena do menu, quando a formiga morre ou captura todos

os cogumelos. Essas duas situações podem ser validadas facilmente no método `Update`, como se mostra de seguida:

```
void Update () {
    GameTime += Time.deltaTime;
    Energy -= Time.deltaTime;

    if (Energy < 0 && !dead) {
        dead = true;
        player.GetComponent<AntSounds>().PlayDying();
    }
    else if (dead || PickedMushrooms == TotalMushrooms) {
        SceneManager.LoadScene("menu");
    }
}
```

Ou seja, bastará que a formiga esteja morta ou que o número de cogumelos apanhados seja o esperado para que seja carregada a cena do menu.

7.3.2 INTERFACE

A interface desenhada nesta secção vai seguir a mesma lógica do menu principal e do menu de preferências. Serão adicionados três novos painéis: um que mostra a pontuação do jogador e lhe solicita o nome, para o adicionar à lista das melhores classificações; outro que também mostra a pontuação do jogador, mas não permite adicionar o nome (quando a pontuação obtida não atingiu os valores necessários para entrar na lista de melhores jogadas); e outro que mostra as pontuações das 10 melhores jogadas, juntamente com os nomes dos jogadores que obtiveram essas classificações.

Todos estes painéis serão criados dentro do objeto `Canvas` já existente e poderão ser obtidos, tal como se fez anteriormente, copiando o painel do menu principal e alterando os diferentes elementos.

A Figura 7.20 apresenta a interface de adição de pontuações à lista das melhores classificações. Esta interface é baseada num painel, de dimensões semelhantes às dos outros dois, designado por `AddScore-Panel`, composto por quatro objetos-filhos diretos: um outro painel (para a parte superior, com a classificação do jogador), um com o componente *Input Field* e dois botões — um para confirmar a adição da classificação e outro para cancelar essa adição. O painel superior irá conter dois objetos-filhos com o componente *Text*. A Figura 7.21 mostra a hierarquia de objetos que deverá ser criada.

FIGURA 7.20 – Interface para adição de classificação

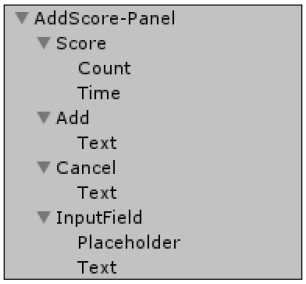


FIGURA 7.21 – Hierarquia de objetos para a interface de adição de classificação

Para facilitar a escrita do código necessário para atualizar este interface, foram atribuídos os nomes `Score` ao painel criado dentro do painel `AddScore-Panel` e `Count` e `Time` aos dois objetos com o componente *Text* adicionados dentro do painel `Score`. Por sua vez, o botão para confirmar a adição da classificação será designado por `Add` e o botão para cancelar será designado por `Cancel`. Também é importante realçar que o componente *Input Field* é composto por dois objetos-filhos: o `Placeholder` que corresponde a um objeto com o componente *Text* que contém o texto a mostrar no campo, antes de o utilizador começar a digitar (ver exemplo na Figura 7.20), e um outro objeto, denominado `Text`, que será responsável por mostrar o texto digitado pelo utilizador. O posicionamento destes objetos é resumido na Tabela 7.3, mostrando os valores para as âncoras dos objetos criados.

Além das âncoras, alguns objetos precisam de alguma configuração adicional. Os objetos `Count`, `Time`, `InputField/Text` e `InputField/Placeholder` deverão ter o alinhamento do parágrafo centrado horizontalmente e verticalmente. O texto introduzido nos dois primeiros é irrelevante, já que será reescrito através de uma *script*. No entanto, o texto do `Placeholder` terá de ser alterado para `Enter Your Name`.

OBJETO	MIN X	MAX X	MIN Y	MAX Y
`Score`	0.000	1.000	0.60	1.00
`Score/Count`	0.000	1.000	0.50	1.00
`Score/Time`	0.000	1.000	0.00	0.50
`Add`	0.525	1.000	0.00	0.30
`Cancel`	0.000	0.475	0.00	0.30
`InputField`	0.000	1.000	0.35	0.55

TABELA 7.3 – Âncoras para os vários objetos da interface de adição de classificações

Em relação aos botões, deve alterar o texto a ser apresentado (`Cancel` e `Add Highscore`), bem como aplicar as cores desejadas, tal como foi feito nos menus anteriores.

De realçar a importância de manter os nomes dos objetos `Score`, `Count` e `Time`, para que as *scripts* criadas funcionem devidamente.

Por sua vez, o jogador que não tiver atingido uma pontuação suficiente verá a interface apresentada na Figura 7.22. Como se pode verificar, uma parte é semelhante à definida anteriormente (`AddScore-Panel`), pelo que poderá aproveitar e copiar a estrutura para um novo painel, de nome `NoScore-Panel`. Este painel é composto por um outro painel superior, com dois objetos de texto (tal como na interface anterior) e dois botões, um que irá apresentar as melhores classificações e outro que irá apenas apresentar o menu inicial.

FIGURA 7.22 – Interface apenas com a classificação

A estrutura de objetos pode ser vista na Figura 7.23 e a Tabela 7.4 resume as âncoras dos vários objetos. Além das âncoras, será também necessário alterar o texto apresentado nos dois botões: `Highscores` e `OK`.

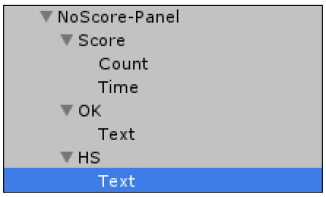


FIGURA 7.23 – *Hierarquia* de objetos para a interface apenas com a classificação

OBJETO	MIN X	MAX X	MIN Y	MAX Y
`Score`	0.00	1.00	0.60	1.00
`Score/Count`	0.00	1.00	0.50	1.00
`Score/Time`	0.00	1.00	0.00	0.50
`OK`	0.00	1.00	0.00	0.25
`HS`	0.00	1.00	0.30	0.55

TABELA 7.4 – Âncoras para os vários objetos da interface apenas com a classificação

As configurações destes objetos são semelhantes às definidas na interface anterior, onde a principal diferença é o texto que deverá ser apresentado em cada botão. Deverão também ser configuradas as cores dos botões, de modo a serem consistentes com as restantes interfaces.

A Figura 7.24 mostra o último painel a construir. Este é o mais simples e é composto apenas pelo painel geral, denominado `HighScores-Panel`, mas com âncoras diferentes, um painel-filho[29] (`Scores`), com o componente *Image*, que terá, por sua vez, um objeto-filho com o componente *Text*, como se pode ver na hierarquia de objetos apresentada na Figura 7.25.

As respetivas âncoras encontram-se descritas na Tabela 7.5. Em termos de configuração gráfica, será apenas uma questão de mudar a transparência da imagem do painel `Scores`, e a configuração do objeto `Text` para que o alinhamento do texto seja centrado, quer no eixo horizontal quer no vertical.

OBJETO	MIN X	MAX X	MIN Y	MAX Y
`HighScores-Panel`	0.05	0.70	0.10	0.65
`Scores`	0.00	1.00	0.00	1.00
`Text`	0.00	1.00	0.00	1.00

TABELA 7.5 – Âncoras para os vários objetos da interface das melhores classificações

Esta última interface não terá qualquer botão, bastando que o jogador carregue numa qualquer tecla ou botão do rato para regressar ao menu principal.

