Desenvolvimento de um videojogo 2D e de uma página web

LABORATÓRIO DE PROJETO

UNIVERSIDADE AUTÓNOMA DE LISBOA

ALUNOS

André Cruz 20151033

André Hipólito 20150698

ORIENTADOR

Professor Daniel Silvestre

DOCENTES

Professor Nuno Brás,

Professor Raul Dionísio,

Professor Mário Marques da Silva

LEI | Licenciantura Engenharia Informática

RESUMO

Este projeto tem como objetivo dissecar o processo de desenvolvimento de um jogo digital 2D *standalone* e *mobile cross-platform* com a ajuda de um *website* para o complementar. Descrevemos o que define um motor de jogo, alguns dos motores mais usados e as decisões que levam ao porquê de ser usado o *Unity*. Desde as opções de licenciamento que são disponibilizadas até às funções que o *Unity* proporciona para a criação do jogo. A partir daqui acrescentamos todos os objetos usados, onde se explica o que fazem, os pontos de vida que têm, caso estejamos a falar dos inimigos ou do jogador, dos pontos de vida que tiram ao acertar no jogador ou ao tocar-lhe (no caso dos inimigos ou obstáculos) e até todos os objetos que o jogador pode apanhar onde podemos ver no ecrã de desempenho. Toda esta informação é processada em C# e ligada entre todos os objetos de modo a que tenhamos um jogo.

De modo a ajudar os utilizadores criamos ainda um *website* para ajudar a divulgar o jogo.

ABSTRACT

This project aims to dissect the process of developing a digital 2D standalone game and mobile cross-platform with the help of a website to complement it. We describe what defines a game engine, some of the most commonly used engines, and the decisions that lead to why Unity is used. From the licensing options that are made available to the functions that Unity provides for game creation. From here we add all the objects used, which explains what they do, the hit points they have, if we are talking about enemies or the player, the hit points they get when hitting the player or touching him case of enemies or obstacles) and even all objects that the player can pick up where we can see on the performance screen. All this information is processed in C # and linked between all objects so that we have a game.

In order to help users, we've also created a website to help spread the word.

índice

ÍNDICE DE IMAGENS

INTRODUÇÃO

O desenvolvimento de jogos digitais é um dos principais responsáveis pela evolução da Informática nos últimos anos, uma vez que requer técnicas sofisticadas – que muitas vezes evocam os mais recentes avanços da tecnologia – e engloba uma multitude de ramos da Informática, como a inteligência artificial, a computação gráfica, a simulação e muitos outros. Os jogos digitais têm de combinar uma série de elementos para criar uma sensação de imersividade nos utilizadores. Por essa razão, desenvolver um jogo digital lida com a Informática mas também por exemplo com a Psicologia, a Música e o Design Gráfico.

A indústria dos jogos digitais tem crescido exponencialmente nos últimos anos como se pode ver na figura 1. A receita gerada por esta indústria passa os milhares de milhão, competindo com outras indústrias bilionárias como, por exemplo, o cinema. Os jogos estão cada vez mais realistas simulando assim pessoas ou floras de maneira quase perfeita. A evolução dos videojogos suscita a curiosidade de cada vez mais pessoas, levando ao aumento de utilizadores, isto leva, portanto, ao crescimento desta indústria, cada vez de dia para dia, de ano para ano.

FIGURA 1

MOTOR DE JOGO

Um motor de jogo é um programa de computador ou conjunto de bibliotecas que possibilitam a junção e construção de elementos necessários para fazer um jogo[10]. Estão incluídos motores gráficos e físicos para renderizar gráficos, fazer animações e detetar colisões. Para além disto, estão ainda incluídos suportes para sons e inteligência artificial. Todas estas ferramentas permitem a criação de um jogo de raíz, com relativa facilidade.

Alguns dos motores de jogo mais conhecidos são[11]:

1. *UNREAL ENGINE* – Talvez o motor de jogo mais conhecido e usado. É sobretudo usado para fazer jogos de grandes dimensões, com gráficos excelentes. É a solução ideal para quem procura criar um jogo em 3D para o computador. Apesar de também ser usado por utilizadores amadores, é talvez mais usado por profissionais, sendo totalmente grátis, embora seja cobrada uma parte do lucro que é feito com um jogo. Podemos ver uma comparação entre este e o *CryEngine* na fig2.

2. *CRYENGINE* – Bastante semelhante ao *Unreal Engine* no que toca à capacidade gráfica, os gráficos realistas fazem deste motor de jogo uma plataforma a considerar. Tal como o anterior, apenas é cobrada uma parte do lucro feito com jogos, sendo grátis para uso pessoal. Apesar de fornecer tutoriais, quando comparado com outros motores de jogo, este é mais difícil de trabalhar no caso de amadores. Podemos ver uma comparação entre este e o *Unreal Engine* na fig 2.

3. *GODOT ENGINE* – É um motor de jogo avançado para quem procura desenvolver jogos 2D/3D. Suporta diversas plataformas, é grátis de usar e tem bastantes facilidades ao nível da criação de jogo.

FIGURA 2

No entanto, e apesar de todas estas opções, o motor de jogo escolhido foi o *Unity*.

PORQUÊ O UNITY?

Como é sabido, criar um jogo pode ser uma grande dor de cabeça, pois há que ter atenção ao mais pequeno pormenor. Para além da necessidade de tornar o jogo mais complexo e diversificado, também se procura uma maior facilidade a nível de *design* e compatibilidade com outras plataformas. Tudo isto são facilidades que o *Unity* nos pode oferecer, sendo um motor de jogo que tanto pode ser usado por pessoas que estão a ter o primeiro contacto com desenvolvimento de jogos, como a nível profissional. Outra vantagem bastante importante é o facto de o jogo poder ser testado em tempo real, oferecendo a possibilidade de testar o que funciona e o que não funciona.

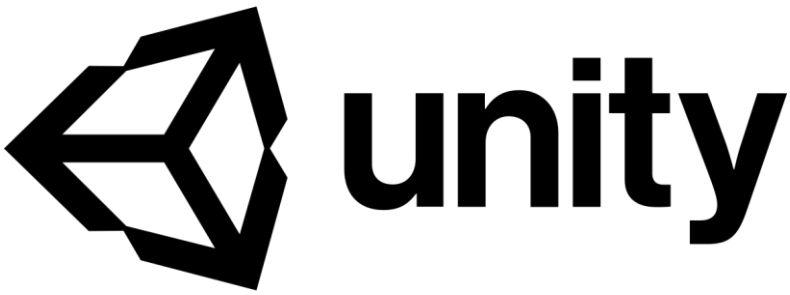

FIGURA 3

MAIS SOBRE O UNITY

Com mais de dez anos de existência (lançado em 2005), o *Unity* é um motor de jogo criado pela *Unity Technologies* que permite criar jogos em duas (2D) ou três (3D) dimensões. A linguagem C# é usada como linguagem principal, tendo sido usadas anteriormente as linguagens *Boo* e *UnityScript*, entretanto removidas. Para desenvolvimento de jogos 2D é permitida a importação de *sprites* e um renderizador de mapa avançado[12].

O editor é suportado pelos Sistemas Operativos *Windows* e *macOS*, tendo uma versão em fases de testes para *Linux*. Em termos de plataformas suportadas pelo motor de jogo em si, são cerca de 27. *iOS, Android, Windows, macOS* e *Linux* são algumas das plataformas mais conhecidas.

A prova de que o *Unity* é uma plataforma bastante usada não só por amadores é os jogos que foram feitos, entre eles *Super Mario Run, Temple Run Trilogy, Hearthstone: heroes of warcraft* e o tão famoso *Pokemon GO.*

EXISTEM ALGUMAS OPÇÕES DE LICENCIAMENTO[12]:

- ***PERSONAL*** – É a opção escolhida para iniciantes ou estudantes. Grátis para uso e com acesso a todos os principais recursos do motor de jogo, bem como atualizações e suporte para todas as plataformas. No entanto, existem algumas limitações, sendo que não é possível usar o *Unity Personal* caso a empresa obtenha mais de 100.000$ em receita buta anual ou receber mais de 100.000$.