[29] Na verdade, este painel é relativamente redundante, já que se poderia adicionar o componente de imagem ao seu objeto-pai. Optou-se por esta abordagem para manter semelhança na estrutura em relação às restantes interfaces desenvolvidas nesta secção.

FIGURA 7.24 – Interface com as melhores classificações

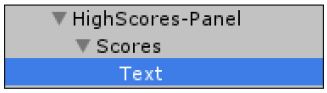


FIGURA 7.25 – Hierarquia de objetos para a interface com as melhores classificações

7.3.3 GESTÃO DE CLASSIFICAÇÕES

Esta secção irá centrar-se na implementação das funcionalidades de cada um dos painéis preparados na secção anterior, bem como no código necessário para a sua interligação. Além disso, será também implementado o código responsável por armazenar as melhores classificações.

O 1.º passo corresponderá a eliminar os comentários das três linhas de variáveis criadas na *script* `MainMenuCode`, referentes aos três painéis que foram criados. Posteriormente, deverão ser arrastados os objetos, a partir da *Hierarquia*, para os respetivos campos no *Inspetor* do `Canvas`. Posteriormente, no método `Awake` serão colocados todos os painéis como inativos:

```
// ...
AddScorePanel.SetActive(false);
MainManuPanel.SetActive(false);
HighScoresPanel.SetActive(false);
NoScorePanel.SetActive(false);
PrefsPanel.SetActive(false);
```

Poderá ainda aproveitar-se para apagar a última linha desse mesmo método, que ativa o painel do menu principal, uma vez que irá ser reescrito de seguida.

A cena referente às várias interfaces pode ser carregada em três situações:

- Quando a aplicação do jogo inicia, haverá interesse em mostrar o menu principal.
- Quando o jogador perde — matando a formiga — ou ganha — comendo todos os cogumelos —, sendo que nesse caso, apresentar-se-á a pontuação e a possível adição às melhores classificações.
- Quando o jogador está a meio de uma partida e usa a tecla **Esc** duas vezes, para cancelar o jogo, situação em que também se pretende apresentar o menu principal.

Assim, o método `Awake` terá de decidir que painel deve ser apresentado. Para tomar esta decisão, será procurado o objeto `GameManager` na cena. Recorda-se que este objeto foi configurado de modo a que não seja destruído no final do jogo. Então, caso o jogador tenha arrancado a aplicação de jogo nesse momento, o objeto não existe e saber-se-á que o menu principal deve ser apresentado. Em alternativa, o jogador terminou uma partida, regressando à cena do menu e, neste caso, o objeto `GameManager` existe, pelo que será apresentada a sua classificação.

Abaixo do código apresentado para desativar todos os painéis, será implementada a seguinte estrutura condicional:

```
GameManager gm = FindObjectOfType<GameManager>();
if (gm != null) {
   // Ramo 1
   // ...(a completar)...
}
else {
   // Ramo 2
   MainManuPanel.SetActive(true);
}
```

em que o *Ramo 1* será executado quando existe um objeto na cena do tipo solicitado e o *Ramo 2* quando esse objeto não existe. O primeiro ramo será responsável por tratar das classificações, enquanto o segundo se resume, como já implementado, a ativar o painel do menu principal.

Antes de preencher o primeiro ramo com o código necessário, convém atender à situação anteriormente descrita, referente ao abandono do jogo, usando a tecla **Esc**. Nesta situação, o objeto `GameManager` vai existir pelo que, com o código apresentado, o *Ramo 1* seria o escolhido, embora se pretenda que seja apresentado o menu principal. Uma forma simples de resolver esse problema é editar a *script* `ExitToMenu` e adicionar uma chamada

ao método `Destroy` para destruir o objeto `GameManager` e garantir que o código anterior não o encontra:

```
// ...
else {
   if (Input.GetKeyDown(KeyCode.Escape)) {
      Destroy(GameManager.instance);
      SceneManager.LoadScene("menu");
   } else {
      timer -= Time.deltaTime;
 // ...
```

Resolvido este problema, será necessário completar o primeiro ramo da estrutura condicional apresentada anteriormente. Esse ramo é responsável por obter as melhores pontuações armazenadas e verificar se a pontuação obtida pelo jogador é suficiente para entrar nessa lista.

Antes de continuar com o código desse ramo, serão criadas duas classes, uma para armazenar a pontuação de um jogador e outra para armazenar a lista de melhores classificações. Estas classes serão definidas diretamente na *script* `MainMenuCode`, já que só serão usadas pelo código dessa *script*:

```
public class MainMenuCode : MonoBehaviour {

  const int top = 10;

  [Serializable]
  public class  HighScores {
    public Score[] scores;

    public HighScores() { scores = new Score[0]; }
  }
```

Neste código foi criada a constante `top`, que corresponde ao número máximo de pontuações a armazenar (as 10 melhores, neste caso). Por sua vez, a classe `HighScores` é apenas um *array* que irá armazenar as melhores classificações. Foi criado um construtor para quando o jogo é executado pela primeira vez e ainda não existem quaisquer classificações guardadas (como se mostrará ao longo desta secção).

O principal detalhe desta classe (e da seguinte) é o facto de estar anotada com a propriedade *Serializable* [30], o que corresponde a indicar ao compilador de C# que a classe deve ser preparada para ser convertida para um formato de intercâmbio:

[30] Para que o compilador reconheça esta anotação é necessário a inclusão da biblioteca `System`.

```
[Serializable]
public class Score : IComparable<Score> {
  public int mushroomCount;
  public float gameTime;
  public string userName;

  public Score(int m, float t) {
    mushroomCount = m;
    gameTime = t;
  }
  public int CompareTo(Score obj) {
    if ( mushroomCount < obj.mushroomCount ||
         (mushroomCount == obj.mushroomCount &&
          gameTime > obj.gameTime)) {
      return -1;
    }
    else if (mushroomCount == obj.mushroomCount &&
             obj.gameTime == gameTime) {
      return 0;
    }
    else {
      return 1;
    }
  }
}
```

A classe `Score` irá guardar informação sobre uma pontuação obtida, ou seja, o tempo de jogo, o número de cogumelos apanhados e o nome do jogador que obteve a pontuação. Além de ter sido criado um construtor (sem nome do jogador, já que inicialmente não se terá essa informação), foi criado o método `CompareTo`, que implementa a interface `IComparable`, responsável pela comparação de duas pontuações. A invocação deste método sobre uma pontuação a, indicando-lhe uma pontuação b — `a.CompareTo(b)` —, irá retornar -1 se a for uma pontuação inferior a b, 0 se forem pontuações idênticas e 1 se a for uma pontuação melhor do que b.

Para facilitar a implementação dos vários métodos necessários para a adição de classificações à lista das melhores, serão criadas duas variáveis na classe `MainMenuCode`, bem como dois métodos, um para obter as melhores classificações das preferências do utilizador, e outro para as guardar:

```
Score score;
HighScores scores;
```

```
public HighScores RetrieveHighScores() {
   if (PlayerPrefs.HasKey("HighScores")) {
      string json = PlayerPrefs.GetString("HighScores");
      return JsonUtility.FromJson<HighScores>(json);
   }
   else {
      return new HighScores();
   }
}
void StoreHighScores() {
   string json = JsonUtility.ToJson(scores);
   PlayerPrefs.SetString("HighScores", json);
}
```

A primeira variável irá armazenar a pontuação obtida pelo jogador durante o seu último jogo. A segunda servirá de ponte entre as preferências do utilizador (`PlayerPrefs`), onde se irão armazenar as melhores pontuações.

O método `RetrieveHighScores` verifica se existe alguma preferência com o nome `HighScores`. Se não existir, cria um *array* vazio de classificações. Porém, se existir, obtém uma *string* que irá ter as pontuações guardadas no formato JSON e convertê-la-á nos objetos respetivos do tipo `HighScores`. Esta conversão é feita usando o método `FromJson`, que permite converter facilmente uma *string* JSON num objeto.

Do mesmo modo, o método `StoreHighScores` converte a classe de classificações numa *string* JSON usando o método `ToJson` e armazena essa *string* como uma preferência do utilizador.


Note que o recurso ao `PlayerPrefs` para armazenar esta informação leva a que as tabelas de classificações não sejam partilhadas entre diferentes instalações do jogo. Para esse tipo de objetivo deverá ser usado um serviço *online*, como o *Play Games Services* para dispositivos *Android*.

Depois desta incursão por classes e métodos auxiliares, torna-se bastante mais simples implementar o primeiro ramo condicional, apresentado anteriormente: obter o tempo e o número de cogumelos apanhados, e validar se essa classificação é suficientemente boa para entrar na lista de melhores classificações:

```
GameManager gm = FindObjectOfType<GameManager>();
if (gm != null) {
   score = new Score(gm.PickedMushrooms, gm.GameTime / 60);
   Destroy(gm);
```

```
        scores = RetrieveHighScores();
        if (scores.scores.Length < top ||
            (scores.scores.Length >= top &&
             score.CompareTo(scores.scores[top-1]) > 0)) {
           // ... pontuação a incluir na lista!
        }
        else {
           // ... pontuação demasiado fraca!
        }
    } else {
        MainManuPanel.SetActive(true);
    }
```

Havendo, então, um objeto do tipo `GameManager`, é criada e guardada a pontuação do jogador, usando o construtor da classe `Score`, na variável `score`. Ao construtor, é passado o número de cogumelos apanhados, bem como o tempo despendido, em minutos. Para que não se esqueça de destruir o `GameManager` e visto que a informação relevante já está salvaguardada, destrói-se o objeto de imediato.

Tendo-se armazenado os dados necessários para calcular a pontuação do jogador, usa-se o método `RetrieveHighScores` para obter a lista das melhores classificações e validar se a nova pontuação deverá integrar essa mesma lista.

A estrutura condicional que se segue verifica se a lista de classificações já conta com o número máximo de pontuações. Em caso negativo, e por pior que tenha sido o resultado do jogador, ainda há espaço para armazenar essa pontuação. Se, por outro lado, a lista de classificações já está completa, verifica-se se a pontuação obtida é melhor do que a pior armazenada. Só nessa situação é que a pontuação deverá ser considerada e adicionada. Caso contrário, a pontuação não é suficiente e o jogador não terá o privilégio de guardar a sua classificação.

No primeiro caso, deverá ser apresentado o painel `AddScore-Panel` e, no segundo, o painel `NoScore-Panel`. Contudo, em ambos os casos, será necessário preencher os respetivos campos com as pontuações obtidas[31]:

```
    if (scores.scores.Length < top ||
        (scores.scores.Length >= top &&
         score.CompareTo(scores.scores[top-1]) > 0)) {
       transform.Find("AddScore-Panel/Score/Count")
                .GetComponent<Text>()
                .text = string.Format("{0} Eaten Mushrooms",
                                      score.mushroomCount);
```

[31] Existe bastante sobreposição de código que poderia ser melhorada. Preferiu-se apresentar deste modo para que se torne mais simples de acompanhar.

```
        transform.Find("AddScore-Panel/Score/Time")
                 .GetComponent<Text>()
                 .text = string.Format("in {0:F2} Minutes",
                                       score.gameTime);

        AddScorePanel.SetActive(true);
    }
    else {
        transform.Find("NoScore-Panel/Score/Count")
                 .GetComponent<Text>()
                 .text = string.Format("{0} Eaten Mushrooms",
                                       score.mushroomCount);
        transform.Find("NoScore-Panel/Score/Time")
                 .GetComponent<Text>()
                 .text = string.Format("in {0:F2} Minutes",
                                       score.gameTime);

        NoScorePanel.SetActive(true);
    }
```

Concluído este código, o método `Awake` fica completo e já deverá apresentar um de três painéis: o menu principal, ao iniciar o jogo ou depois de cancelar uma jogada; o painel para adicionar o nome do jogador, se a pontuação for suficientemente boa; ou um painel a indicar apenas a pontuação, por não ter chegado aos valores mínimos necessários.

Falta, no entanto, código que permita a interligação entre os diferentes painéis.

Para a situação em que o jogador conseguiu um bom resultado e lhe foi solicitado o nome, existem dois botões: um para confirmar a adição do resultado e um outro para cancelar, caso o jogador não pretenda gravar a sua classificação.

Para o botão de cancelar, poderá ser adicionada a referência, no evento `On Click` do seu *Inspetor*, ao método `CancelButton`, definido na secção 7.2.2.1 O código aí implementado permitia apenas transitar entre o painel de preferências e o menu inicial. Ao adicionar uma instrução extra, o método torna-se reutilizável também na situação aqui descrita:

```
public void CancelButton() {
    AddScorePanel.SetActive(false);
    PrefsPanel.SetActive(false);
    MainManuPanel.SetActive(true);
}
```

A alteração limitou-se a desativar não só o painel de preferências, mas também o de adição de uma nova classificação e a ativação do painel do menu principal.

Por outro lado, o botão para confirmar a adição da pontuação, terá de aceder ao nome do jogador e armazenar essa informação. Deverá associar-se o evento `On Click` desse botão ao método `AddHighScore`:

```
public void AddHighScore() {
    score.userName = transform.Find("AddScore-Panel/InputField")
                              .GetComponent<InputField>().text;
    List<Score> l = scores.scores.ToList();
    l.Add(score);
    scores.scores = l.OrderByDescending(x => x).Take(top).ToArray();
    StoreHighScores();

    AddScorePanel.SetActive(false);
    HighScoresPanel.SetActive(true);
}
```

Este bloco de código tem duas partes. A primeira calcula a nova lista de pontuações a ser guardada. A segunda faz a troca entre painéis, mostrando o painel com as melhores pontuações.

No cálculo da nova lista de pontuações é obtido o nome do jogador e armazenado na variável `score`, que já tem o resto da informação da sua classificação. Depois, usam-se alguns métodos de LINQ[32]: converte-se o *array* para uma lista, para facilitar a adição de elementos; adiciona-se a nova pontuação; solicita-se a ordenação direta dos elementos; e, depois de ordenados, escolhem-se os 10 primeiros. A lista resultante é convertida novamente num *array* e as pontuações são armazenadas.