- ***PLUS*** – Uma opção para quem procura levar o desenvolvimento de jogos mais a sério, com o custo entre 13$ e 20$ por mês. Para além das funcionalidades garantidas na versão *Personal*, são facultados cursos de desenvolvimento de jogos pela *Unity*, bem como desconto de 20% na loja, para compras de *sprites*, áudio, etc. Permite ainda recursos multijogador até 50 utilizadores simultaneamente ligados pela *Unity*. Tal como a versão anterior, esta tem algumas limitações, entre as quais a receita anual ou os benefícios não poderem ser superiores a 200.000$.

- *PRO* – Tem um custo de cerca de 100$ por mês. É uma versão que tem as vantagens das versões anteriores, com a diferença de não ter limites de receita anual. Existem ainda mais algumas vantagens, como o facto de ser uma versão que facilita o trabalho em equipa e permite resolver rapidamente qualquer erro que surja.
- *ENTERPRISE* – É uma versão direcionada para equipas grandes, com mais de 21 membros, que oferece vantagens e facilidades em trabalhar no mesmo projeto em conjunto.

FUNÇÕES DO UNITY

Apesar do *Unity* usar a linguagem C# como linguagem de programação, existem algumas funções e classes que necessitamos de saber por forma a facilitar-nos o uso desta plataforma.

MONOBEHAVIOUR

É a classe base sobre a qual todos os *scripts* do *Unity* derivam, quando é usada a linguagem C#. Quando é usada *UnityScript* (uma espécie de *JavaScript*), não é necessário derivar desta classe. Durante o desenvolvimento deste jogo, todos os *scripts* usados derivaram da classe *MonoBehaviour*.

FUNÇÕES START E AWAKE

A função *Start* é inicializada no *frame* no qual o *script* é ativado, antes de serem inicializados quaisquer métodos *Update*. É uma função que apenas é chamada uma vez durante todo o tempo de execução do *script*.

Tal como a função *Start*, a função *Awake* também só é chamada uma vez. Esta é chamada quando a instância do *script* está a ser carregada. No entanto, esta última difere da anterior no sentido em que é chamada no momento em que o objeto no qual o *script* está inserido é inicializado, independentemente se o *script* está ativo ou não. Por este motivo, ambas

as funções podem não ser chamadas no mesmo frame. Note-se que *frame* é uma atualização no ecrã.

A função *Awake* é chamada em todos os objetos de cada cena antes da função *Start*. Nos casos em que a inicialização do objeto A dependa da anterior inicialização do objeto B, esta última deve ser feita na função *Awake* e a inicialização do objeto A deve ser feita na função *Start*.

Cada função *Awake* dos objetos é chamada por ordem aleatória. Tendo isto em conta, deve-se usar esta função para estabelecer referências entre os *scripts* e usar a função *Start* para passar informações para a frente e para trás.

Quando os objetos são instanciados durante o jogo, a respetiva função *Awake* é chamada depois das funções *Start* dos objetos presentes na cena que já foram concluídas.

FUNÇÕES UPDATE E FIXEDUPDATE

A função *Update* é chamada uma vez por cada *frame*, nos *scripts* em que é usada. Normalmente é usada para fazer atualizações regulares, tais como mover objetos nos quais a física não é aplicada, receber entrada de dados do utilizador ou fazer cronómetros simples. Esta função não é chamada numa cronologia constante, uma vez que depende da duração de cada *frame*. Ora, se um *frame* demorar mais tempo a processar do que o seguinte, o tempo entre as chamadas desta função será diferente.

Por outro lado, a função *FixedUpdate*, embora seja parecida à anterior, segue uma cronologia constante e o tempo entre as chamadas desta função é igual. É normalmente usada para atualizações regulares tais como ajustar a física dos objetos.

FUNÇÃO LATEUPDATE

É uma função que é chamada depois de todas as funções *Update*. É bastante útil caso seja necessário ordenar os *scripts*. Temos como exemplo uma câmara que siga um objeto, devendo esse movimento ser sempre definido na função *LateUpdate*.

OnTriggerEnter2D (Collision2D other)

É uma função que é chamada quando um objeto entra numa zona de colisão do objeto com o qual colide, que funciona como um gatilho. Apenas funciona para objetos de dimensionalidade 2D. Esta função tem um argumento, um outro objeto colisor, sobre o qual são passadas mais informações nesta função, para o mesmo ser identificado. Um dos usos desta função pode ser detetar a colisão de um determinado objeto, de modo a desencadear uma ação, como por exemplo guardar a posição de uma personagem no caso de esta morrer (funcionar como um *checkpoint*).

OnTriggerStay2D (Collision2D other)

Diferente da função anterior, uma vez que neste caso a função não é chamada aquando existe uma colisão, mas sim enquanto um objeto está presente na zona de colisão de outro, em todos os *frames*. Também é passado um argumento, com as mesmas características do que é passado na função anterior. Esta função pode ser usada para alterar o movimento de uma personagem, de forma a simular o movimento dentro de água, por exemplo.

OnTriggerExit2D (Collision2D other)

Aceita como argumento um objeto que tenha um *collider2D* associado às suas propriedades. Esta função é semelhante à anterior, com a diferença que em vez de ser chamada enquanto o objeto está presente na zona de colisão de outro, é chamada quando o objeto sai dessa mesma zona de colisão. Pode ser usada, em complemento com a função anterior, como forma de repor o movimento normal da personagem.

OnCollisionEnter2D (Collision2D other)

Toma como argumento um objeto que tenha um *collider2D* associado às suas propriedades. Difere da função *OnTriggerEnter2D* no sentido em que neste caso o objeto com o qual colide não "entra" dentro deste, como é o caso dessa função. Esta função pode

ser usada, por exemplo, para detetar a presença do jogador numa plataforma e consequentemente, fazê-la cair.

OnCollisionStay2D (Collision2D other)

Ao contrário da função anterior, esta função apenas é chamada enquanto o objeto está em contacto com o objeto ao qual esta função está associada. Aceita um argumento com as mesmas propriedades do argumento da função anterior. Pode ser usada, entre outras situações, para fazer uma plataforma mexer apenas quando o jogador está em contacto com a mesma.

OnCollisionExit2D (Collision2D other)

Aceita igualmente um argumento com as mesmas características. Pode funcionar como um complemento da função anterior, pois esta apenas é chamada quando já não existe contacto entre os dois objetos. Uma das situações em que pode ser aplicada é, por exemplo, uma superfície que volta à forma normal depois do contacto com o jogador.

Para tentar perceber melhor as diferenças entre *collision* e *trigger*, temos na figura 4, um breve resumo dos comportamentos numa e noutra situação.

Collision detection occurs and messages are sent upon collision

	Static Collider	Rigidbody Collider	Kinematic Rigidbody Collider	Static Trigger Collider	Rigidbody Trigger Collider	Kinematic Rigidbody Trigger Collider
Static Collider		Y				
Rigidbody Collider	Y	Y	Y			
Kinematic Rigidbody Collider		Y				
Static Trigger Collider						
Rigidbody Trigger Collider						
Kinematic Rigidbody Trigger Collider						

Trigger messages are sent upon collision

	Static Collider	Rigidbody Collider	Kinematic Rigidbody Collider	Static Trigger Collider	Rigidbody Trigger Collider	Kinematic Rigidbody Trigger Collider
Static Collider					Y	Y
Rigidbody Collider				Y	Y	Y
Kinematic Rigidbody Collider				Y	Y	Y
Static Trigger Collider		Y	Y		Y	Y
Rigidbody Trigger Collider	Y	Y	Y	Y	Y	Y
Kinematic Rigidbody Trigger Collider	Y	Y	Y	Y	Y	Y

FIGURA 4

COROUTINES

Quando uma função é chamada, esta é finalizada antes de retornar algo. Quer isto dizer que todas as ações dentro de uma função têm de ter a duração de um único *frame*, não podendo portanto, conter uma animação sequencial ou uma sequência de ações.

Este problema pode ser resolvido escrevendo código na função *Update* que executa as ações pretendidas. No entanto, é normalmente usada uma *Courotine* para este efeito.

Uma *Coroutine* é como uma função que tem a capacidade de fazer uma pausa na execução, devolvendo o resultado visível (ou não) ao utilizador. Quando é retomada a execução no *frame* seguinte, a ação continua no exato estado onde parou. Essencialmente é uma função que retorna um objeto do tipo *IEnumerator* (ou em alguns casos, não retorna nada), ou seja, tudo o que a função retornar tem de implementar a interface *IEnumerator*.