Embora o código anterior ative o painel das melhores classificações, ainda não foi implementada a funcionalidade responsável por preenchê-lo com essa informação. Esta será implementada numa nova *script*, de nome `ShowHighScores`, associada ao painel `High-Scores-Panel`, com o seguinte código:

```
public class ShowHighScores : MonoBehaviour {

  public void Awake() {
    var hs = transform.parent
                      .GetComponent<MainMenuCode>()
                      .RetrieveHighScores();
    string text = "<b>HighScores</b>\n\n";
```

[32] O LINQ (*Language Integrated Query*) é uma biblioteca (`using System.Linq`) que permite a escrita de operações sobre coleções de objetos, sejam em memória, sejam em bases de dados, de forma mais expedita. Uma vez que não é o foco deste livro, o código é explicado de forma superficial e o leitor mais interessado deverá consultar bibliografia adequada, como *C# 7.0 com Visual Studio – Curso Completo*, de Henrique Loureiro, também editado pela FCA.

```
    foreach (var s in hs.scores) {
      text += string.Format("{0} got {1} mushrooms in {2:F2} minutes\n",
                            s.userName, s.mushroomCount, s.gameTime);
    }
    transform.Find("Scores/Text")
             .GetComponent<Text>().text = text;
  }
  void Update () {
    if (Input.anyKeyDown)
      SceneManager.LoadScene("menu");
  }
}
```

No método `Awake` acede-se ao objeto-pai (o `Canvas`), obtém-se o seu componente `MainMenuCode` e executa-se o método `RetrieveHighScores` para obter o *array* das melhores classificações. Posteriormente, constrói-se uma *string* com o texto a apresentar: um título e uma linha por cada classificação. Repare no uso de etiquetas HTML no título. O Unity permite que o texto apresentado nos elementos de interface tenham algumas etiquetas, como o `<b>` para negrito e `<i>` para itálico. Para que isto seja possível, a opção `Rich Text` do componente *Text* deverá estar ativada.

Construído o texto, acede-se ao objeto que deverá apresentar as classificações e altera-se o campo `text` que corresponde ao texto a ser apresentado.

No método `Update` também é implementado o regresso ao menu principal. Desta feita, foi usado o acessor `anyKeyDown`, que retorna um valor booleano verdadeiro se o jogador carregar numa tecla ou num botão, seja do rato ou de um comando. Nesse momento, será apresentado, de novo, o menu principal.

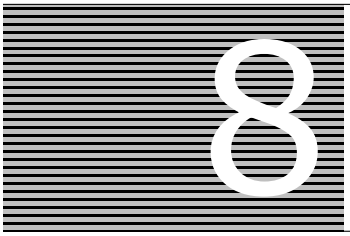

Neste caso, também se poderia ter optado por esconder o painel das classificações e ativar o painel do menu principal, o que seria mais eficiente. Usou-se novamente o método `LoadScene` apenas para salientar que é possível recarregar a cena atual.

Os botões para consultar a lista de melhores classificações, quer no painel do menu principal (`MainMenu-Panel`), quer no painel da classificação insuficiente (`NoScore-Panel`), deverão ter o evento `On Click` associado ao método `ShowHighScores` que será ainda implementado na *script* `MainMenuCode`:

```
public void ShowHighScores() {
  MainManuPanel.SetActive(false);
  NoScorePanel.SetActive(false);
  HighScoresPanel.SetActive(true);
}
```

Fica apenas a faltar a ligação do painel `NoScore-Panel`, quando o jogador usa o botão `Ok` para regressar ao menu principal. Poderá usar-se, uma vez mais, o método `CancelButton`, alterando-o para:

```
public void CancelButton() {
    AddScorePanel.SetActive(false);
    PrefsPanel.SetActive(false);
    NoScorePanel.SetActive(false);
    MainManuPanel.SetActive(true);
}
```



Distribuição

Este capítulo apresenta um dos últimos passos na construção de um jogo: a sua distribuição. Discutem-se alguns aspetos relativos à preparação de um executável do jogo, para plataformas desktop, e à construção de instaladores para Windows e macOS.

 Uma das principais vantagens do Unity é a facilidade com que se constrói um jogo para diferentes plataformas usando a mesma base de código. Embora seja uma verdade, existem muitos detalhes que têm de se ter em conta ao desenvolver um jogo para diferentes tipos de plataformas, em particular, para plataformas móveis. Deste modo, e dado o cariz introdutório deste livro, não serão discutidos os processos de preparação de executáveis e de distribuição de jogos para dispositivos móveis e para consolas.

8.1 PREPARAÇÃO DE UM EXECUTÁVEL

A criação de um executável para determinado jogo é feita usando a opção `File` → `Build Settings`, que apresenta a janela da Figura 7.11, que já foi parcialmente analisada para a definição das cenas a serem preparadas pelo Unity.

Enquanto a parte superior indica quais as cenas que devem ser construídas e colocadas no executável, a zona de baixo permite escolher a plataforma para a qual se pretende preparar o executável e, para cada plataforma, escolher alguns detalhes desse executável. Exemplo disso é a construção de um executável para Windows que permite a escolha de arquitetura de 32 ou 64 bits, bem como a construção, ou não, de um executável para depuração (*debugging*).

Usando o botão `Build`, poderá ser criado o executável e usando o botão `Build and Run`, poderá ser criado o executável e, caso o processo não apresente erros, o jogo poderá ser executado.

Nas plataformas Windows, o jogo é composto por um executável (por exemplo, `antroid.exe`) e por uma pasta de recursos (`antroid_Data`). Para que o jogo funcione, será sempre necessário distribuir quer o executável, quer a pasta de recursos, sem a alteração dos seus nomes.

8.1.1 CONFIGURAÇÕES AVANÇADAS

Além da criação direta de um executável usando o botão `Build`, é possível a definição de um grande conjunto de opções bastante interessantes. Para aceder a estas opções, deve ser usado o botão `Player Settings`. De seguida, surgirá um *Inspetor* em que se poderão definir vários aspetos do executável, de acordo com a plataforma desejada (Figura 8.1).

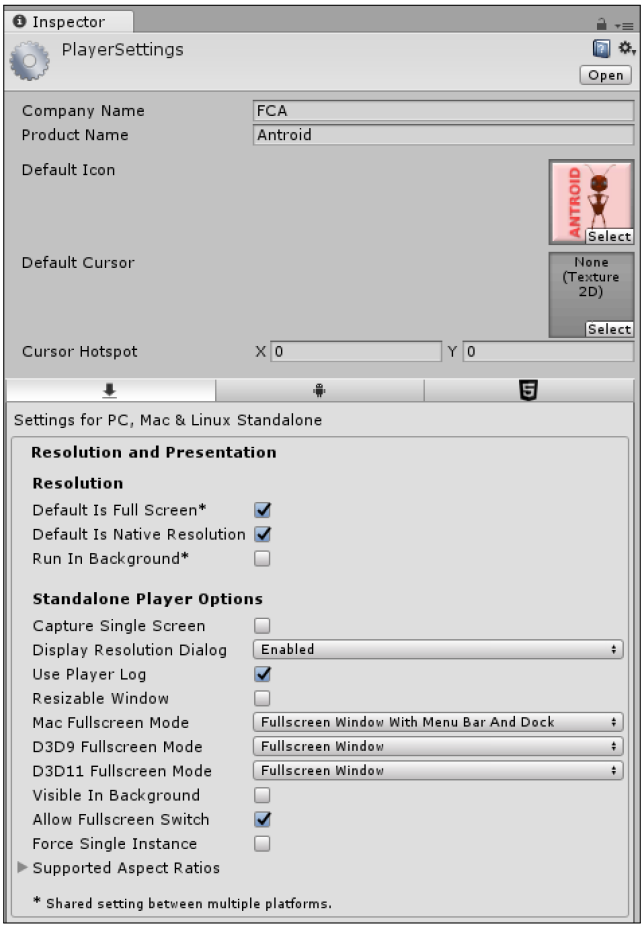


FIGURA 8.1 – *Inspetor* das propriedades do executável

No topo são apresentadas algumas configurações gerais, nomeadamente o nome da empresa que desenvolve o jogo, o nome do jogo, o ícone a ser usado e, se assim se desejar, uma imagem para ser aplicada como cursor do rato. Caso a defina, poderá alterar a posição da zona relevante para os cliques do rato. Ou seja, no cursor habitual, uma seta, o canto superior esquerdo é o ponto que deverá ser colocado pelo utilizador sobre o botão ou item sobre o qual deseja clicar. Por sua vez, se optar por um cursor ao estilo de uma mira, então

deverá ser o ponto central a corresponder à zona ativa. Não esquecer que, em ambos os casos, a imagem terá de ser importada para o projeto como um recurso convencional.

Seguem-se três separadores: um para as plataformas desktop (Windows, macOS e Linux), outro para plataformas Android e um último para a versão *Web*.

Considerando as plataformas desktop, o primeiro grupo corresponde à configuração da resolução gráfica usada, bem como do modo de apresentação. Em relação à resolução (primeiras três opções) é possível indicar se o jogo deverá iniciar com ecrã completo (`Fullscreen mode`) ou como uma janela (`Windowed mode`). Também é possível definir se a resolução a ser usada, por omissão, é a resolução atual do sistema (resolução nativa). Embora esta seja a opção mais comum, também é possível definir uma resolução mais baixa. A última opção deste trio indica se o jogo deve continuar a sua execução quando o jogador muda de aplicação ou se a execução deve entrar em pausa.

Por sua vez, o segundo grupo, inclui um conjunto de opções mais detalhadas, das quais se salientam as seguintes:

- `Display Resolution Dialog` — Indica se, ao iniciar o jogo, deve ser apresentada ao utilizador uma janela predefinida do Unity a permitir a escolha de uma resolução. Se optar por não o fazer, poderá implementar o seu próprio sistema de controlo de resoluções, ou usar sempre a resolução nativa do sistema. Poderá ainda usar a opção `Hidden by Default`, que irá fazer com que a janela só surja se a aplicação for executada com a tecla **Alt** premida.

- `Resizable Window` — Indica se, ao iniciar o jogo em modo janela, o jogador pode redimensionar a janela, ou não. O principal problema em permitir que seja redimensionada reside no facto de que o rácio largura/altura irá mudar, o que poderá tornar a gestão do jogo mais complicada. Por exemplo, pode fazer com que inimigos que não deveriam ser visíveis pelo jogador a determinada distância se vejam.

- `Visible In Background` — É uma opção apenas para plataformas Windows e define se a aplicação, quando está a usar o ecrã completo, deverá ser visível quando não está com o foco (existe outra aplicação à sua frente).

- `Force Single Instance` — Garantirá que só existe uma instância do jogo a ser executada a cada momento.

- `Supported Aspect Ratios` — Define quais os rácios de largura/altura para os quais o jogo está preparado, de modo a que o jogador seja obrigado a alterar a resolução para um rácio suportado. Limitar rácios pode ser uma forma simples de controlar o que é visível a cada momento, mas que será pouco agradável para o jogador. A escolha destes rácios irá filtrar quais as resoluções apresentadas ao jogador na janela de configuração da resolução desejada.

O segundo grupo de configurações, denominado `Icon`, é muito pouco interessante, e limita-se a permitir a definição de ícones diferentes para diferentes tamanhos, o que permite que o mesmo ícone seja apresentado com diferentes graus de qualidades, dependendo da situação. Permite também que o programador defina ícones mais simples para resoluções menores e mais complexos para resoluções maiores.

O grupo seguinte, referente ao `Splash Screen` (Figura 8.2), é mais interessante. Embora o programador possa implementar os seus próprios ecrãs de apresentação (*splash screen*), o Unity permite fazê-lo de forma bastante simples. Tal é especialmente útil porque na sua versão gratuita o Unity obriga à apresentação do seu logótipo no início do jogo. Usando as configurações de `Splash Screen` é possível apresentar um ou mais logótipos definidos pelo programador, ao mesmo tempo que é apresentado o logótipo do Unity, o que o torna menos intrusivo.

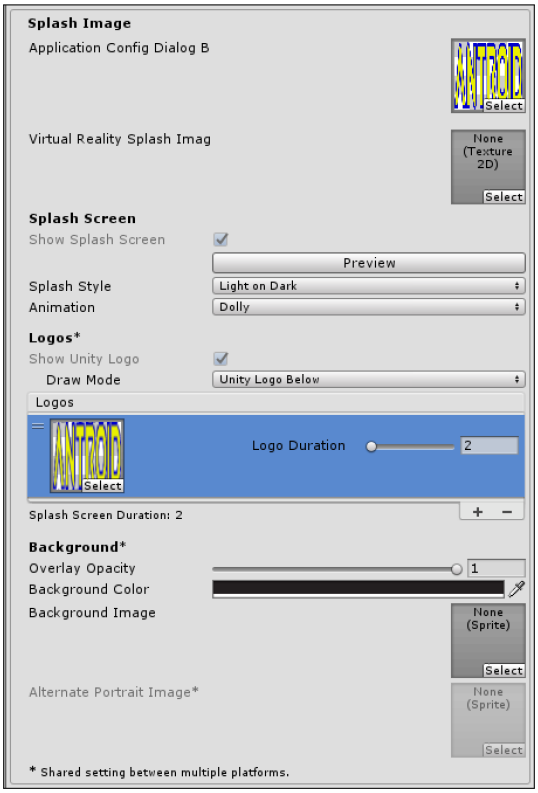


FIGURA 8.2 – *Inspetor* das propriedades de `Splash Screen` do executável

A primeira opção, `Application Config Dialog Banner`, permite definir uma imagem para ser apresentada na janela de configuração das resoluções gráficas de jogo e a segunda, `Virtual Reality Splash Image`, é usada para jogos em realidade virtual.

No caso da primeira imagem, é importante atentar no facto de que a imagem deverá ter como dimensões 432×163 píxeis. Depois de importada a imagem com essas dimensões para a pasta de projeto e, porque o Unity transforma automaticamente todas as texturas para dimensões que sejam potências de 2, essa funcionalidade deve ser alterada. Para o efeito, deverá selecionar a imagem no separador *Projeto* e alterar a opção `Advanced / Non power of 2` para `None`, bem como desativada a opção `Advanced / Generate Mip Maps`, tal como demonstrado na Figura 8.3.

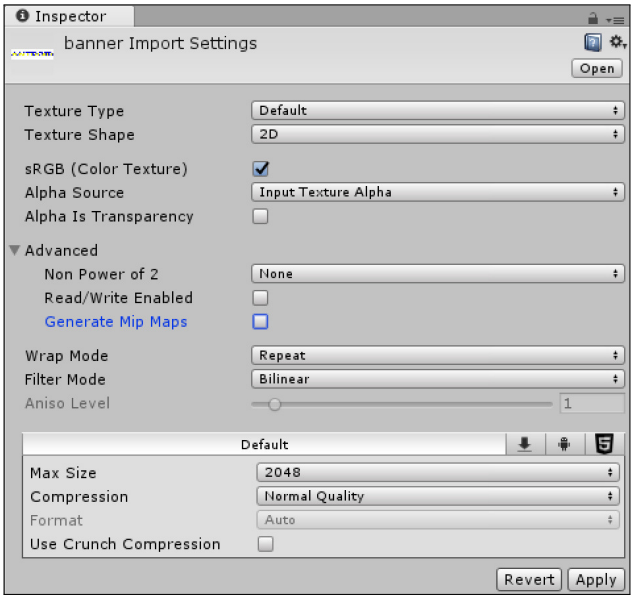


FIGURA 8.3 – *Inspetor* das propriedades da imagem usada como *banner*

As opções sob `Splash Screen` permitem definir os logótipos a serem apresentados e a forma como são apresentados. A primeira opção, `Show Splash Screen`, não é passível de ser desligada na versão gratuita do Unity, já que a apresentação do logótipo do Unity é obrigatória. O botão `Preview` permite simular, no separador *Cena*, o resultado final. Em `Splash Style` pode definir se pretende um logótipo claro sobre um fundo preto, ou um logótipo escuro sobre um fundo claro e, em `Animation`, pode definir se os logótipos terão alguma animação (movimento suave) ou se estarão parados. A Figura 8.4 mostra a pré-visualização com base nas configurações apresentadas na Figura 8.2.