Para esta função funcionar corretamente, é necessário a linha de comando *yield return*. O que este comando faz é parar ou pausar a execução nesse exato momento e retornar o objeto que se pretende (ou nenhum). Caso tenha sido pausada, a função retorna no mesmo ponto onde ficou, no *frame* seguinte. Se existir um grande número de ações pode provocar alguma lentidão. Uma solução é implementar um atraso no retorno do objeto através do comando *WaitForSeconds*, que aceita um número de segundos e que adia o retorno pelo tempo especificado.

No entanto, para uma *Coroutine* iniciar, é necessário o comando *StartCoroutine*, que aceita como argumento o nome da *Coroutine*. No sentido inverso, caso exista a necessidade de parar a função, podemos usar o comando *StopCoroutine*, especificando o nome da *Coroutine* a finalizar, ou finalizando todas através do comando *StopAllCoroutines.*

ESCALABILIDADE

Normalmente, quando existe uma pequena quantidade de dados, qualquer *software* funciona corretamente. O problema forma-se quando a quantidade de dados aumenta,

sendo por vezes difícil o manuseamento dessa grande porção de dados. Felizmente, o *Unity* oferece-nos uma grande ajuda no que toca a esse aspeto, através do uso de *prefabs*. *Prefabs* – ou em português, pré-fabricados – tal como o nome indica, são objetos que são pré-fabricados e que podem ser usados mais facilmente ao longo do projeto. Isto cria-nos imensas facilidades, como por exemplo, ao adicionarmos inimigos. Basta criarmos uma vez um objeto desse tipo, com as classes que se adequam ao mesmo e depois é só torná-lo um pré-fabricado e usá-lo onde e quantas vezes quisermos. Se por algum motivo for necessário efetuar alguma alteração no mesmo, basta efetuar essa alteração uma vez e guardar, que será aplicada a todos os objetos desse tipo presentes em todos os níveis.

DESEMPENHO

Talvez a maior exigência no que toca às qualidades de um *software* prende-se no seu desempenho. Como no nosso caso podiam existir objetos que nunca seriam destruídos (inimigos ou o jogador a disparar constantemente, por exemplo), isso poderia provocar alguma lentidão. Como tal, surgiu a necessidade de resolver esses problemas. A solução que implementámos consiste na criação de duas classes: *DestroyObjectsOverTime* e *DestroyFinishedParticles*. O que a primeira faz é destruir os objetos que são lançados quer pelo jogador, quer pelos inimigos ao fim de um determinado tempo, de modo a que não fiquem infinitamente presentes no nível e reduza o desempenho do jogo. Por sua vez, a segunda classe tem o objetivo de destruir as partículas que já não vão ser usadas. Partículas essas que são instanciadas cada vez que um inimigo morre, por exemplo, e que não são novamente usadas, podendo ser destruídas de modo a não prejudicar o desempenho do jogo.

OBJETOS

PROPRIEDADES

De forma a tornar os objetos (sejam eles o jogador, inimigos, caixas, etc) mais realistas, o *Unity* proporciona-nos algumas classes que ajudam a alcançar esse objetivo. Comecemos pelas propriedades físicas em si. Como forma de simular um objeto afetado pelas leis da física, temos a classe *RigidBody2D*, que podemos ver mais ao detalhe na figura 5.

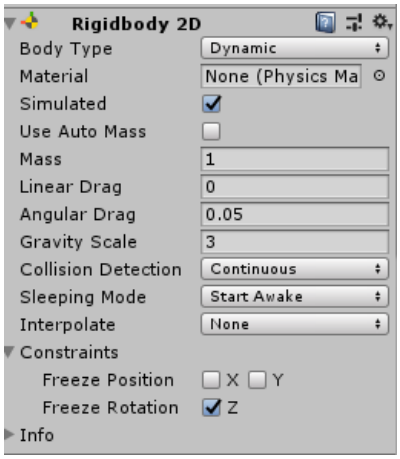

FIGURA 5

Entre outras, podemos ver que algumas das propriedades que vão ser adicionadas são a força que a gravidade exerce sobre este objeto, a sua massa e o seu tipo. Podem ser de três tipos: *Static*, *Dynamic* e *Kinematic*. Só pelo nome é possível perceber que comportamentos têm objetos do tipo *Static* e *Dynamic*. Por sua vez, objetos do tipo *Kinematic*, não se mexem se outro objeto exercer algum tipo de força sobre estes, no entanto, a sua posição pode ser alterada através de código.

De seguida, temos provavelmente a classe do *Unity* que mais vezes foi aplicada aos objetos, a classe *Collider2D*. Sejam do tipo *BoxCollider2D*, *CircleCollider2D* ou *EdgeCollider2D*, todas estas classes herdam da classe *Collider2D*. O que estas acrescentam, é uma forma (invisível) que determina as superfícies do objeto que colidem com outros objetos. Não necessita de ser exatamente da forma do objeto, por vezes uma aproximação até é mais eficiente. Podemos ver um exemplo na figura 6.

FIGURA 6

Como andar de um lado para o outro não basta para criar o efeito de movimento, é preciso recorrer às animações. Usando a classe *Animator* é possível criar uma animação parecida à representada na figura 7.

FIGURA 7

Por fim, temos ainda a possibilidade de adicionar som ao jogo. Quer seja uma música de fundo, ou sons que são reproduzidos quando existe algum movimento específico, em ambos os casos podem ser adicionados com recurso à classe *AudioSource* (Figura 8). Entre outras opções, é-nos possibilitado ajustar o volume, a prioridade ou selecionar se queremos que o som seja reproduzido assim que o objeto é inicializado.

FIGURA 8

JOGADOR

Na figura 9 temos representada aquela que é a nossa personagem principal. É a personagem que pode ser controlada pelo utilizador. Entre vários movimentos, é possível andar, saltar ou até mesmo disparar. Pode ainda interagir com outros objetos que se encontram no jogo, tal como moedas, vidas ou inimigos.

FIGURA 9

Vida: 100

Dano quando dispara: 10

INIMIGOS E OBSTÁCULOS

Um dos aspetos mais recorrentes num jogo de plataformas é a presença de inimigos para aumentar a dificuldade. Neste caso não podia ser exceção. Assim sendo, temos três inimigos, com formas e propriedades diferentes.

Comecemos então pelo primeiro inimigo.

FIGURA 10

Vida: 10

Dano quando em contacto com o jogador: 20

Pontos quando é morto: 100

Como o inimigo anterior era um bocado básico, pois só tirava vida quando tocava no jogador, havia a necessidade de dificultar o nível do jogo. Então, foi adicionado outro tipo de inimigo, um inimigo que pode disparar contra a personagem.

FIGURA 11

Vida: 20

Dano quando em contacto com o jogador: 30

Dano no jogador quando dispara: 40

Pontos quando é morto: 200

Continuando a aumentar a dificuldade, surgiu um novo tipo de inimigo. Desta vez não se limita a andar pelo chão: ele voa e desloca-se em direção ao jogador quando este está dentro do alcance do inimigo.

FIGURA 12

Vida: 20

Dano quando em contacto com o jogador: 40

Pontos quando é morto: 350

No entanto, apesar de haver alguma diversidade no que toca a inimigos, achámos por bem incluir alguns obstáculos. Ainda que não sejam inimigos propriamente ditos, assumem comportamentos que tiram vida ao jogador. Na figura 13, está representado o primeiro

tipo de obstáculo – picos. Simplesmente colocados sob o chão ou podendo mover-se para cima e para baixo, matam instantaneamente se o jogador entrar em contacto com estes. Temos ainda um bloco de picos, como mostra a figura 14, que tem a particularidade de cair assim que é detetado o movimento do jogador por baixo do mesmo. Tal como o anterior, também mata se colidir com o jogador.

FIGURA 13

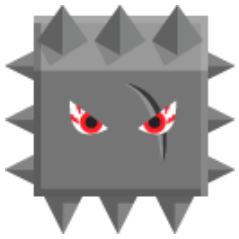

FIGURA 14

Para aumentar a lista de obstáculos com os quais é preciso ter o máximo cuidado, temos o exemplo do pêndulo. Como o nome indica, apresenta o movimento de um pêndulo e tem picos na extremidade, como é possível de ver na figura 15. Por último, mas não menos importante temos uma serra que se vai movendo pelo chão (figura 16). Em ambos os casos é necessário evitar o contacto, uma vez que também matam instantaneamente o jogador.