Nas opções sob `Logos` existe, mais uma vez, uma opção que não pode ser desligada na versão gratuita: se o logótipo do próprio Unity deve, ou não, ser apresentado. Além disso, o `Draw Mode` permite definir se o logótipo do Unity deve ser apresentado isolado, no centro do ecrã, antes de apresentar os restantes logótipos, ou se deve ser apresentado um pouco mais pequeno, abaixo dos restantes logótipos. A opção `Logos` lista os logótipos definidos pelo utilizador para serem apresentados.

FIGURA 8.4 – Pré-visualização do `Splash Screen`

Finalmente, em `Background` define-se se deve ser apresentada uma imagem de fundo, sob os logótipos (`Background Image`) e qual a opacidade e cor dessa imagem (`Overlay Opacity` e `Background Color`). A opção `Alternate Portrait Image` é útil para dispositivos móveis, para definir qual a imagem de fundo a ser apresentada quando o ecrã se encontra na vertical.

A Figura 8.5 mostra o quarto e último grupo de configurações, que poderemos designar por avançadas. Destas, será apenas destacada a possibilidade de escolher a versão do .net e, implicitamente, a versão do C# a ser suportada, na opção `Scripting Runtime Version`. Esta é uma opção nova na versão 2017 do Unity, mas que é solicitada por grande parte dos seus utilizadores há bastantes anos, na medida em que permite que se use um dialeto mais recente do C# que suporte algumas das suas mais recentes adições, como o controlo de métodos assíncronos com `async`/`await`, a interpolação de variáveis em *strings* ou ainda a inicialização de propriedades.

8.2 INSTALADORES

A distribuição de um jogo apenas como um documento comprimido com os recursos necessários não é a forma mais bonita. Em Windows, seria interessante a criação de um instalador, que permitisse, também, fazer a desinstalação quando necessário, e em macOS, a possibilidade de criar uma imagem de disco, agradável, de onde o utilizador pudesse arrastar a aplicação.

Nesta secção explica-se, sucintamente, como se pode criar este tipo de instaladores, embora não seja algo específico do Unity. Em particular, para criar o instalador da plataforma Windows será usada ferramenta independente.

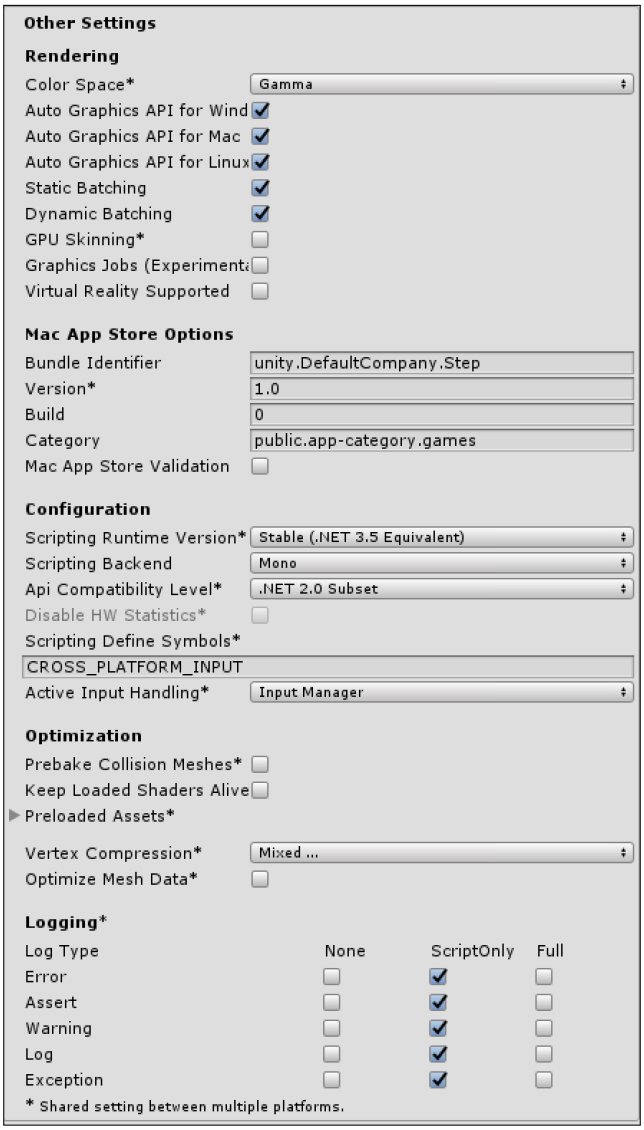


FIGURA 8.5 – *Inspetor* das propriedades avançadas do executável

8.2.1 MICROSOFT WINDOWS

Para a instalação de aplicações em Microsoft Windows, será usada a aplicação Install Forge[33], que é uma aplicação gratuita e de simples utilização. A sua instalação é trivial, bastando seguir o assistente de instalação.

[33] Disponível em `http://installforge.net/`.

Depois de ter sido criado o executável do jogo a distribuir, será necessário iniciar o Install Forge e seguir um conjunto de passos. A Figura 8.6 mostra a janela inicial da aplicação.

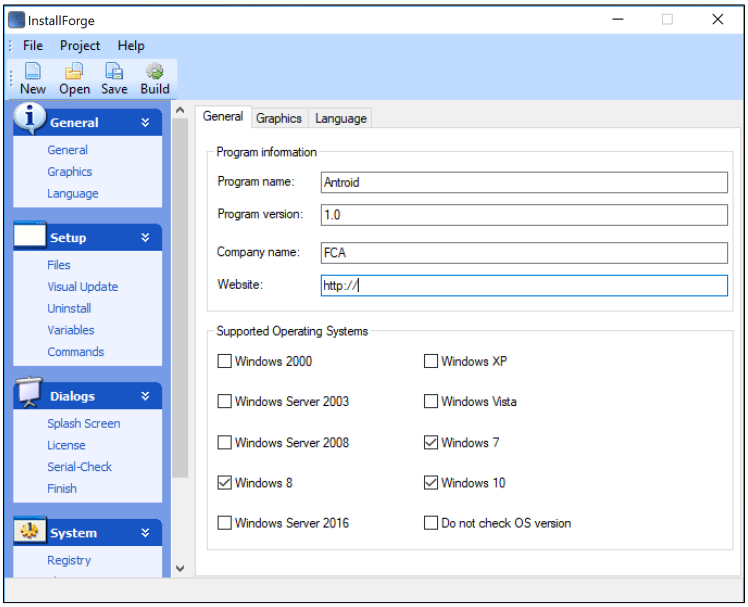


FIGURA 8.6 – Janela inicial do Install Forge

Na margem esquerda, é apresentada uma lista de passos, ou de configurações, que podem ser realizados no instalador. Cada grupo corresponde a um ecrã, em que cada item surge como um separador. Note-se que não serão detalhadas todas as opções, mas apenas as mais interessantes para a distribuição de um jogo.

No separador *General* deverá ser indicado o nome do programa (jogo), a versão, caso exista, e o nome da organização que disponibiliza o jogo. Esta informação será usada pelo instalador para definir qual o caminho sob a pasta `Program Files` em que o jogo será instalado. Também poderá indicar um endereço para um sítio referente ao jogo, bem como uma lista de sistemas operativos suportados.


Sugere-se que se desliguem todas as versões do Windows Server, bem como das versões 2000, XP e Vista, embora seja possível que alguns jogos desenvolvidos em Unity funcionem corretamente nestas versões do sistema operativo.

O separador *Graphics* permite definir a aparência da janela do instalador e o separador *Language* definir em que línguas o instalador estará disponível. Estas opções são bastante simples de usar e pouco relevantes para o principal objetivo de um instalador.

O primeiro separador do segundo grupo, *Files*, é talvez o mais importante e corresponde à lista dos ficheiros que deverão ser instalados. A Figura 8.7 mostra o aspeto dessa janela.

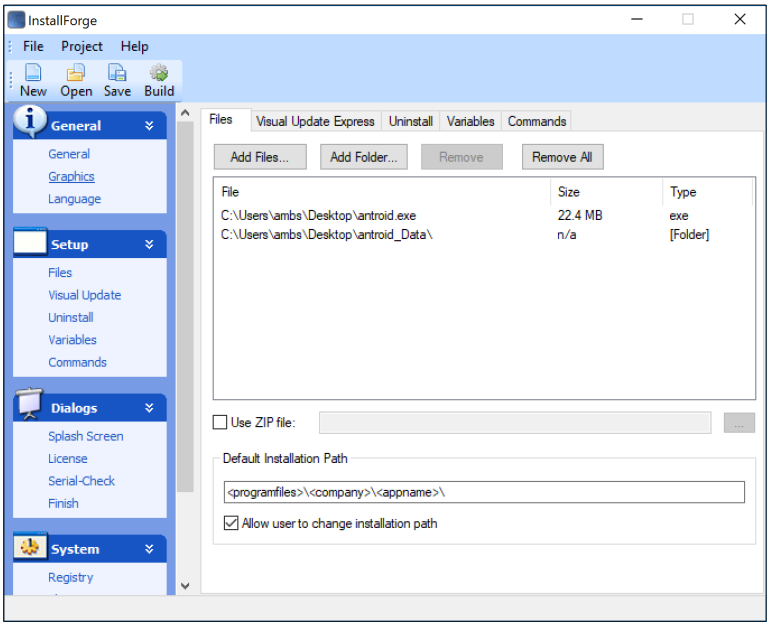


FIGURA 8.7 – Opções de ficheiros do instalador

No lista apresentada deverão ser colocados todos os ficheiros a serem instalados. Pode adicionar ficheiros usando o botão `Add Files` ou usar o botão `Add Folder` para adicionar toda uma pasta. Os ficheiros e as pastas adicionados serão colocados na raiz da pasta de instalação. É evidente que, em relação às pastas, a estrutura do seu conteúdo é mantida. Para um jogo Unity, terá de adicionar o executável e a pasta de dados.

Ainda nesta janela, pode selecionar em que pasta o jogo será instalado e se o utilizador pode definir, ou não, uma outra pasta que prefira.

O separador *Visual Update Express* permite a especificação de mecanismos automáticos para que o utilizador possa atualizar o seu jogo sem a sua completa reinstalação. Assim, permite que o programador prepare novas versões e as disponibilize rapidamente aos seus utilizadores. Para obter mais informação sobre como usar esta funcionalidade, deverá consultar o manual do Install Forge.

Já o separador *Unistall* permite indicar se deverá ser criado e instalado um programa que permita a desinstalação do jogo. Esta é uma funcionalidade útil e deve ser ativada.

O separador *Variables* permite que o instalador possa redefinir algumas das variáveis de ambiente do sistema e o separador *Commands* lista um conjunto de comandos que deverão ser executados no final do processo de instalação. Estas duas funcionalidades são bastante úteis para alguns tipos de aplicações, mas não são típicas para a distribuição de um jogo Unity.

O grupo *Dialogs* permite mais alguma configuração do instalador. A primeira opção, `Splash Screen`, define um ecrã inicial para ser apresentado antes do início do instalador. A opção `License` possibilita a definição da licença do jogo. O separador *Serial-Check* é usado para configurar um mecanismo básico de validação de cópias, solicitando ao utilizador a introdução de um código aquando da sua instalação. Finalmente, a opção `Finish` permite definir o que será feito quando a instalação terminar a execução de uma aplicação (por exemplo, do jogo) ou o reinício da máquina, entre outras opções.

No grupo *System*, a opção `Registry` é usada para manipular o sistema de registo do Windows e a opção `Shortcuts` especifica como serão criados atalhos para a aplicação (e onde serão colocados). A Figura 8.8 mostra a adição de um atalho e, na Figura 8.9, na lista que se encontra no topo, deverão aparecer os executáveis que terão direito a um atalho.

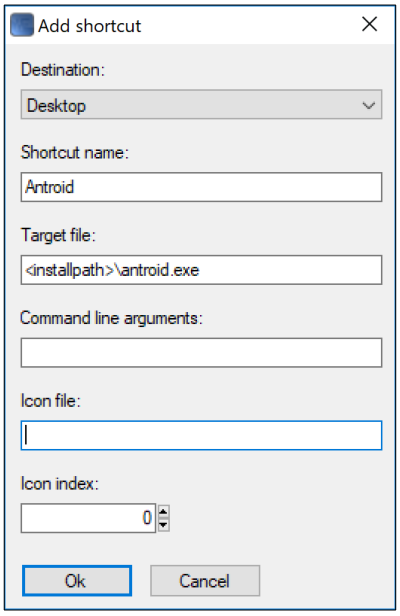


FIGURA 8.8 – Exemplo de criação de um atalho

Terminadas as opções de configuração do instalador, o último grupo contém um único item que corresponde à construção do executável instalador. Aí deverá indicar onde será colocado o ficheiro, se o conteúdo deve, ou não, ser comprimido e pressionar o botão para a criação do executável, que ficará pronto a ser distribuído.

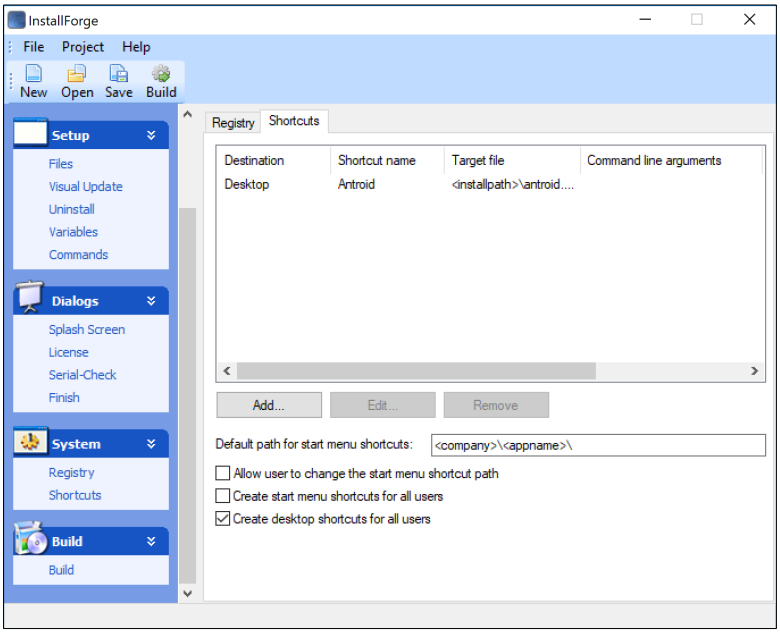


FIGURA 8.9 – Opções para a criação de atalhos

8.2.2 macOS

Em macOS as aplicações são tipicamente distribuídas em imagens de discos, com a extensão `.dmg` que, depois de serem montadas no sistema operativo, aparecem como uma janela com um ou mais ícones, que correspondem ou à aplicação a instalar (situação em que o utilizador deverá arrastar esse ícone para a pasta de aplicações do sistema) ou como um instalador (situação em que o utilizador deverá fazer duplo clique no ícone, para iniciar a aplicação de instalação).