FIGURA 15

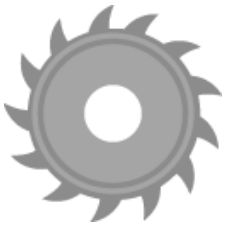

FIGURA 16

Terminados todos os objetos que se comportam como obstáculos, é necessário ainda ter atenção a algumas superfícies de contacto, como a água (figura 17) e a lava (figura 18).

Se quando o jogador entra em contacto com a lava é fácil de perceber o que acontece – é logo morto – não é muito difícil perceber o que acontece quando este entra em contacto com a água. Tal como na vida real, quando estamos debaixo de água, ao fim de algum tempo começa-nos a faltar o ar. Neste caso não deixa de ser igual, traduzindo-se "faltar o ar" para ir perdendo uns pontos de vida.

FIGURA 17

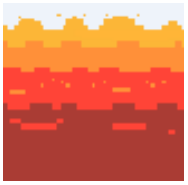

FIGURA 18

VIDAS, MOEDAS, PONTOS E OUTROS

Em cada nível está presente no ecrã informação sobre o desempenho do jogador. Nele é apresentado o número de vidas, a barra de vida, os pontos, as moedas e o tempo restante, quando aplicável. Na figura 19 podemos ver um exemplo de como tudo isto é apresentado.

FIGURA 19

Tudo isto estaria incompleto se não existissem objetos que fossem possíveis de apanhar. Existem moedas, vidas e *packs* de vida, que estão representados nas figuras 20, 21 e 22, respetivamente. *Packs* de vida são nada mais nada menos que objetos que dão vida ao jogador.

FIGURA 20

FIGURA 21

FIGURA 22

MENU DE PAUSA

Salvo raras exceções – jogos *multiplayer* – todos os jogos têm uma opção de pausar o jogo. Neste não podia ser diferente. Podemos ver na figura 23 como é composto o menu de pausa.

FIGURA 23

São apresentadas três opções. A primeira, e talvez a mais básica é a opção de retomar o jogo. Quando o utilizador clica nesta opção o jogo volta a retomar normalmente. A última opção é a de voltar para o ecrã inicial. Quando esta opção é escolhida (e tal como o nome indica) o jogo volta para o ecrã inicial. Por fim, a segunda opção – opções de som, figura 24 – abre-nos um segundo menu, onde podemos escolher o volume da música de fundo ou simplesmente silenciar o jogo por completo. No final, temos ainda uma opção que nos possibilita de voltar ao menu anterior depois de fazermos (ou não) as nossas alterações no som.

FIGURA 24

SCENES

ECRÃ INICIAL

Primeira janela que o jogador vê assim que abre o jogo e podemos observá-la na figura 26, aqui pode ir à loja, começar um novo jogo, continuar de onde parou ou sair da aplicação.

LOJA

Aqui o jogador vai ter opções para poder gastar as moedas que apanhou durante o jogo como podemos ver na figura 25. Vai poder comprar melhorias para o dano que a faca dá e também para aumentar a sua vida base.

FIGURA 25

NOVO JOGO

O jogador ao entrar por esta porta vai estar a começar o jogo pela primeira vez ou a reiniciar o jogo, ou seja todas as moedas que foram apanhadas, todo o *score* alcançado e todas as *keys* estarão de volta a zero.

CONTINUAR

Nesta porta o jogador vai apenas continuar de onde parou, isto quer dizer que pode desligar o jogo ou simplesmente voltar ao ecrã inicial e todo o seu progresso ficará guardado podendo apenas continuar a partir daqui.

SAIR

Aqui é simplesmente sair do jogo, fechar a aplicação.

FIGURA 26

MANUAL DO PROGRAMADOR JOGO 2D

ECRÃ INCICIAL

LOAD

Classe responsável por levar o jogador para as diferentes scenes(níveis).

Variáveis mais importantes:

- *Public string loadLevel* – scene para onde o jogador é levado;

Métodos:

- *OnTriggerEnter2D(Collider2D other)* – aceita como argumento um objeto que tem associado um *collider2D* às suas propriedades. Caso esse objeto seja o jogador, define as moedas, score e keys para 0 através de *PlayerPrefs.SetInt("PlayerCurrentCoins", 0)*, no caso das coins, mesmo coisa para as moedas e keys mas em vez de usar "*PlayerCurrentCoins*" usa "*PlayerCurrentScore*" e "*PlayerCurrentKeys*". Faz também o load do nível através de *SceneManager.LoadScene(loadLevel),* e reinicia as moedas, keys e score através de *PlayerPrefs.DeleteAll()*.

LOADCONTINUE

Classe responsável por levar o jogador para as diferentes scenes(níveis) mas ao contrário de *Load*, leva o jogador com todas as moedas, score e keys que já apanhou até ali.

Variáveis mais importantes:

- *Public string loadLevel* – scene para onde o jogador é levado;

Métodos:

- *OnTriggerEnter2D(Collider2D other)* – aceita como argumento um objeto que tem associado um *collider2D* às suas propriedades. Caso esse objeto seja o jogador, vai buscar as moedas, score e keys que o jogador através de *PlayerPrefs.GetInt()* e depois carrega o nível através de *SceneManager.LoadScene(loadLevel).*

SAIR

Classe responsável por fechar a aplicação.

Métodos:

- *OnTriggerEnter2D(Collider2D other)* – aceita como argumento um objeto que tem associado um *collider2D* às suas propriedades. Caso esse objeto seja o jogador, vai fechar a aplicação através de *Application.Quit().*

STORE

Responsável por fazer os upgrades do dano que a faca faz e da vida base do jogador.

Variáveis mais importantes:

- *public string mainMenu* -

Métodos:

- *Start()* - apenas inicializa variáveis;
- *public void GiveMoreHealth()* – se o jogador tiver 50 coins ou mais ao carregar no upgrade a sua vida máxima passa para 150 através de *playerHealth.playerHealthBar.maxValue = 150* e são-lhe retiradas 50 coins com *coinsManager.coins* -= 50;

- *public void TakeMoreDamage()* – se o jogador tiver 65 coins ou mais ao carregar no upgrade o dano que a faca passa a tirar é 20 isto é possível através de *knife.damageToGive = 20* e são-lhe retiradas 65 coins com *coinsManager.coins -= 65;*

- *public void Voltar()* – Para podermos sair da loja usamos *SceneManager.LoadScene(mainMenu).*

CONTROLO, AÇÕES E PROPRIEDADES DA PERSONAGEM

CHARACTERCONTROLLER2D

Classe responsável por alguns dos movimentos da personagem, tais como saltar, controlar a personagem quando está em contacto com paredes nas quais pode subir e descer e detetar as superfícies onde se pode mover.

Variáveis mais importantes:

- *public float m_JumpForce* – força aplicada ao salto;
- *private LayerMask m_WhatIsGround* – define quais são as camadas sobre as quais a personagem se pode mover;
- *private LayerMask m_WhatIsWall* – define quais são as paredes que a personagem pode subir e descer.

Métodos:

- *Awake()* – apenas inicializa variáveis;
- *FixedUpdate()* – é usada para detetar as superfícies nas quais a personagem se pode mover;
- *Move (float move, bool crouch, bool jump)* – toma como argumentos uma variável *float* responsável pela direção do movimento e duas variáveis *bool* para controlar quando a personagem salta ou anda mais devagar. É neste método que se controla o salto, deteta o contacto com a parede e deteta quando é que a imagem da personagem tem de virar, de forma a estar virada para o sentido em que se movimenta;
- *HandleWallSliding()* – responsável pelo movimento da personagem em contacto com a parede;
- *Flip()* – muda a imagem da personagem de um sentido para o outro.

PLAYERMOVEMENT

É um complemento da classe anterior. Nesta classe é aplicada a velocidade à personagem (seja no chão ou debaixo de água, por exemplo), é dada a habilidade de disparar, dar um duplo salto e um efeito de *knockback* quando se colide com um inimigo. Nesta classe é ainda dada ordem de reprodução de sons e animações referentes à personagem.

Variáveis mais importantes:

- *public float runSpeed* – velocidade com que a personagem se move;
- *public float waterSpeed* – velocidade com que a personagem se move dentro de água;
- *public float knockback* – força aplicada na personagem quando esta entra em contacto com um inimigo;
- *public Transform firePoint* – ponto a partir do qual é disparada a arma.

Métodos:

- *Start()* – apenas inicializa variáveis;
- *Update()* – é onde está a parte mais importante. É aqui que são aplicadas todas as características descritas anteriormente. Esta função é usada para receber o *input* do utilizador;
- *FixedUpdate()* – responsável por aplicar o *input* do utilizador ao movimento da personagem.

LADDERZONE

Deteta se o jogador está ou não numas escadas.

Métodos:

- *OnTriggerEnter2D(Collider2D other)* – aceita como argumento um objeto que tem associado um *collider2D* às suas propriedades. Caso esse objeto seja o

jogador, muda o estado da variável *onLadder* (variável definida na classe *PlayerMovement*) para verdadeiro;

- *OnTriggerExit2D(Collider2D other)* – aceita como argumento um objeto que tem associado um *collider2D* às suas propriedades. É um complemento do método anterior, uma vez que quando o jogador deixa de colidir com o objeto ao qual está associada esta classe, a variável *onLadder* toma o valor de falso;

VERTICALPLATFORM

Responsável por fazer com que o jogador possa subir ou descer a plataforma posicionada por cima das escadas.

Métodos:

- *Start()* – inicializa variáveis;
- *Update()* – altera as propriedades da plataforma de modo a que o jogador possa subir ou descer para as escadas.

ONWATER

Classe responsável por detetar quando o jogador se encontra debaixo de água.

Variáveis mais importantes:

- *public int damageToGive* – pontos de vida que são tirados à personagem;
- *private float timer* – cronómetro para definir o tempo que o jogador pode estar debaixo de água sem perder vida

Métodos:

- *Start()* – inicializa variáveis;
- *Update()* – método que aumenta o tempo do cronómetro e cancela a chamada da função responsável por tirar vida quando o jogador se encontra muito tempo debaixo de água;

- *OnTriggerEnter2D(Collider2D other)* – aceita como argumento um objeto que tem associado um *collider2D* às suas propriedades. Caso seja o jogador, chama o método *GiveDamageOnWater* após cinco segundos e caso o contacto permaneça, é repetidamente chamado a cada um segundo. Muda ainda o valor da variável *onWater* (definida na classe *PlayerMovement*) para verdadeiro;

- *OnTriggerExit2D(Collider2D other)* – aceita como argumento um objeto que tem associado um *collider2D* às suas propriedades. Define quando é que o método *GiveDamageOnWater* deixa de ser chamado, que é quando deixa de haver contacto. Muda o valor da variável *onWater* para falso;

- *GiveDamageOnWater* – retira um número de pontos de vida definidos ao jogador.

PLAYERPUSH

Onde podemos alterar a interação da personagem com a caixa e os seus comportamentos.

Variáveis mais importantes:

- *public float distance* – distância máxima do raio;
- *public LayerMask boxMask* – define quais as camadas que representam as caixas.

Métodos:

- *Update()* – deteta se a caixa está no raio de ação da personagem e caso esteja dá a possibilidade de a agarrar, carregando na tecla respetiva;

- *OnDrawGizmos()* – apenas cria uma linha que serve de auxílio para observar o raio de ação da personagem.

PLATFORMMOVE

Aplica o movimento da plataforma que se move ao jogador, quando este está em contacto com a mesma.

Métodos:

- *Start() – inicializa variáveis;*
- *OnCollisionEnter2D(Collider2D other)* – aceita como argumento um objeto que tem associado um *collider2D* às suas propriedades. Caso esse objeto seja a plataforma que se move, transforma a posição do jogador na posição da plataforma, de modo a que este se movimente com a plataforma;
- *OnCollisionExit2D(Collider2D other)* – aceita como argumento um objeto que tem associado um *collider2D* às suas propriedades. É um complemento do método anterior, uma vez que deixa de definir a posição do jogador assim que este deixa de estar em contacto com a plataforma.

CONTROLOS E MOVIMENTOS DOS ATAQUES DO JOGADOR

KNIFECONTROLLER

É definido o movimento e propriedades da arma, após o utilizador carregar na tecla associada à ação de disparar.

Variáveis mais importantes:

- *public float speed* – velocidade com que a arma se move;
- *public int damageToGive* – pontos de vida que são tirados aos inimigos.

Métodos:

- *Start()* - inicializa variáveis e muda a arma de sentido quando necessário;
- *Update()* – aplica a velocidade à arma, dando-lhe movimento;
- *OnTriggerEnter2D(Collider2D other)* – toma como argumento um objeto que tenha associado um *collider2D* às suas propriedades. Tira vida se entrar em contacto com um inimigo e pode partir um objeto que seja destrutível.

KILLENEMYJUMP

O jogador mata o inimigo 1 quando colide com a parte de cima do mesmo.

Variáveis mais importantes:

- *public float bounceOnEnemy* – força que é aplicada ao jogador quando salta em cima do inimigo, para dar efeito de ressalto;
- *public int damageToGive* - pontos de vida que são tirados aos inimigos.

Métodos:

- *Start()* – inicializa variáveis;
- *OnTriggerEnter2D(Collider2D other)* – aceita como argumento um objeto que tenha associado um *collider2D* às suas propriedades. Se colidir com um inimigo do tipo 1, aplica o efeito de ressalto e tira o número de pontos de vida definidos ao inimigo.

SISTEMA DE VIDA DO JOGADOR

PLAYERHEALTHMANAGER

Classe responsável pelo controlo da vida do jogador.

Variáveis mais importantes:

- *public static int playerHealth* – vida do jogador;
- *public int maxPlayerHealth* – vida máxima que o jogador pode ter;
- *public bool isDead* – deteta quando o jogador está ou não morto;
- *public Slider playerHealthBar* – barra onde a vida do jogador aparece. Diminui ou aumenta caso o jogador perca ou ganhe vida.

Métodos:

- *Start()* – inicializa variáveis;
- *Update()* – método que controla todas as ações que correspondem à vida do jogador. É neste método que são reduzidas o número de vidas do jogador quando este morre, é atualizada a sua vida, é dada ordem para renascer caso ainda tenha vidas suficientes e volta a colocar a chave final de cada nível no seu sítio em caso de morte do jogador;
- *HurtPlayer(int damageToGive)* – retira o número de pontos de vida definidos ao jogador;
- *HealthPlayer(int healthToGive)* – adiciona o número de pontos de vida definidos ao jogador quando este apanha o *item* correspondente;
- *FullHealth()* – repõe a vida máxima ao jogador.

HEALTHPICKUP

Adiciona vida ao sistema de vida do jogador.

Variáveis mais importantes:

- *public int healthToGive* – número de pontos de vida a adicionar.

Métodos:

- *OnTriggerEnter2D(Collider2D other)* – toma como argumento um objeto que tenha associado um *collider2D* às suas propriedades. Adiciona vida ao jogador quando este colide com o *item* responsável por dar vida. Reproduz ainda o som respetivo.

KILLPLAYER

Classe responsável por matar diretamente o jogador.

Métodos:

- *Start()* – inicializa variáveis;

- *OnTriggerEnter2D(Collider2D other)* – aceita como argumento um objeto que tem associado um *collider2D* às suas propriedades. Caso esse objeto seja o jogador, mata instantaneamente o mesmo, chamando depois o método responsável por fazer o jogador renascer.

MOVIMENTOS E AÇÕES DOS INIMIGOS

INIMIGO 1

ENEMYMOVEMENT

Classe onde é definido o movimento do inimigo.

Variáveis mais importantes:

- *public float moveSpeed* – velocidade com que o inimigo se move;
- *public bool moveRight* – deteta se o inimigo está a andar para a direita ou não;
- *private bool notAtEdge* – deteta se o inimigo não está perto do final da superfície;
- *private LayerMask whatIsWall* – define quais as camadas que são consideradas paredes (que fazem com que o inimigo mude de sentido).

Métodos:

- *Start()* – inicializa variáveis;
- *Update()* – responsável por detetar se o inimigo colide com uma parede ou se está perto do final da superfície e consequentemente é obrigado a mudar o sentido do movimento. É ainda aplicada a velocidade ao mesmo

EnemyHealthManager

Classe que controla a vida do inimigo.

Variáveis mais importantes:

- *public int enemyHealth* – vida do inimigo;
- *public int pointsOnDeath* – número de pontos atribuídos ao jogador cada vez que um inimigo é morto.