Esta segunda alternativa é útil quando a instalação obriga à configuração de quaisquer ficheiros ou serviços do sistema operativo, em que o simples arrastar da aplicação não é suficiente. Um exemplo deste tipo de imagens de disco é apresentado na Figura 8.10.

A grande maioria das aplicações não tem este tipo de necessidades, pelo que pode ser distribuída em imagens de disco que contêm apenas um ícone relativo à aplicação que deve ser arrastado (copiado) para a pasta de aplicações. Tipicamente, estas janelas incluem um atalho para a pasta de aplicações, de modo a facilitar o trabalho do utilizador, como se mostra na Figura 8.11[34].

[34] Figura reproduzida a partir do blogue de Andy Maloney (`https://asmaloney.com/`), onde é apresentada uma abordagem possível para a criação deste tipo de imagens de disco para a instalação de aplicações.

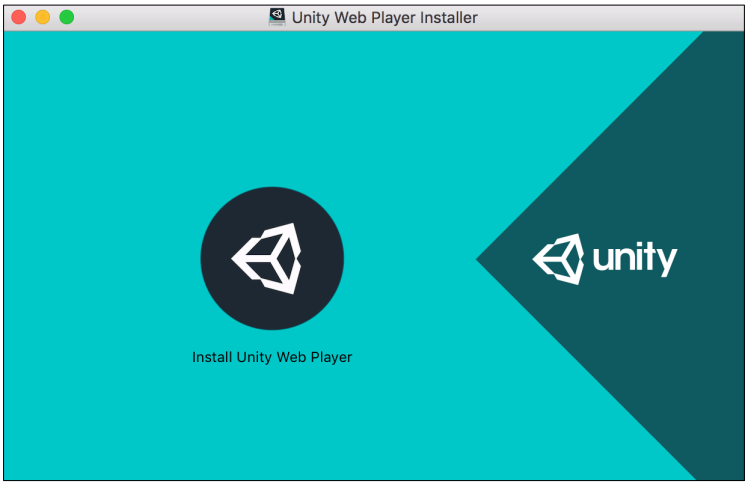


FIGURA 8.10 – Janela da imagem de disco do instalador de Unity Web Player

FIGURA 8.11 – Janela de uma imagem de disco de uma aplicação hipotética

A criação deste tipo de recurso corresponde aos seguintes passos:

1) Criar um disco virtual com o tamanho necessário, tamanho esse que poderá verificar clicando com o botão direito do rato sobre o executável ou usando a combinação de teclas **Command-I** no *Finder*.

Inicie o *Disk Utility* e escolha a opção do menu `File` → `New` → `Blank Disk Image`. Na janela que se segue, na parte superior, escolhe-se um nome para o

disco virtual, por exemplo, `antroid.dmg`, e seleciona-se o local onde ele será guardado. A parte inferior permite configurar o disco, que terá o nome `Antroid` e um tamanho adequado, de acordo com o espaço ocupado pelo jogo. As restantes opções não devem ser alteradas (Figura 8.12).

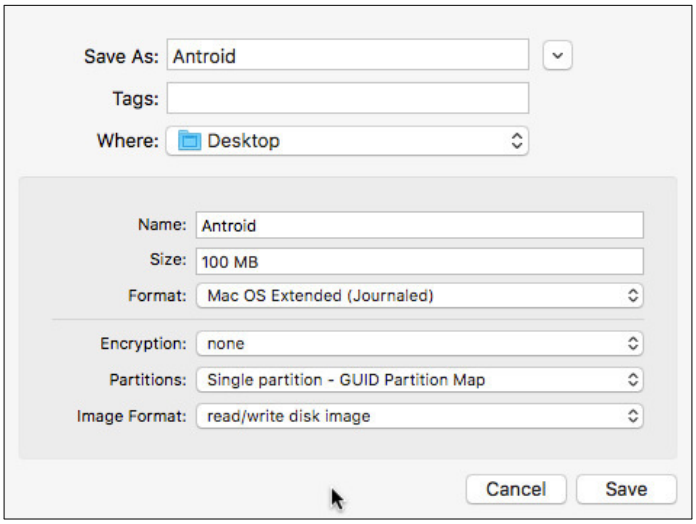


FIGURA 8.12 – Opções na criação de um disco virtual

2) O disco, depois de criado, será automaticamente montado no sistema, sendo agora possível copiar o jogo para dentro desse mesmo disco.

3) A criação de uma ligação (*link*) para a pasta `Applications` pode ser feita arrastando essa pasta para dentro do disco virtual criado anteriormente, mantendo a tecla **Option** pressionada (conhecida por **Alt** na plataforma Windows).

4) A mensagem apresentada na janela, que sugere ao utilizador que arraste a aplicação para a respetiva pasta de aplicações, é, por incrível que possa parecer, uma imagem. Esta deve ser criada numa ferramenta de edição de imagens de acordo com o tamanho da janela desejado, tendo apenas o cuidado de mantê-la a 72 dpi.

Posteriormente, é necessário aplicar a imagem como fundo à janela do disco virtual criado. Para o fazer, deverá alterar o modo de visualização da janela para `ícones` (`View` → `as Icons`) e abrir as opções de visualização (`View` → `Show View Options`). Nessas opções, deverá ser alterada a opção `Background` para `Picture` e escolhida a imagem em causa.


Se a opção para alterar a imagem de fundo estiver inativa, será necessário alterar o modo de ordenação dos ícones (`Arrange by`) para `None`.

5) O tamanho da janela e o tamanho da imagem devem coincidir e é importante que os ícones estejam na posição desejada. Assim, primeiro deverá ajustar o tamanho da janela, de acordo com a imagem de fundo, e depois colocar os ícones alinhados.

6) Terminado este processo, o disco virtual está pronto. Pode ejetá-lo e distribuí-lo.

Glossário de Termos
Português Europeu* / Português do Brasil

Para facilitar a utilização deste livro pelos leitores brasileiros incluímos este glossário de termos.

Português Europeu	Português do Brasil
Aceder	Acessar
Aplicação	Aplicativo
Barra de estado	Barra de status
Ecrã	Tela
Ficheiro	Arquivo
Gestão	Gerenciamento
Guardar/Gravar	Salvar
Ligação	Conexão
Modelo	Padrão
Rato	Mouse
Registo	Registro
Separador	Aba
Sistema operativo	Sistema operacional
Utilizador	Usuário

* 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

E

F

G

H

I

J

M

N

O

P