Métodos:

- *Update()* – é neste método que o inimigo morre, caso não tenha vida, sendo posteriormente reproduzido um efeito de simulação de sangue e adicionados os pontos ao jogador;
- *GiveDamage(int damageToGive)* – são retirados à vida do inimigo o número de pontos definidos.

HurtPlayerOnContact

É a classe responsável por retirar vida à personagem caso esta colida com algum inimigo.

Variáveis mais importantes:

- *public int damageToGive* – pontos de vida que são tirados à personagem.

Métodos:

- *Start()* – inicializa variáveis;
- *OnTriggerEnter2D(Collider2D other)* – toma como argumento um objeto que tenha associado um *collider2D* às suas propriedades. Caso o objeto que colida seja o jogador, é retirada vida ao mesmo, aplicando o efeito de *knockback* e reproduzindo o som associado à perda de vida do jogador.

INIMIGO 2

Contem as classes *EnemyHealthManager* e *HurtPlayerOnContact*. Contem ainda a classe *Enemy2Movement*, que é em tudo semelhante à classe *EnemyMovement*, apenas com a diferença que nesta classe as animações são tratadas de maneira diferente, visto que este inimigo tem mais do que uma animação.

CONTROLOS E MOVIMENTOS DA ARMA DO INIMIGO

ENEMYFIRECONTROLLER

Semelhante à classe *KnifeController*. É definido o movimento e propriedades da arma.

Variáveis mais importantes:

- *public float speed* – velocidade com que a arma se move;
- *public int damageToGive* – pontos de vida que são tirados ao jogador.

Métodos:

- *Start()* - inicializa variáveis e muda a arma de sentido quando necessário;
- *Update()* – aplica a velocidade à arma, dando-lhe movimento;
- *OnTriggerEnter2D(Collider2D other)* – toma como argumento um objeto que tenha associado um *collider2D* às suas propriedades. Tira vida ao jogador ao entrar em contacto com o mesmo.

SHOOTPLAYERINRANGE

Define quando é que um inimigo pode disparar. Cada inimigo tem o mesmo alcance e só dispara quando o jogador está dentro do alcance.

Variáveis mais importantes:

- *public float playerRange* – alcance máximo do inimigo;
- *public Transform firePoint* – ponto a partir do qual o inimigo dispara;
- *public float waitBetweenShots* – tempo de intervalo entre cada disparo.

Métodos:

- *Start()* – inicializa variáveis;
- *Update()* – é desenhada uma linha para ser visível o alcance de cada inimigo. É ainda responsável por fazer cada inimigo disparar.

INIMIGO 3

Contém as classes *EnemyHealthManager* e *HurtPlayerOnContact* tal como nos inimigos 1 e 2.

FLYERENEMYMOVEMENT

Variáveis mais importantes:

- *Public float moveSpeed* - velocidade com que o inimigo se move;
- *Public float playerRange* – alcance a que o inimigo detecta o jogador;
- *Public bool playerInRange* – para saber se o jogador esta dentro do *playerRange* ou não;

Métodos:

- *Start()* – inicializa variáveis;
- *Update()* – é responsável por definir o *playerInRange* e fazer o inimigo movimentar-se para o jogador consoante essa variável;

Boss

BossPatrol

Responsável pelo movimento deste inimigo.

Variáveis mais importantes:

- *Public float moveSpeed* - velocidade com que o inimigo se move;
- *Public bool moveRight* – deteta se o inimigo está a andar para a direita;
- *Public int pointsOnDeath* – pontos que o jogador ganha quando o mata.

Métodos:

- *Start()* – inicializa variáveis;
- *Update()* – é responsável por movimentar o inimigo.

BossHealthmANAGER

Coordena o sistema de saúde do inimigo.

Variáveis mais importantes:

- *Public int enemyHealth* – vida do inimigo;
- *Public int enemyHealth2* – vida dos inimigos quando se separam;
- *Public int pointsOnDeath* – pontos que o jogador ganha quando o mata.

Métodos:

- *Update()* – é responsável por separar o inimigo quando o matamos e separar o mesmo em dois. É ainda responsável por matar esses dois.
- *GiveDamage()* – aplica dano ao inimigo.

BOSSWALL

Métodos:

- *Update()* – abre a porta quando o inimigo é completamente morto.

PICOS

UPANDDOWN

Aplica o movimento de subir e descer aos picos.

Variáveis mais importantes:

- *public float downDelay* – tempo que os picos demoram a ir para baixo;
- *public float moveSpeed* – velocidade a que os picos se movimentam.

Métodos:

- *Start()* – inicializa variáveis;
- *Update()* – chama a função *GoDown;*
- *IEnumerator GoDown()* – responsável pelo movimento para cima e para baixo dos picos

PICOS A CAIR

FALL DOWN

Classe responsável por fazer cair os picos quando o jogador passa por baixo destes.

Variáveis mais importantes:

- *public float fallVelocity* – velocidade com que os picos caem.

Métodos:

- *Start()* – inicializa variáveis;
- *Update()* – deteta quando o jogador passa por baixo dos picos e fá-los cair com a velocidade definida.

PÊNDULO

PENDULUM

Responsável pelo movimento do pêndulo.

Variáveis mais importantes:

- *public float leftPushRange* – alcance máximo do pêndulo no movimento para a esquerda;
- *public float rightPushRange* – alcance máximo do pêndulo no movimento para a direita;
- *public float velocityThreshold* – velocidade máxima que o pêndulo pode alcançar.

Métodos:

- *Start()* – inicializa as variáveis;
- *Update()* – chama o método *Push();*
- *Push()* – define e aplica o movimento ao pêndulo.

SISTEMA E GESTÃO DE PONTOS, VIDAS, MOEDAS, CHAVES E TEMPO

SISTEMA DE PONTOS

SCOREMANAGER

Responsável pela contagem dos pontos, pontos esses que são visíveis para o utilizador e vão sendo atualizados.

Variáveis mais importantes:

- *public static int score* – número de pontos do jogador;

Métodos:

- *Start()* – inicialização das variáveis;
- *Update()* – método que faz com que os pontos sejam atualizados no ecrã;
- *AddPoints(int pointsToAdd)* – são adicionados à variável *score* o número de pontos definidos;
- *Reset()* – redefine o número de pontos para zero.

SISTEMA DE VIDAS

LIFEMANAGER

Onde o número de vidas (tentativas) restantes do jogador é controlado. O número de vidas é visível para o utilizador e vai sendo atualizado.

Variáveis mais importantes:

- *public int startingLifes* – número de vidas que o jogador tem no começo do jogo;
- *private int lifeCounter* – número de vidas, atualizado, que o jogador tem ao longo do jogo;

- *public GameObject gameOverScreen* – ecrã de fim de jogo que é visível quando o jogador morre e não tem mais vidas;
- *public float waitAfterGameOver* – tempo até aparecer o ecrã de fim de jogo.

Métodos:

- *Start()* – inicializa variáveis;
- *Update()* – responsável por mostrar o número de vidas do jogador e ativar o ecrã de final de jogo. Retorna ainda ao menu inicial após *input* respetivo do utilizador;
- *GiveLife()* – método que adiciona uma vida ao total;
- *TakeLife()* – método que retira uma vida ao total;

LIFEPICKUP

Adiciona uma vida quando o jogador apanha um *item* que corresponde ao número de vidas.

Métodos:

- *Start()* – inicializa variáveis;
- *OnTriggerEnter2D(Collider2D other)* – aceita como argumento um objeto que tem associado um *collider2D* às suas propriedades. Caso esse objeto seja o jogador, adiciona uma vida ao sistema de vidas. Reproduz ainda um som respetivo.

SISTEMA DE MOEDAS

COINSMANAGER

Responsável pela contagem das moedas, que são visíveis para o utilizador e vão sendo atualizadas.

Variáveis mais importantes:

- *private int coins* – número de moedas do jogador;

Métodos:

- *Start()* – inicializa variáveis;
- *Update()* – apresenta o número de moedas atualizadas no ecrã;
- *AddCoins()* – adiciona uma moeda ao número total de moedas;
- *Reset()* – redefine o número total de moedas em zero.

CoinPickup

Adiciona uma moeda quando o jogador apanha o *item* correspondente.

Variáveis mais importantes:

- *public int pointsToAdd* – número de pontos a adicionar quando se apanha uma moeda

Métodos:

- *Start()* – inicializa variáveis;
- *OnTriggerEnter2D(Collider2D other)* – aceita como argumento um objeto que tem associado um *collider2D* às suas propriedades. Caso esse objeto seja o jogador, adiciona uma moeda ao número total de moedas e o número de pontos definidos ao sistema de pontos. Reproduz ainda um som respetivo.

Sistema de chaves

KeyManager

Guarda o número de chaves apanhadas pelo jogador.

Variáveis mais importantes:

- *public int keys* – número total de chaves que o jogador apanhou.

Métodos:

- *Start()* – define o número de chaves em zero;
- *Update()* – normaliza o número de chaves para tomar valores negativos;
- *AddKey()* – adiciona uma chave ao número total de chaves;
- *TakeKey()* – retira uma chave ao número total de chaves.

KeyPickup

Responsável por adicionar uma chave quando o jogador apanha o *item* correspondente.

Variáveis mais importantes:

- *public bool hasKey* – deteta se o jogador apanhou a chave do nível correspondente.

Métodos:

- *Start()* – inicializa variáveis;
- *OnTriggerEnter2D(Collider2D other)* – aceita como argumento um objeto que tem associado um *collider2D* às suas propriedades. Caso o jogador colida com o *item* associado à chave, adiciona uma chave ao número total de chaves.

Sistema de tempo

TimeManager

Quando aplicável, apresenta o tempo restante para completar uma tarefa.

Variáveis mais importantes:

- *public float startingTime* – tempo inicial disponível para completar uma tarefa;
- *private float time* – tempo restante para completar a tarefa;

Métodos:

- *Start()* – inicializa variáveis;
- *Update()* – responsável por apresentar no ecrã o tempo disponível, bem como tirar a vida ao jogador caso ele não complete a tarefa no tempo definido.

CONTROLO DO NÍVEL

LEVELMANAGER

Responsável por controlar pormenores referentes a cada nível, como o renascimento do jogador ou tirar o som do jogo.

Variáveis mais importantes:

- *public int respawnDelay* – tempo que o jogador demora a renascer após a morte;
- *public GameObject currentCheckpoint* – posição no nível onde o jogador renasce.

Métodos:

- *Start()* – inicializa variáveis;
- *Mute()* – tira o som do jogo;
- *RespawnPlayer()* – chama a função responsável pelo renascimento do jogador;
- *IEnumerator RespawnPlayerCo()* – controla a posição onde o jogador renasce. Reproduz ainda alguns efeitos quando o mesmo morre ou ressuscita.

CHECKPOINT

Guarda a posição onde o jogador vai renascer.

Métodos:

- *Start()* – inicializa variáveis;
- *OnTriggerEnter2D(Collider2D other)* – aceita como argumento um objeto que tem associado um *collider2D* às suas propriedades. Caso o jogador colida com objeto ao qual está associado esta função, a posição vai ser guardada e o mesmo vai renascer nesse sítio. Quando o jogador colide com outro *checkpoint*, passa a renascer nessa posição.

LEVELLOADER

Carrega uma nova cena (nível). Ativa um ecrã onde aparece uma barra que informa o estado do carregamento.

Variáveis mais importantes:

- *private bool playerInZone* – deteta se o jogador está na zona certa para poder mudar de nível;
- *public Slider slider* – barra onde aparece o estado de carregamento de um nível.

Métodos:

- *Awake()* – desativa o ecrã de carregamento;
- *Start()* – inicializa variáveis;
- *Update()* – chama a função responsável pelo carregamento de um nível diferente;
- *OnTriggerEnter2D(Collider2D other)* – aceita como argumento um objeto que tem associado um *collider2D* às suas propriedades. Caso esse objeto seja o jogador, muda o estado da variável *playerInZone* para verdadeiro;
- *OnTriggerExit2D(Collider2D other)* – aceita como argumento um objeto que tem associado um *collider2D* às suas propriedades. É um complemento do método

anterior, uma vez que quando o jogador deixa de colidir com o objeto ao qual está associada esta classe, a variável *playerInZone* toma o valor de falso;

- *IEnumerator LoadAsynchronously(string levelToLoad)* – toma como argumento o nome do nível que se pretende carregar. Este método é responsável por carregar o nível desejado e ativar o ecrã de carregamento.

CAMERAFOLLOW

Classe que torna o jogador sempre o ponto central da câmara.

Variáveis mais importantes:

- *private float xMin, xMax, yMin, yMax* – variáveis que limitam as posições máximas e mínimas da câmara, nos eixos x e y;
- *private Transform target* – alvo a tornar como centro da câmara.

Métodos:

- *Start()* – determina qual o ponto central da câmara;
- *LateUpdate()* – atualiza a posição da câmara.

CONTROLO DE OBJETOS DO JOGO

MOVEPLATFORM

Move uma plataforma entre dois ou mais pontos.

Variáveis mais importantes:

- *public float moveSpeed* – velocidade com que a plataforma se desloca;

Métodos:

- *Start()* – inicializa variáveis;
- *Update()* – método responsável por aplicar o movimento à plataforma.

MOVEWALL

Quando uma alavanca é puxada, a parede respetiva move-se.

Variáveis mais importantes:

- *private bool playerInZone* – deteta se o jogador está em zona possível de puxar a alavanca;
- *public Sprite up* – imagem da alavanca na sua posição natural;
- *public Sprite down* – imagem da alavanca após ser puxada pelo jogador;
- *public GameObject wall* – parede a ser movida.

Métodos:

- *Awake()* – define a imagem da alavanca na sua posição natural;
- *Start()* – inicializa as variáveis;
- *Update()* – verifica se o jogador está em condições de puxar a alavanca e se sim, altera a imagem da mesma e chama a função responsável por mover a parede;
- *IEnumerator Move()* – move a parede e mantém a mesma nessa posição durante algum tempo. Quando esse tempo passa, a parede volta à posição inicial e a imagem da alavanca muda;
- *OnTriggerEnter2D(Collider2D other)* – aceita como argumento um objeto que tem associado um *collider2D* às suas propriedades. Caso esse objeto seja o jogador, muda o estado da variável *playerInZone* para verdadeiro;
- *OnTriggerExit2D(Collider2D other)* – aceita como argumento um objeto que tem associado um *collider2D* às suas propriedades. É um complemento do método anterior, uma vez que quando o jogador deixa de colidir com o objeto ao qual está associada esta classe, a variável *playerInZone* toma o valor de falso;

FALLING PLATFORM

Classe que tem como função fazer com que uma plataforma caia quando o jogador entra em contacto com a mesma.

Variáveis mais importantes:

- *public float fallDelay* – tempo que demora, após o contacto, para a plataforma cair;
- *public bool isFalling* – deteta se a plataforma está a cair ou não;
- *public Vector2 initialPosition* – posição inicial da plataforma.

Métodos:

- *Awake()* – inicializa variáveis;
- *Start()* – guarda a posição inicial da plataforma;
- *OnCollisionEnter2D(Collision2D other)* – aceita como argumento um objeto que tenha a propriedade *collision2D* associada. Se o objeto que colidir com o objeto ao qual esta classe está associada for o jogador, chama a função *Fall()*, demorando essa chamada o tempo definido anteriormente;
- *Fall()* – muda algumas propriedades da plataforma e aplica-lhe velocidade, para que possa cair;
- *Respawn()* – chama a função *RespawnCo()*;
- *IEnumerator RespawnCo()* – repõe as propriedades iniciais da plataforma e volta a coloca-la na posição inicial;
- *OnTriggerEnter2D(Collision2D other)* – toma como argumento um objeto que tem a propriedade *collision2D* associada. Se esse objeto for o limite do nível, chama a função *Respawn()*.

CONTROLO DO MENU DE PAUSA

PAUSEMENU

Pausa o jogo e ativa o *menu* de pausa.

Variáveis mais importantes:

- *public bool isPaused* – deteta se o jogo está na pausa ou não;
- *public bool inSoundMenu* – deteta se o ecrã de opções de som está ativo;
- *public GameObject pauseMenuCanvas* – ecrã de pausa;
- *public GameObject soundMenuCanvas* – ecrã de opções de som;

Métodos:

- *Update()* – ativa/desativa o ecrã de pausa e pausa/retoma o jogo;
- *Resume()* – cria as condições necessárias para retomar o jogo;
- *Options()* – desativa o ecrã de pausa e ativa o ecrã de opções de som;
- *Back()* – desativa o ecrã de opções de som e ativa o ecrã de pausa novamente;
- *Quit()* – carrega o nível desejado.

MANUAL DE UTILIZADOR JOGO 2D

- Carregar no *setup.exe*;
- Seguir as instruções do *setup*, que vai criar um atalho para o jogo no ambiente de trabalho;
- Carregar duas vezes em cima do atalho do jogo e a partir daqui é só desfrutar.

MANUAL DO PROGRAMADOR DO WEBSITE

PÁGINA HTML:

Bastante simples *one page website*, com uma *navbar* que nos leva para 4 zonas da página.

NAVBAR:

- *Home* – que leva de volta para o topo da página
- Sobre Nós – que leva para a parte onde tem uma introdução
- *Download* – onde tem 2 icones que leva para a *appstore* ou *applestore* de modo a fazer *download* do jogo.
- Ajuda – onde os utilizadores podem expôr as suas duvidas

SIGN UP BUTTON

Index.php:

- Página que aparece depois de ser feito um *login* válido.

Register.php:

- Página que aparece ao carregar no botão "*Sign Up*" que está na *navbar*, onde para se registar tem que escolher um *username, email, password* e *confirm password* através de um *input*, e no fim confirmar o registo carregando no *register* que é um *button.*

Login.php:

- Onde se insere o *username* e *password*, através de um *input*, já criados de modo a poder fazer *login in.*

Server.php:

Onde se faz a ligação a base de dados para registar um utilizador novo ou para deixar um que já exista fazer *login in*.

- *$user_check_query* – para verificar se já existe um utilizador com o *username* ou *email* que o novo utilizador está a usar para fazer um novo registo.
- Se já existir aquele *username* ou *email* temos 2 condições: uma que manda um erro se já existir aquele *user* e outra que manda um erro se já existir aquele *email.*
- Se não tiver erros faz o registo do utilizador, onde encripta a *password* com md5, e insere os dados que acabaram de ser introduzidos na base de dados.
- Se o utilizador estiver a fazer o *login* vai inserir dados em dois campos, *username* e *password,* se esses campos estiverem vazios ele depara-se com dois erros: um que é mandado quando se tem o campo do *username* vazio, que diz que não se pode ter aquele campo vazio, e outro quando se tem o campo da *password* vazio que manda o mesmo erro.
- Se os dois campos estiverem preenchidos a *password* é desencriptada, ele vai a base de dados procurar onde aquela *pass* e *user* e se estiverem o utilizador está oficialmente *logged in* mas se o contrário se verificar é mandado um erro que diz que aquela combinação está errada.

JAVASCRIPT:

VAR CFG:

Para escolhermos o tempo do scroll.

Variaveis:

- *scrollDuration* - onde definimos o tempo que o scroll demora.

VAR DOC:

Para detetar diferentes *browsers*.

VAR SSPRELOADER:

Onde é definido o que acontece ao *preloader*.

- *$('html, body').animate({ scrollTop: 0 }, 'normal')* - para forçar a página a voltar ao topo quando é atualizada
- *$("#preloader").delay(500).fadeOut('slow')* - e fazer com o *preloader* que cobre todo o site desapareça.

VAR SSWAYPOINTS:

Serve para as secções que estão na *navbar* mudem de cor quando temos o cursor sobre elas.

Variáveis:

- *Var active_section* – parte do *site* que está ativa;
- *Var active_link* – parte da *navbar* que está ativa.

VAR SSSMOOTHSCROLL:

Para que ao fazer *scroll* na página este seja suave.

FUNCTION SSINIT:

- Onde todos os var que tem o script são inicializados.

MANUAL DE UTILIZADOR DO WEBSITE

- *Site* de uma página, tem uma secção de onde falamos sobré nós, outra onde pode fazer *download* do jogo, tanto na *appstore* como no *applestore*, e uma de ajuda.
- Pode fazer inscrição no *site* carregando no botão "*sign up*", onde vai inserir um nome de usuário, *email* e password.

DIFICULDADES

Apesar de esteticamente o jogo ter ficado do nosso agrado, e termos alcançado algo perto do que imaginámos, em termos de desempenho não podemos dizer o mesmo. Fomos experienciando algumas dificuldades ao longo do desenvolvimento deste projeto, e se conseguimos resolver maior parte delas com ajuda do fórum do *Unity*[7] ou através de vídeos do *Youtube*[4], outras nem tanto. A parte que correu menos bem neste projeto foi o facto de não conseguirmos passar os dados do jogo para o *site*, objetivo que foi proposto inicialmente. Fomos tentando que as informações sobre as moedas ou os pontos, por exemplo, fossem mostradas no *site*, mas sem sucesso. Outro dos pontos negativos deste projeto, foi o facto das mudanças feitas na loja não se aplicarem no resto do jogo.

CONCLUSÃO

O objetivo proposto por nós foi desenvolver um jogo 2D e de uma página *web* associada ao mesmo. Sem qualquer tipo de bases no que toca aos processos de desenvolvimento de um jogo – apenas tínhamos conhecimento da linguagem de programação que o suporta – resolvemos arriscar. Chegado ao fim do projeto, só podemos estar contentes com o resultado final. É certo que havia coisas para melhorar (aspetos que mencionámos anteriormente), mas no geral o resultado foi positivo. Sem dúvida que a página *web* está muito mais simples do que o jogo em si, gostaríamos de ter desenvolvido mais esse aspeto. No entanto, a parte do jogo em si está bastante satisfatória, funcionando fluentemente.

Todo este projeto ajudou-nos a crescer enquanto alunos e sem dúvida a enriquecer os nossos conhecimentos.

REFERÊNCIAS

1.How to BUILD / EXPORT your Game in Unity (Windows | Mac | WebGL). Disponível em: HTTPS://WWW.YOUTUBE.COM/WATCH?V=7NXKATXGSN8. Visto a: 16 set. 2018.

2.Unity Android Tutorial : SDK, Build and Publish. Disponível em HTTPS://WWW.YOUTUBE.COM/WATCH?V=1YLDXIMURLO. Visto a: 16 set. 2018.

3.Complete User Registration system using PHP and MySQL database. Disponível em HTTPS://WWW.YOUTUBE.COM/WATCH?V=C--MU07UHQW. Visto a 16 set. 2018.

4. UNITY 2D PLATFORMER TUTORIAL - LEARN THE BASICS OF MAKING A GAME!. Disponível em: HTTPS://WWW.YOUTUBE.COM/WATCH?V=IXK2UXEC94&LIST=PLIYFVMTJWC_UP8X NVM3OSQGBJOMQGHKVZ. Visto a 10 julho. 2018

5.How to Make a One Page Website. Disponível em : HTTPS://CODEPLANET.IO/HOW-TO-MAKE-A-SINGLE-PAGE-WEBSITE/. Visto a 6 agosto 2018

6.Quick Delete PlayerPrefs with Menu Button – Unity. Disponível em: HTTPS://WWW.TANGLEDREALITYSTUDIOS.COM/CODE-EXAMPLES/QUICK-DELETE-PLAYERPREFS-WITH-EDITOR-MENU-BUTTON/ . Visto a 13 set. 2018

7.Dúvidas . Disponível em: HTTPS://STACKOVERFLOW.COM/

8.Dúvidas . Disponível em: HTTPS://FORUM.UNITY.COM/

9.HOW TO MAKE A 2D PLATFORMER - UNITY COURSE. Disponível em: HTTPS://WWW.YOUTUBE.COM/WATCH?V=UBPICGCKHTE&LIST=PLPV2KYIB3JR42O VBU6K2DIL6Y22RY9J1C. Visto a 11 julho 2018

10. O que é um engine ou um motor gráfico? Disponível em: HTTPS://WWW.TECMUNDO.COM.BR/VIDEO-GAME-E-JOGOS/9263-O-QUE-E-ENGINE-OU-MOTOR-GRAFICO-.HTM Visto a 18 junho 2018

11. Top Game Engines. Disponível em: HTTPS://INSTABUG.COM/BLOG/GAME-ENGINES/ Visto a 18 junho 2018

12. *Unity site.* Disponível em: HTTPS://UNITY3D.COM/PT Visto a 20 junho 2018